MACHINE LEARNING MASTERY

Generative Adversarial Networks

WITH PYTHON

Deep Learning Generative Models for Image Synthesis and Image Translation

Jason Brownlee

Disclaimer

The information contained within this eBook is strictly for educational purposes. If you wish to apply ideas contained in this eBook, you are taking full responsibility for your actions.
The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.
No part of this eBook may be reproduced or transmitted in any form or by any means, electronic or mechanical, recording or by any information storage and retrieval system, without written permission from the author.

Acknowledgements

Special thanks to my proofreader Sarah Martin and my technical editors Arun Koshy, Andrei Cheremskoy, and Michael Sanderson.

Copyright

Generative Adversarial Networks with Python
© Copyright 2019 Jason Brownlee. All Rights Reserved.

Edition: v1.81

Contents

Copyright i

Contents ii

Preface iii

Introductions v

Welcome v

I Foundations 1

1 What are Generative Adversarial Networks 3
1.1 Overview . . . 3
1.2 What Are Generative Models? . . . 4
1.3 What Are Generative Adversarial Networks? . . . 7
1.4 Why Generative Adversarial Networks? . . . 12
1.5 Further Reading . . . 13
1.6 Summary . . . 14

2 How to Develop Deep Learning Models With Keras 16
2.1 Tutorial Overview . . . 16
2.2 Keras Model Life-Cycle . . . 17
2.3 Keras Functional Models . . . 22
2.4 Standard Network Models . . . 23
2.5 Further Reading . . . 29
2.6 Summary . . . 29

3 How to Upsample with Convolutional Neural Networks 30
3.1 Tutorial Overview . . . 30
3.2 Need for Upsampling in GANs . . . 31
3.3 How to Use the Upsampling Layer . . . 31
3.4 How to Use the Transpose Convolutional Layer . . . 36
3.5 Further Reading . . . 42
3.6 Summary . . . 43

4 How to Implement the GAN Training Algorithm 44
4.1 Tutorial Overview . . . 44
4.2 How to Implement the GAN Training Algorithm . . . 45
4.3 Understanding the GAN Loss Function . . . 48
4.4 How to Train GAN Models in Practice . . . 50
4.5 Further Reading . . . 51
4.6 Summary . . . 51

5 How to Implement GAN Hacks to Train Stable Models 53
5.1 Tutorial Overview . . . 53
5.2 Challenge of Training GANs . . . 54
5.3 Heuristics for Training Stable GANs . . . 55
5.4 Deep Convolutional GANs (DCGANs) . . . 55
5.5 Soumith Chintala's GAN Hacks . . . 61
5.6 Further Reading . . . 65
5.7 Summary . . . 66

II GAN Basics 67

6 How to Develop a 1D GAN from Scratch 69
6.1 Tutorial Overview . . . 70
6.2 Select a One-Dimensional Function . . . 70
6.3 Define a Discriminator Model . . . 73
6.4 Define a Generator Model . . . 77
6.5 Training the Generator Model . . . 82
6.6 Evaluating the Performance of the GAN . . . 86
6.7 Complete Example of Training the GAN . . . 89
6.8 Extensions . . . 93
6.9 Further Reading . . . 94
6.10 Summary . . . 94

7 How to Develop a DCGAN for Grayscale Handwritten Digits 95
7.1 Tutorial Overview . . . 95
7.2 MNIST Handwritten Digit Dataset . . . 96
7.3 How to Define and Train the Discriminator Model . . . 98
7.4 How to Define and Use the Generator Model . . . 105
7.5 How to Train the Generator Model . . . 112
7.6 How to Evaluate GAN Model Performance . . . 117
7.7 Complete Example of GAN for MNIST . . . 119
7.8 How to Use the Final Generator Model . . . 126
7.9 Extensions . . . 129
7.10 Further Reading . . . 130
7.11 Summary . . . 130

8 How to Develop a DCGAN for Small Color Photographs 132
8.1 Tutorial Overview . . . 132
8.2 CIFAR-10 Small Object Photograph Dataset . . . 133
8.3 How to Define and Train the Discriminator Model . . . 135
8.4 How to Define and Use the Generator Model . . . 143
8.5 How to Train the Generator Model . . . 150
8.6 How to Evaluate GAN Model Performance . . . 155
8.7 Complete Example of GAN for CIFAR-10 . . . 158
8.8 How to Use the Final Generator Model . . . 166
8.9 Extensions . . . 169
8.10 Further Reading . . . 169
8.11 Summary . . . 170

9 How to Explore the Latent Space When Generating Faces 171
9.1 Tutorial Overview . . . 171
9.2 Vector Arithmetic in Latent Space . . . 172
9.3 Large-Scale CelebFaces Dataset (CelebA) . . . 174
9.4 How to Prepare the CelebA Faces Dataset . . . 174
9.5 How to Develop a GAN for CelebA . . . 181
9.6 How to Explore the Latent Space for Generated Faces . . . 191
9.7 Extensions . . . 205
9.8 Further Reading . . . 205
9.9 Summary . . . 206

10 How to Identify and Diagnose GAN Failure Modes 207
10.1 Tutorial Overview . . . 207
10.2 How To Train a Stable GAN . . . 208
10.3 How To Identify a Mode Collapse . . . 223
10.4 How To Identify Convergence Failure . . . 229
10.5 Further Reading . . . 242
10.6 Summary . . . 242

III GAN Evaluation 243

11 How to Evaluate Generative Adversarial Networks 245
11.1 Overview . . . 245
11.2 Problem with Evaluating Generator Models . . . 246
11.3 Manual GAN Generator Evaluation . . . 246
11.4 Qualitative GAN Generator Evaluation . . . 247
11.5 Quantitative GAN Generator Evaluation . . . 248
11.6 Which GAN Evaluation Scheme to Use . . . 250
11.7 Further Reading . . . 251
11.8 Summary . . . 252

12 How to Implement the Inception Score **253**
12.1 Tutorial Overview . . . 253
12.2 What Is the Inception Score? . . . 254
12.3 How to Calculate the Inception Score . . . 254
12.4 How to Implement the Inception Score With NumPy . . . 256
12.5 How to Implement the Inception Score With Keras . . . 258
12.6 Problems With the Inception Score . . . 264
12.7 Further Reading . . . 264
12.8 Summary . . . 265

13 How to Implement the Frechet Inception Distance **266**
13.1 Tutorial Overview . . . 266
13.2 What Is the Frechet Inception Distance? . . . 267
13.3 How to Calculate the FID . . . 268
13.4 How to Implement the FID With NumPy . . . 269
13.5 How to Implement the FID With Keras . . . 271
13.6 How to Calculate the FID for Real Images . . . 275
13.7 Further Reading . . . 277
13.8 Summary . . . 278

IV GAN Loss **279**

14 How to Use Different GAN Loss Functions **281**
14.1 Overview . . . 281
14.2 Challenge of GAN Loss . . . 282
14.3 Standard GAN Loss Functions . . . 282
14.4 Alternate GAN Loss Functions . . . 284
14.5 Effect of Different GAN Loss Functions . . . 286
14.6 Further Reading . . . 287
14.7 Summary . . . 287

15 How to Develop a Least Squares GAN (LSGAN) **288**
15.1 Tutorial Overview . . . 288
15.2 What Is Least Squares GAN . . . 289
15.3 How to Develop an LSGAN for MNIST . . . 291
15.4 How to Generate Images With LSGAN . . . 302
15.5 Further Reading . . . 304
15.6 Summary . . . 305

16 How to Develop a Wasserstein GAN (WGAN) **306**
16.1 Tutorial Overview . . . 306
16.2 What Is a Wasserstein GAN? . . . 307
16.3 How to Implement Wasserstein Loss . . . 308
16.4 Wasserstein GAN Implementation Details . . . 310
16.5 How to Train a Wasserstein GAN Model . . . 314
16.6 How to Generate Images With WGAN . . . 328

16.7 Further Reading . . . 330
16.8 Summary . . . 331

V Conditional GANs 332

17 How to Develop a Conditional GAN (cGAN) 334
17.1 Tutorial Overview . . . 334
17.2 Conditional Generative Adversarial Networks . . . 335
17.3 Fashion-MNIST Clothing Photograph Dataset . . . 337
17.4 Unconditional GAN for Fashion-MNIST . . . 339
17.5 Conditional GAN for Fashion-MNIST . . . 348
17.6 Conditional Clothing Generation . . . 360
17.7 Extensions . . . 361
17.8 Further Reading . . . 362
17.9 Summary . . . 363

18 How to Develop an Information Maximizing GAN (InfoGAN) 364
18.1 Tutorial Overview . . . 364
18.2 What Is the Information Maximizing GAN . . . 365
18.3 How to Implement the InfoGAN Loss Function . . . 367
18.4 How to Develop an InfoGAN for MNIST . . . 369
18.5 How to Use Control Codes With an InfoGAN . . . 386
18.6 Extensions . . . 391
18.7 Further Reading . . . 391
18.8 Summary . . . 393

19 How to Develop an Auxiliary Classifier GAN (AC-GAN) 394
19.1 Tutorial Overview . . . 394
19.2 Auxiliary Classifier Generative Adversarial Networks . . . 395
19.3 Fashion-MNIST Clothing Photograph Dataset . . . 397
19.4 How to Define AC-GAN Models . . . 397
19.5 How to Develop an AC-GAN for Fashion-MNIST . . . 407
19.6 How to Generate Items of Clothing With the AC-GAN . . . 417
19.7 Extensions . . . 420
19.8 Further Reading . . . 420
19.9 Summary . . . 421

20 How to Develop a Semi-Supervised GAN (SGAN) 423
20.1 Tutorial Overview . . . 423
20.2 What Is the Semi-Supervised GAN? . . . 424
20.3 How to Implement the Semi-Supervised Discriminator . . . 425
20.4 How to Develop a Semi-Supervised GAN for MNIST . . . 439
20.5 How to Use the Final SGAN Classifier Model . . . 449
20.6 Extensions . . . 450
20.7 Further Reading . . . 451
20.8 Summary . . . 452

VI Image Translation 453

21 Introduction to Pix2Pix 455
21.1 Overview . . . 455
21.2 The Problem of Image-to-Image Translation . . . 456
21.3 Pix2Pix GAN for Image-to-Image Translation . . . 456
21.4 Pix2Pix Architectural Details . . . 457
21.5 Applications of the Pix2Pix GAN . . . 459
21.6 Insight into Pix2Pix Architectural Choices . . . 463
21.7 Further Reading . . . 465
21.8 Summary . . . 466

22 How to Implement Pix2Pix Models 467
22.1 Tutorial Overview . . . 467
22.2 What Is the Pix2Pix GAN? . . . 468
22.3 How to Implement the PatchGAN Discriminator Model . . . 468
22.4 How to Implement the U-Net Generator Model . . . 474
22.5 How to Implement Adversarial and L1 Loss . . . 478
22.6 How to Update Model Weights . . . 483
22.7 Further Reading . . . 485
22.8 Summary . . . 486

23 How to Develop a Pix2Pix End-to-End 487
23.1 Tutorial Overview . . . 487
23.2 What Is the Pix2Pix GAN? . . . 488
23.3 Satellite to Map Image Translation Dataset . . . 488
23.4 How to Develop and Train a Pix2Pix Model . . . 492
23.5 How to Translate Images With a Pix2Pix Model . . . 505
23.6 How to Translate Google Maps to Satellite Images . . . 512
23.7 Extensions . . . 515
23.8 Further Reading . . . 516
23.9 Summary . . . 516

24 Introduction to the CycleGAN 518
24.1 Overview . . . 518
24.2 Problem With Image-to-Image Translation . . . 519
24.3 Unpaired Image-to-Image Translation With CycleGAN . . . 520
24.4 What Is the CycleGAN Model Architecture . . . 521
24.5 Applications of CycleGAN . . . 522
24.6 Implementation Tips for CycleGAN . . . 526
24.7 Further Reading . . . 527
24.8 Summary . . . 527

25 How to Implement CycleGAN Models 529
25.1 Tutorial Overview . . . 529
25.2 What Is the CycleGAN Architecture? . . . 530
25.3 How to Implement the CycleGAN Discriminator Model . . . 531

25.4 How to Implement the CycleGAN Generator Model . . . 536
25.5 How to Implement Composite Models and Loss . . . 540
25.6 How to Update Model Weights . . . 546
25.7 Further Reading . . . 550
25.8 Summary . . . 551

26 How to Develop the CycleGAN End-to-End 552
26.1 Tutorial Overview . . . 552
26.2 What Is the CycleGAN? . . . 553
26.3 How to Prepare the Horses to Zebras Dataset . . . 553
26.4 How to Develop a CycleGAN to Translate Horse to Zebra . . . 556
26.5 How to Perform Image Translation with CycleGAN . . . 571
26.6 Extensions . . . 579
26.7 Further Reading . . . 579
26.8 Summary . . . 580

VII Advanced GANs 581

27 Introduction to the BigGAN 583
27.1 Overview . . . 583
27.2 Brittleness of GAN Training . . . 583
27.3 Develop Better GANs by Scaling Up . . . 584
27.4 How to Scale-Up GANs With BigGAN . . . 585
27.5 Example of Images Generated by BigGAN . . . 590
27.6 Further Reading . . . 592
27.7 Summary . . . 593

28 Introduction to the Progressive Growing GAN 594
28.1 Overview . . . 594
28.2 GANs Are Generally Limited to Small Images . . . 595
28.3 Generate Large Images by Progressively Adding Layers . . . 595
28.4 How to Progressively Grow a GAN . . . 596
28.5 Images Generated by the Progressive Growing GAN . . . 598
28.6 How to Configure Progressive Growing GAN Models . . . 600
28.7 Further Reading . . . 602
28.8 Summary . . . 602

29 Introduction to the StyleGAN 603
29.1 Overview . . . 603
29.2 Lacking Control Over Synthesized Images . . . 604
29.3 Control Style Using New Generator Model . . . 604
29.4 What Is the StyleGAN Model Architecture . . . 605
29.5 Examples of StyleGAN Generated Images . . . 608
29.6 Further Reading . . . 610
29.7 Summary . . . 611

VIII Appendix 612

A Getting Help 613
A.1 Applied Neural Networks . . . 613
A.2 Programming Computer Vision Books . . . 613
A.3 Official Keras Destinations . . . 614
A.4 Where to Get Help with Keras . . . 614
A.5 How to Ask Questions . . . 615
A.6 Contact the Author . . . 615

B How to Setup Python on Your Workstation 616
B.1 Overview . . . 616
B.2 Download Anaconda . . . 616
B.3 Install Anaconda . . . 618
B.4 Start and Update Anaconda . . . 620
B.5 Install Deep Learning Libraries . . . 623
B.6 Further Reading . . . 624
B.7 Summary . . . 624

C How to Setup Amazon EC2 for Deep Learning on GPUs 625
C.1 Overview . . . 625
C.2 Setup Your AWS Account . . . 626
C.3 Launch Your Server Instance . . . 627
C.4 Login, Configure and Run . . . 631
C.5 Build and Run Models on AWS . . . 632
C.6 Close Your EC2 Instance . . . 633
C.7 Tips and Tricks for Using Keras on AWS . . . 634
C.8 Further Reading . . . 635
C.9 Summary . . . 635

IX Conclusions 636

How Far You Have Come 637

Preface

The study of Generative Adversarial Networks (GANs) is new, just a few years old. Yet, in just a few years GANs have achieved results so remarkable that they have become the state-of-the-art in generative modeling. The field is so new that there are no good theories on how to implement and configure the models. All advice for applying GAN models is based on hard-earned empirical findings, the same as any nascent field of study. This makes it both exciting and frustrating.

It's exciting because although the results achieved so far are significant. Such as the automatic synthesis of large photo-realistic faces and translation of photographs from day to night. We have only scratched the surface on the capabilities of these methods. It's frustrating because the models are fussy and prone to failure modes, even after all care is taken in the choice of model architecture, model configuration hyperparameters, and data preparation.

I carefully designed this book to give you the foundation required to begin developing and applying generative adversarial networks quickly. We skip over many false starts and interesting diversions in the field and focus on the precise tips, tricks, and hacks you need to know in order to develop and train stable GAN models. We also visit state-of-the-art models and discover how we can achieve the same impressive results ourselves. There are three key areas that you must understand when working with GANs; they are:

- **How to develop basic GAN models**. This includes training unsupervised GANs for image synthesis and the heuristics required to configure and successfully train the generative models.
- **How to evaluate GAN models**. This includes widely adopted qualitative and quantitative techniques for evaluating the quality of synthesized images.
- **How to develop alternate GAN models**. This includes the use of alternate loss functions and the use of class-conditional information in the models.

These three key areas are a required foundation for understanding and developing the more advanced GAN techniques. We then build upon these key areas and explore state-of-the-art models for image translation: specifically the Pix2Pix and CycleGAN models. These key topics provide the backbone for the book and the tutorials you will work through. I believe that after completing this book, you will have the skills and knowledge required to bring generative adversarial networks to your own projects.

Jason Brownlee
2019

Introduction

Welcome

Welcome to *Generative Adversarial Networks with Python*. Generative Adversarial Networks, or GANs for short, are a deep learning technique for training generative models. GANs are most commonly used for the generation of synthetic images for a specific domain that are different and practically indistinguishable from other real images. The study and application of GANs are only a few years old, yet the results achieved have been nothing short of remarkable. Because the field is so young, it can be challenging to know how to get started, what to focus on, and how to best use the available techniques. This book is designed to teach you step-by-step how to bring the best and most promising aspects of the exciting field of Generative Adversarial Networks to your own projects.

Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that know some deep learning. Maybe you want or need to start using Generative Adversarial Networks on your research project or on a project at work. This guide was written to help you do that quickly and efficiently by compressing years of knowledge and experience into a laser-focused course of hands-on tutorials. The lessons in this book assume a few things about you, such as:

- You know your way around basic Python for programming.
- You know your way around basic NumPy for array manipulation.
- You know your way around basic Keras for deep learning.

For some bonus points, perhaps some of the below criteria apply to you. Don't panic if they don't.

- You may know how to work through a predictive modeling problem end-to-end.
- You may know a little bit of computer vision, such as convolutional neural networks.

This guide was written in the top-down and results-first machine learning style that you're used to from MachineLearningMastery.com.

About Your Outcomes

This book will teach you how to get results as a machine learning practitioner interested in using Generative Adversarial Networks on your computer vision project. After reading and working through this book, you will know:

- How to use upsampling and inverse convolutional layers in deep convolutional neural network models.
- How to implement the training procedure for fitting GAN models with the Keras deep learning library.
- How to implement best practice heuristics for the successful configuration and training of GAN models.
- How to develop and train simple GAN models for image synthesis for black and white and color images.
- How to explore the latent space for image generation with point interpolation and vector arithmetic.
- How to evaluate GAN models using qualitative and quantitative measures, such as the inception score.
- How to train GAN models with alternate loss functions, such as least squares and Wasserstein loss.
- How to structure the latent space and influence the generation of synthetic images with conditional GANs.
- How to develop image translation models with Pix2Pix for paired images and CycleGAN for unpaired images.
- How sophisticated GAN models, such as Progressive Growing GAN, are used to achieve remarkable results.

This book will NOT teach you how to be a research scientist, nor all the theory behind why specific methods work. For that, I would recommend good research papers and textbooks. See the *Further Reading* section at the end of each tutorial for a solid starting point.

How to Read This Book

This book was written to be read linearly, from start to finish. That being said, if you know the basics and need help with a specific method or type of problem, then you can flip straight to that section and get started. This book was designed for you to read on your workstation, on the screen, not on a tablet or eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.

This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then

applying your new understanding with working Python examples. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them. Try some of the extensions presented at the end of each lesson and let me know how you do.

About the Book Structure

This book was designed around major techniques that are directly relevant to Generative Adversarial Networks. There are a lot of things you could learn about GANs, from theory to abstract concepts to APIs. My goal is to take you straight to developing an intuition for the elements you must understand with laser-focused tutorials. The tutorials were designed to focus on how to get results. As such, the tutorials give you the tools to both rapidly understand and apply each technique or operation. There is a mixture of both tutorial lessons and projects to both introduce the methods and give plenty of examples and opportunities to practice using them.

Each of the tutorials is designed to take you about one hour to read through and complete, excluding the execution time of some of the larger models, as well as extensions and further reading sections. You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and this book is intended to be read and used, not to sit idle. I would recommend picking a schedule and sticking to it. The tutorials are divided into seven parts; they are:

- **Part 1: Foundations**. Discover the convolutional layers for upsampling, the GAN architecture, the algorithms for training the model, and the best practices for configuring and training GANs.
- **Part 2: GAN Basics**. Discover how to develop GANs starting with a 1D GAN, progressing through black and white and color images and ending with performing vector arithmetic in latent space.
- **Part 3: GAN Evaluation**. Discover qualitative and quantitative methods for evaluating GAN models based on their generated images.
- **Part 4: GAN Loss**. Discover alternate loss functions for GAN models and how to implement some of the more widely used approaches.
- **Part 5: Conditional GANs**. Discover how to incorporate class conditional information into GANs and add controls over the types of images generated by the model.
- **Part 6: Image Translation**. Discover advanced GAN models used for image translation both with and without paired examples in the training data.
- **Part 7. Advanced GANs**. Discover the advanced GAN models that push the limits of the architecture and are the basis for some of the impressive state-of-the-art results.

Each part targets specific learning outcomes, and so does each tutorial within each part. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and do not get bogged down in the math or near-infinite number of digressions.

The tutorials were not designed to teach you everything there is to know about each of the methods. They were designed to give you an understanding of how they work, how to use them, and how to interpret the results the fastest way I know how: to learn by doing.

About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. For this reason, the examples are highly targeted.

- Models were demonstrated on real-world datasets to give you the context and confidence to bring the techniques to your own projects.
- Model configurations used were based on best practices but were not optimized. This leaves the door open for you to explore new and possibly better configurations.
- Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third parties needed beyond the installation of the required packages.

A complete working example is presented with each tutorial for you to inspect and copy-paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues. The provided code was developed in a text editor and intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead.

Neural network algorithms are stochastic. This means that they will make different predictions when the same model configuration is trained on the same training data. On top of that, each experimental problem in this book is based around generating stochastic predictions. As a result, this means you will not get exactly the same sample output presented in this book. This is by design. I want you to get used to the stochastic nature of the neural network algorithms. If this bothers you, please note:

- You can re-run a given example a few times and your results should be close to the values reported.
- You can make the output consistent by fixing the NumPy and TensorFlow random number seed.
- You can develop a robust estimate of the skill of a model by fitting and evaluating it multiple times and taking the average of the final skill score (highly recommended).

All code examples were tested on a POSIX-compatible machine with Python 3 and Keras 2 with the TensorFlow backend. All code examples will run on modest and modern computer hardware and were executed on a CPU or GPU. A GPUs is not required but is highly recommended for most of the presented examples. Advice on how to access cheap GPUs via cloud computing is provided in the appendix. I am only human and there may be a bug in the sample code. If you discover a bug, please let me know so I can fix it, correct the book, and send out a free update.

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Research papers.
- Books and book chapters.
- Web Pages and Articles.
- API documentation.
- Open Source Projects.

Wherever possible, I try to list and link to the relevant API documentation for key objects and functions used in each lesson so you can learn more about them. When it comes to research papers, I try to list papers that are first to use a specific technique or first in a specific problem domain. These are not required reading but can give you more technical details, theory, and configuration details if you're looking for it. Wherever possible, I have tried to link to the freely available version of the paper on `arxiv.org`. You can search for and download any of the papers listed on Google Scholar Search `scholar.google.com`. Wherever possible, I have tried to link to books on Amazon. I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

About Getting Help

You might need help along the way. Don't worry; you are not alone.

- **Help with a Technique?** If you need help with the technical aspects of a specific operation or technique, see the *Further Reading* section at the end of each tutorial.
- **Help with APIs?** If you need help with using the Keras library, see the list of resources in the *Further Reading* section at the end of each lesson, and also see *Appendix A*.
- **Help with your workstation?** If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in *Appendix B*.
- **Help running large models?** I recommend renting time on Amazon Web Service (AWS) EC2 instances to run large models. If you need help getting started on AWS, see the tutorial in *Appendix C*.
- **Help in general?** You can shoot me an email. My details are in *Appendix A*.

Summary

Are you ready? Let's dive in! Next up, you will discover a concrete overview of Generative Adversarial Networks.

Part I

Foundations

Overview

In this part you will discover the foundations of Generative Adversarial Networks, or GANs. After reading the chapters in this part, you will know:

- About generative models, Generative Adversarial Networks and why GANs are worth studying (Chapter 1).
- How to implement and use deep learning models using the Keras deep learning library (Chapter 2).
- How to use convolutional layers for upsampling required in the generator model (Chapter 3).
- How to implement the training algorithm used to update the discriminator and generator models (Chapter 4).
- How to implement pratical heuristic tips, tricks and hacks required for the stable training of GAN models (Chapter 5).

Chapter 1

What are Generative Adversarial Networks

Generative Adversarial Networks, or GANs for short, are an approach to generative modeling using deep learning methods, such as convolutional neural networks. Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset. GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together in an adversarial zero-sum game until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

GANs are an exciting and rapidly changing field, delivering on the promise of generative models in their ability to generate realistic examples across a range of problem domains, most notably in image-to-image translation tasks such as translating photos of summer to winter or day to night, and in generating photorealistic photos of objects, scenes, and people that even humans cannot tell are fake. In this tutorial, you will discover a gentle introduction to Generative Adversarial Networks, or GANs. After reading this tutorial, you will know:

- Context for GANs, including supervised vs. unsupervised learning and discriminative vs. generative modeling.
- GANs are an architecture for automatically training a generative model by treating the unsupervised problem as supervised and using both a generative and a discriminative model.
- GANs provide a path to sophisticated domain-specific data augmentation and a solution to problems that require a generative solution, such as image-to-image translation.

Let's get started.

1.1 Overview

This tutorial is divided into three parts; they are:

1. What Are Generative Models?
2. What Are Generative Adversarial Networks?
3. Why Generative Adversarial Networks?

1.2 What Are Generative Models?

In this section, we will review the idea of generative models, considering supervised vs. unsupervised learning paradigms and discriminative vs. generative modeling.

1.2.1 Supervised vs. Unsupervised Learning

A typical machine learning problem involves using a model to make a prediction, e.g. predictive modeling. This requires a training dataset that is used to train a model, comprised of multiple examples, called samples, each with input variables (X) and output class labels (y). A model is trained by showing examples of inputs, having it predict outputs, and correcting the model to make the outputs more like the expected outputs.

> In the predictive or supervised learning approach, the goal is to learn a mapping from inputs x to outputs y, given a labeled set of input-output pairs ...
>
> — Page 2, *Machine Learning: A Probabilistic Perspective*, 2012.

This correction of the model is generally referred to as a supervised form of learning, or supervised learning. It is *supervised* because there is a real expected outcome to which a prediction is compared.

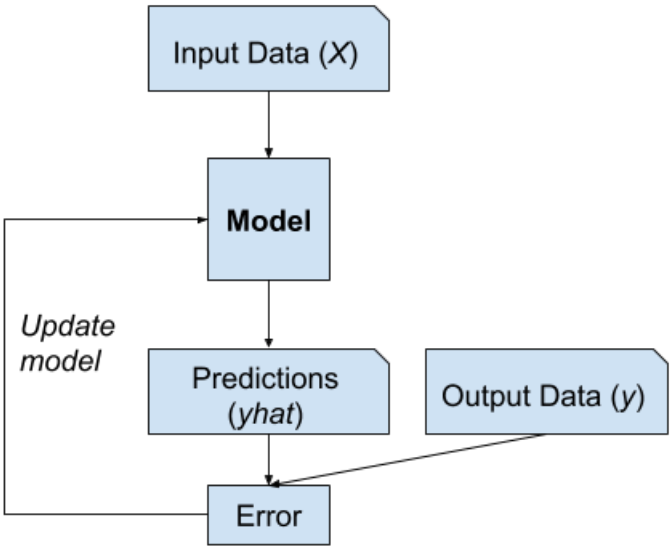

Figure 1.1: Example of Supervised Learning.

Examples of supervised learning problems include classification and regression, and examples of supervised learning algorithms include logistic regression and random forests. There is another paradigm of learning where the model is only given the input variables (X) and the problem does not have any output variables (y). A model is constructed by extracting or summarizing the patterns in the input data. There is no correction of the model, as the model is not predicting anything.

> The second main type of machine learning is the descriptive or unsupervised learning approach. Here we are only given inputs, and the goal is to find "interesting patterns" in the data. [...] This is a much less well-defined problem, since we are not told what kinds of patterns to look for, and there is no obvious error metric to use (unlike supervised learning, where we can compare our prediction of y for a given x to the observed value).
>
> — Page 2, *Machine Learning: A Probabilistic Perspective*, 2012.

This lack of correction is generally referred to as an unsupervised form of learning, or unsupervised learning.

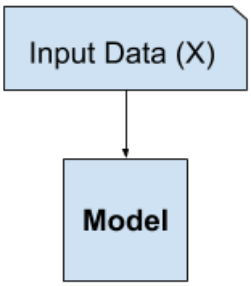

Figure 1.2: Example of Unsupervised Learning.

Examples of unsupervised learning problems include clustering and generative modeling, and examples of unsupervised learning algorithms are K-means and Generative Adversarial Networks.

1.2.2 Discriminative vs. Generative Modeling

In supervised learning, we may be interested in developing a model to predict a class label given an example of input variables. This predictive modeling task is called classification. Classification is also traditionally referred to as discriminative modeling.

> ... we use the training data to find a discriminant function $f(x)$ that maps each x directly onto a class label, thereby combining the inference and decision stages into a single learning problem.
>
> — Page 44, *Pattern Recognition and Machine Learning*, 2006.

This is because a model must discriminate examples of input variables across classes; it must choose or make a decision as to what class a given example belongs.

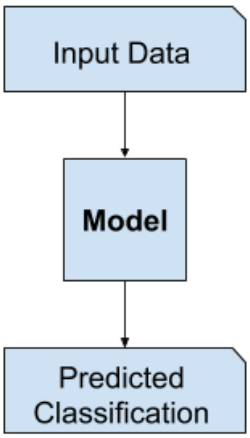

Figure 1.3: Example of Discriminative Modeling.

Alternately, unsupervised models that summarize the distribution of input variables may be able to be used to create or generate new examples in the input distribution. As such, these types of models are referred to as generative models.

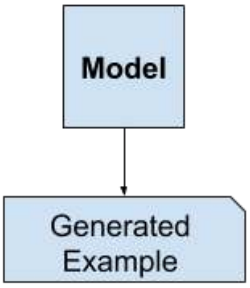

Figure 1.4: Example of Generative Modeling.

For example, a single variable may have a known data distribution, such as a Gaussian distribution, or bell shape. A generative model may be able to sufficiently summarize this data distribution, and then be used to generate new examples that plausibly fit into the distribution of the input variable.

> Approaches that explicitly or implicitly model the distribution of inputs as well as outputs are known as generative models, because by sampling from them it is possible to generate synthetic data points in the input space.
>
> — Page 43, *Pattern Recognition and Machine Learning*, 2006.

In fact, a really good generative model may be able to generate new examples that are not just plausible, but indistinguishable from real examples from the problem domain.

1.2.3 Examples of Generative Models

Naive Bayes is an example of a generative model that is more often used as a discriminative model. For example, Naive Bayes works by summarizing the probability distribution of each input variable and the output class. When a prediction is made, the probability for each possible outcome is calculated for each variable, the independent probabilities are combined, and the most likely outcome is predicted. Used in reverse, the probability distributions for each variable can be sampled to generate new plausible (independent) feature values.

Other examples of generative models include Latent Dirichlet Allocation, or LDA, and the Gaussian Mixture Model, or GMM. Deep learning methods can be used as generative models. Two popular examples include the Restricted Boltzmann Machine, or RBM, and the Deep Belief Network, or DBN. Two modern examples of deep learning generative modeling algorithms include the Variational Autoencoder, or VAE, and the Generative Adversarial Network, or GAN.

1.3 What Are Generative Adversarial Networks?

Generative Adversarial Networks, or GANs, are a deep-learning-based generative model. More generally, GANs are a model architecture for training a generative model, and it is most common to use deep learning models in this architecture. The GAN architecture was first described in the 2014 paper by Ian Goodfellow, et al. titled *Generative Adversarial Networks*. The initial models worked but were unstable and difficult to train. A standardized approach called Deep Convolutional Generative Adversarial Networks, or DCGAN, that led to more stable models was later formalized by Alec Radford, et al. in the 2015 paper titled *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*.

> Most GANs today are at least loosely based on the DCGAN architecture ...
>
> — *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

The GAN model architecture involves two sub-models: a generator model for generating new examples and a discriminator model for classifying whether generated examples are real (from the domain) or fake (generated by the generator model).

- **Generator**. Model that is used to generate new plausible examples from the problem domain.
- **Discriminator**. Model that is used to classify examples as real (from the domain) or fake (generated).

> Generative adversarial networks are based on a game theoretic scenario in which the generator network must compete against an adversary. The generator network directly produces samples. Its adversary, the discriminator network, attempts to distinguish between samples drawn from the training data and samples drawn from the generator.
>
> — Page 699, *Deep Learning*, 2016.

1.3.1 The Generator Model

The generator model takes a fixed-length random vector as input and generates a sample in the domain, such as an image. A vector is drawn randomly from a Gaussian distribution and is used to seed or source of noise for the generative process. To be clear, the input is a vector of random numbers. It is not an image or a flattened image and has no meaning other than the meaning applied by the generator model. After training, points in this multidimensional vector space will correspond to points in the problem domain, forming a compressed representation of the data distribution. This vector space is referred to as a latent space, or a vector space comprised of latent variables. Latent variables, or hidden variables, are those variables that are important for a domain but are not directly observable.

> A latent variable is a random variable that we cannot observe directly.
>
> — Page 67, *Deep Learning*, 2016.

We often refer to latent variables, or a latent space, as a projection or compression of a data distribution. That is, a latent space provides a compression or high-level concepts of the observed raw data such as the input data distribution. In the case of GANs, the generator model applies meaning to points in a chosen latent space, such that new points drawn from the latent space can be provided to the generator model as input and used to generate new and different output examples.

> Machine-learning models can learn the statistical latent space of images, music, and stories, and they can then sample from this space, creating new artworks with characteristics similar to those the model has seen in its training data.
>
> — Page 270, *Deep Learning with Python*, 2017.

After training, the generator model is kept and used to generate new samples.

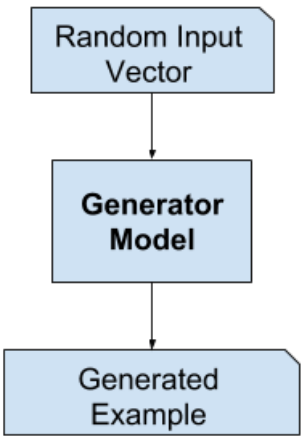

Figure 1.5: Example of the GAN Generator Model.

1.3.2 The Discriminator Model

The discriminator model takes an example from the problem domain as input (real or generated) and predicts a binary class label of real or fake (generated). The real example comes from the training dataset. The generated examples are output by the generator model. The discriminator is a normal classification model.

After the training process, the discriminator model is discarded as we are interested in the generator. Sometimes, the generator can be repurposed as it has learned to effectively extract features from examples in the problem domain. Some or all of the feature extraction layers can be used in transfer learning applications using the same or similar input data.

> We propose that one way to build good image representations is by training Generative Adversarial Networks (GANs), and later reusing parts of the generator and discriminator networks as feature extractors for supervised tasks

— *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, 2015.

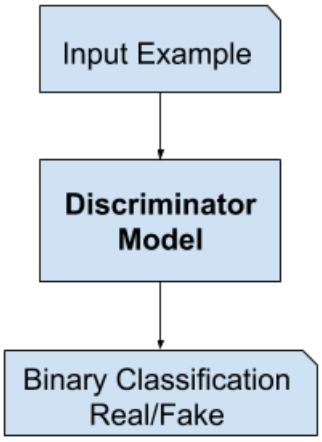

Figure 1.6: Example of the GAN Discriminator Model.

1.3.3 GANs as a Two Player Game

Generative modeling is an unsupervised learning problem, as we discussed in the previous section, although a clever property of the GAN architecture is that the training of the generative model is framed as a supervised learning problem. The two models, the generator and discriminator, are trained together. The generator generates a batch of samples, and these, along with real examples from the domain, are provided to the discriminator and classified as real or fake.

The discriminator is then updated to get better at discriminating real and fake samples in the next round, and importantly, the generator is updated based on how well, or not, the generated samples fooled the discriminator.

> We can think of the generator as being like a counterfeiter, trying to make fake money, and the discriminator as being like police, trying to allow legitimate money

> and catch counterfeit money. To succeed in this game, the counterfeiter must learn to make money that is indistinguishable from genuine money, and the generator network must learn to create samples that are drawn from the same distribution as the training data.
>
> — *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

In this way, the two models are competing against each other. They are adversarial in the game theory sense and are playing a zero-sum game.

> Because the GAN framework can naturally be analyzed with the tools of game theory, we call GANs "adversarial".
>
> — *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

In this case, zero-sum means that when the discriminator successfully identifies real and fake samples, it is rewarded and no change is needed to the model parameters, whereas the generator is penalized with large updates to model parameters. Alternately, when the generator fools the discriminator, it is rewarded and no change is needed to the model parameters, but the discriminator is penalized and its model parameters are updated.

At a limit, the generator generates perfect replicas from the input domain every time, and the discriminator cannot tell the difference and predicts *unsure* (e.g. 50% for real and fake) in every case. This is just an example of an idealized case; we do not need to get to this point to arrive at a useful generator model.

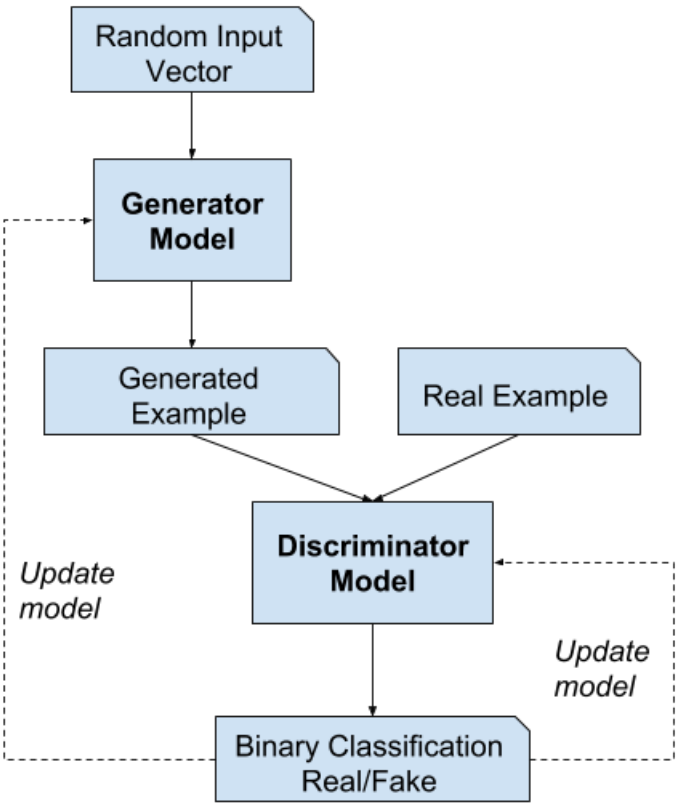

Figure 1.7: Example of the Generative Adversarial Network Model Architecture.

> [training] drives the discriminator to attempt to learn to correctly classify samples as real or fake. Simultaneously, the generator attempts to fool the classifier into believing its samples are real. At convergence, the generator's samples are indistinguishable from real data, and the discriminator outputs $\frac{1}{2}$ everywhere. The discriminator may then be discarded.
>
> — Page 700, *Deep Learning*, 2016.

1.3.4 GANs and Convolutional Neural Networks

GANs typically work with image data and use Convolutional Neural Networks, or CNNs, as the generator and discriminator models. The reason for this may be both because the first description of the technique was in the field of computer vision and it used image data, and because of the remarkable progress that has been seen in recent years using CNNs more generally to achieve state-of-the-art results on a suite of computer vision tasks such as object detection and face recognition.

Modeling image data means that the latent space, the input to the generator, provides a compressed representation of the set of images or photographs used to train the model. It also means that the generator generates new images, providing an output that can be easily viewed and assessed by developers or users of the model. It may be this fact above others, the ability to visually assess the quality of the generated output, that has both led to the focus of computer vision applications with CNNs and on the massive leaps in the capability of GANs as compared to other generative models, deep-learning-based or otherwise.

1.3.5 Conditional GANs

An important extension to the GAN is in their use for conditionally generating an output. The generative model can be trained to generate new examples from the input domain, where the input, the random vector from the latent space, is provided with (conditioned by) some additional input. The additional input could be a class value, such as male or female in the generation of photographs of people, or a digit, in the case of generating images of handwritten digits.

> Generative adversarial nets can be extended to a conditional model if both the generator and discriminator are conditioned on some extra information y. y could be any kind of auxiliary information, such as class labels or data from other modalities. We can perform the conditioning by feeding y into the both the discriminator and generator as [an] additional input layer.
>
> — *Conditional Generative Adversarial Nets*, 2014.

The discriminator is also conditioned, meaning that it is provided both with an input image that is either real or fake and the additional input. In the case of a classification label type conditional input, the discriminator would then expect that the input would be of that class, in turn teaching the generator to generate examples of that class in order to fool the discriminator. In this way, a conditional GAN can be used to generate examples from a domain of a given type (this is covered further in Chapter 17).

Taken one step further, the GAN models can be conditioned on an example from the domain, such as an image. This allows for applications of GANs such as text-to-image translation, or image-to-image translation. This allows for some of the more impressive applications of GANs, such as style transfer, photo colorization, transforming photos from summer to winter or day to night, and so on. In the case of conditional GANs for image-to-image translation, such as transforming day to night, the discriminator is provided examples of real and generated nighttime photos as well as (conditioned on) real daytime photos as input. The generator is provided with a random vector from the latent space as well as (conditioned on) real daytime photos as input (this is covered further in many chapters later in the book).

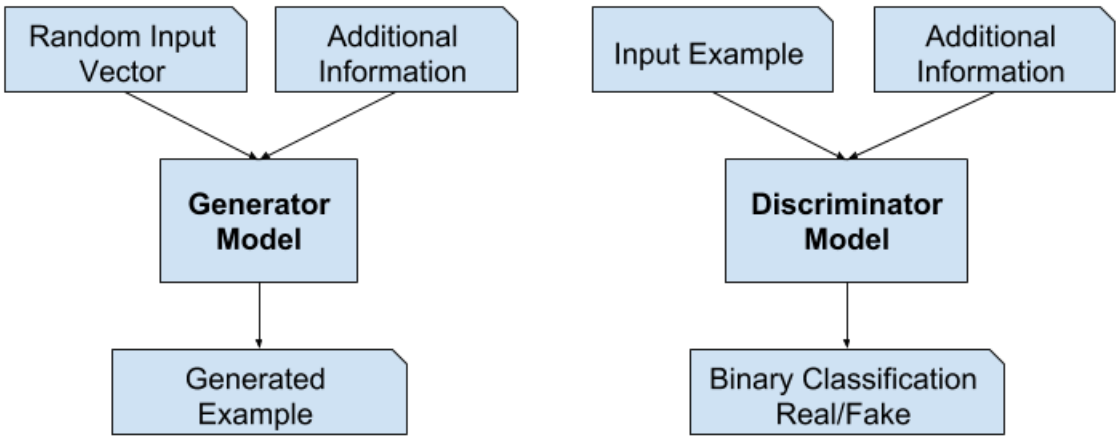

Figure 1.8: Example of a Conditional Generative Adversarial Network Model Architecture.

1.4 Why Generative Adversarial Networks?

One of the many major advancements in the use of deep learning methods in domains such as computer vision is a technique called data augmentation. Data augmentation results in better performing models, both increasing model skill and providing a regularizing effect, reducing generalization error. It works by creating new, artificial but plausible examples from the input problem domain on which the model is trained. The techniques are primitive in the case of image data, involving crops, flips, zooms, and other simple transforms of existing images in the training dataset.

Successful generative modeling provides an alternative and potentially more domain-specific approach for data augmentation. In fact, data augmentation is a simplified version of generative modeling, although it is rarely described this way.

> ... enlarging the sample with latent (unobserved) data. This is called data augmentation. [...] In other problems, the latent data are actual data that should have been observed but are missing.
>
> — Page 276, *The Elements of Statistical Learning*, 2016.

In complex domains or domains with a limited amount of data, generative modeling provides a path towards more training for modeling. GANs have seen much success in this use case in

domains such as deep reinforcement learning. There are many research reasons why GANs are interesting, important, and require further study. Ian Goodfellow outlines a number of these in his 2016 conference keynote and associated technical report titled *NIPS 2016 Tutorial: Generative Adversarial Networks*. Among these reasons, he highlights GANs' successful ability to model high-dimensional data, handle missing data, and the capacity of GANs to provide multi-modal outputs or multiple plausible answers. Perhaps the most compelling application of GANs is in conditional GANs for tasks that require the generation of new examples. Here, Goodfellow indicates three main examples:

- **Image Super-Resolution**. The ability to generate high-resolution versions of input images.
- **Creating Art**. The ability to create new and artistic images, sketches, painting, and more.
- **Image-to-Image Translation**. The ability to translate photographs across domains, such as day to night, summer to winter, and more.

Perhaps the most compelling reason that GANs are widely studied, developed, and used is because of their success. GANs have been able to generate photos so realistic that humans are unable to tell that they are of objects, scenes, and people that do not exist in real life. *Astonishing* is not a sufficient adjective for their capability and success.

Figure 1.9: Example of the Progression in the Capabilities of GANs From 2014 to 2017. Taken from The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation, 2018.

1.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

1.5.1 Books

- Chapter 20. Deep Generative Models, Deep Learning, 2016.
  https://amzn.to/2YuwVjL
- Chapter 8. Generative Deep Learning, Deep Learning with Python, 2017.
  https://amzn.to/2U2bHuP
- Machine Learning: A Probabilistic Perspective, 2012.
  https://amzn.to/2HV6cYx

- Pattern Recognition and Machine Learning, 2006.
  https://amzn.to/2K3EcE3
- The Elements of Statistical Learning, 2016.
  https://amzn.to/2UcPeva

1.5.2 Papers

- Generative Adversarial Networks, 2014.
  https://arxiv.org/abs/1406.2661
- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.
  https://arxiv.org/abs/1511.06434
- NIPS 2016 Tutorial: Generative Adversarial Networks, 2016.
  https://arxiv.org/abs/1701.00160
- Conditional Generative Adversarial Nets, 2014.
  https://arxiv.org/abs/1411.1784
- The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation, 2018.
  https://arxiv.org/abs/1802.07228

1.5.3 Articles

- Generative model, Wikipedia.
  https://en.wikipedia.org/wiki/Generative_model
- Latent Variable, Wikipedia.
  https://en.wikipedia.org/wiki/Latent_variable
- Generative Adversarial Network, Wikipedia.
  https://en.wikipedia.org/wiki/Generative_adversarial_network

1.6 Summary

In this tutorial, you discovered a gentle introduction to Generative Adversarial Networks, or GANs. Specifically, you learned:

- Context for GANs, including supervised vs. unsupervised learning and discriminative vs. generative modeling.
- GANs are an architecture for automatically training a generative model by treating the unsupervised problem as supervised and using both a generative and a discriminative model.
- GANs provide a path to sophisticated domain-specific data augmentation and a solution to problems that require a generative solution, such as image-to-image translation.

1.6.1 Next

In the next tutorial, you will discover how to develop simple models using the Keras deep learning Python library.

Chapter 2

How to Develop Deep Learning Models With Keras

Deep learning neural networks are very easy to create and evaluate in Python with Keras, but you must follow a strict model life-cycle. In this tutorial you will discover the step-by-step life-cycle for creating, training and evaluating deep learning neural networks in Keras and how to make predictions with a trained model. You will also discover how to use the functional API that provides more flexibility when designing models. After reading this tutorial you will know:

- How to define, compile, fit and evaluate a deep learning neural network in Keras.
- How to select default loss functions for regression and classification predictive modeling problems.
- How to use the functional API to develop standard Multilayer Perceptron, convolutional and recurrent neural networks.

Let's get started.

Note: It is assumed that you have a basic familiarity with deep learning and Keras. Nevertheless, this tutorial should provide a refresher for the Keras Sequential API, and perhaps an introduction to the Keras functional API. See the Appendix B for installation instructions, if needed.

2.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Keras Model Life-Cycle
2. Keras Functional Models
3. Standard Network Models

2.2 Keras Model Life-Cycle

Below is an overview of the 5 steps in the neural network model life-cycle in Keras:

1. Define Network.
2. Compile Network.
3. Fit Network.
4. Evaluate Network.
5. Make Predictions.

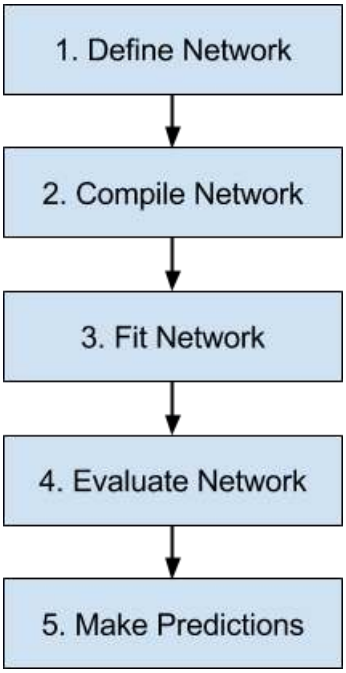

Figure 2.1: 5 Step Life-Cycle for Neural Network Models in Keras.

Let's take a look at each step in turn using the easy-to-use Keras Sequential API.

2.2.1 Step 1. Define Network

The first step is to define your neural network. Neural networks are defined in Keras as a sequence of layers. The container for these layers is the `Sequential` class. Create an instance of the `Sequential` class. Then you can add your layers in the order that they should be connected. For example, we can do this in two steps:

```
...
model = Sequential()
model.add(Dense(2))
```

Listing 2.1: Sequential model with one `Dense` layer with 2 neurons.

But we can also do this in one step by creating an array of layers and passing it to the constructor of the `Sequential` class.

```
...
layers = [Dense(2)]
model = Sequential(layers)
```

Listing 2.2: Layers for a Sequential model defined as an array.

The first layer in the network must define the number of inputs to expect. The way that this is specified can differ depending on the network type, but for a Multilayer Perceptron model this is specified by the `input_dim` attribute. For example, a small Multilayer Perceptron model with 2 inputs in the visible layer, 5 neurons in the hidden layer and one neuron in the output layer can be defined as:

```
...
model = Sequential()
model.add(Dense(5, input_dim=2))
model.add(Dense(1))
```

Listing 2.3: Sequential model with 2 inputs.

Think of a Sequential model as a pipeline with your raw data fed in at the bottom and predictions that come out at the top. This is a helpful conception in Keras as components that were traditionally associated with a layer can also be split out and added as separate layers or layer-like objects, clearly showing their role in the transform of data from input to prediction. For example, activation functions that transform a summed signal from each neuron in a layer can be added to the Sequential as a layer-like object called the `Activation` class.

```
...
model = Sequential()
model.add(Dense(5, input_dim=2))
model.add(Activation('relu'))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Listing 2.4: Sequential model with `Activation` functions defined separately from layers.

The choice of activation function is most important for the output layer as it will define the format that predictions will take. For example, below are some common predictive modeling problem types and the format and standard activation function that you can use in the output layer:

- **Regression**: Linear activation function, or `linear` (or `None`), and the number of neurons matching the number of outputs.
- **Binary Classification (2 class)**: Logistic activation function, or `sigmoid`, and one neuron the output layer.
- **Multiclass Classification (>2 class)**: Softmax activation function, or `softmax`, and one output neuron per class value, assuming a one hot encoded output pattern.

2.2.2 Step 2. Compile Network

Once we have defined our network, we must compile it. Compilation is an efficiency step. It transforms the simple sequence of layers that we defined into a highly efficient series of matrix transforms in a format intended to be executed on your GPU or CPU, depending on how Keras is configured. Think of compilation as a precompute step for your network. It is always required after defining a model.

Compilation requires a number of parameters to be specified, tailored to training your network. Specifically, the optimization algorithm to use to train the network and the loss function used to evaluate the network that is minimized by the optimization algorithm. For example, below is a case of compiling a defined model and specifying the stochastic gradient descent (`sgd`) optimization algorithm and the mean squared error (`mean_squared_error`) loss function, intended for a regression type problem.

```
...
model.compile(optimizer='sgd', loss='mean_squared_error')
```

Listing 2.5: Example of compiling a defined model.

Alternately, the optimizer can be created and configured before being provided as an argument to the compilation step.

```
...
algorithm = SGD(lr=0.1, momentum=0.3)
model.compile(optimizer=algorithm, loss='mean_squared_error')
```

Listing 2.6: Example of defining the optimization algorithm separately.

The type of predictive modeling problem imposes constraints on the type of loss function that can be used. For example, below are some standard loss functions for different predictive model types:

- **Regression**: Mean Squared Error or `mean_squared_error`.
- **Binary Classification (2 class)**: Logarithmic Loss, also called cross-entropy or `binary_crossentropy`.
- **Multiclass Classification (>2 class)**: Multiclass Logarithmic Loss or `categorical_crossentropy`.

The most common optimization algorithm is stochastic gradient descent, but Keras also supports a suite of other state-of-the-art optimization algorithms that work well with little or no configuration. Perhaps the most commonly used optimization algorithms because of their generally better performance are:

- **Stochastic Gradient Descent**, or `sgd`, that requires the tuning of a learning rate and momentum.
- **Adam**, or `adam`, that requires the tuning of learning rate.
- **RMSprop**, or `rmsprop`, that requires the tuning of learning rate.

Finally, you can also specify metrics to collect while fitting your model in addition to the loss function. Generally, the most useful additional metric to collect is accuracy for classification problems. The metrics to collect are specified by name in an array. For example:

```
...
model.compile(optimizer='sgd', loss='mean_squared_error', metrics=['accuracy'])
```

Listing 2.7: Example of defining metrics when compiling the model.

2.2.3 Step 3. Fit Network

Once the network is compiled, it can be fit, which means adapting the model weights in response to a training dataset. Fitting the network requires the training data to be specified, both a matrix of input patterns, `X`, and an array of matching output patterns, `y`. The network is trained using the backpropagation algorithm and optimized according to the optimization algorithm and loss function specified when compiling the model.

The backpropagation algorithm requires that the network be trained for a specified number of epochs or exposures to the training dataset. Each epoch can be partitioned into groups of input-output pattern pairs called batches. This defines the number of patterns that the network is exposed to before the weights are updated within an epoch. It is also an efficiency optimization, ensuring that not too many input patterns are loaded into memory at a time. A minimal example of fitting a network is as follows:

```
...
history = model.fit(X, y, batch_size=10, epochs=100)
```

Listing 2.8: Example of fitting a compiled model.

Once fit, a history object is returned that provides a summary of the performance of the model during training. This includes both the loss and any additional metrics specified when compiling the model, recorded each epoch. Training can take a long time, from seconds to hours to days depending on the size of the network and the size of the training data.

By default, a progress bar is displayed on the command line for each epoch. This may create too much noise for you, or may cause problems for your environment, such as if you are in an interactive notebook or IDE. You can reduce the amount of information displayed to just the loss each epoch by setting the verbose argument to 2. You can turn off all output by setting verbose to 0. For example:

```
...
history = model.fit(X, y, batch_size=10, epochs=100, verbose=0)
```

Listing 2.9: Example of turning off verbose output when fitting the model.

At the end of the training process, you may want to save your model for later use. This can be achieved by calling the `save()` function on the model itself and specifying a filename.

```
...
model.save('final_model.h5')
```

Listing 2.10: Example of saving a fit model to file.

Saving a model requires that the `h5py` library is installed on your workstation. This can be achieved easily using `pip`, for example:

```
sudo pip install h5py
```

Listing 2.11: Example installing the `h5py` library with `pip`.

The model can be loaded later by calling the `load_model()` function and specifying the filename.

```
from keras.models import load_model
model = load_model('final_model.h5')
```

Listing 2.12: Example of loading a saved model from file.

2.2.4 Step 4. Evaluate Network

Once the network is trained, it can be evaluated. The network can be evaluated on the training data, but this will not provide a useful indication of the performance of the network as a predictive model, as it has seen all of this data before. We can evaluate the performance of the network on a separate dataset, unseen during training. This will provide an estimate of the performance of the network at making predictions for unseen data in the future.

The model evaluates the loss across all of the test patterns, as well as any other metrics specified when the model was compiled, like classification accuracy. A list of evaluation metrics is returned. For example, for a model compiled with the accuracy metric, we could evaluate it on a new dataset as follows:

```
...
loss, accuracy = model.evaluate(X, y)
```

Listing 2.13: Example of evaluating a fit model.

As with fitting the network, verbose output is provided to give an idea of the progress of evaluating the model. We can turn this off by setting the verbose argument to 0.

```
...
loss, accuracy = model.evaluate(X, y, verbose=0)
```

Listing 2.14: Example of turning off verbose output when evaluating a fit model.

2.2.5 Step 5. Make Predictions

Once we are satisfied with the performance of our fit model, we can use it to make predictions on new data. This is as easy as calling the `predict()` function on the model with an array of new input patterns. For example:

```
...
predictions = model.predict(X)
```

Listing 2.15: Example of making a prediction with a fit model.

The predictions will be returned in the format provided by the output layer of the network. In the case of a regression problem, these predictions may be in the format of the problem directly, provided by a linear activation function. For a binary classification problem, the predictions may be an array of probabilities for the first class that can be converted to a 1 or 0 by rounding.

For a multiclass classification problem, the results may be in the form of an array of probabilities (assuming a one hot encoded output variable) that may need to be converted to a single class output prediction using the `argmax()` NumPy function. Alternately, for classification problems, we can use the `predict_classes()` function that will automatically convert the predicted probabilities into class integer values.

```
...
predictions = model.predict_classes(X)
```

Listing 2.16: Example of predicting classes with a fit model.

As with fitting and evaluating the network, verbose output is provided to give an idea of the progress of the model making predictions. We can turn this off by setting the verbose argument to 0.

```
...
predictions = model.predict(X, verbose=0)
```

Listing 2.17: Example of disabling verbose output when making predictions.

2.3 Keras Functional Models

The sequential API allows you to create models layer-by-layer for most problems. It is limited in that it does not allow you to create models that share layers or have multiple input or output layers. The functional API in Keras is an alternate way of creating models that offers a lot more flexibility, including creating more complex models.

It specifically allows you to define multiple input or output models as well as models that share layers. More than that, it allows you to define ad hoc acyclic network graphs. Models are defined by creating instances of layers and connecting them directly to each other in pairs, then defining a Model that specifies the layers to act as the input and output to the model. Let's look at the three unique aspects of Keras functional API in turn:

2.3.1 Defining Input

Unlike the Sequential model, you must create and define a standalone `Input` layer that specifies the shape of input data. The input layer takes a shape argument that is a tuple that indicates the dimensionality of the input data. When input data is one-dimensional (rows of samples), such as for a Multilayer Perceptron, the shape must explicitly leave room for the shape of the minibatch size used when splitting the data when training the network. Therefore, the shape tuple is always defined with a hanging last dimension `(2,)`, this is the way you must define a one-dimensional tuple in Python, for example:

```
...
visible = Input(shape=(2,))
```

Listing 2.18: Example of defining input for a functional model.

2.3.2 Connecting Layers

The layers in the model are connected pairwise. This is done by specifying where the input comes from when defining each new layer. A bracket or functional notation is used, such that after the layer is created, the layer from which the input to the current layer comes from is specified. Let's make this clear with a short example. We can create the input layer as above, then create a hidden layer as a `Dense` layer that receives input only from the input layer.

```
...
visible = Input(shape=(2,))
hidden = Dense(2)(visible)
```

Listing 2.19: Example of connecting a hidden layer to the visible layer.

It is the `(visible)` layer after the creation of the `Dense` layer that connects the input layer's output as the input to the `Dense` hidden layer. It is this way of connecting layers pairwise that gives the functional API its flexibility. For example, you can see how easy it would be to start defining ad hoc graphs of layers.

2.3.3 Creating the Model

After creating all of your model layers and connecting them together, you must define the model. As with the Sequential API, the model is the thing you can summarize, fit, evaluate, and use to make predictions. Keras provides a Model class that you can use to create a model from your created layers. It requires that you only specify the input and output layers. For example:

```
...
visible = Input(shape=(2,))
hidden = Dense(2)(visible)
model = Model(inputs=visible, outputs=hidden)
```

Listing 2.20: Example of creating a full model with the functional API.

Now that we know all of the key pieces of the Keras functional API, let's work through defining a suite of different models and build up some practice with it. Each example is executable and prints the structure and creates a diagram of the graph. I recommend doing this for your own models to make it clear what exactly you have defined. My hope is that these examples provide templates for you when you want to define your own models using the functional API in the future.

2.4 Standard Network Models

When getting started with the functional API, it is a good idea to see how some standard neural network models are defined. In this section, we will look at defining a simple Multilayer Perceptron, convolutional neural network, and recurrent neural network. These examples will provide a foundation for understanding the more elaborate examples later.

2.4.1 Multilayer Perceptron

In this section, we define a Multilayer Perceptron model for binary classification. The model has 10 inputs, 3 hidden layers with 10, 20, and 10 neurons, and an output layer with 1 output.

Rectified linear activation functions are used in each hidden layer and a sigmoid activation function is used in the output layer, for binary classification.

```
# example of a multilayer perceptron
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
visible = Input(shape=(10,))
hidden1 = Dense(10, activation='relu')(visible)
hidden2 = Dense(20, activation='relu')(hidden1)
hidden3 = Dense(10, activation='relu')(hidden2)
output = Dense(1, activation='sigmoid')(hidden3)
model = Model(inputs=visible, outputs=output)
# summarize layers
model.summary()
# plot graph
plot_model(model, to_file='multilayer_perceptron_graph.png')
```

Listing 2.21: Example of defining an MLP with the functional API.

Running the example prints the structure of the network.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 10)                0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                110
_________________________________________________________________
dense_2 (Dense)              (None, 20)                220
_________________________________________________________________
dense_3 (Dense)              (None, 10)                210
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 11
=================================================================
Total params: 551
Trainable params: 551
Non-trainable params: 0
_________________________________________________________________
```

Listing 2.22: Summary of MLP model defined with the functional API.

A plot of the model graph is also created and saved to file.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

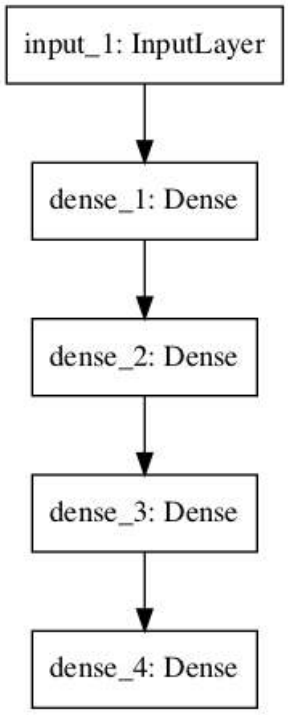

Figure 2.2: Plot of the MLP Model Graph.

2.4.2 Convolutional Neural Network

In this section, we will define a convolutional neural network for image classification. The model receives black and white 64×64 images as input, then has a sequence of two convolutional and pooling layers as feature extractors, followed by a fully connected layer to interpret the features and an output layer with a sigmoid activation for two-class predictions.

```
# example of a convolutional neural network
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D
visible = Input(shape=(64,64,1))
conv1 = Conv2D(32, (4,4), activation='relu')(visible)
pool1 = MaxPooling2D()(conv1)
conv2 = Conv2D(16, (4,4), activation='relu')(pool1)
pool2 = MaxPooling2D()(conv2)
flat1 = Flatten()(pool2)
hidden1 = Dense(10, activation='relu')(flat1)
output = Dense(1, activation='sigmoid')(hidden1)
model = Model(inputs=visible, outputs=output)
# summarize layers
model.summary()
# plot graph
plot_model(model, to_file='convolutional_neural_network.png')
```

Listing 2.23: Example of defining an CNN with the functional API.

Running the example summarizes the model layers.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 64, 64, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 61, 61, 32)        544
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 30, 30, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 27, 27, 16)        8208
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 13, 13, 16)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2704)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                27050
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 11
=================================================================
Total params: 35,813
Trainable params: 35,813
Non-trainable params: 0
_________________________________________________________________
```

Listing 2.24: Summary of CNN model defined with the functional API.

A plot of the model graph is also created and saved to file.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

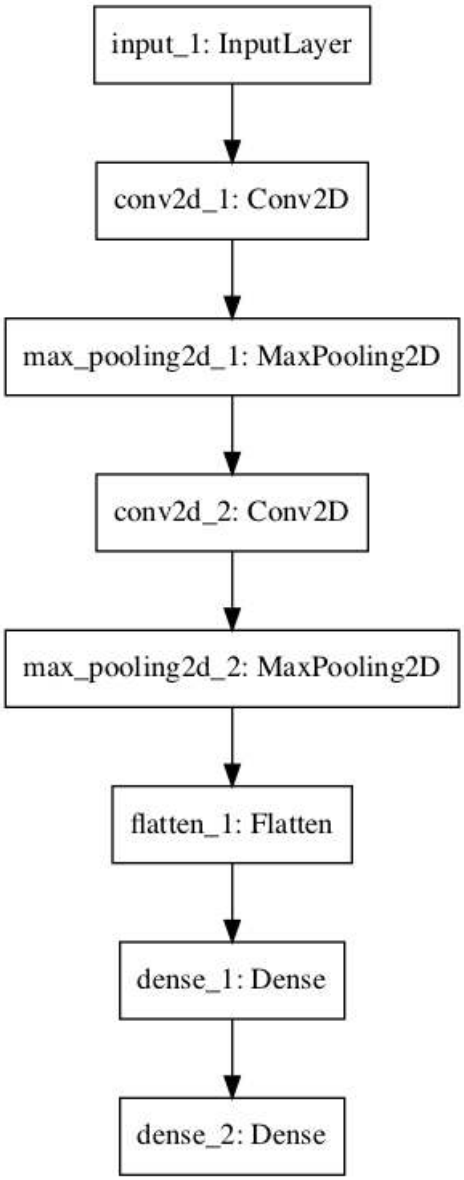

Figure 2.3: Plot of the CNN Model Graph.

2.4.3 Recurrent Neural Network

In this section, we will define a long short-term memory recurrent neural network for sequence classification. The model expects 100 time steps of one feature as input. The model has a single LSTM hidden layer to extract features from the sequence, followed by a fully connected layer to interpret the LSTM output, followed by an output layer for making binary predictions.

```
# example of a recurrent neural network
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers.recurrent import LSTM
visible = Input(shape=(100,1))
hidden1 = LSTM(10)(visible)
```

```
hidden2 = Dense(10, activation='relu')(hidden1)
output = Dense(1, activation='sigmoid')(hidden2)
model = Model(inputs=visible, outputs=output)
# summarize layers
model.summary()
# plot graph
plot_model(model, to_file='recurrent_neural_network.png')
```

Listing 2.25: Example of defining an RNN with the functional API.

Running the example summarizes the model layers.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 100, 1)            0
_________________________________________________________________
lstm_1 (LSTM)                (None, 10)                480
_________________________________________________________________
dense_1 (Dense)              (None, 10)                110
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 11
=================================================================
Total params: 601
Trainable params: 601
Non-trainable params: 0
_________________________________________________________________
```

Listing 2.26: Summary of RNN model defined with the functional API.

A plot of the model graph is also created and saved to file.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

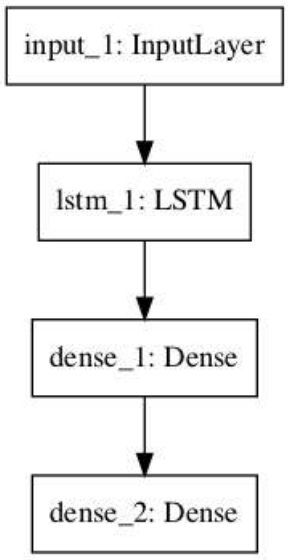

Figure 2.4: Plot of the RNN Model Graph.

2.5 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Keras documentation for Sequential Models.
  https://keras.io/models/sequential/
- Keras documentation for Functional Models.
  https://keras.io/models/model/
- Getting started with the Keras Sequential model.
  https://keras.io/models/model/
- Getting started with the Keras functional API.
  https://keras.io/models/model/
- Keras documentation for optimization algorithms.
  https://keras.io/optimizers/
- Keras documentation for loss functions.
  https://keras.io/losses/

2.6 Summary

In this tutorial, you discovered the step-by-step life-cycle for creating, training and evaluating deep learning neural networks in Keras and how to use the functional API that provides more flexibility when deigning models. Specifically, you learned:

- How to define, compile, fit and evaluate a deep learning neural network in Keras.
- How to select default loss functions for regression and classification predictive modeling problems.
- How to use the functional API to develop standard Multilayer Perceptron, convolutional and recurrent neural networks.

2.6.1 Next

In the next tutorial, you will discover the special layers to use in a convolutional neural network to output an image.

Chapter 3

How to Upsample with Convolutional Neural Networks

Generative Adversarial Networks, or GANs, are an architecture for training generative models, such as deep convolutional neural networks for generating images. The GAN architecture is comprised of both a generator and a discriminator model. The generator is responsible for creating new outputs, such as images, that plausibly could have come from the original dataset. The generator model is typically implemented using a deep convolutional neural network and results-specialized layers that learn to fill in features in an image rather than extract features from an input image.

Two common types of layers that can be used in the generator model are a upsample layer that simply doubles the dimensions of the input and the transpose convolutional layer that performs an inverse convolution operation. In this tutorial, you will discover how to use Upsampling and Transpose Convolutional Layers in Generative Adversarial Networks when generating images. After completing this tutorial, you will know:

- Generative models in the GAN architecture are required to upsample input data in order to generate an output image.
- The Upsampling layer is a simple layer with no weights that will double the dimensions of input and can be used in a generative model when followed by a traditional convolutional layer.
- The Transpose Convolutional layer is an inverse convolutional layer that will both upsample input and learn how to fill in details during the model training process.

Let's get started.

3.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Need for Upsampling in GANs
2. How to Use the Upsampling Layer
3. How to Use the Transpose Convolutional Layer

3.2 Need for Upsampling in GANs

Generative Adversarial Networks are an architecture for neural networks for training a generative model. The architecture is comprised of a generator and a discriminator model, each of which are implemented as a deep convolutional neural network. The discriminator is responsible for classifying images as either real (from the domain) or fake (generated). The generator is responsible for generating new plausible examples from the problem domain. The generator works by taking a random point from the latent space as input and outputting a complete image, in a one-shot manner.

A traditional convolutional neural network for image classification, and related tasks, will use pooling layers to downsample input images. For example, an average pooling or max pooling layer will reduce the feature maps from a convolutional by half on each dimension, resulting in an output that is one quarter the area of the input. Convolutional layers themselves also perform a form of downsampling by applying each filter across the input images or feature maps; the resulting activations are an output feature map that is smaller because of the border effects. Often padding is used to counter this effect. The generator model in a GAN requires an inverse operation of a pooling layer in a traditional convolutional layer. It needs a layer to translate from coarse salient features to a more dense and detailed output.

A simple version of an unpooling or opposite pooling layer is called an upsampling layer. It works by repeating the rows and columns of the input. A more elaborate approach is to perform a backwards convolutional operation, originally referred to as a deconvolution, which is incorrect, but is more commonly referred to as a fractional convolutional layer or a transposed convolutional layer. Both of these layers can be used on a GAN to perform the required upsampling operation to transform a small input into a large image output. In the following sections, we will take a closer look at each and develop an intuition for how they work so that we can use them effectively in our GAN models.

3.3 How to Use the Upsampling Layer

Perhaps the simplest way to upsample an input is to double each row and column. For example, an input image with the shape 2×2 would be output as 4×4.

```
        1, 2
Input = 3, 4

         1, 1, 2, 2
Output = 1, 1, 2, 2
         3, 3, 4, 4
         3, 3, 4, 4
```

Listing 3.1: Example input and output using an upsampling layer.

3.3.1 Worked Example Using the `UpSampling2D` Layer

The Keras deep learning library provides this capability in a layer called `UpSampling2D`. It can be added to a convolutional neural network and repeats the rows and columns provided as input in the output. For example:

```
...
# define model
model = Sequential()
model.add(UpSampling2D())
```

Listing 3.2: Example of adding an `UpSampling2D` to a model.

We can demonstrate the behavior of this layer with a simple contrived example. First, we can define a contrived input image that is 2×2 pixels. We can use specific values for each pixel so that after upsampling, we can see exactly what effect the operation had on the input.

```
...
# define input data
X = asarray([[1, 2],
      [3, 4]])
# show input data for context
print(X)
```

Listing 3.3: Example of defining a small input matrix.

Once the image is defined, we must add a channel dimension (e.g. grayscale) and also a sample dimension (e.g. we have 1 sample) so that we can pass it as input to the model. The data dimensions in order are: samples, rows, columns, and channels.

```
...
# reshape input data into one sample with one channel
X = X.reshape((1, 2, 2, 1))
```

Listing 3.4: Example of reshaping a matrix to be input to a CNN.

We can now define our model. The model has only the `UpSampling2D` layer which takes 2×2 grayscale images as input directly and outputs the result of the upsampling operation.

```
...
# define model
model = Sequential()
model.add(UpSampling2D(input_shape=(2, 2, 1)))
# summarize the model
model.summary()
```

Listing 3.5: Example of defining a small upsampling CNN.

We can then use the model to make a prediction, that is upsample a provided input image.

```
...
# make a prediction with the model
yhat = model.predict(X)
```

Listing 3.6: Example of making a prediction with the defined model.

The output will have four dimensions, like the input, therefore, we can convert it back to a 2×2 array to make it easier to review the result.

```
...
# reshape output to remove sample and channel to make printing easier
yhat = yhat.reshape((4, 4))
# summarize output
print(yhat)
```

Listing 3.7: Example of interpreting the model prediction.

Tying all of this together, the complete example of using the `UpSampling2D` layer in Keras is provided below.

```
# example of using the upsampling layer
from numpy import asarray
from keras.models import Sequential
from keras.layers import UpSampling2D
# define input data
X = asarray([[1, 2],
       [3, 4]])
# show input data for context
print(X)
# reshape input data into one sample a sample with a channel
X = X.reshape((1, 2, 2, 1))
# define model
model = Sequential()
model.add(UpSampling2D(input_shape=(2, 2, 1)))
# summarize the model
model.summary()
# make a prediction with the model
yhat = model.predict(X)
# reshape output to remove channel to make printing easier
yhat = yhat.reshape((4, 4))
# summarize output
print(yhat)
```

Listing 3.8: Example of demonstrating a model with an `UpSampling2D` layer.

Running the example first creates and summarizes our 2×2 input data. Next, the model is summarized. We can see that it will output a 4×4 result as we expect, and importantly, the layer has no parameters or model weights. This is because it is not learning anything; it is just doubling the input. Finally, the model is used to upsample our input, resulting in a doubling of each row and column for our input data, as we expected.

```
[[1 2]
 [3 4]]


_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
up_sampling2d_1 (UpSampling2 (None, 4, 4, 1)      0
=================================================================
Total params: 0
Trainable params: 0
Non-trainable params: 0
_________________________________________________________________


[[1. 1. 2. 2.]
 [1. 1. 2. 2.]
 [3. 3. 4. 4.]
 [3. 3. 4. 4.]]
```

Listing 3.9: Example output from demonstrating a model with an `UpSampling2D` layer.

By default, the `UpSampling2D` will double each input dimension. This is defined by the `size` argument that is set to the tuple (2,2). You may want to use different factors on each dimension, such as double the width and triple the height. This could be achieved by setting the `size` argument to `(2, 3)`. The result of applying this operation to a 2×2 image would be a 4×6 output image (e.g. 2×2 and 2×3). For example:

```
...
# example of using different scale factors for each dimension
model.add(UpSampling2D(size=(2, 3)))
```

Listing 3.10: Example of defining the shape of the upsampling operation.

Additionally, by default, the `UpSampling2D` layer will use a nearest neighbor algorithm to fill in the new rows and columns. This has the effect of simply doubling rows and columns, as described and is specified by the `interpolation` argument set to `'nearest'`. Alternately, a bilinear interpolation method can be used which draws upon multiple surrounding points. This can be specified via setting the `interpolation` argument to `'bilinear'`. For example:

```
...
# example of using bilinear interpolation when upsampling
model.add(UpSampling2D(interpolation='bilinear'))
```

Listing 3.11: Example of defining the interpolation model for the upsampling.

3.3.2 Simple Generator Model With the `UpSampling2D` Layer

The `UpSampling2D` layer is simple and effective, although does not perform any learning. It is not able to fill in useful detail in the upsampling operation. To be useful in a GAN, each `UpSampling2D` layer must be followed by a `Conv2D` layer that will learn to interpret the doubled input and be trained to translate it into meaningful detail. We can demonstrate this with an example.

In this example, our little GAN generator model must produce a 10×10 image as output and take a 100 element vector of random numbers from the latent space as input. First, a `Dense` fully connected layer can be used to interpret the input vector and create a sufficient number of activations (outputs) that can be reshaped into a low-resolution version of our output image, in this case, 128 versions of a 5×5 image.

```
...
# define model
model = Sequential()
# define input shape, output enough activations for 128 5x5 images
model.add(Dense(128 * 5 * 5, input_dim=100))
# reshape vector of activations into 128 feature maps with 5x5
model.add(Reshape((5, 5, 128)))
```

Listing 3.12: Example of defining the input activations for the generator model.

Next, the 5×5 feature maps can be upsampled to a 10×10 feature map.

```
...
# quadruple input from 128 5x5 to 1 10x10 feature map
model.add(UpSampling2D())
```

Listing 3.13: Example of adding a `UpSampling2D` layer to the model.

Finally, the upsampled feature maps can be interpreted and filled in with hopefully useful detail by a `Conv2D` layer. The `Conv2D` has a single feature map as output to create the single image we require.

```
...
# fill in detail in the upsampled feature maps
model.add(Conv2D(1, (3,3), padding='same'))
```

Listing 3.14: Example of adding a `Conv2D` layer to the model.

Tying this together, the complete example is listed below.

```
# example of using upsampling in a simple generator model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import UpSampling2D
from keras.layers import Conv2D
# define model
model = Sequential()
# define input shape, output enough activations for for 128 5x5 image
model.add(Dense(128 * 5 * 5, input_dim=100))
# reshape vector of activations into 128 feature maps with 5x5
model.add(Reshape((5, 5, 128)))
# double input from 128 5x5 to 1 10x10 feature map
model.add(UpSampling2D())
# fill in detail in the upsampled feature maps and output a single image
model.add(Conv2D(1, (3,3), padding='same'))
# summarize model
model.summary()
```

Listing 3.15: Example of defining a simple generator model using the `UpSampling2D` layer.

Running the example creates the model and summarizes the output shape of each layer. We can see that the `Dense` layer outputs 3,200 activations that are then reshaped into 128 feature maps with the shape 5×5. The widths and heights are doubled to 10×10 by the `UpSampling2D` layer, resulting in a feature map with quadruple the area. Finally, the `Conv2D` processes these feature maps and adds in detail, outputting a single 10×10 image.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 3200)              323200
_________________________________________________________________
reshape_1 (Reshape)          (None, 5, 5, 128)         0
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 10, 10, 128)  0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 1)         1153
=================================================================
```

```
Total params: 324,353
Trainable params: 324,353
Non-trainable params: 0
_________________________________________________________________
```

Listing 3.16: Example output from defining a simple generator model using the `UpSampling2D` layer.

3.4 How to Use the Transpose Convolutional Layer

The transpose convolutional layer is more complex than a simple upsampling layer. A simple way to think about it is that it both performs the upsample operation and interprets the coarse input data to fill in the detail while it is upsampling. It is like a layer that combines the `UpSampling2D` and `Conv2D` layers into one layer. This is a crude understanding, but a practical starting point.

> The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution
>
> — *A Guide To Convolution Arithmetic For Deep Learning*, 2016.

In fact, the transpose convolutional layer performs an inverse convolution operation. Specifically, the forward and backward passes of the convolutional layer are reversed.

> One way to put it is to note that the kernel defines a convolution, but whether it's a direct convolution or a transposed convolution is determined by how the forward and backward passes are computed.
>
> — *A Guide To Convolution Arithmetic For Deep Learning*, 2016.

It is sometimes called a deconvolution or deconvolutional layer and models that use these layers can be referred to as deconvolutional networks, or deconvnets.

> A deconvnet can be thought of as a convnet model that uses the same components (filtering, pooling) but in reverse, so instead of mapping pixels to features does the opposite.
>
> — *Visualizing and Understanding Convolutional Networks*, 2013.

Referring to this operation as a deconvolution is technically incorrect as a deconvolution is a specific mathematical operation not performed by this layer. In fact, the traditional convolutional layer does not technically perform a convolutional operation, it performs a cross-correlation.

> The deconvolution layer, to which people commonly refer, first appears in Zeiler's paper as part of the deconvolutional network but does not have a specific name. [...] It also has many names including (but not limited to) subpixel or fractional convolutional layer, transposed convolutional layer, inverse, up or backward convolutional layer.
>
> — *Is The Deconvolution Layer The Same As A Convolutional Layer?*, 2016.

It is a very flexible layer, although we will focus on its use in generative models for upsampling an input image. The transpose convolutional layer is much like a normal convolutional layer. It requires that you specify the number of filters and the kernel size of each filter. The key to the layer is the stride. Typically, the stride of a convolutional layer is (1×1), that is a filter is moved along one pixel horizontally for each read from left-to-right, then down pixel for the next row of reads. A stride of 2×2 on a normal convolutional layer has the effect of downsampling the input, much like a pooling layer. In fact, a 2×2 stride can be used instead of a pooling layer in the discriminator model.

The transpose convolutional layer is like an inverse convolutional layer. As such, you would intuitively think that a 2×2 stride would upsample the input instead of downsample, which is exactly what happens. Stride or strides refers to the manner of a filter scanning across an input in a traditional convolutional layer. Whereas, in a transpose convolutional layer, stride refers to the manner in which outputs in the feature map are laid down. This effect can be implemented with a normal convolutional layer using a fractional input stride (f), e.g. with a stride of $f = \frac{1}{2}$. When inverted, the output stride is set to the numerator of this fraction, e.g. $f = 2$.

> In a sense, upsampling with factor f is convolution with a fractional input stride of $\frac{1}{f}$. So long as f is integral, a natural way to upsample is therefore backwards convolution (sometimes called deconvolution) with an output stride of f.
>
> — *Fully Convolutional Networks for Semantic Segmentation*, 2014.

One way that this effect can be achieved with a normal convolutional layer is by inserting new rows and columns of 0.0 values in the input data.

> Finally note that it is always possible to emulate a transposed convolution with a direct convolution. The disadvantage is that it usually involves adding many columns and rows of zeros to the input ...
>
> — *A Guide To Convolution Arithmetic For Deep Learning*, 2016.

Let's make this concrete with an example. Consider an input image with the size 2×2 as follows:

```
        1, 2
Input = 3, 4
```

Listing 3.17: Example of an input matrix.

Assuming a single filter with a 1×1 kernel and model weights that result in no changes to the inputs when output (e.g. a model weight of 1.0 and a bias of 0.0), a transpose convolutional operation with an output stride of 1×1 will reproduce the output as-is:

```
         1, 2
Output = 3, 4
```

Listing 3.18: Example of an output matrix with a 1×1 stride.

With an output stride of (2,2), the 1×1 convolution requires the insertion of additional rows and columns into the input image so that the reads of the operation can be performed. Therefore, the input looks as follows:

```
        1, 0, 2, 0
Input = 0, 0, 0, 0
        3, 0, 4, 0
        0, 0, 0, 0
```

Listing 3.19: Example of an input matrix with a 2×2 stride.

The model can then read across this input using an output stride of (2,2) and will output a 4×4 image, in this case with no change as our model weights have no effect by design:

```
         1, 0, 2, 0
Output = 0, 0, 0, 0
         3, 0, 4, 0
         0, 0, 0, 0
```

Listing 3.20: Example of an output matrix with a 2×2 stride.

3.4.1 Worked Example Using the `Conv2DTranspose` Layer

Keras provides the transpose convolution capability via the `Conv2DTranspose` layer. It can be added to your model directly; for example:

```
...
# define model
model = Sequential()
model.add(Conv2DTranspose(...))
```

Listing 3.21: Example of defining a model with a `Conv2DTranspose` layer.

We can demonstrate the behavior of this layer with a simple contrived example. First, we can define a contrived input image that is 2×2 pixels, as we did in the previous section. We can use specific values for each pixel so that after the transpose convolutional operation, we can see exactly what effect the operation had on the input.

```
...
# define input data
X = asarray([[1, 2],
       [3, 4]])
# show input data for context
print(X)
```

Listing 3.22: Example of defining an input matrix for the model.

Once the image is defined, we must add a channel dimension (e.g. grayscale) and also a sample dimension (e.g. we have 1 sample) so that we can pass it as input to the model.

```
...
# reshape input data into one sample a sample with a channel
X = X.reshape((1, 2, 2, 1))
```

Listing 3.23: Example of reshaping the input matrix as an image input.

We can now define our model. The model has only the `Conv2DTranspose` layer, which takes 2×2 grayscale images as input directly and outputs the result of the operation. The `Conv2DTranspose` both upsamples and performs a convolution. As such, we must specify both the number of filters and the size of the filters as we do for `Conv2D` layers. Additionally, we must specify a stride of (2,2) because the upsampling is achieved by the stride behavior of the convolution on the input. Specifying a stride of (2,2) has the effect of spacing out the input. Specifically, rows and columns of 0.0 values are inserted to achieve the desired stride. In this example, we will use one filter, with a 1×1 kernel and a stride of 2×2 so that the 2×2 input image is upsampled to 4×4.

```
...
# define model
model = Sequential()
model.add(Conv2DTranspose(1, (1,1), strides=(2,2), input_shape=(2, 2, 1)))
# summarize the model
model.summary()
```

Listing 3.24: Example of defining a model with a `Conv2DTranspose` layer.

To make it clear what the `Conv2DTranspose` layer is doing, we will fix the single weight in the single filter to the value of 1.0 and use a bias value of 0.0. These weights, along with a kernel size of (1,1) will mean that values in the input will be multiplied by 1 and output as-is, and the 0 values in the new rows and columns added via the stride of 2×2 will be output as 0 (e.g. 1×0 in each case).

```
...
# define weights that do nothing
weights = [asarray([[[[1]]]]), asarray([0])]
# store the weights in the model
model.set_weights(weights)
```

Listing 3.25: Example of fixing the model weights.

We can then use the model to make a prediction, that is upsample a provided input image.

```
...
# make a prediction with the model
yhat = model.predict(X)
```

Listing 3.26: Example of making a prediction with the model.

The output will have four dimensions, like the input, therefore, we can convert it back to a 2×2 array to make it easier to review the result.

```
...
# reshape output to remove channel to make printing easier
yhat = yhat.reshape((4, 4))
# summarize output
print(yhat)
```

Listing 3.27: Example of interpreting the prediction from the model.

Tying all of this together, the complete example of using the `Conv2DTranspose` layer in Keras is provided below.

```
# example of using the transpose convolutional layer
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2DTranspose
# define input data
X = asarray([[1, 2],
       [3, 4]])
# show input data for context
print(X)
# reshape input data into one sample a sample with a channel
X = X.reshape((1, 2, 2, 1))
# define model
model = Sequential()
model.add(Conv2DTranspose(1, (1,1), strides=(2,2), input_shape=(2, 2, 1)))
# summarize the model
model.summary()
# define weights that they do nothing
weights = [asarray([[[[1]]]]), asarray([0])]
# store the weights in the model
model.set_weights(weights)
# make a prediction with the model
yhat = model.predict(X)
# reshape output to remove channel to make printing easier
yhat = yhat.reshape((4, 4))
# summarize output
print(yhat)
```

Listing 3.28: Example of defining a simple model using the `Conv2DTranspose` layer.

Running the example first creates and summarizes our 2×2 input data. Next, the model is summarized. We can see that it will output a 4×4 result as we expect, and importantly, the layer two parameters or model weights. One for the single 1×1 filter and one for the bias. Unlike the `UpSampling2D` layer, the `Conv2DTranspose` will learn during training and will attempt to fill in detail as part of the upsampling process. Finally, the model is used to upsample our input. We can see that the calculations of the cells that involve real values as input result in the real value as output (e.g. 1×1, 1×2, etc.). We can see that where new rows and columns have been inserted by the stride of 2×2, that their 0.0 values multiplied by the 1.0 values in the single 1×1 filter have resulted in 0 values in the output.

```
[[1 2]
 [3 4]]


_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_transpose_1 (Conv2DTr (None, 4, 4, 1)           2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

```
[[1. 0. 2. 0.]
 [0. 0. 0. 0.]
 [3. 0. 4. 0.]
 [0. 0. 0. 0.]]
```

Listing 3.29: Example output from defining a simple model using the `Conv2DTranspose` layer.

Recall that this is a contrived case where we artificially specified the model weights so that we could see the effect of the transpose convolutional operation. In practice, we will use a large number of filters (e.g. 64 or 128), a larger kernel (e.g. 3×3, 5×5, etc.), and the layer will be initialized with random weights that will learn how to effectively upsample with detail during training. In fact, you might imagine how different sized kernels will result in different sized outputs, more than doubling the width and height of the input. In this case, the `padding` argument of the layer can be set to `'same'` to force the output to have the desired (doubled) output shape; for example:

```
...
# example of using padding to ensure that the output are only doubled
model.add(Conv2DTranspose(1, (3,3), strides=(2,2), padding='same', input_shape=(2, 2, 1)))
```

Listing 3.30: Example of defining the `Conv2DTranspose` layer with same padding.

3.4.2 Simple Generator Model With the `Conv2DTranspose` Layer

The `Conv2DTranspose` is more complex than the `UpSampling2D` layer, but it is also effective when used in GAN models, specifically the generator model. Either approach can be used, although the `Conv2DTranspose` layer is preferred, perhaps because of the simpler generator models and possibly better results, although GAN performance and skill is notoriously difficult to quantify. We can demonstrate using the `Conv2DTranspose` layer in a generator model with another simple example.

In this case, our little GAN generator model must produce a 10×10 image and take a 100-element vector from the latent space as input, as in the previous `UpSampling2D` example. First, a `Dense` fully connected layer can be used to interpret the input vector and create a sufficient number of activations (outputs) that can be reshaped into a low-resolution version of our output image, in this case, 128 versions of a 5×5 image.

```
...
# define model
model = Sequential()
# define input shape, output enough activations for 128 5x5 images
model.add(Dense(128 * 5 * 5, input_dim=100))
# reshape vector of activations into 128 feature maps with 5x5
model.add(Reshape((5, 5, 128)))
```

Listing 3.31: Example of defining the base activations for a generator model.

Next, the 5×5 feature maps can be upsampled to a 10×10 feature map. We will use a 3×3 kernel size for the single filter, which will result in a slightly larger than doubled width and height in the output feature map (11×11). Therefore, we will set the `padding` argument to `'same'` to ensure the output dimensions are 10×10 as required.

```
...
# double input from 128 5x5 to 1 10x10 feature map
model.add(Conv2DTranspose(1, (3,3), strides=(2,2), padding='same'))
```

Listing 3.32: Example of adding the `Conv2DTranspose` layer with same padding.

Tying this together, the complete example is listed below.

```
# example of using transpose conv in a simple generator model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2DTranspose
# define model
model = Sequential()
# define input shape, output enough activations for for 128 5x5 image
model.add(Dense(128 * 5 * 5, input_dim=100))
# reshape vector of activations into 128 feature maps with 5x5
model.add(Reshape((5, 5, 128)))
# double input from 128 5x5 to 1 10x10 feature map
model.add(Conv2DTranspose(1, (3,3), strides=(2,2), padding='same'))
# summarize model
model.summary()
```

Listing 3.33: Example of defining a generator model using the `Conv2DTranspose` layer.

Running the example creates the model and summarizes the output shape of each layer. We can see that the `Dense` layer outputs 3,200 activations that are then reshaped into 128 feature maps with the shape 5×5. The widths and heights are doubled to 10×10 by the `Conv2DTranspose` layer resulting in a single feature map with quadruple the area.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 3200)              323200
_________________________________________________________________
reshape_1 (Reshape)          (None, 5, 5, 128)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 10, 10, 1)         1153
=================================================================
Total params: 324,353
Trainable params: 324,353
Non-trainable params: 0
_________________________________________________________________
```

Listing 3.34: Example output from defining a generator model using the `Conv2DTranspose` layer.

3.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

3.5.1 Papers

- A Guide To Convolution Arithmetic For Deep Learning, 2016.
  https://arxiv.org/abs/1603.07285
- Deconvolutional Networks, 2010.
  https://ieeexplore.ieee.org/document/5539957
- Is The Deconvolution Layer The Same As A Convolutional Layer?, 2016.
  https://arxiv.org/abs/1609.07009
- Visualizing and Understanding Convolutional Networks, 2013.
  https://arxiv.org/abs/1311.2901
- Fully Convolutional Networks for Semantic Segmentation, 2014.
  https://arxiv.org/abs/1411.4038

3.5.2 API

- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/

3.5.3 Articles

- Convolution Arithmetic Project, GitHub.
  https://github.com/vdumoulin/conv_arithmetic
- What Are Deconvolutional Layers?, Data Science Stack Exchange.
  https://datascience.stackexchange.com/questions/6107/what-are-deconvolutional-layers

3.6 Summary

In this tutorial, you discovered how to use Upsampling and Transpose Convolutional Layers in Generative Adversarial Networks when generating images. Specifically, you learned:

- Generative models in the GAN architecture are required to upsample input data in order to generate an output image.
- The Upsampling layer is a simple layer with no weights that will double the dimensions of input and can be used in a generative model when followed by a traditional convolutional layer.
- The Transpose Convolutional layer is an inverse convolutional layer that will both upsample input and learn how to fill in details during the model training process.

3.6.1 Next

In the next tutorial, you will discover the algorithm for training generative adversarial network models.

Chapter 4

How to Implement the GAN Training Algorithm

The Generative Adversarial Network, or GAN for short, is an architecture for training a generative model. The architecture is comprised of two models. The generator that we are interested in, and a discriminator model that is used to assist in the training of the generator. Initially, both of the generator and discriminator models were implemented as Multilayer Perceptrons (MLP), although more recently, the models are implemented as deep convolutional neural networks. It can be challenging to understand how a GAN is trained and exactly how to understand and implement the loss function for the generator and discriminator models. In this tutorial, you will discover how to implement the generative adversarial network training algorithm and loss functions. After completing this tutorial, you will know:

- How to implement the training algorithm for a generative adversarial network.
- How the loss function for the discriminator and generator work.
- How to implement weight updates for the discriminator and generator models in practice.

Let's get started.

4.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. How to Implement the GAN Training Algorithm
2. Understanding the GAN Loss Function
3. How to Train GAN Models in Practice

Note: The code examples in this tutorial are snippets only, not standalone runnable examples. They are designed to help you develop an intuition for the algorithm and they can be used as the starting point for implementing the GAN training algorithm on your own project.

4.2 How to Implement the GAN Training Algorithm

The GAN training algorithm involves training both the discriminator and the generator model in parallel. The algorithm is summarized in the figure below, taken from the original 2014 paper by Goodfellow, et al. titled *Generative Adversarial Networks*.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 4.1: Summary of the Generative Adversarial Network Training Algorithm. Taken from: Generative Adversarial Networks.

Let's take some time to unpack and get comfortable with this algorithm. The outer loop of the algorithm involves iterating over steps to train the models in the architecture. One cycle through this loop is not an epoch: it is a single update comprised of specific batch updates to the discriminator and generator models. An epoch is defined as one cycle through a training dataset, where the samples in a training dataset are used to update the model weights in minibatches. For example, a training dataset of 100 samples used to train a model with a minibatch size of 10 samples would involve 10 minibatch updates per epoch. The model would be fit for a given number of epochs, such as 500.

This is often hidden from you via the automated training of a model via a call to the `fit()` function and specifying the number of epochs and the size of each minibatch. In the case of the GAN, the number of training iterations must be defined based on the size of your training dataset and batch size. In the case of a dataset with 100 samples, a batch size of 10, and 500 training epochs, we would first calculate the number of batches per epoch and use this to calculate the total number of training iterations using the number of epochs. For example:

```
...
batches_per_epoch = floor(dataset_size / batch_size)
total_iterations = batches_per_epoch * total_epochs
```

Listing 4.1: Example of calculating the total training iterations.

In the case of a dataset of 100 samples, a batch size of 10, and 500 epochs, the GAN would be trained for floor($\frac{100}{10}$) $\times$ 500 or 5,000 total iterations. Next, we can see that one iteration of training results in possibly multiple updates to the discriminator and one update to the generator, where the number of updates to the discriminator is a hyperparameter that is set to 1.

> The training process consists of simultaneous SGD. On each step, two minibatches are sampled: a minibatch of x values from the dataset and a minibatch of z values drawn from the model's prior over latent variables. Then two gradient steps are made simultaneously ...
>
> — *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

We can therefore summarize the training algorithm with Python pseudocode as follows:

```
# gan training algorithm
def train_gan(dataset, n_epochs, n_batch):
  # calculate the number of batches per epoch
  batches_per_epoch = int(len(dataset) / n_batch)
  # calculate the number of training iterations
  n_steps = batches_per_epoch * n_epochs
  # gan training algorithm
  for i in range(n_steps):
    # update the discriminator model
    # ...
    # update the generator model
    # ...
```

Listing 4.2: Example of the high-level GAN training algorithm.

An alternative approach may involve enumerating the number of training epochs and splitting the training dataset into batches for each epoch. Updating the discriminator model involves a few steps. First, a batch of random points from the latent space must be created for use as input to the generator model to provide the basis for the generated or *fake* samples. Then a batch of samples from the training dataset must be selected for input to the discriminator as the *real* samples.

Next, the discriminator model must make predictions for the real and fake samples and the weights of the discriminator must be updated proportional to how correct or incorrect those predictions were. The predictions are probabilities and we will get into the nature of the predictions and the loss function that is minimized in the next section. For now, we can outline what these steps actually look like in practice. We need a generator and a discriminator model, e.g. such as a Keras model. These can be provided as arguments to the training function. Next, we must generate points from the latent space and then use the generator model in its current form to generate some fake images. For example:

```
...
# generate points in the latent space
z = randn(latent_dim * n_batch)
# reshape into a batch of inputs for the network
z = z.reshape(n_batch, latent_dim)
# generate fake images
fake = generator.predict(z)
```

Listing 4.3: Example of generating fake examples.

The size of the latent dimension is also provided as a hyperparameter to the training algorithm. We then must select a batch of real samples, and this too will be wrapped into a function.

```
...
# select a batch of random real images
ix = randint(0, len(dataset), n_batch)
# retrieve real images
real = dataset[ix]
```

Listing 4.4: Example of selecting real examples.

The discriminator model must then make a prediction for each of the generated and real images and the weights must be updated.

```
# gan training algorithm
def train_gan(generator, discriminator, dataset, latent_dim, n_epochs, n_batch):
  # calculate the number of batches per epoch
  batches_per_epoch = int(len(dataset) / n_batch)
  # calculate the number of training iterations
  n_steps = batches_per_epoch * n_epochs
  # gan training algorithm
  for i in range(n_steps):
    # generate points in the latent space
    z = randn(latent_dim * n_batch)
    # reshape into a batch of inputs for the network
    z = z.reshape(n_batch, latent_dim)
    # generate fake images
    fake = generator.predict(z)
    # select a batch of random real images
    ix = randint(0, len(dataset), n_batch)
    # retrieve real images
    real = dataset[ix]
    # update weights of the discriminator model
    # ...

    # update the generator model
    # ...
```

Listing 4.5: Example of the GAN training algorithm with sample selection and generation.

Next, the generator model must be updated. Again, a batch of random points from the latent space must be selected and passed to the generator to generate fake images, and then passed to the discriminator to classify.

```
...
# generate points in the latent space
z = randn(latent_dim * n_batch)
# reshape into a batch of inputs for the network
z = z.reshape(n_batch, latent_dim)
# generate fake images
fake = generator.predict(z)
# classify as real or fake
result = discriminator.predict(fake)
```

Listing 4.6: Example of updating the generator model.

The response can then be used to update the weights of the generator model.

```
# gan training algorithm
def train_gan(generator, discriminator, dataset, latent_dim, n_epochs, n_batch):
  # calculate the number of batches per epoch
  batches_per_epoch = int(len(dataset) / n_batch)
  # calculate the number of training iterations
  n_steps = batches_per_epoch * n_epochs
  # gan training algorithm
  for i in range(n_steps):
    # generate points in the latent space
    z = randn(latent_dim * n_batch)
    # reshape into a batch of inputs for the network
    z = z.reshape(n_batch, latent_dim)
    # generate fake images
    fake = generator.predict(z)
    # select a batch of random real images
    ix = randint(0, len(dataset), n_batch)
    # retrieve real images
    real = dataset[ix]
    # update weights of the discriminator model
    # ...
    # generate points in the latent space
    z = randn(latent_dim * n_batch)
    # reshape into a batch of inputs for the network
    z = z.reshape(n_batch, latent_dim)
    # generate fake images
    fake = generator.predict(z)
    # classify as real or fake
    result = discriminator.predict(fake)
    # update weights of the generator model
    # ...
```

Listing 4.7: Example of the GAN training algorithm with with preparation for updating model weights.

It is interesting that the discriminator is updated with two batches of samples each training iteration whereas the generator is only updated with a single batch of samples per training iteration. Now that we have defined the training algorithm for the GAN, we need to understand how the model weights are updated. This requires understanding the loss function used to train the GAN.

4.3 Understanding the GAN Loss Function

The discriminator is trained to correctly classify real and fake images. This is achieved by maximizing the log of predicted probability of real images and the log of the inverted probability of fake images, averaged over each minibatch of examples. Recall that we add log probabilities, which is the same as multiplying probabilities, although without vanishing into small numbers. Therefore, we can understand this loss function as seeking probabilities close to 1.0 for real images and probabilities close to 0.0 for fake images, inverted to become larger numbers. The addition of these values means that lower average values of this loss function result in better performance of the discriminator. Inverting this to a minimization problem, it should not be surprising if you are familiar with developing neural networks for binary classification, as this is

exactly the approach used.

> This is just the standard cross-entropy cost that is minimized when training a standard binary classifier with a sigmoid output. The only difference is that the classifier is trained on two minibatches of data; one coming from the dataset, where the label is 1 for all examples, and one coming from the generator, where the label is 0 for all examples.
>
> — *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

The generator is more tricky. The GAN algorithm defines the generator model's loss as minimizing the log of the inverted probability of the discriminator's prediction of fake images, averaged over a minibatch. This is straightforward, but according to the authors, it is not effective in practice when the generator is poor and the discriminator is good at rejecting fake images with high confidence. The loss function no longer gives good gradient information that the generator can use to adjust weights and instead saturates.

> In this case, $\log(1 - D(G(z)))$ saturates. Rather than training G to minimize $\log(1 - D(G(z)))$ we can train G to maximize $\log D(G(z))$. This objective function results in the same fixed point of the dynamics of G and D but provides much stronger gradients early in learning.
>
> — *Generative Adversarial Networks*, 2014.

Instead, the authors recommend maximizing the log of the discriminator's predicted probability for fake images. The change is subtle. In the first case, the generator is trained to minimize the probability of the discriminator being correct. With this change to the loss function, the generator is trained to maximize the probability of the discriminator being incorrect.

> In the minimax game, the generator minimizes the log-probability of the discriminator being correct. In this game, the generator maximizes the log probability of the discriminator being mistaken.
>
> — *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

The sign of this loss function can then be inverted to give a familiar minimizing loss function for training the generator. As such, this is sometimes referred to as the $-\log(D)$ trick for training GANs.

> Our baseline comparison is DCGAN, a GAN with a convolutional architecture trained with the standard GAN procedure using the $-\log D$ trick.
>
> — *Wasserstein GAN*, 2017.

GAN loss is covered in further detail in Chapter 14. Now that we understand the GAN loss function, we can look at how the discriminator and the generator model can be updated in practice.

4.4 How to Train GAN Models in Practice

The practical implementation of the GAN loss function and model updates is straightforward. We will look at examples using the Keras library. We can implement the discriminator directly by configuring the discriminator model to predict a probability of 1 for real images and 0 for fake images and minimizing the cross-entropy loss, specifically the binary cross-entropy loss. For example, a snippet of our model definition with Keras for the discriminator might look as follows for the output layer and the compilation of the model with the appropriate loss function.

```
...
# output layer
model.add(Dense(1, activation='sigmoid'))
# compile model
model.compile(loss='binary_crossentropy', ...)
```

Listing 4.8: Example of defining discriminator output activation function and loss function.

The defined model can be trained for each batch of real and fake samples providing arrays of 1s and 0s for the expected outcome. The `ones()` and `zeros()` NumPy functions can be used to create these target labels, and the Keras function `train_on_batch()` can be used to update the model for each batch of samples.

```
...
X_fake = ...
X_real = ...
# define target labels for fake images
y_fake = zeros((n_batch, 1))
# update the discriminator for fake images
discriminator.train_on_batch(X_fake, y_fake)
# define target labels for real images
y_real = ones((n_batch, 1))
# update the discriminator for real images
discriminator.train_on_batch(X_real, y_real)
```

Listing 4.9: Example of updating the weights of the discriminator model.

The discriminator model will be trained to predict the probability of *realness* of a given input image that can be interpreted as a class label of $class = 0$ for fake and $class = 1$ for real. The generator is trained to maximize the discriminator predicting a high probability of $class = 1$ for generated (fake) images. This is achieved by updating the generator via the discriminator with the class label of 1 for the generated images. The discriminator is not updated in this operation but provides the gradient information required to update the weights of the generator model. For example, if the discriminator predicts a low average probability of being real for the batch of generated images, then this will result in a large error signal propagated backward into the generator given the *expected probability* for the samples was 1.0 for real. This large error signal, in turn, results in relatively large changes to the generator to hopefully improve its ability at generating fake samples on the next batch.

This can be implemented in Keras by creating a composite model that combines the generator and discriminator models, allowing the output images from the generator to flow into discriminator directly, and in turn, allow the error signals from the predicted probabilities of the discriminator to flow back through the weights of the generator model. For example:

```
# define a composite gan model for the generator and discriminator
```

```
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam')
  return model
```

Listing 4.10: Example of defining a composite model for updating the generator.

The composite model can then be updated using fake images and real class labels.

```
...
# generate points in the latent space
z = randn(latent_dim * n_batch)
# reshape into a batch of inputs for the network
z = z.reshape(n_batch, latent_dim)
# define target labels for real images
y_real = ones((n_batch, 1))
# update generator model
gan_model.train_on_batch(z, y_real)
```

Listing 4.11: Example of updating the weights of the generator model via the discriminator model.

That completes our tour of the GAN training algorithm, loss function and weight update details for the discriminator and generator models.

4.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Generative Adversarial Networks, 2014.
  https://arxiv.org/abs/1406.2661
- NIPS 2016 Tutorial: Generative Adversarial Networks, 2016.
  https://arxiv.org/abs/1701.00160
- Wasserstein GAN, 2017.
  https://arxiv.org/abs/1701.07875

4.6 Summary

In this tutorial, you discovered how to implement the generative adversarial network training algorithm and loss functions. Specifically, you learned:

- How to implement the training algorithm for a generative adversarial network.

- How the loss function for the discriminator and generator work.
- How to implement weight updates for the discriminator and generator models in practice.

4.6.1 Next

In the next tutorial, you will discover tips, tricks, and hacks that you can use to train stable GAN models.

Chapter 5

How to Implement GAN Hacks to Train Stable Models

Generative Adversarial Networks, or GANs, are challenging to train. This is because the architecture involves both a generator and a discriminator model that compete in a zero-sum game. It means that improvements to one model come at the cost of a degrading of performance in the other model. The result is a very unstable training process that can often lead to failure, e.g. a generator that generates the same image all the time or generates nonsense. As such, there are a number of heuristics or best practices (called *GAN hacks*) that can be used when configuring and training your GAN models. These heuristics have been hard won by practitioners testing and evaluating hundreds or thousands of combinations of configuration operations on a range of problems over many years.

Some of these heuristics can be challenging to implement, especially for beginners. Further, some or all of them may be required for a given project, although it may not be clear which subset of heuristics should be adopted, requiring experimentation. This means a practitioner must be ready to implement a given heuristic with little notice. In this tutorial, you will discover how to implement a suite of best practices or GAN hacks that you can copy-and-paste directly into your GAN project. After reading this tutorial, you will know:

- The simultaneous training of generator and discriminator models in GANs is inherently unstable.
- How to implement seven best practices for the deep convolutional GAN model architecture from scratch.
- How to implement four additional best practices from Soumith Chintala's GAN Hacks presentation and list.

Let's get started.

5.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Challenge of Training GANs

2. Heuristics for Training Stable GANs
3. Deep Convolutional GANs (DCGANs).
4. Soumith Chintala's GAN Hacks.

5.2 Challenge of Training GANs

GANs are difficult to train. The reason they are difficult to train is that both the generator model and the discriminator model are trained simultaneously in a game. This means that improvements to one model come at the expense of the other model. The goal of training two models involves finding a point of equilibrium between the two competing concerns.

> Training GANs consists in finding a Nash equilibrium to a two-player non-cooperative game. [...] Unfortunately, finding Nash equilibria is a very difficult problem. Algorithms exist for specialized cases, but we are not aware of any that are feasible to apply to the GAN game, where the cost functions are non-convex, the parameters are continuous, and the parameter space is extremely high-dimensional

— *Improved Techniques for Training GANs*, 2016.

It also means that every time the parameters of one of the models are updated, the nature of the optimization problem that is being solved is changed. This has the effect of creating a dynamic system.

> But with a GAN, every step taken down the hill changes the entire landscape a little. It's a dynamic system where the optimization process is seeking not a minimum, but an equilibrium between two forces.

— Page 306, *Deep Learning with Python*, 2017.

In neural network terms, the technical challenge of training two competing neural networks at the same time is that they can fail to converge.

> The largest problem facing GANs that researchers should try to resolve is the issue of non-convergence.

— *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

Instead of converging, GANs may suffer from one of a small number of failure modes. A common failure mode is that instead of finding a point of equilibrium, the generator oscillates between generating specific examples in the domain.

> In practice, GANs often seem to oscillate, [...] meaning that they progress from generating one kind of sample to generating another kind of sample without eventually reaching an equilibrium.

— *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

Perhaps the most challenging model failure is the case where multiple inputs to the generator result in the generation of the same output. This is referred to as *mode collapse*, and may represent one of the most challenging issues when training GANs.

> Mode collapse, also known as the scenario, is a problem that occurs when the generator learns to map several different input z values to the same output point.
>
> — *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

GAN failure modes are further covered in Chapter 10. Finally, there are no good objective metrics for evaluating whether a GAN is performing well during training. E.g. reviewing loss is not sufficient. Instead, the best approach is to visually inspect generated examples and use subjective evaluation.

> Generative adversarial networks lack an objective function, which makes it difficult to compare performance of different models. One intuitive metric of performance can be obtained by having human annotators judge the visual quality of samples.
>
> — *Improved Techniques for Training GANs*, 2016.

5.3 Heuristics for Training Stable GANs

GANs are difficult to train. At the time of writing, there is no good theoretical foundation as to how to design and train GAN models, but there is established literature of heuristics, or *hacks*, that have been empirically demonstrated to work well in practice. As such, there are a range of best practices to consider and implement when developing a GAN model. Perhaps the two most important sources of suggested configuration and training parameters are:

- Alec Radford, et al's 2015 paper that introduced the DCGAN architecture.
- Soumith Chintala's 2016 presentation and associated *GAN Hacks* list.

In this tutorial, we will explore how to implement the most important best practices from these two sources.

5.4 Deep Convolutional GANs (DCGANs)

Perhaps one of the most important steps forward in the design and training of stable GAN models was the 2015 paper by Alec Radford, et al. titled *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. In the paper, they describe the Deep Convolutional GAN, or DCGAN, approach to GAN development that has become the de facto standard.

> Stabilization of GAN learning remains an open problem. Fortunately, GAN learning performs well when the model architecture and hyperparameters are carefully selected. Radford et al. (2015) crafted a deep convolutional GAN (DCGAN) that performs very well for image synthesis tasks ...

— Page 701, *Deep Learning*, 2016.

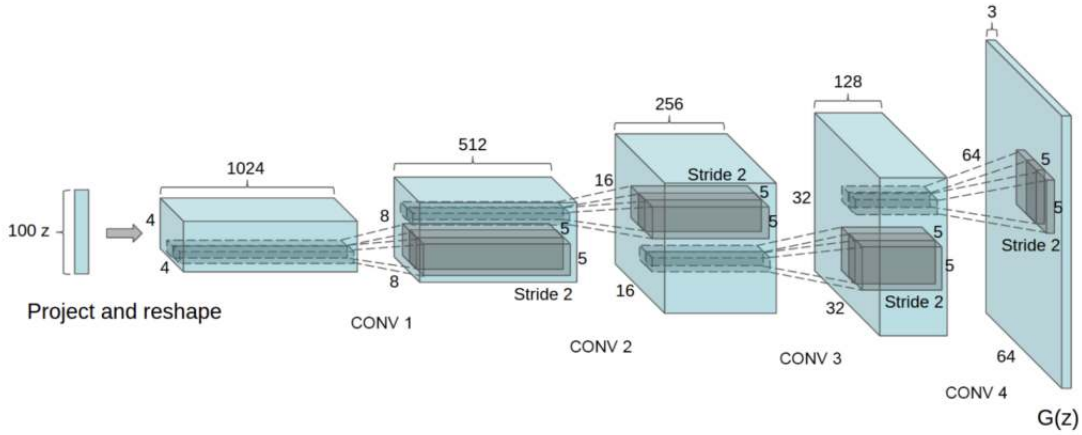

Figure 5.1: Example of the Generator Model Architecture for the DCGAN. Taken from: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

The findings in this paper were hard earned, developed after extensive empirical trial and error with different model architectures, configurations, and training schemes. Their approach remains highly recommended as a starting point when developing new GANs, at least for image-synthesis-based tasks.

> ... after extensive model exploration we identified a family of architectures that resulted in stable training across a range of datasets and allowed for training higher resolution and deeper generative models.

— *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, 2015.

Below provides a summary of the GAN architecture recommendations from the paper.

Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

Figure 5.2: Summary of Architectural Guidelines for Training Stable Deep Convolutional Generative Adversarial Networks. Taken from: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

We will look at how to implement seven best practices for the DCGAN model architecture in this section.

5.4.1 Downsample Using Strided Convolutions

The discriminator model is a standard convolutional neural network model that takes an image as input and must output a binary classification as to whether it is real or fake. It is standard practice with deep convolutional networks to use pooling layers to downsample the input and feature maps with the depth of the network. This is not recommended for the DCGAN, and instead, they recommend downsampling using strided convolutions.

This involves defining a convolutional layer as per normal, but instead of using the default two-dimensional stride of (1,1) to change it to (2,2). This has the effect of downsampling the input, specifically halving the width and height of the input, resulting in output feature maps with one quarter the area. The example below demonstrates this with a single hidden convolutional layer that uses downsampling strided convolutions by setting the `strides` argument to (2,2). The effect is the model will downsample the input from 64×64 to 32×32.

```
# example of downsampling with strided convolutions
from keras.models import Sequential
from keras.layers import Conv2D
# define model
model = Sequential()
model.add(Conv2D(64, (3,3), strides=(2,2), padding='same', input_shape=(64,64,3)))
# summarize model
model.summary()
```

Listing 5.1: Example of using strided convolutions for downsampling.

Running the example shows the shape of the output of the convolutional layer, where the feature maps have one quarter of the area.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 64)        1792
=================================================================
Total params: 1,792
Trainable params: 1,792
Non-trainable params: 0
_________________________________________________________________
```

Listing 5.2: Example output from using strided convolutions for downsampling.

5.4.2 Upsample Using Strided Convolutions

The generator model must generate an output image given a random point from the latent space as input. The recommended approach for achieving this is to use a transpose convolutional layer with a strided convolution. This is a special type of layer that performs the convolution operation in reverse. Intuitively, this means that setting a stride of 2×2 will have the opposite effect, upsampling the input instead of downsampling it in the case of a normal convolutional layer.

By stacking a transpose convolutional layer with strided convolutions, the generator model is able to scale a given input to the desired output dimensions. The example below demonstrates this with a single hidden transpose convolutional layer that uses upsampling strided convolutions

by setting the `strides` argument to (2,2). The effect is the model will upsample the input from 64×64 to 128×128.

```
# example of upsampling with strided convolutions
from keras.models import Sequential
from keras.layers import Conv2DTranspose
# define model
model = Sequential()
model.add(Conv2DTranspose(64, (4,4), strides=(2,2), padding='same', input_shape=(64,64,3)))
# summarize model
model.summary()
```

Listing 5.3: Example of using strided convolutions for upsampling.

Running the example shows the shape of the output of the convolutional layer, where the feature maps have quadruple the area.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_transpose_1 (Conv2DTr (None, 128, 128, 64) 3136
=================================================================
Total params: 3,136
Trainable params: 3,136
Non-trainable params: 0
_________________________________________________________________
```

Listing 5.4: Example output from using strided convolutions for upsampling.

5.4.3 Use Leaky ReLU

The rectified linear activation unit, or ReLU for short, is a simple calculation that returns the value provided as input directly, or the value 0.0 if the input is 0.0 or less. It has become a best practice when developing deep convolutional neural networks generally. The best practice for GANs is to use a variation of the ReLU that allows some values less than zero and learns where the cut-off should be in each node. This is called the leaky rectified linear activation unit, or the `LeakyReLU` layer.

A negative slope can be specified for the `LeakyReLU` and the default value of 0.2 is recommended. Originally, ReLU was recommend for use in the generator model and `LeakyReLU` was recommended for use in the discriminator model, although more recently, the `LeakyReLU` may be recommended in both models. The example below demonstrates using the `LeakyReLU` with the default slope of 0.2 after a convolutional layer in a discriminator model.

```
# example of using leakyrelu in a discriminator model
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import LeakyReLU
# define model
model = Sequential()
model.add(Conv2D(64, (3,3), strides=(2,2), padding='same', input_shape=(64,64,3)))
model.add(LeakyReLU(0.2))
# summarize model
model.summary()
```

Listing 5.5: Example of using the `LeakyReLU` activation function.

Running the example demonstrates the structure of the model with a single convolutional layer followed by the activation layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 64)        1792
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 32, 32, 64)        0
=================================================================
Total params: 1,792
Trainable params: 1,792
Non-trainable params: 0
_________________________________________________________________
```

Listing 5.6: Example output from using the `LeakyReLU` activation function.

5.4.4 Use Batch Normalization

Batch normalization standardizes the activations from a prior layer to have a zero mean and unit variance. This has the effect of stabilizing the training process. Batch normalization is used after the activation of convolution and transpose convolutional layers in the discriminator and generator models respectively. It is added to the model after the hidden layer, but before the activation, such as `LeakyReLU`. The example below demonstrates adding a `BatchNormalization` layer after a `Conv2D` layer in a discriminator model but before the activation.

```
# example of using batch norm in a discriminator model
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import BatchNormalization
from keras.layers import LeakyReLU
# define model
model = Sequential()
model.add(Conv2D(64, (3,3), strides=(2,2), padding='same', input_shape=(64,64,3)))
model.add(BatchNormalization())
model.add(LeakyReLU(0.2))
# summarize model
model.summary()
```

Listing 5.7: Example of using `BatchNormalization`.

Running the example shows the desired usage of batch norm between the outputs of the convolutional layer and the activation function.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 64)        1792
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 64)        256
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 32, 32, 64)        0
=================================================================
Total params: 2,048
Trainable params: 1,920
```

```
Non-trainable params: 128
_________________________________________________________________
```

Listing 5.8: Example output from using `BatchNormalization`.

5.4.5 Use Gaussian Weight Initialization

Before a neural network can be trained, the model weights (parameters) must be initialized to small random variables. The best practice for DCAGAN models reported in the paper is to initialize all weights using a zero-centered Gaussian distribution (the normal or bell-shaped distribution) with a standard deviation of 0.02. The example below demonstrates defining a random Gaussian weight initializer with a mean of 0 and a standard deviation of 0.02 for use in a transpose convolutional layer in a generator model. The same weight initializer instance could be used for each layer in a given model.

```
# example of gaussian weight initialization in a generator model
from keras.models import Sequential
from keras.layers import Conv2DTranspose
from keras.initializers import RandomNormal
# define model
model = Sequential()
init = RandomNormal(mean=0.0, stddev=0.02)
model.add(Conv2DTranspose(64, (4,4), strides=(2,2), padding='same',
    kernel_initializer=init, input_shape=(64,64,3)))
```

Listing 5.9: Example of using Gaussian weight initialization.

5.4.6 Use Adam Stochastic Gradient Descent

Stochastic gradient descent, or SGD for short, is the standard algorithm used to optimize the weights of convolutional neural network models. There are many variants of the training algorithm. The best practice for training DCGAN models is to use the Adam version of stochastic gradient descent with the learning rate `lr` of 0.0002 and the `beta1` momentum value of 0.5 instead of the default of 0.9. The Adam optimization algorithm with this configuration is recommended when both optimizing the discriminator and generator models. The example below demonstrates configuring the Adam stochastic gradient descent optimization algorithm for training a discriminator model.

```
# example of using adam when training a discriminator model
from keras.models import Sequential
from keras.layers import Conv2D
from keras.optimizers import Adam
# define model
model = Sequential()
model.add(Conv2D(64, (3,3), strides=(2,2), padding='same', input_shape=(64,64,3)))
# compile model
opt = Adam(lr=0.0002, beta_1=0.5)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 5.10: Example of using the Adam version of stochastic gradient descent.

5.4.7 Scale Images to the Range [-1,1]

It is recommended to use the hyperbolic tangent activation function as the output from the generator model. As such, it is also recommended that real images used to train the discriminator are scaled so that their pixel values are in the range [-1,1]. This is so that the discriminator will always receive images as input, real and fake, that have pixel values in the same range. Typically, image data is loaded as a NumPy array such that pixel values are 8-bit unsigned integer (uint8) values in the range [0, 255]. First, the array must be converted to floating point values, then rescaled to the required range. The example below provides a function that will appropriately scale a NumPy array of loaded image data to the required range of [-1,1].

```
# example of a function for scaling images
from numpy.random import randint

# scale image data from [0,255] to [-1,1]
def scale_images(images):
  # convert from unit8 to float32
  images = images.astype('float32')
  # scale from [0,255] to [-1,1]
  images = (images - 127.5) / 127.5
  return images

# define one 28x28 color image
images = randint(0, 256, 28 * 28 * 3)
images = images.reshape((1, 28, 28, 3))
# summarize pixel values
print(images.min(), images.max())
# scale
scaled = scale_images(images)
# summarize pixel scaled values
print(scaled.min(), scaled.max())
```

Listing 5.11: Example of scaling images to a new range.

Running the example contrives a single color image with random pixel values in [0,255]. The pixel values are then scaled to the range [-1,1] and the minimum and maximum pixel values are then reported.

```
0 255
-1.0 1.0
```

Listing 5.12: Example output from scaling images to a new range.

5.5 Soumith Chintala's GAN Hacks

Soumith Chintala, one of the co-authors of the DCGAN paper, made a presentation at NIPS 2016 titled *How to Train a GAN?* summarizing many tips and tricks. The video is available on YouTube and is highly recommended[1]. A summary of the tips is also available as a GitHub repository titled *How to Train a GAN? Tips and tricks to make GANs work*[2]. The tips draw

[1] https://www.youtube.com/watch?v=X1mUN6dD8uE

[2] https://github.com/soumith/ganhacks

upon the suggestions from the DCGAN paper as well as elsewhere. In this section, we will review how to implement four additional GAN best practices not covered in the previous section.

5.5.1 Use a Gaussian Latent Space

The latent space defines the shape and distribution of the input to the generator model used to generate new images. The DCGAN recommends sampling from a uniform distribution, meaning that the shape of the latent space is a hypercube. The more recent best practice is to sample from a standard Gaussian distribution, meaning that the shape of the latent space is a hypersphere, with a mean of zero and a standard deviation of one. The example below demonstrates how to generate 500 random Gaussian examples from a 100-dimensional latent space that can be used as input to a generator model; each point could be used to generate an image.

```
# example of sampling from a gaussian latent space
from numpy.random import randn

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape((n_samples, latent_dim))
  return x_input

# size of latent space
n_dim = 100
# number of samples to generate
n_samples = 500
# generate samples
samples = generate_latent_points(n_dim, n_samples)
# summarize
print(samples.shape, samples.mean(), samples.std())
```

Listing 5.13: Example of sampling a Gaussian latent space.

Running the example summarizes the generation of 500 points, each comprised of 100 random Gaussian values with a mean close to zero and a standard deviation close to 1, e.g. a standard Gaussian distribution.

```
(500, 100) -0.004791256735601787 0.9976912528950904
```

Listing 5.14: Example output from sampling a Gaussian latent space.

5.5.2 Separate Batches of Real and Fake Images

The discriminator model is trained using stochastic gradient descent with mini-batches. The best practice is to update the discriminator with separate batches of real and fake images rather than combining real and fake images into a single batch. This can be achieved by updating the model weights for the discriminator model with two separate calls to the `train_on_batch()` function. The code snippet below demonstrates how you can do this within the inner loop of code when training your discriminator model.

```
...
# get randomly selected 'real' samples
X_real, y_real = ...
# update discriminator model weights
discriminator.train_on_batch(X_real, y_real)
# generate 'fake' examples
X_fake, y_fake = ...
# update discriminator model weights
discriminator.train_on_batch(X_fake, y_fake)
```

Listing 5.15: Example of separate batches for real and fake images.

5.5.3 Use Label Smoothing

It is common to use the class label 1 to represent real images and class label 0 to represent fake images when training the discriminator model. These are called hard labels, as the label values are precise or crisp. It is a good practice to use soft labels, such as values slightly more or less than 1.0 or slightly more than 0.0 for real and fake images respectively, where the variation for each image is random. This is often referred to as label smoothing and can have a regularizing effect when training the model. The example below demonstrates defining 1,000 labels for the positive class ($class = 1$) and smoothing the label values uniformly into the range [0.7,1.2] as recommended.

```
# example of positive label smoothing
from numpy import ones
from numpy.random import random

# example of smoothing class=1 to [0.7, 1.2]
def smooth_positive_labels(y):
  return y - 0.3 + (random(y.shape) * 0.5)

# generate 'real' class labels (1)
n_samples = 1000
y = ones((n_samples, 1))
# smooth labels
y = smooth_positive_labels(y)
# summarize smooth labels
print(y.shape, y.min(), y.max())
```

Listing 5.16: Example demonstrating positive label smoothing.

Running the example summarizes the min and max values for the smooth values, showing they are close to the expected values.

```
(1000, 1) 0.7003103006957805 1.1997858934066357
```

Listing 5.17: Example output from demonstrating positive label smoothing.

There have been some suggestions that only positive-class label smoothing is required and to values less than 1.0. Nevertheless, you can also smooth negative class labels. The example below demonstrates generating 1,000 labels for the negative class ($class = 0$) and smoothing the label values uniformly into the range [0.0, 0.3] as recommended.

```
# example of negative label smoothing
from numpy import zeros
from numpy.random import random

# example of smoothing class=0 to [0.0, 0.3]
def smooth_negative_labels(y):
  return y + random(y.shape) * 0.3

# generate 'fake' class labels (0)
n_samples = 1000
y = zeros((n_samples, 1))
# smooth labels
y = smooth_negative_labels(y)
# summarize smooth labels
print(y.shape, y.min(), y.max())
```

Listing 5.18: Example demonstrating negative label smoothing.

```
(1000, 1) 0.00019305316963429408 0.299785314665858
```

Listing 5.19: Example output from demonstrating negative label smoothing.

5.5.4 Use Noisy Labels

The labels used when training the discriminator model are always correct. This means that fake images are always labeled with class 0 and real images are always labeled with class 1. It is recommended to introduce some errors to these labels where some fake images are marked as real, and some real images are marked as fake. If you are using separate batches to update the discriminator for real and fake images, this may mean randomly adding some fake images to the batch of real images, or randomly adding some real images to the batch of fake images. If you are updating the discriminator with a combined batch of real and fake images, then this may involve randomly flipping the labels on some images. The example below demonstrates this by creating 1,000 samples of real ($class = 1$) labels and flipping them with a 5% probability, then doing the same with 1,000 samples of fake ($class = 0$) labels.

```
# example of noisy labels
from numpy import ones
from numpy import zeros
from numpy.random import choice

# randomly flip some labels
def noisy_labels(y, p_flip):
  # determine the number of labels to flip
  n_select = int(p_flip * y.shape[0])
  # choose labels to flip
  flip_ix = choice([i for i in range(y.shape[0])], size=n_select)
  # invert the labels in place
  y[flip_ix] = 1 - y[flip_ix]
  return y

# generate 'real' class labels (1)
n_samples = 1000
y = ones((n_samples, 1))
```

```
# flip labels with 5% probability
y = noisy_labels(y, 0.05)
# summarize labels
print(y.sum())

# generate 'fake' class labels (0)
y = zeros((n_samples, 1))
# flip labels with 5% probability
y = noisy_labels(y, 0.05)
# summarize labels
print(y.sum())
```

Listing 5.20: Example demonstrating noisy labels.

Try running the example a few times. The results show that approximately 50 of the 1s are flipped to 0s for the positive labels (e.g. 5% of 1,000) and approximately 50 0s are flopped to 1s in for the negative labels.

```
951.0
49.0
```

Listing 5.21: Example output from demonstrating noisy labels.

5.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

5.6.1 Papers

- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.
  https://arxiv.org/abs/1511.06434
- Tutorial: Generative Adversarial Networks, NIPS, 2016.
  https://arxiv.org/abs/1701.00160
- Improved Techniques for Training GANs, 2016.
  https://arxiv.org/abs/1606.03498

5.6.2 API

- Keras API.
  https://keras.io/
- NumPy Random sampling (numpy.random) API.
  https://docs.scipy.org/doc/numpy/reference/routines.random.html
- NumPy Array manipulation routines.
  https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html

5.6.3 Articles

- ganhacks: How to Train a GAN? Tips and tricks to make GANs work.
  https://github.com/soumith/ganhacks
- Ian Goodfellow, Introduction to GANs, NIPS 2016.
  https://www.youtube.com/watch?v=9JpdAg6uMXs
- Soumith Chintala, How to train a GAN, NIPS 2016 Workshop on Adversarial Training.
  https://www.youtube.com/watch?v=X1mUN6dD8uE
- Deconvolution and Checkerboard Artifacts, 2016.
  https://distill.pub/2016/deconv-checkerboard/

5.7 Summary

In this tutorial, you discovered how to implement a suite of best practices or GAN hacks that you can copy-and-paste directly into your GAN project. Specifically, you learned:

- The simultaneous training of generator and discriminator models in GANs is inherently unstable.
- How to implement seven best practices for the deep convolutional GAN model architecture from scratch.
- How to implement four additional best practices from Soumith Chintala's GAN Hacks presentation and list.

5.7.1 Next

This was the final tutorial in this part. In the next part you will begin developing your own GAN models from scratch.

Part II

GAN Basics

Overview

In this part you will discover how to implement, train and use GAN models. After reading the chapters in this part, you will know:

- How to develop a GAN model for a one-dimensional target function (Chapter 6).
- How to develop and use a GAN for generating black and white handwritten digits (Chapter 7).
- How to develop and use a GAN for generating small color photographs of objects (Chapter 8).
- How to explore the latent space of a trained GAN using interpolation and vector arithmetic (Chapter 9).
- How to identify different failure modes when training GAN models (Chapter 10).

Chapter 6

How to Develop a 1D GAN from Scratch

Generative Adversarial Networks, or GANs for short, are a deep learning architecture for training powerful generator models. A generator model is capable of generating new artificial samples that plausibly could have come from an existing distribution of samples. GANs are comprised of both generator and discriminator models. The generator is responsible for generating new samples from the domain, and the discriminator is responsible for classifying whether samples are real or fake (generated). Importantly, the performance of the discriminator model is used to update both the model weights of the discriminator itself and the generator model. This means that the generator never actually sees examples from the domain and is adapted based on how well the discriminator performs.

This is a complex type of model both to understand and to train. One approach to better understand the nature of GAN models and how they can be trained is to develop a model from scratch for a very simple task. A simple task that provides a good context for developing a simple GAN from scratch is a one-dimensional function. This is because both real and generated samples can be plotted and visually inspected to get an idea of what has been learned. A simple function also does not require sophisticated neural network models, meaning the specific generator and discriminator models used on the architecture can be easily understood. In this tutorial, we will select a simple one-dimensional function and use it as the basis for developing and evaluating a generative adversarial network from scratch using the Keras deep learning library. After completing this tutorial, you will know:

- The benefit of developing a generative adversarial network from scratch for a simple one-dimensional function.
- How to develop separate discriminator and generator models, as well as a composite model for training the generator via the discriminator's predictive behavior.
- How to subjectively evaluate generated samples in the context of real examples from the problem domain.

Let's get started.

6.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. Select a One-Dimensional Function
2. Define a Discriminator Model
3. Define a Generator Model
4. Training the Generator Model
5. Evaluating the Performance of the GAN
6. Complete Example of Training the GAN

6.2 Select a One-Dimensional Function

The first step is to select a one-dimensional function to model. Something of the form:

$$y = f(x) \tag{6.1}$$

Where x are input values and y are the output values of the function. Specifically, we want a function that we can easily understand and plot. This will help in both setting an expectation of what the model should be generating and in using a visual inspection of generated examples to get an idea of their quality. We will use a simple function of x^2; that is, the function will return the square of the input. You might remember this function from high school algebra as the u-shaped function. We can define the function in Python as follows:

```
# simple function
def calculate(x):
  return x * x
```

Listing 6.1: Example of defining a target function.

We can define the input domain as real values between -0.5 and 0.5 and calculate the output value for each input value in this linear range, then plot the results to get an idea of how inputs relate to outputs. The complete example is listed below.

```
# demonstrate simple x^2 function
from matplotlib import pyplot

# simple function
def calculate(x):
  return x * x

# define inputs
inputs = [-0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5]
# calculate outputs
outputs = [calculate(x) for x in inputs]
# plot the result
pyplot.plot(inputs, outputs)
pyplot.show()
```

Listing 6.2: Example of plotting the domain for the target function.

Running the example calculates the output value for each input value and creates a plot of input vs. output values. We can see that values far from 0.0 result in larger output values, whereas values close to zero result in smaller output values, and that this behavior is symmetrical around zero. This is the well-known u-shape plot of the X^2 one-dimensional function.

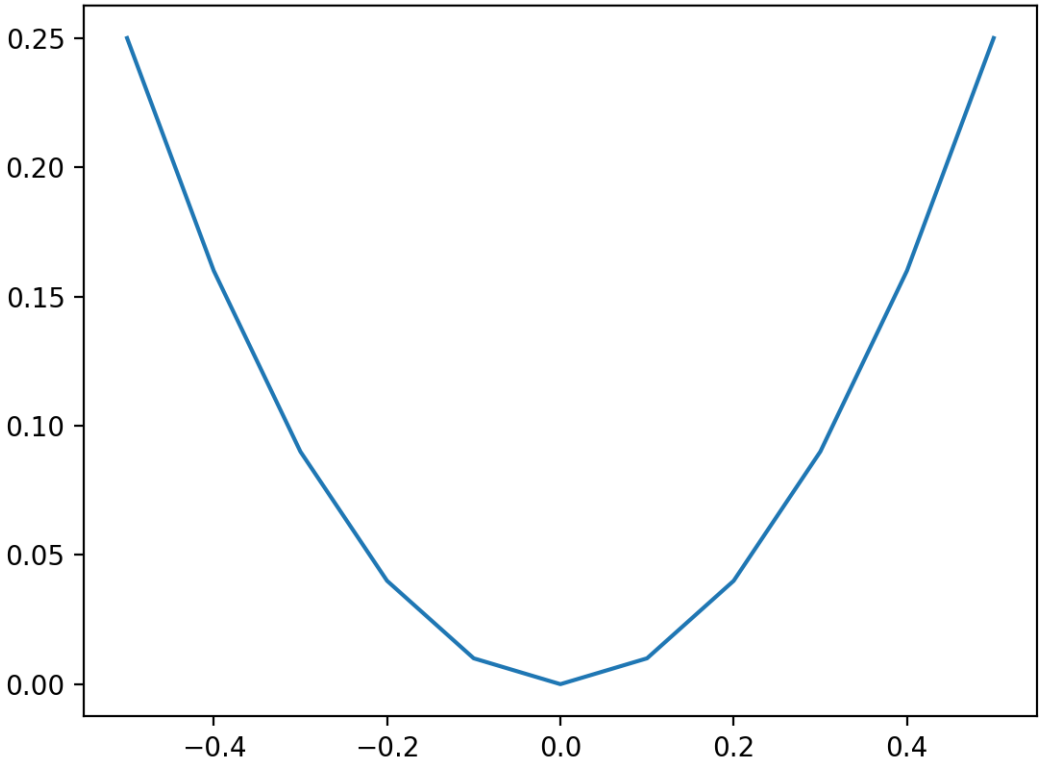

Figure 6.1: Plot of inputs vs. outputs for X^2 function.

We can generate random samples or points from the function. This can be achieved by generating random values between -0.5 and 0.5 and calculating the associated output value. Repeating this many times will give a sample of points from the function, e.g. *real samples*. Plotting these samples using a scatter plot will show the same u-shape plot, although comprised of the individual random samples. The complete example is listed below. First, we generate uniformly random values between 0 and 1, then shift them to the range -0.5 and 0.5. We then calculate the output value for each randomly generated input value and combine the arrays into a single NumPy array with n rows (100) and two columns.

```
# example of generating random samples from X^2
from numpy.random import rand
from numpy import hstack
from matplotlib import pyplot

# generate randoms sample from x^2
def generate_samples(n=100):
  # generate random inputs in [-0.5, 0.5]
```

```
  X1 = rand(n) - 0.5
  # generate outputs X^2 (quadratic)
  X2 = X1 * X1
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  return hstack((X1, X2))

# generate samples
data = generate_samples()
# plot samples
pyplot.scatter(data[:, 0], data[:, 1])
pyplot.show()
```

Listing 6.3: Example of sampling the domain for the target function.

Running the example generates 100 random inputs and their calculated output and plots the sample as a scatter plot, showing the familiar u-shape.

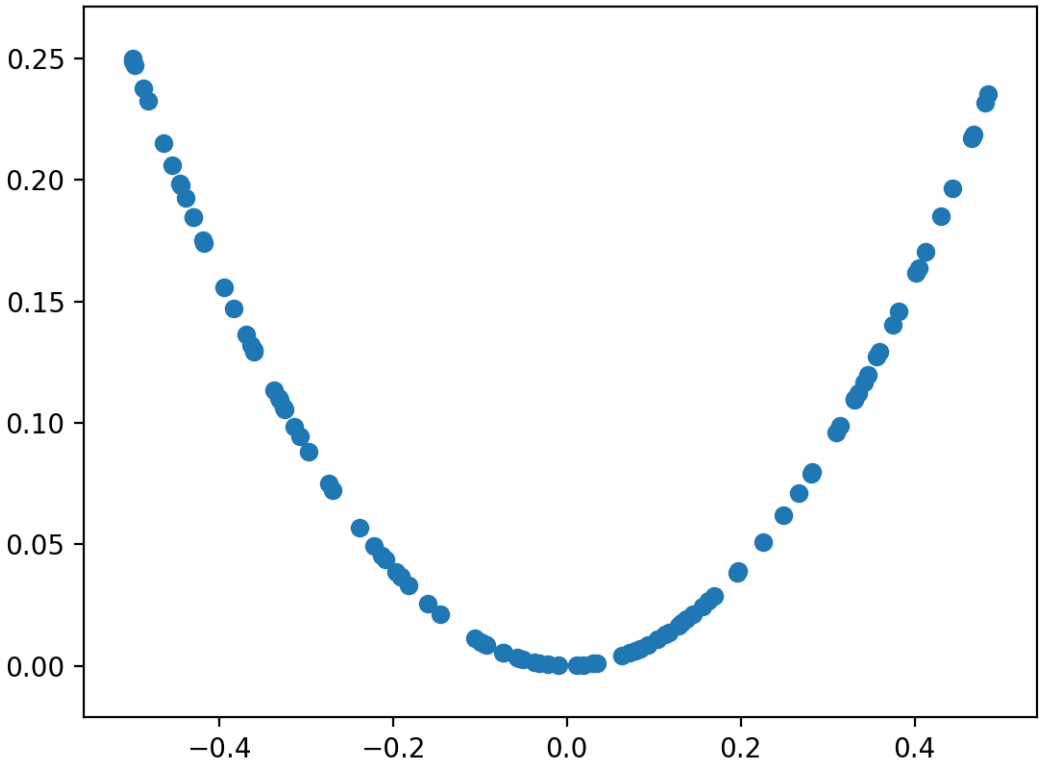

Figure 6.2: Plot of randomly generated sample of inputs vs. calculated outputs for X^2 function.

We can use this function as a starting point for generating real samples for our discriminator function. Specifically, a sample is comprised of a vector with two elements, one for the input and one for the output of our one-dimensional function. We can also imagine how a generator model could generate new samples that we can plot and compare to the expected u-shape of

the X^2 function. Specifically, a generator would output a vector with two elements: one for the input and one for the output of our one-dimensional function.

6.3 Define a Discriminator Model

The next step is to define the discriminator model. The model must take a sample from our problem, such as a vector with two elements, and output a classification prediction as to whether the sample is real or fake. This is a binary classification problem.

- **Inputs**: Sample with two real values.
- **Outputs**: Binary classification, likelihood the sample is real (or fake).

The problem is very simple, meaning that we don't need a complex neural network to model it. The discriminator model will have one hidden layer with 25 nodes and we will use the ReLU activation function and an appropriate weight initialization method called He weight initialization. The output layer will have one node for the binary classification using the sigmoid activation function. The model will minimize the binary cross-entropy loss function, and the Adam version of stochastic gradient descent will be used because it is very effective. The `define_discriminator()` function below defines and returns the discriminator model. The function parameterizes the number of inputs to expect, which defaults to two.

```
# define the standalone discriminator model
def define_discriminator(n_inputs=2):
  model = Sequential()
  model.add(Dense(25, activation='relu', kernel_initializer='he_uniform',
      input_dim=n_inputs))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
```

Listing 6.4: Example of a function for defining the discriminator model.

We can use this function to define the discriminator model and summarize it. The complete example is listed below.

```
# define the discriminator model
from keras.models import Sequential
from keras.layers import Dense
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(n_inputs=2):
  model = Sequential()
  model.add(Dense(25, activation='relu', kernel_initializer='he_uniform',
      input_dim=n_inputs))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model

# define the discriminator model
```

```
model = define_discriminator()
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='discriminator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 6.5: Example of defining and summarizing the discriminator model.

Running the example defines the discriminator model and summarizes it.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 25)                75
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 26
=================================================================
Total params: 101
Trainable params: 101
Non-trainable params: 0
_________________________________________________________________
```

Listing 6.6: Example output from defining and summarizing the discriminator model.

A plot of the model is also created and we can see that the model expects two inputs and will predict a single output.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

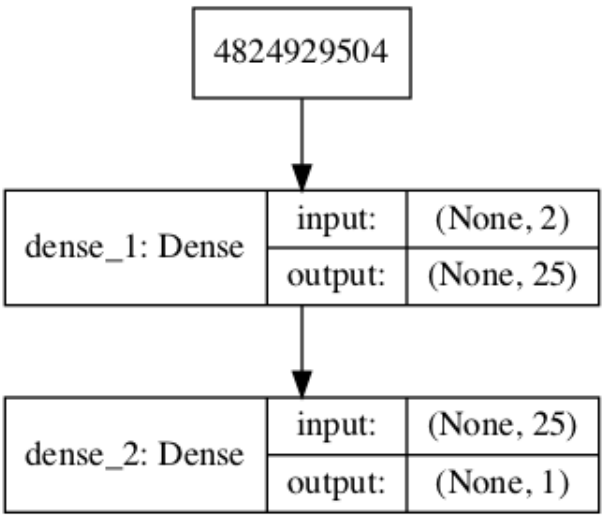

Figure 6.3: Plot of the Discriminator Model in the GAN.

We could start training this model now with real examples with a class label of one and randomly generated samples with a class label of zero. There is no need to do this, but the elements we will develop will be useful later, and it helps to see that the discriminator is just a

normal neural network model. First, we can update our `generate_samples()` function from the prediction section and call it `generate_real_samples()` and have it also return the output class labels for the real samples, specifically, an array of 1 values, where $class = 1$ means real.

```
# generate n real samples with class labels
def generate_real_samples(n):
  # generate inputs in [-0.5, 0.5]
  X1 = rand(n) - 0.5
  # generate outputs X^2
  X2 = X1 * X1
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # generate class labels
  y = ones((n, 1))
  return X, y
```

Listing 6.7: Example of a function for creating random samples for the target function.

Next, we can create a copy of this function for creating fake examples. In this case, we will generate random values in the range -1 and 1 for both elements of a sample. The output class label for all of these examples is 0. This function will act as our fake generator model.

```
# generate n fake samples with class labels
def generate_fake_samples(n):
  # generate inputs in [-1, 1]
  X1 = -1 + rand(n) * 2
  # generate outputs in [-1, 1]
  X2 = -1 + rand(n) * 2
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # generate class labels
  y = zeros((n, 1))
  return X, y
```

Listing 6.8: Example of a function for creating random fake samples for the target function.

Next, we need a function to train and evaluate the discriminator model. This can be achieved by manually enumerating the training epochs and for each epoch generating a half batch of real examples and a half batch of fake examples, and updating the model on each, e.g. one whole batch of examples. The `train()` function could be used, but in this case, we will use the `train_on_batch()` function directly. The model can then be evaluated on the generated examples and we can report the classification accuracy on the real and fake samples. The `train_discriminator()` function below implements this, training the model for 1,000 batches and using 128 samples per batch (64 fake and 64 real).

```
# train the discriminator model
def train_discriminator(model, n_epochs=1000, n_batch=128):
  half_batch = int(n_batch / 2)
  # run epochs manually
  for i in range(n_epochs):
    # generate real examples
    X_real, y_real = generate_real_samples(half_batch)
```

```
    # update model
    model.train_on_batch(X_real, y_real)
    # generate fake examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update model
    model.train_on_batch(X_fake, y_fake)
    # evaluate the model
    _, acc_real = model.evaluate(X_real, y_real, verbose=0)
    _, acc_fake = model.evaluate(X_fake, y_fake, verbose=0)
    print(i, acc_real, acc_fake)
```

Listing 6.9: Example of a function for training the discriminator model.

We can tie all of this together and train the discriminator model on real and fake examples. The complete example is listed below.

```
# define and fit a discriminator model
from numpy import zeros
from numpy import ones
from numpy import hstack
from numpy.random import rand
from keras.models import Sequential
from keras.layers import Dense

# define the standalone discriminator model
def define_discriminator(n_inputs=2):
  model = Sequential()
  model.add(Dense(25, activation='relu', kernel_initializer='he_uniform',
      input_dim=n_inputs))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model

# generate n real samples with class labels
def generate_real_samples(n):
  # generate inputs in [-0.5, 0.5]
  X1 = rand(n) - 0.5
  # generate outputs X^2
  X2 = X1 * X1
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # generate class labels
  y = ones((n, 1))
  return X, y

# generate n fake samples with class labels
def generate_fake_samples(n):
  # generate inputs in [-1, 1]
  X1 = -1 + rand(n) * 2
  # generate outputs in [-1, 1]
  X2 = -1 + rand(n) * 2
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
```

```
  X = hstack((X1, X2))
  # generate class labels
  y = zeros((n, 1))
  return X, y

# train the discriminator model
def train_discriminator(model, n_epochs=1000, n_batch=128):
  half_batch = int(n_batch / 2)
  # run epochs manually
  for i in range(n_epochs):
    # generate real examples
    X_real, y_real = generate_real_samples(half_batch)
    # update model
    model.train_on_batch(X_real, y_real)
    # generate fake examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update model
    model.train_on_batch(X_fake, y_fake)
    # evaluate the model
    _, acc_real = model.evaluate(X_real, y_real, verbose=0)
    _, acc_fake = model.evaluate(X_fake, y_fake, verbose=0)
    print(i, acc_real, acc_fake)

# define the discriminator model
model = define_discriminator()
# fit the model
train_discriminator(model)
```

Listing 6.10: Example of fitting the discriminator model.

Running the example generates real and fake examples and updates the model, then evaluates the model on the same examples and prints the classification accuracy.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the model rapidly learns to correctly identify the real examples with perfect accuracy and is very good at identifying the fake examples with 80% to 90% accuracy.

```
...
995 1.0 0.875
996 1.0 0.921875
997 1.0 0.859375
998 1.0 0.9375
999 1.0 0.8125
```

Listing 6.11: Example output from fitting the discriminator model.

Training the discriminator model is straightforward. The goal is to train a generator model, not a discriminator model, and that is where the complexity of GANs truly lies.

6.4 Define a Generator Model

The next step is to define the generator model. The generator model takes as input a point from the latent space and generates a new sample, e.g. a vector with both the input and output

elements of our function, e.g. x and x^2. A latent variable is a hidden or unobserved variable, and a latent space is a multi-dimensional vector space of these variables. We can define the size of the latent space for our problem and the shape or distribution of variables in the latent space.

This is because the latent space has no meaning until the generator model starts assigning meaning to points in the space as it learns. After training, points in the latent space will correspond to points in the output space, e.g. in the space of generated samples. We will define a small latent space of five dimensions and use the standard approach in the GAN literature of using a Gaussian distribution for each variable in the latent space. We will generate new inputs by drawing random numbers from a standard Gaussian distribution, i.e. mean of zero and a standard deviation of one.

- **Inputs**: Point in latent space, e.g. a five-element vector of Gaussian random numbers.
- **Outputs**: Two-element vector representing a generated sample for our function (x and x^2).

The generator model will be small like the discriminator model. It will have a single `Dense` hidden layer with fifteen nodes and will use the ReLU activation function and He weight initialization. The output layer will have two nodes for the two elements in a generated vector and will use a linear activation function. A linear activation function is used because we know we want the generator to output a vector of real values and the scale will be [-0.5, 0.5] for the first element and about [0.0, 0.25] for the second element.

The model is not compiled. The reason for this is that the generator model is not fit directly. The `define_generator()` function below defines and returns the generator model. The size of the latent dimension is parameterized in case we want to play with it later, and the output shape of the model is also parameterized, matching the function for defining the discriminator model.

```
# define the standalone generator model
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(15, activation='relu', kernel_initializer='he_uniform',
      input_dim=latent_dim))
  model.add(Dense(n_outputs, activation='linear'))
  return model
```

Listing 6.12: Example of a function for defining the generator model.

We can summarize the model to help better understand the input and output shapes. The complete example is listed below.

```
# define the generator model
from keras.models import Sequential
from keras.layers import Dense
from keras.utils.vis_utils import plot_model

# define the standalone generator model
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(15, activation='relu', kernel_initializer='he_uniform',
      input_dim=latent_dim))
  model.add(Dense(n_outputs, activation='linear'))
```

```
  return model

# define the discriminator model
model = define_generator(5)
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='generator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 6.13: Example of defining and summarizing the generator model.

Running the example defines the generator model and summarizes it.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 15)                90
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 32
=================================================================
Total params: 122
Trainable params: 122
Non-trainable params: 0
_________________________________________________________________
```

Listing 6.14: Example output from defining and summarizing the generator model.

A plot of the model is also created and we can see that the model expects a five-element point from the latent space as input and will predict a two-element vector as output.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

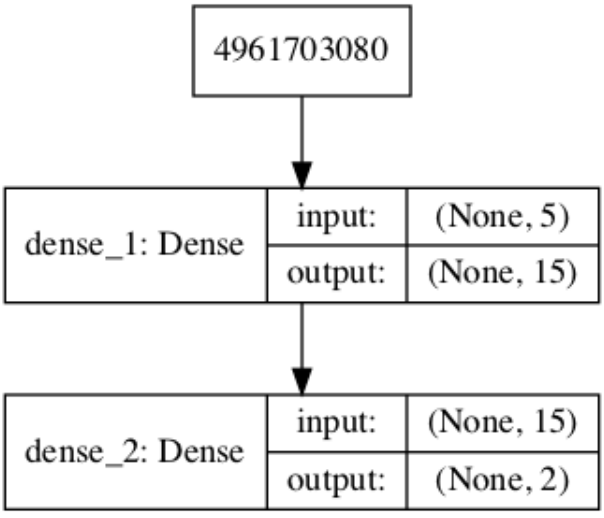

Figure 6.4: Plot of the Generator Model in the GAN.

We can see that the model takes as input a random five-element vector from the latent space and outputs a two-element vector for our one-dimensional function. This model cannot do much

at the moment. Nevertheless, we can demonstrate how to use it to generate samples. This is not needed, but again, some of these elements may be useful later. The first step is to generate new random points in the latent space. We can achieve this by calling the `randn()` NumPy function for generating arrays of random numbers drawn from a standard Gaussian.

The array of random numbers can then be reshaped into samples: that is n rows with five elements per row. The `generate_latent_points()` function below implements this and generates the desired number of points in the latent space that can be used as input to the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n):
  # generate points in the latent space
  x_input = randn(latent_dim * n)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n, latent_dim)
  return x_input
```

Listing 6.15: Example of a function sampling the latent space.

Next, we can use the generated points as input the generator model to generate new samples, then plot the samples. The `generate_fake_samples()` function below implements this, where the defined generator and size of the latent space are passed as arguments, along with the number of points for the model to generate.

```
# use the generator to generate n fake examples and plot the results
def generate_fake_samples(generator, latent_dim, n):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n)
  # predict outputs
  X = generator.predict(x_input)
  # plot the results
  pyplot.scatter(X[:, 0], X[:, 1])
  pyplot.show()
```

Listing 6.16: Example of a function for generating fake examples.

Tying this together, the complete example is listed below.

```
# define and use the generator model
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot

# define the standalone generator model
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(15, activation='relu', kernel_initializer='he_uniform',
      input_dim=latent_dim))
  model.add(Dense(n_outputs, activation='linear'))
  return model

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n):
  # generate points in the latent space
  x_input = randn(latent_dim * n)
```

```
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n, latent_dim)
  return x_input

# use the generator to generate n fake examples and plot the results
def generate_fake_samples(generator, latent_dim, n):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n)
  # predict outputs
  X = generator.predict(x_input)
  # plot the results
  pyplot.scatter(X[:, 0], X[:, 1])
  pyplot.show()

# size of the latent space
latent_dim = 5
# define the discriminator model
model = define_generator(latent_dim)
# generate and plot generated samples
generate_fake_samples(model, latent_dim, 100)
```

Listing 6.17: Example of defining and using the generator model.

Running the example generates 100 random points from the latent space, uses this as input to the generator and generates 100 fake samples from our one-dimensional function domain. As the generator has not been trained, the generated points are complete rubbish, as we expect, but we can imagine that as the model is trained, these points will slowly begin to resemble the target function and its u-shape.

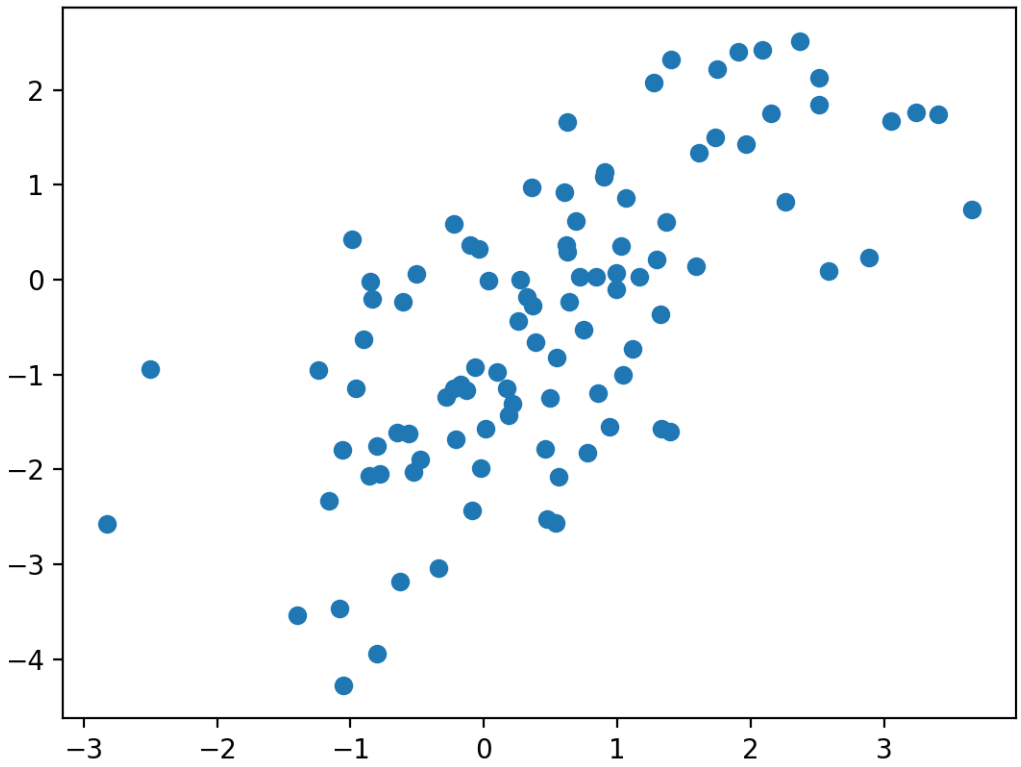

Figure 6.5: Scatter plot of Fake Samples Predicted by the Generator Model.

We have now seen how to define and use the generator model. We will need to use the generator model in this way to create samples for the discriminator to classify. We have not seen how the generator model is trained; that is next.

6.5 Training the Generator Model

The weights in the generator model are updated based on the performance of the discriminator model. When the discriminator is good at detecting fake samples, the generator is updated more (via a larger error gradient), and when the discriminator model is relatively poor or confused when detecting fake samples, the generator model is updated less. This defines the zero-sum or adversarial relationship between these two models. There may be many ways to implement this using the Keras API, but perhaps the simplest approach is to create a new model that subsumes or encapsulates the generator and discriminator models. Specifically, a new GAN model can be defined that stacks the generator and discriminator such that the generator receives as input random points in the latent space, generates samples that are fed into the discriminator model directly, classified, and the output of this larger model can be used to update the model weights of the generator. To be clear, we are not talking about a new third model, just a logical third model that uses the already-defined layers and weights from the standalone generator and discriminator models.

Only the discriminator is concerned with distinguishing between real and fake examples; therefore, the discriminator model can be trained in a standalone manner on examples of each. The generator model is only concerned with the discriminator's performance on fake examples. Therefore, we will mark all of the layers in the discriminator as not trainable when it is part of the GAN model so that they cannot be updated and overtrained on fake examples.

When training the generator via this subsumed GAN model, there is one more important change. The generator wants the discriminator to think that the samples output by the generator are real, not fake. Therefore, when the generator is trained as part of the GAN model, we will mark the generated samples as real ($class = 1$). We can imagine that the discriminator will then classify the generated samples as not real ($class = 0$) or a low probability of being real (0.3 or 0.5). The backpropagation process used to update the model weights will see this as a large error and will update the model weights (i.e. only the weights in the generator) to correct for this error, in turn making the generator better at generating plausible fake samples. Let's make this concrete.

- **Inputs**: Point in latent space, e.g. a five-element vector of Gaussian random numbers.
- **Outputs**: Binary classification, likelihood the sample is real (or fake).

The `define_gan()` function below takes as arguments the already-defined generator and discriminator models and creates the new logical third model subsuming these two models. The weights in the discriminator are marked as not trainable, which only affects the weights as seen by the GAN model and not the standalone discriminator model. The GAN model then uses the same binary cross-entropy loss function as the discriminator and the efficient Adam version of stochastic gradient descent.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam')
  return model
```

Listing 6.18: Example of a function for defining the composite model for training the generator.

Making the discriminator not trainable is a clever trick in the Keras API. The trainable property impacts the model when it is compiled. The discriminator model was compiled with trainable layers, therefore the model weights in those layers will be updated when the standalone model is updated via calls to `train_on_batch()`. The discriminator model was marked as not trainable, added to the GAN model, and compiled. In this model, the model weights of the discriminator model are not trainable and cannot be changed when the GAN model is updated via calls to `train_on_batch()`. The complete example of creating the discriminator, generator, and composite model is listed below.

```
# demonstrate creating the three models in the gan
from keras.models import Sequential
from keras.layers import Dense
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(n_inputs=2):
  model = Sequential()
  model.add(Dense(25, activation='relu', kernel_initializer='he_uniform',
      input_dim=n_inputs))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(15, activation='relu', kernel_initializer='he_uniform',
      input_dim=latent_dim))
  model.add(Dense(n_outputs, activation='linear'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam')
  return model

# size of the latent space
latent_dim = 5
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# summarize gan model
gan_model.summary()
# plot gan model
plot_model(gan_model, to_file='gan_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 6.19: Example of defining and summarizing the composite model for training the generator.

Running the example first creates a summary of the composite model. You might get a `UserWarning` about not calling the `compile()` function that you can safely ignore.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_2 (Sequential) (None, 2)                 122
_________________________________________________________________
sequential_1 (Sequential) (None, 1)                 101
=================================================================
Total params: 223
Trainable params: 122
Non-trainable params: 101
_________________________________________________________________
```

Listing 6.20: Example output from defining and summarizing the composite model for training the generator.

A plot of the model is also created and we can see that the model expects a five-element point in latent space as input and will predict a single output classification label.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

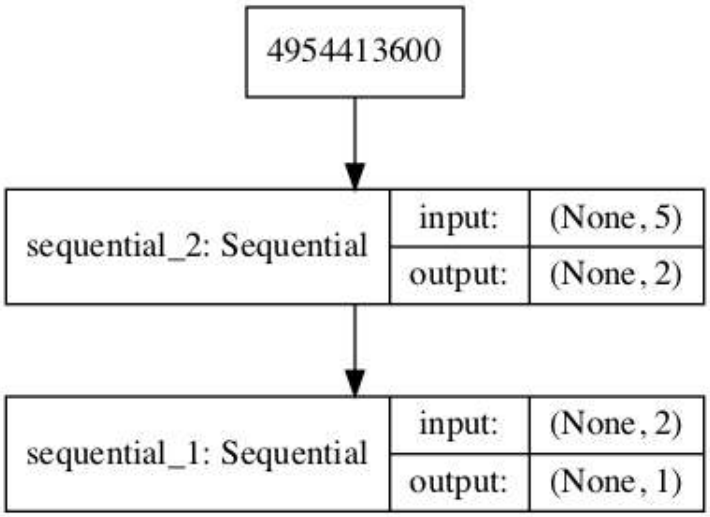

Figure 6.6: Plot of the Composite Generator and Discriminator Model in the GAN.

Training the composite model involves generating a batch-worth of points in the latent space via the `generate_latent_points()` function in the previous section, and $class = 1$ labels and calling the `train_on_batch()` function. The `train_gan()` function below demonstrates this, although it is pretty uninteresting as only the generator will be updated each epoch, leaving the discriminator with default model weights.

```
# train the composite model
def train_gan(gan_model, latent_dim, n_epochs=10000, n_batch=128):
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare points in latent space as input for the generator
```

```
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    gan_model.train_on_batch(x_gan, y_gan)
```

Listing 6.21: Example of a function for training the composite model.

Instead, what is required is that we first update the discriminator model with real and fake samples, then update the generator via the composite model. This requires combining elements from the `train_discriminator()` function defined in the discriminator section and the `train_gan()` function defined above. It also requires that the `generate_fake_samples()` function use the generator model to generate fake samples instead of generating random numbers. The complete train function for updating the discriminator model and the generator (via the composite model) is listed below.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, latent_dim, n_epochs=10000, n_batch=128):
  # determine half the size of one batch, for updating the discriminator
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare real samples
    x_real, y_real = generate_real_samples(half_batch)
    # prepare fake examples
    x_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator
    d_model.train_on_batch(x_real, y_real)
    d_model.train_on_batch(x_fake, y_fake)
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    gan_model.train_on_batch(x_gan, y_gan)
```

Listing 6.22: Example of a function for training the discriminator and generator models.

We almost have everything we need to develop a GAN for our one-dimensional function. One remaining aspect is the evaluation of the model.

6.6 Evaluating the Performance of the GAN

Generally, there are no objective ways to evaluate the performance of a GAN model. In this specific case, we can devise an objective measure for the generated samples as we know the true underlying input domain and target function and can calculate an objective error measure. Nevertheless, we will not calculate this objective error score in this tutorial. Instead, we will use the subjective approach used in most GAN applications. Specifically, we will use the generator to generate new samples and inspect them relative to real samples from the domain. First, we can use the `generate_real_samples()` function developed in the discriminator part above to generate real examples. Creating a scatter plot of these examples will create the familiar u-shape of our target function.

```
# generate n real samples with class labels
def generate_real_samples(n):
  # generate inputs in [-0.5, 0.5]
  X1 = rand(n) - 0.5
  # generate outputs X^2
  X2 = X1 * X1
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # generate class labels
  y = ones((n, 1))
  return X, y
```

Listing 6.23: Example of a function for generating real samples for training.

Next, we can use the generator model to generate the same number of fake samples. This requires first generating the same number of points in the latent space via the `generate_latent_points()` function developed in the generator section above. These can then be passed to the generator model and used to generate samples that can also be plotted on the same scatter plot.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n):
  # generate points in the latent space
  x_input = randn(latent_dim * n)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n, latent_dim)
  return x_input
```

Listing 6.24: Example of a function for generating random points in latent space.

The `generate_fake_samples()` function below generates these fake samples and the associated class label of 0 which will be useful later.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n, 1))
  return X, y
```

Listing 6.25: Example of a function for generating fake samples.

Having both samples plotted on the same graph allows them to be directly compared to see if the same input and output domain are covered and whether the expected shape of the target function has been appropriately captured, at least subjectively. The `summarize_performance()` function below can be called any time during training to create a scatter plot of real and generated points to get an idea of the current capability of the generator model.

```
# plot real and fake points
def summarize_performance(generator, latent_dim, n=100):
  # prepare real samples
  x_real, y_real = generate_real_samples(n)
```

```
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(generator, latent_dim, n)
  # scatter plot real and fake data points
  pyplot.scatter(x_real[:, 0], x_real[:, 1], color='red')
  pyplot.scatter(x_fake[:, 0], x_fake[:, 1], color='blue')
  pyplot.show()
```

Listing 6.26: Example of a function for plotting generated points.

We may also be interested in the performance of the discriminator model at the same time. Specifically, we are interested to know how well the discriminator model can correctly identify real and fake samples. A good generator model should make the discriminator model confused, resulting in a classification accuracy closer to 50% on real and fake examples. We can update the `summarize_performance()` function to also take the discriminator and current epoch number as arguments and report the accuracy on the sample of real and fake examples. It will also generate a plot of synthetic plots and save it to file for later review.

```
# evaluate the discriminator and plot real and fake points
def summarize_performance(epoch, generator, discriminator, latent_dim, n=100):
  # prepare real samples
  x_real, y_real = generate_real_samples(n)
  # evaluate discriminator on real examples
  _, acc_real = discriminator.evaluate(x_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(generator, latent_dim, n)
  # evaluate discriminator on fake examples
  _, acc_fake = discriminator.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print(epoch, acc_real, acc_fake)
  # scatter plot real and fake data points
  pyplot.scatter(x_real[:, 0], x_real[:, 1], color='red')
  pyplot.scatter(x_fake[:, 0], x_fake[:, 1], color='blue')
  # save plot to file
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()
```

Listing 6.27: Example of a function for summarizing the performance of the generator model.

This function can then be called periodically during training. For example, if we choose to train the models for 10,000 iterations, it may be interesting to check-in on the performance of the model every 2,000 iterations. We can achieve this by parameterizing the frequency of the check-in via `n_eval` argument, and calling the `summarize_performance()` function from the `train()` function after the appropriate number of iterations. The updated version of the `train()` function with this change is listed below.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, latent_dim, n_epochs=10000, n_batch=128,
    n_eval=2000):
  # determine half the size of one batch, for updating the discriminator
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare real samples
    x_real, y_real = generate_real_samples(half_batch)
```

```
    # prepare fake examples
    x_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator
    d_model.train_on_batch(x_real, y_real)
    d_model.train_on_batch(x_fake, y_fake)
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    gan_model.train_on_batch(x_gan, y_gan)
    # evaluate the model every n_eval epochs
    if (i+1) % n_eval == 0:
      summarize_performance(i, g_model, d_model, latent_dim)
```

Listing 6.28: Example of a function for training the GAN.

6.7 Complete Example of Training the GAN

We now have everything we need to train and evaluate a GAN on our chosen one-dimensional function. The complete example is listed below.

```
# train a generative adversarial network on a one-dimensional function
from numpy import hstack
from numpy import zeros
from numpy import ones
from numpy.random import rand
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(n_inputs=2):
  model = Sequential()
  model.add(Dense(25, activation='relu', kernel_initializer='he_uniform',
      input_dim=n_inputs))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim, n_outputs=2):
  model = Sequential()
  model.add(Dense(15, activation='relu', kernel_initializer='he_uniform',
      input_dim=latent_dim))
  model.add(Dense(n_outputs, activation='linear'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
```

```
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam')
  return model

# generate n real samples with class labels
def generate_real_samples(n):
  # generate inputs in [-0.5, 0.5]
  X1 = rand(n) - 0.5
  # generate outputs X^2
  X2 = X1 * X1
  # stack arrays
  X1 = X1.reshape(n, 1)
  X2 = X2.reshape(n, 1)
  X = hstack((X1, X2))
  # generate class labels
  y = ones((n, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n):
  # generate points in the latent space
  x_input = randn(latent_dim * n)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n, 1))
  return X, y

# evaluate the discriminator and plot real and fake points
def summarize_performance(epoch, generator, discriminator, latent_dim, n=100):
  # prepare real samples
  x_real, y_real = generate_real_samples(n)
  # evaluate discriminator on real examples
  _, acc_real = discriminator.evaluate(x_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(generator, latent_dim, n)
  # evaluate discriminator on fake examples
  _, acc_fake = discriminator.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print(epoch, acc_real, acc_fake)
  # scatter plot real and fake data points
  pyplot.scatter(x_real[:, 0], x_real[:, 1], color='red')
```

```
  pyplot.scatter(x_fake[:, 0], x_fake[:, 1], color='blue')
  # save plot to file
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()

# train the generator and discriminator
def train(g_model, d_model, gan_model, latent_dim, n_epochs=10000, n_batch=128,
    n_eval=2000):
  # determine half the size of one batch, for updating the discriminator
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare real samples
    x_real, y_real = generate_real_samples(half_batch)
    # prepare fake examples
    x_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator
    d_model.train_on_batch(x_real, y_real)
    d_model.train_on_batch(x_fake, y_fake)
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    gan_model.train_on_batch(x_gan, y_gan)
    # evaluate the model every n_eval epochs
    if (i+1) % n_eval == 0:
      summarize_performance(i, g_model, d_model, latent_dim)

# size of the latent space
latent_dim = 5
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# train model
train(generator, discriminator, gan_model, latent_dim)
```

Listing 6.29: Complete example of training a GAN model on a one-dimensional target function.

Running the example reports model performance every 2,000 training iterations (batches) and creates a plot.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that the training process is relatively unstable. The first column reports the iteration number, the second the classification accuracy of the discriminator for real examples, and the third column the classification accuracy of the discriminator for generated (fake) examples. In this case, we can see that the discriminator remains relatively confused about real examples, and performance on identifying fake examples varies.

```
1999 0.45 1.0
3999 0.45 0.91
5999 0.86 0.16
7999 0.6 0.41
9999 0.15 0.93
```

Listing 6.30: Example output from the complete example of training a GAN model on a one-dimensional target function.

We will omit providing the five created plots here for brevity; instead we will look at only two. The first plot is created after 2,000 iterations and shows real (red) vs. fake (blue) samples. The model performs poorly initially with a cluster of generated points only in the positive input domain, although with the right functional relationship.

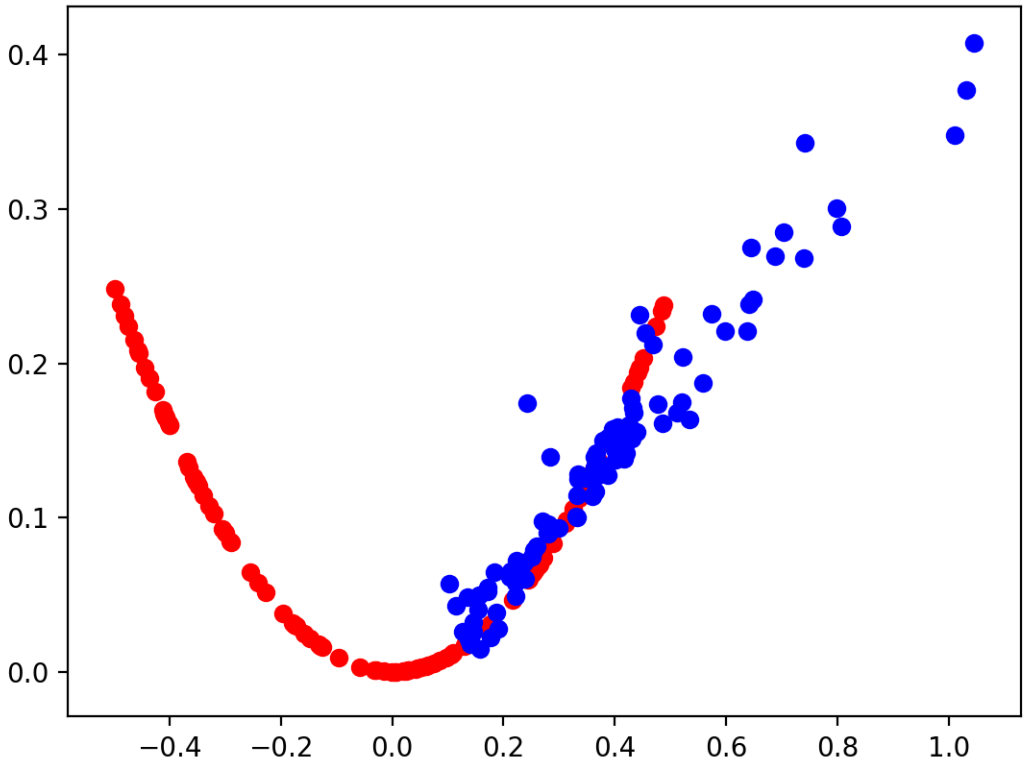

Figure 6.7: Scatter Plot of Real and Generated Examples for the Target Function After 2,000 Iterations.

The second plot shows real (red) vs. fake (blue) after 10,000 iterations. Here we can see that the generator model does a reasonable job of generating plausible samples, with the input values in the right domain between [-0.5 and 0.5] and the output values showing the X^2 relationship, or close to it.

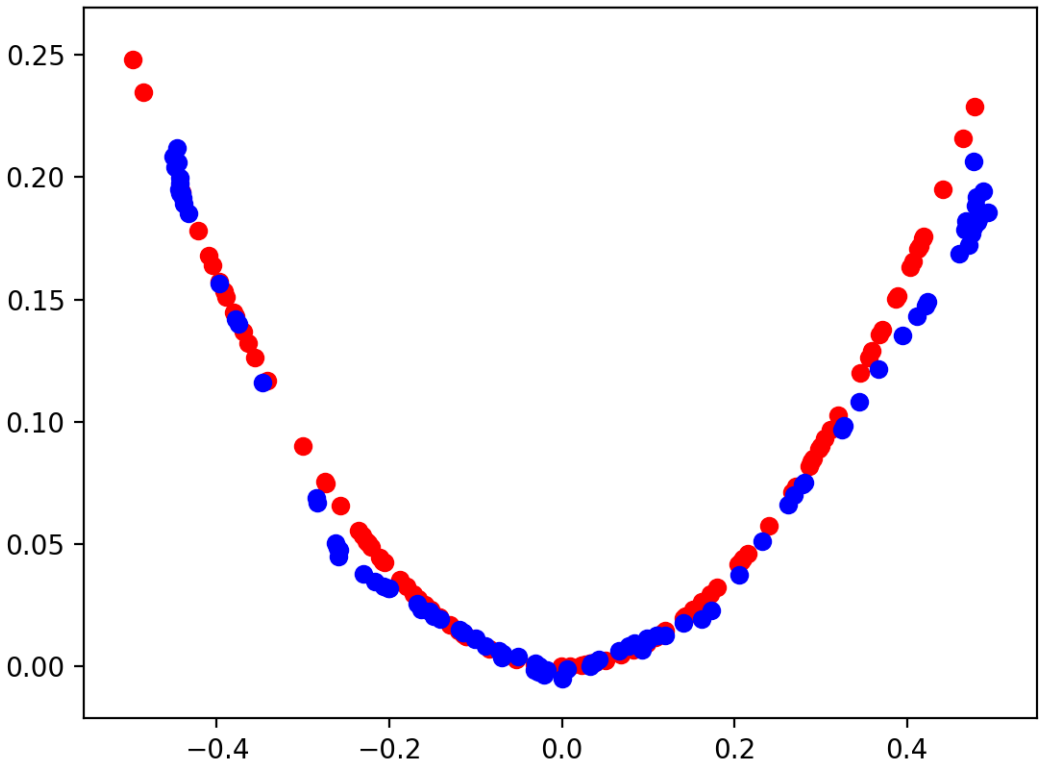

Figure 6.8: Scatter Plot of Real and Generated Examples for the Target Function After 10,000 Iterations.

6.8 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Model Architecture**. Experiment with alternate model architectures for the discriminator and generator, such as more or fewer nodes, layers, and alternate activation functions such as leaky ReLU.
- **Data Scaling**. Experiment with alternate activation functions such as the hyperbolic tangent (tanh) and any required scaling of training data.
- **Alternate Target Function**. Experiment with an alternate target function, such a simple sine wave, Gaussian distribution, a different quadratic, or even a multi-modal polynomial function.

If you explore any of these extensions, I'd love to know.

6.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Keras API.
  https://keras.io/
- How can I "freeze" Keras layers?.
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers
- MatplotLib API.
  https://matplotlib.org/api/
- `numpy.random.rand` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html
- `numpy.random.randn` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html
- `numpy.zeros` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html
- `numpy.ones` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.ones.html
- `numpy.hstack` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.hstack.html

6.10 Summary

In this tutorial, you discovered how to develop a generative adversarial network from scratch for a one-dimensional function. Specifically, you learned:

- The benefit of developing a generative adversarial network from scratch for a simple one-dimensional function.
- How to develop separate discriminator and generator models, as well as a composite model for training the generator via the discriminator's predictive behavior.
- How to subjectively evaluate generated samples in the context of real examples from the problem domain.

6.10.1 Next

In the next tutorial, you will develop a deep convolutional GAN for the MNIST handwritten digit dataset.

Chapter 7

How to Develop a DCGAN for Grayscale Handwritten Digits

Generative Adversarial Networks, or GANs, are an architecture for training generative models, such as deep convolutional neural networks for generating images. Developing a GAN for generating images requires both a discriminator convolutional neural network model for classifying whether a given image is real or generated and a generator model that uses inverse convolutional layers to transform an input to a full two-dimensional image of pixel values.

It can be challenging to understand both how GANs work and how deep convolutional neural network models can be trained in a GAN architecture for image generation. A good starting point for beginners is to practice developing and using GANs on standard image datasets used in the field of computer vision, such as the MNIST handwritten digit dataset. Using small and well-understood datasets means that smaller models can be developed and trained quickly, allowing the focus to be put on the model architecture and image generation process itself. In this tutorial, you will discover how to develop a generative adversarial network with deep convolutional networks for generating handwritten digits. After completing this tutorial, you will know:

- How to define and train the standalone discriminator model for learning the difference between real and fake images.
- How to define the standalone generator model and train the composite generator and discriminator model.
- How to evaluate the performance of the GAN and use the final standalone generator model to generate new images.

Let's get started.

7.1 Tutorial Overview

This tutorial is divided into seven parts; they are:

1. MNIST Handwritten Digit Dataset
2. How to Define and Train the Discriminator Model

3. How to Define and Use the Generator Model
4. How to Train the Generator Model
5. How to Evaluate GAN Model Performance
6. Complete Example of GAN for MNIST
7. How to Use the Final Generator Model

7.2 MNIST Handwritten Digit Dataset

The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It is a dataset of 70,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9. The task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively. Keras provides access to the MNIST dataset via the `mnist.load_dataset()` function. It returns two tuples, one with the input and output elements for the standard training dataset, and another with the input and output elements for the standard test dataset. The example below loads the dataset and summarizes the shape of the loaded dataset.

Note: the first time you load the dataset, Keras will automatically download a compressed version of the images and save them under your home directory in `~/.keras/datasets/`. The download is fast as the dataset is only about eleven megabytes in its compressed form.

```
# example of loading the mnist dataset
from keras.datasets.mnist import load_data
# load the images into memory
(trainX, trainy), (testX, testy) = load_data()
# summarize the shape of the dataset
print('Train', trainX.shape, trainy.shape)
print('Test', testX.shape, testy.shape)
```

Listing 7.1: Example of loading and summarizing the MNIST dataset.

Running the example loads the dataset and prints the shape of the input and output components of the train and test splits of images. We can see that there are 60K examples in the training set and 10K in the test set and that each image is a square of 28 by 28 pixels.

```
Train (60000, 28, 28) (60000,)
Test (10000, 28, 28) (10000,)
```

Listing 7.2: Example output from loading and summarizing the MNIST dataset.

The images are grayscale with a black background (0 pixel value) and the handwritten digits in white (pixel values near 255). This means if the images were plotted, they would be mostly black with a white digit in the middle. We can plot some of the images from the training dataset using the Matplotlib library using the `imshow()` function and specify the color map via the `cmap` argument as `'gray'` to show the pixel values correctly.

```
...
# plot raw pixel data
pyplot.imshow(trainX[i], cmap='gray')
```

Listing 7.3: Example of plotting a single image using the gray color map.

Alternately, the images are easier to review when we reverse the colors and plot the background as white and the handwritten digits in black. They are easier to view as most of the image is now white with the area of interest in black. This can be achieved using a reverse grayscale color map, as follows:

```
...
# plot raw pixel data
pyplot.imshow(trainX[i], cmap='gray_r')
```

Listing 7.4: Example of plotting a single image using the reverse gray color map.

The example below plots the first 25 images from the training dataset in a 5 by 5 square.

```
# example of loading the mnist dataset
from keras.datasets.mnist import load_data
from matplotlib import pyplot
# load the images into memory
(trainX, trainy), (testX, testy) = load_data()
# plot images from the training dataset
for i in range(25):
  # define subplot
  pyplot.subplot(5, 5, 1 + i)
  # turn off axis
  pyplot.axis('off')
  # plot raw pixel data
  pyplot.imshow(trainX[i], cmap='gray_r')
pyplot.show()
```

Listing 7.5: Example of plotting images from the MNIST dataset.

Running the example creates a plot of 25 images from the MNIST training dataset, arranged in a 5×5 square.

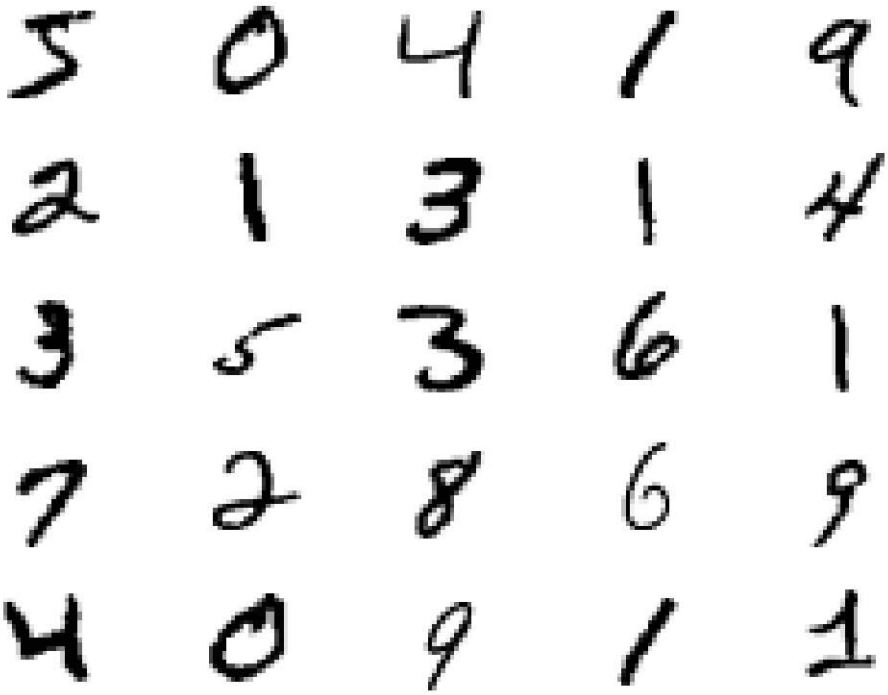

Figure 7.1: Plot of the First 25 Handwritten Digits From the MNIST Dataset.

We will use the images in the training dataset as the basis for training a Generative Adversarial Network. Specifically, the generator model will learn how to generate new plausible handwritten digits between 0 and 9, using a discriminator that will try to distinguish between real images from the MNIST training dataset and new images output by the generator model. This is a relatively simple problem that does not require a sophisticated generator or discriminator model, although it does require the generation of a grayscale output image.

7.3 How to Define and Train the Discriminator Model

The first step is to define the discriminator model. The model must take a sample image from our dataset as input and output a classification prediction as to whether the sample is real or fake. This is a binary classification problem:

- **Inputs**: Image with one channel and 28×28 pixels in size.
- **Outputs**: Binary classification, likelihood the sample is real (or fake).

The discriminator model has two convolutional layers with 64 filters each, a small kernel size of 3, and larger than normal stride of 2. The model has no pooling layers and a single node in the output layer with the sigmoid activation function to predict whether the input sample is real

or fake. The model is trained to minimize the binary cross-entropy loss function, appropriate for binary classification. We will use some best practices in defining the discriminator model, such as the use of `LeakyReLU` instead of ReLU, using `Dropout`, and using the Adam version of stochastic gradient descent with a learning rate of 0.0002 and a momentum of 0.5. The function `define_discriminator()` below defines the discriminator model and parametrizes the size of the input image.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

Listing 7.6: Example of a function for defining the discriminator model.

We can use this function to define the discriminator model and summarize it. The complete example is listed below.

```
# example of defining the discriminator model
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define model
model = define_discriminator()
```

```
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='discriminator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 7.7: Example of defining and summarizing the discriminator model.

Running the example first summarizes the model architecture, showing the input and output from each layer. We can see that the aggressive 2×2 stride acts to downsample the input image, first from 28×28 to 14×14, then to 7×7, before the model makes an output prediction. This pattern is by design as we do not use pooling layers and use the large stride as to achieve a similar downsampling effect. We will see a similar pattern, but in reverse, in the generator model in the next section.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 14, 14, 64)        640
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 3137
=================================================================
Total params: 40,705
Trainable params: 40,705
Non-trainable params: 0
_________________________________________________________________
```

Listing 7.8: Example output from defining and summarizing the discriminator model.

A plot of the model is also created and we can see that the model expects two inputs and will predict a single output.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

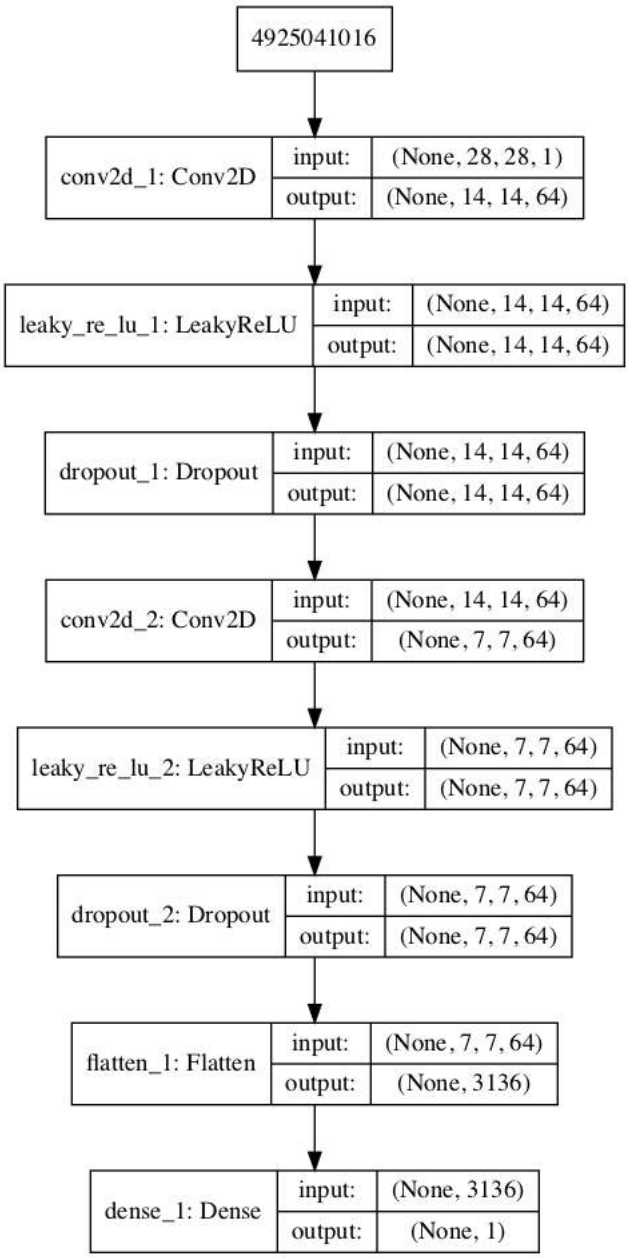

Figure 7.2: Plot of the Discriminator Model in the MNIST GAN.

We could start training this model now with real examples with a class label of one, and randomly generated samples with a class label of zero. The development of these elements will be useful later, and it helps to see that the discriminator is just a normal neural network model for binary classification. First, we need a function to load and prepare the dataset of real images. We will use the `mnist.load_data()` function to load the MNIST dataset and just use the input part of the training dataset as the real images.

```
...
# load mnist dataset
(trainX, _), (_, _) = load_data()
```

Listing 7.9: Example of loading the MNIST training dataset.

The images are 2D arrays of pixels and convolutional neural networks expect 3D arrays of images as input, where each image has one or more channels We must update the images to have an additional dimension for the grayscale channel. We can do this using the `expand_dims()` NumPy function and specify the final dimension for the channels-last image format.

```
...
# expand to 3d, e.g. add channels dimension
X = expand_dims(trainX, axis=-1)
```

Listing 7.10: Example of adding a channels dimension to the dataset.

Finally, we must scale the pixel values from the range of unsigned integers in [0,255] to the normalized range of [0,1]. It is best practice to use the range [-1,1], but in this case the range [0,1] works just fine.

```
# convert from unsigned ints to floats
X = X.astype('float32')
# scale from [0,255] to [0,1]
X = X / 255.0
```

Listing 7.11: Example of normalizing pixel values.

The `load_real_samples()` function below implements this.

```
# load and prepare mnist training images
def load_real_samples():
  # load mnist dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels dimension
  X = expand_dims(trainX, axis=-1)
  # convert from unsigned ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [0,1]
  X = X / 255.0
  return X
```

Listing 7.12: Example of a function for loading and preparing the MNIST training dataset.

The model will be updated in batches, specifically with a collection of real samples and a collection of generated samples. On training, an epoch is defined as one pass through the entire training dataset. We could systematically enumerate all samples in the training dataset, and that is a good approach, but good training via stochastic gradient descent requires that the training dataset be shuffled prior to each epoch. A simpler approach is to select random samples of images from the training dataset. The `generate_real_samples()` function below will take the training dataset as an argument and will select a random subsample of images; it will also return class labels for the sample, specifically a class label of 1, to indicate real images.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y
```

Listing 7.13: Example of a function for selecting a sample of real images.

Now, we need a source of fake images. We don't have a generator model yet, so instead, we can generate images comprised of random pixel values, specifically random pixel values in the range [0,1] like our scaled real images. The `generate_fake_samples()` function below implements this behavior and generates images of random pixel values and their associated class label of 0, for fake.

```
# generate n fake samples with class labels
def generate_fake_samples(n_samples):
  # generate uniform random numbers in [0,1]
  X = rand(28 * 28 * n_samples)
  # reshape into a batch of grayscale images
  X = X.reshape((n_samples, 28, 28, 1))
  # generate 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y
```

Listing 7.14: Example of a function for generating random fake images.

Finally, we need to train the discriminator model. This involves repeatedly retrieving samples of real images and samples of generated images and updating the model for a fixed number of iterations. We will ignore the idea of epochs for now (e.g. complete passes through the training dataset) and fit the discriminator model for a fixed number of batches. The model will learn to discriminate between real and fake (randomly generated) images rapidly, therefore, not many batches will be required before it learns to discriminate perfectly.

The `train_discriminator()` function implements this, using a batch size of 256 images where 128 are real and 128 are fake each iteration. We update the discriminator separately for real and fake examples so that we can calculate the accuracy of the model on each sample prior to the update. This gives insight into how the discriminator model is performing over time.

```
# train the discriminator model
def train_discriminator(model, dataset, n_iter=100, n_batch=256):
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_iter):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator on real samples
    _, real_acc = model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update discriminator on fake samples
    _, fake_acc = model.train_on_batch(X_fake, y_fake)
    # summarize performance
    print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))
```

Listing 7.15: Example of a function for training the discriminator model.

Tying all of this together, the complete example of training an instance of the discriminator model on real and generated (fake) images is listed below.

```
# example of training the discriminator model on real and random mnist images
from numpy import expand_dims
```

```
from numpy import ones
from numpy import zeros
from numpy.random import rand
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LeakyReLU

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# load and prepare mnist training images
def load_real_samples():
  # load mnist dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels dimension
  X = expand_dims(trainX, axis=-1)
  # convert from unsigned ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [0,1]
  X = X / 255.0
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y

# generate n fake samples with class labels
def generate_fake_samples(n_samples):
  # generate uniform random numbers in [0,1]
  X = rand(28 * 28 * n_samples)
  # reshape into a batch of grayscale images
```

```
  X = X.reshape((n_samples, 28, 28, 1))
  # generate 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# train the discriminator model
def train_discriminator(model, dataset, n_iter=100, n_batch=256):
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_iter):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator on real samples
    _, real_acc = model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update discriminator on fake samples
    _, fake_acc = model.train_on_batch(X_fake, y_fake)
    # summarize performance
    print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))

# define the discriminator model
model = define_discriminator()
# load image data
dataset = load_real_samples()
# fit the model
train_discriminator(model, dataset)
```

Listing 7.16: Example of defining and training the discriminator model.

Running the example first defines the model, loads the MNIST dataset, then trains the discriminator model.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the discriminator model learns to tell the difference between real and generated MNIST images very quickly, in about 50 batches.

```
...
>96 real=100% fake=100%
>97 real=100% fake=100%
>98 real=100% fake=100%
>99 real=100% fake=100%
>100 real=100% fake=100%
```

Listing 7.17: Example output from defining and training the discriminator model.

Now that we know how to define and train the discriminator model, we need to look at developing the generator model.

7.4 How to Define and Use the Generator Model

The generator model is responsible for creating new, fake but plausible images of handwritten digits. It does this by taking a point from the latent space as input and outputting a square

grayscale image. The latent space is an arbitrarily defined vector space of Gaussian-distributed values, e.g. 100 dimensions. It has no meaning, but by drawing points from this space randomly and providing them to the generator model during training, the generator model will assign meaning to the latent points. At the end of training, the latent vector space represents a compressed representation of the output space, MNIST images, that only the generator knows how to turn into plausible MNIST images.

- **Inputs**: Point in latent space, e.g. a 100 element vector of Gaussian random numbers.
- **Outputs**: Two-dimensional square grayscale image of 28×28 pixels with pixel values in [0,1].

We don't have to use a 100 element vector as input; it is a round number and widely used, but I would expect that 10, 50, or 500 would work just as well. Developing a generator model requires that we transform a vector from the latent space with, 100 dimensions to a 2D array with 28×28 or 784 values. There are a number of ways to achieve this but there is one approach that has proven effective at deep convolutional generative adversarial networks. It involves two main elements. The first is a `Dense` layer as the first hidden layer that has enough nodes to represent a low-resolution version of the output image. Specifically, an image half the size (one quarter the area) of the output image would be 14×14 or 196 nodes, and an image one quarter the size (one eighth the area) would be 7×7 or 49 nodes.

We don't just want one low-resolution version of the image; we want many parallel versions or interpretations of the input. This is a pattern in convolutional neural networks where we have many parallel filters resulting in multiple parallel activation maps, called feature maps, with different interpretations of the input. We want the same thing in reverse: many parallel versions of our output with different learned features that can be collapsed in the output layer into a final image. The model needs space to invent, create, or generate. Therefore, the first hidden layer, the `Dense` layer needs enough nodes for multiple low-resolution versions of our output image, such as 128.

```
...
# foundation for 7x7 image
model.add(Dense(128 * 7 * 7, input_dim=100))
```

Listing 7.18: Example of defining the base activations in the generator model.

The activations from these nodes can then be reshaped into something image-like to pass into a convolutional layer, such as 128 different 7×7 feature maps.

```
...
model.add(Reshape((7, 7, 128)))
```

Listing 7.19: Example of reshaping the activations into a suitable shape for a `Conv2D` layer.

The next major architectural innovation involves upsampling the low-resolution image to a higher resolution version of the image. There are two common ways to do this upsampling process, sometimes called deconvolution. One way is to use an `UpSampling2D` layer (like a reverse pooling layer) followed by a normal `Conv2D` layer. The other and perhaps more modern way is to combine these two operations into a single layer, called a `Conv2DTranspose`. We will use this latter approach for our generator.

The `Conv2DTranspose` layer can be configured with a stride of (2×2) that will quadruple the area of the input feature maps (double their width and height dimensions). It is also good practice to use a kernel size that is a factor of the stride (e.g. double) to avoid a checkerboard pattern that can be observed when upsampling (for more on upsampling layers, see Chapter 3).

```
...
# upsample to 14x14
model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
```

Listing 7.20: Example of defining an upsampling layer.

This can be repeated to arrive at our 28×28 output image. Again, we will use the `LeakyReLU` activation with a default slope of 0.2, reported as a best practice when training GAN models. The output layer of the model is a `Conv2D` with one filter and a kernel size of 7×7 and `'same'` padding, designed to create a single feature map and preserve its dimensions at 28×28 pixels. A sigmoid activation is used to ensure output values are in the desired range of [0,1]. The `define_generator()` function below implements this and defines the generator model. The generator model is not compiled and does not specify a loss function or optimization algorithm. This is because the generator is not trained directly. We will learn more about this in the next section.

```
# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
  return model
```

Listing 7.21: Example of a function for defining the generator model.

We can summarize the model to help better understand the input and output shapes. The complete example is listed below.

```
# example of defining the generator model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
```

```
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
  return model

# define the size of the latent space
latent_dim = 100
# define the generator model
model = define_generator(latent_dim)
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='generator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 7.22: Example of defining and summarizing the generator model.

Running the example summarizes the layers of the model and their output shape. We can see that, as designed, the first hidden layer has 6,272 parameters or $128 \times 7 \times 7$, the activations of which are reshaped into 128 7×7 feature maps. The feature maps are then upscaled via the two `Conv2DTranspose` layers to the desired output shape of 28×28, until the output layer, where a single activation map is output.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 6272)              633472
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 6272)              0
_________________________________________________________________
reshape_1 (Reshape)          (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 14, 14, 128)       262272
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 28, 28, 128)       262272
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 28, 28, 128)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 1)         6273
=================================================================
Total params: 1,164,289
Trainable params: 1,164,289
Non-trainable params: 0
_________________________________________________________________
```

Listing 7.23: Example output from defining and summarizing the generator model.

A plot of the model is also created and we can see that the model expects a 100-element vector from the latent space as input and will generate an image as output.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

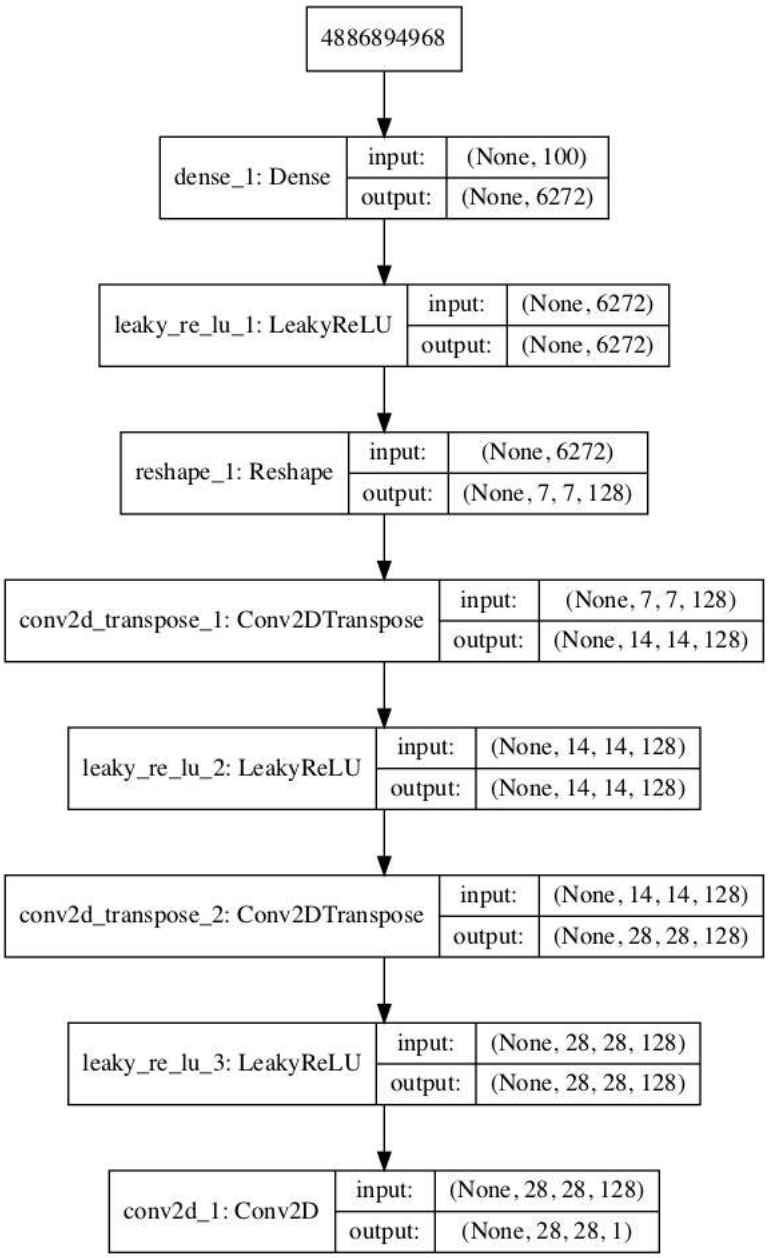

Figure 7.3: Plot of the Generator Model in the MNIST GAN.

This model cannot do much at the moment. Nevertheless, we can demonstrate how to use it to generate samples. This is a helpful demonstration to understand the generator as just another model, and some of these elements will be useful later. The first step is to draw new

points from the latent space. We can achieve this by calling the `randn()` NumPy function for generating arrays of random numbers drawn from a standard Gaussian. The array of random numbers can then be reshaped into samples, that is n rows with 100 elements per row. The `generate_latent_points()` function below implements this and generates the desired number of points in the latent space that can be used as input to the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

Listing 7.24: Example of a function for generating random points in the latent space.

Next, we can use the generated points as input to the generator model to generate new samples, then plot the samples. We can update the `generate_fake_samples()` function from the previous section to take the generator model as an argument and use it to generate the desired number of samples by first calling the `generate_latent_points()` function to generate the required number of points in latent space as input to the model. The updated `generate_fake_samples()` function is listed below and returns both the generated samples and the associated class labels.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y
```

Listing 7.25: Example of a function for generating synthetic images using the generator model.

We can then plot the generated samples as we did the real MNIST examples in the first section by calling the `imshow()` function with the reversed grayscale color map. The complete example of generating new MNIST images with the untrained generator model is listed below.

```
# example of defining and using the generator model
from numpy import zeros
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from matplotlib import pyplot

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
```

```
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
  return model

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# size of the latent space
latent_dim = 100
# define the discriminator model
model = define_generator(latent_dim)
# generate samples
n_samples = 25
X, _ = generate_fake_samples(model, latent_dim, n_samples)
# plot the generated samples
for i in range(n_samples):
  # define subplot
  pyplot.subplot(5, 5, 1 + i)
  # turn off axis labels
  pyplot.axis('off')
  # plot single image
  pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
# show the figure
pyplot.show()
```

Listing 7.26: Example of using the untrained generator to output random images.

Running the example generates 25 examples of fake MNIST images and visualizes them on a single plot of 5 by 5 images. As the model is not trained, the generated images are completely random pixel values in [0, 1].

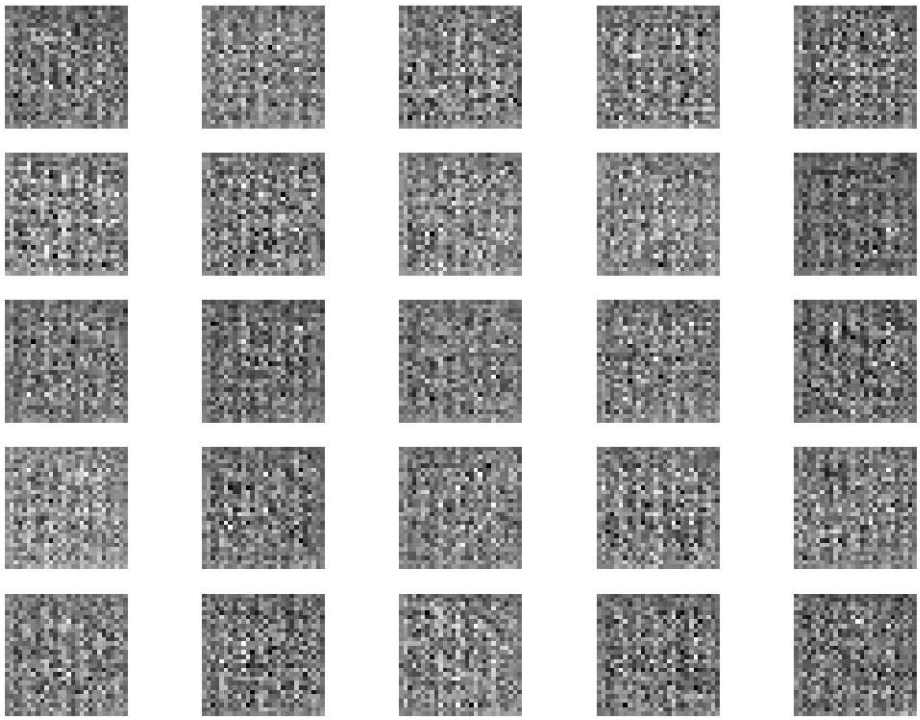

Figure 7.4: Example of 25 MNIST Images Output by the Untrained Generator Model.

Now that we know how to define and use the generator model, the next step is to train the model.

7.5 How to Train the Generator Model

The weights in the generator model are updated based on the performance of the discriminator model. When the discriminator is good at detecting fake samples, the generator is updated more, and when the discriminator model is relatively poor or confused when detecting fake samples, the generator model is updated less. This defines the zero-sum or adversarial relationship between these two models. There may be many ways to implement this using the Keras API, but perhaps the simplest approach is to create a new model that combines the generator and discriminator models. Specifically, a new GAN model can be defined that stacks the generator and discriminator such that the generator receives as input random points in the latent space and generates samples that are fed into the discriminator model directly, classified, and the output of this larger model can be used to update the model weights of the generator.

To be clear, we are not talking about a new third model, just a new logical model that uses the already-defined layers and weights from the standalone generator and discriminator models. Only the discriminator is concerned with distinguishing between real and fake examples, therefore the discriminator model can be trained in a standalone manner on examples of each, as

we did in the section on the discriminator model above. The generator model is only concerned with the discriminator's performance on fake examples. Therefore, we will mark all of the layers in the discriminator as not trainable when it is part of the GAN model so that they cannot be updated and overtrained on fake examples. When training the generator via this logical GAN model, there is one more important change. We want the discriminator to think that the samples output by the generator are real, not fake. Therefore, when the generator is trained as part of the GAN model, we will mark the generated samples as real ($class = 1$).

Why would we want to do this? We can imagine that the discriminator will then classify the generated samples as not real ($class = 0$) or a low probability of being real (0.3 or 0.5). The backpropagation process used to update the model weights will see this as a large error and will update the model weights (i.e. only the weights in the generator) to correct for this error, in turn making the generator better at generating good fake samples. Let's make this concrete.

- **Inputs**: Point in latent space, e.g. a 100 element vector of Gaussian random numbers.
- **Outputs**: Binary classification, likelihood the sample is real (or fake).

The `define_gan()` function below takes as arguments the already-defined generator and discriminator models and creates the new logical third model subsuming these two models. The weights in the discriminator are marked as not trainable, which only affects the weights as seen by the GAN model and not the standalone discriminator model. The GAN model then uses the same binary cross-entropy loss function as the discriminator and the efficient Adam version of stochastic gradient descent with the learning rate of 0.0002 and momentum 0.5, recommended when training deep convolutional GANs.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model
```

Listing 7.27: Example of a function for defining the composite model to update the generator model via the discriminator model.

Making the discriminator not trainable is a clever trick in the Keras API. The trainable property impacts the model after it is compiled. The discriminator model was compiled with trainable layers, therefore the model weights in those layers will be updated when the standalone model is updated via calls to the `train_on_batch()` function. The discriminator model was then marked as not trainable, added to the GAN model, and compiled. In this model, the model weights of the discriminator model are not trainable and cannot be changed when the GAN model is updated via calls to the `train_on_batch()` function. This change in the trainable property does not impact the training of standalone discriminator model. The complete example of creating the discriminator, generator, and composite model is listed below.

```
# demonstrate creating the three models in the gan
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
```

```
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# summarize gan model
gan_model.summary()
# plot gan model
plot_model(gan_model, to_file='gan_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 7.28: Example of defining and summarizing the composite model.

Running the example first creates a summary of the composite model. We can see that the model expects MNIST images as input and predicts a single value as output.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_2 (Sequential) (None, 28, 28, 1)       1164289
_________________________________________________________________
sequential_1 (Sequential) (None, 1)               40705
=================================================================
Total params: 1,204,994
Trainable params: 1,164,289
Non-trainable params: 40,705
_________________________________________________________________
```

Listing 7.29: Example output from defining and summarizing the composite model.

A plot of the model is also created and we can see that the model expects a 100-element point in latent space as input and will predict a single output classification label.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

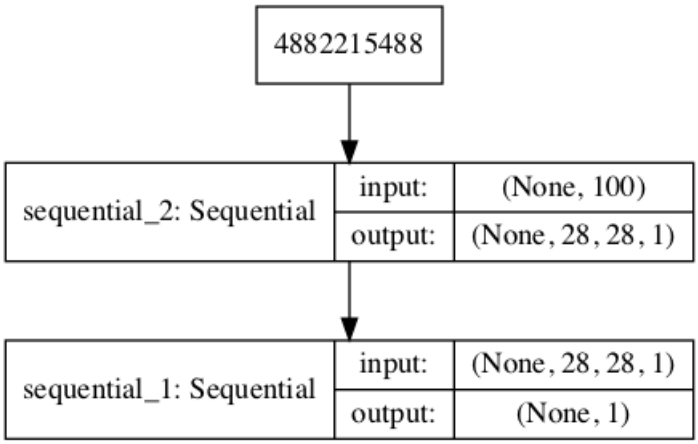

Figure 7.5: Plot of the Composite Generator and Discriminator Model in the MNIST GAN.

Training the composite model involves generating a batch worth of points in the latent space via the `generate_latent_points()` function in the previous section, and $class = 1$ labels and calling the `train_on_batch()` function. The `train_gan()` function below demonstrates this, although it is pretty simple as only the generator will be updated each epoch, leaving the discriminator with default model weights.

```
# train the composite model
def train_gan(gan_model, latent_dim, n_epochs=100, n_batch=256):
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    gan_model.train_on_batch(x_gan, y_gan)
```

Listing 7.30: Example of a function for training the composite model.

Instead, what is required is that we first update the discriminator model with real and fake samples, then update the generator via the composite model. This requires combining elements from the `train_discriminator()` function defined in the discriminator section above and the `train_gan()` function defined above. It also requires that we enumerate over both epochs and batches within in an epoch. The complete train function for updating the discriminator model and the generator (via the composite model) is listed below. There are a few things to note in this model training function. First, the number of batches within an epoch is defined by how many times the batch size divides into the training dataset. We have a dataset size of 60K samples and a batch size of 256, so with rounding down, there are $\frac{60000}{256}$ or 234 batches per epoch.

The discriminator model is updated once per batch by combining one half a batch (128) of fake and real (128) examples into a single batch via the `vstack()` NumPy function. You could update the discriminator with each half batch separately (recommended for more complex datasets) but combining the samples into a single batch will be faster over a long run, especially when training on GPU hardware. Finally, we report the loss for each batch. It is critical to keep an eye on the loss over batches. The reason for this is that a crash in the discriminator

loss indicates that the generator model has started generating rubbish examples that the discriminator can easily discriminate. Monitor the discriminator loss and expect it to hover around 0.5 to 0.8 per batch on this dataset. The generator loss is less critical and may hover between 0.5 and 2 or higher on this dataset. A clever programmer might even attempt to detect the crashing loss of the discriminator, halt, and then restart the training process.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # create training set for the discriminator
      X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))
      # update discriminator model weights
      d_loss, _ = d_model.train_on_batch(X, y)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d=%.3f, g=%.3f' % (i+1, j+1, bat_per_epo, d_loss, g_loss))
```

Listing 7.31: Example of a function for training the GAN models.

We almost have everything we need to develop a GAN for the MNIST handwritten digits dataset. One remaining aspect is the evaluation of the model.

7.6 How to Evaluate GAN Model Performance

Generally, there are no objective ways to evaluate the performance of a GAN model. We cannot calculate this objective error score for generated images. It might be possible in the case of MNIST images because the images are so well constrained, but in general, it is not possible (yet). Instead, images must be subjectively evaluated for quality by a human operator. This means that we cannot know when to stop training without looking at examples of generated images. In turn, the adversarial nature of the training process means that the generator is changing after every batch, meaning that once *good enough* images can be generated, the subjective quality of the images may then begin to vary, improve, or even degrade with subsequent updates. There are three ways to handle this complex training situation.

1. Periodically evaluate the classification accuracy of the discriminator on real and fake images.

2. Periodically generate many images and save them to file for subjective review.

3. Periodically save the generator model.

All three of these actions can be performed at the same time for a given training epoch, such as every five or 10 training epochs. The result will be a saved generator model for which we have a way of subjectively assessing the quality of its output and objectively knowing how well the discriminator was fooled at the time the model was saved. Training the GAN over many epochs, such as hundreds or thousands of epochs, will result in many snapshots of the model that can be inspected and from which specific outputs and models can be cherry-picked for later use.

First, we can define a function called `summarize_performance()` that will summarize the performance of the discriminator model. It does this by retrieving a sample of real MNIST images, as well as generating the same number of fake MNIST images with the generator model, then evaluating the classification accuracy of the discriminator model on each sample and reporting these scores.

```
# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
```

Listing 7.32: Example of a function for summarizing the performance of the models.

This function can be called from the `train()` function based on the current epoch number, such as every 10 epochs.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
  ...
  # evaluate the model performance, sometimes
  if (i+1) % 10 == 0:
    summarize_performance(i, g_model, d_model, dataset, latent_dim)
```

Listing 7.33: Example of how model performance can be summarized from the train function.

Next, we can update the `summarize_performance()` function to both save the model and to create and save a plot generated examples. The generator model can be saved by calling the `save()` function on the generator model and providing a unique filename based on the training epoch number.

```
...
# save the generator model tile file
filename = 'generator_model_%03d.h5' % (epoch + 1)
g_model.save(filename)
```

Listing 7.34: Example of saving the generator model.

We can develop a function to create a plot of the generated samples. As we are evaluating the discriminator on 100 generated MNIST images, we can plot all 100 images as a 10 by 10 grid. The `save_plot()` function below implements this, again saving the resulting plot with a unique filename based on the epoch number.

```
# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, epoch, n=10):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()
```

Listing 7.35: Example of a function for saving a plot of generated images and the generator model.

The updated `summarize_performance()` function with these additions is listed below.

```
# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
  # save plot
  save_plot(x_fake, epoch)
  # save the generator model tile file
  filename = 'generator_model_%03d.h5' % (epoch + 1)
  g_model.save(filename)
```

Listing 7.36: Example of an updated function for summarizing the performance of the GAN.

7.7 Complete Example of GAN for MNIST

We now have everything we need to train and evaluate a GAN on the MNIST handwritten digit dataset. The complete example is listed below.

```
# example of training a gan on mnist
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy import vstack
```

```
from numpy.random import randn
from numpy.random import randint
from tensorflow.keras.datasets.mnist import load_data
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Reshape
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import Conv2DTranspose
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.layers import Dropout
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
```

```
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# load and prepare mnist training images
def load_real_samples():
  # load mnist dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels dimension
  X = expand_dims(trainX, axis=-1)
  # convert from unsigned ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [0,1]
  X = X / 255.0
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, epoch, n=10):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
```

```
  # save plot to file
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()

# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
  # save plot
  save_plot(x_fake, epoch)
  # save the generator model tile file
  filename = 'generator_model_%03d.h5' % (epoch + 1)
  g_model.save(filename)

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # create training set for the discriminator
      X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))
      # update discriminator model weights
      d_loss, _ = d_model.train_on_batch(X, y)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d=%.3f, g=%.3f' % (i+1, j+1, bat_per_epo, d_loss, g_loss))
    # evaluate the model performance, sometimes
    if (i+1) % 10 == 0:
      summarize_performance(i, g_model, d_model, dataset, latent_dim)

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
```

```
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim)
```

Listing 7.37: Complete example of training a GAN to generate grayscale handwritten digits.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

The chosen configuration results in the stable training of both the generative and discriminative model. The model performance is reported every batch, including the loss of both the discriminative (d) and generative (g) models.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the loss remains stable over the course of training.

```
>1, 1/234, d=0.711, g=0.678
>1, 2/234, d=0.703, g=0.698
>1, 3/234, d=0.694, g=0.717
>1, 4/234, d=0.684, g=0.740
>1, 5/234, d=0.679, g=0.757
>1, 6/234, d=0.668, g=0.777
...
>100, 230/234, d=0.690, g=0.710
>100, 231/234, d=0.692, g=0.705
>100, 232/234, d=0.698, g=0.701
>100, 233/234, d=0.697, g=0.688
>100, 234/234, d=0.693, g=0.698
```

Listing 7.38: Example output of loss from training a GAN to generate grayscale handwritten digits.

The generator is evaluated every 10 epochs, resulting in 10 evaluations, 10 plots of generated images, and 10 saved models. In this case, we can see that the accuracy fluctuates over training. When viewing the discriminator model's accuracy score in concert with generated images, we can see that the accuracy on fake examples does not correlate well with the subjective quality of images, but the accuracy for real examples may. It is crude and possibly unreliable metric of GAN performance, along with loss.

```
>Accuracy real: 51%, fake: 78%
>Accuracy real: 30%, fake: 95%
>Accuracy real: 75%, fake: 59%
>Accuracy real: 98%, fake: 11%
>Accuracy real: 27%, fake: 92%
>Accuracy real: 21%, fake: 92%
>Accuracy real: 29%, fake: 96%
>Accuracy real: 4%, fake: 99%
```

```
>Accuracy real: 18%, fake: 97%
>Accuracy real: 28%, fake: 89%
```

Listing 7.39: Example output of accuracy from training a GAN to generate grayscale handwritten digits.

More training, beyond some point, does not mean better quality generated images. In this case, the results after 10 epochs are low quality, although we can see that the generator has learned to generate centered figures in white on a black background (recall we have inverted the grayscale in the plot).

Figure 7.6: Plot of 100 GAN Generated MNIST Figures After 10 Epochs.

After 20 or 30 more epochs, the model begins to generate very plausible MNIST figures, suggesting that 100 epochs are probably not required for the chosen model configurations.

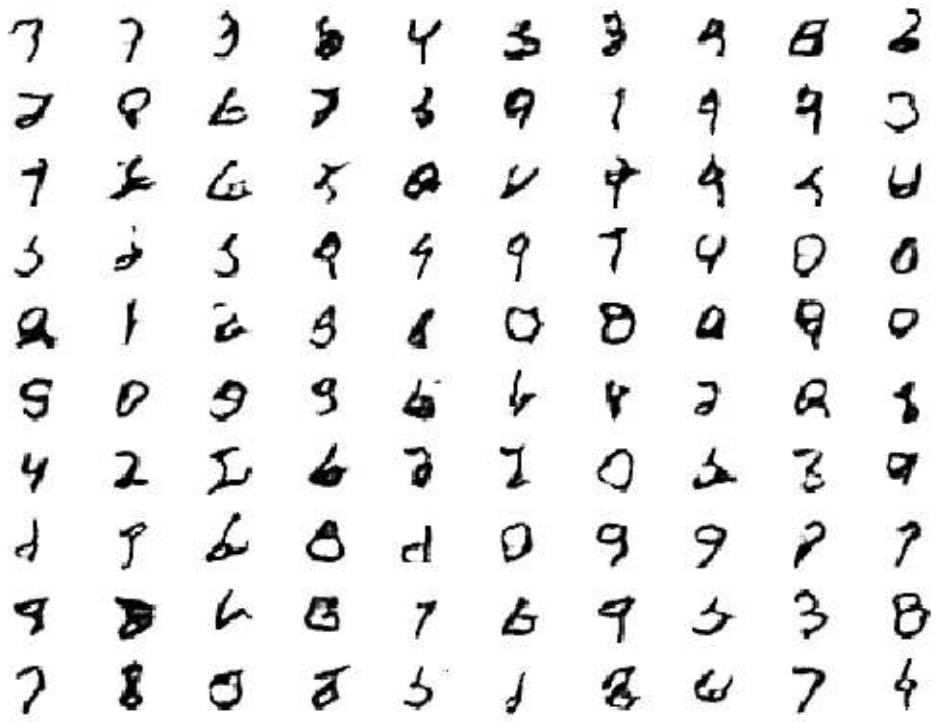

Figure 7.7: Plot of 100 GAN Generated MNIST Figures After 40 Epochs.

The generated images after 100 epochs are not greatly different, but I believe I can detect less blocky-ness in the curves.

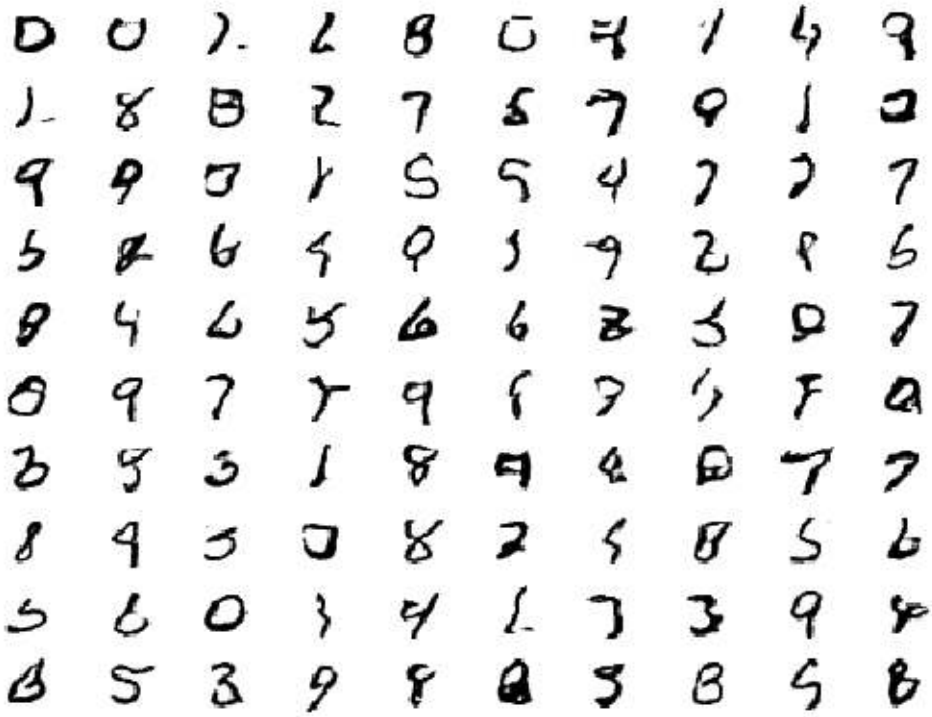

Figure 7.8: Plot of 100 GAN Generated MNIST Figures After 100 Epochs.

7.8 How to Use the Final Generator Model

Once a final generator model is selected, it can be used in a standalone manner for your application. This involves first loading the model from file, then using it to generate images. The generation of each image requires a point in the latent space as input. The complete example of loading the saved model and generating images is listed below. In this case, we will use the model saved after 100 training epochs, but the model saved after 40 or 50 epochs would work just as well.

```
# example of loading the generator model and generating images
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

```
# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  pyplot.show()

# load model
model = load_model('generator_model_100.h5')
# generate images
latent_points = generate_latent_points(100, 25)
# generate images
X = model.predict(latent_points)
# plot the result
save_plot(X, 5)
```

Listing 7.40: Complete example of loading and using the saved generator model.

Running the example first loads the model, samples 25 random points in the latent space, generates 25 images, then plots the results as a single image.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that most of the images are plausible, or plausible pieces of handwritten digits.

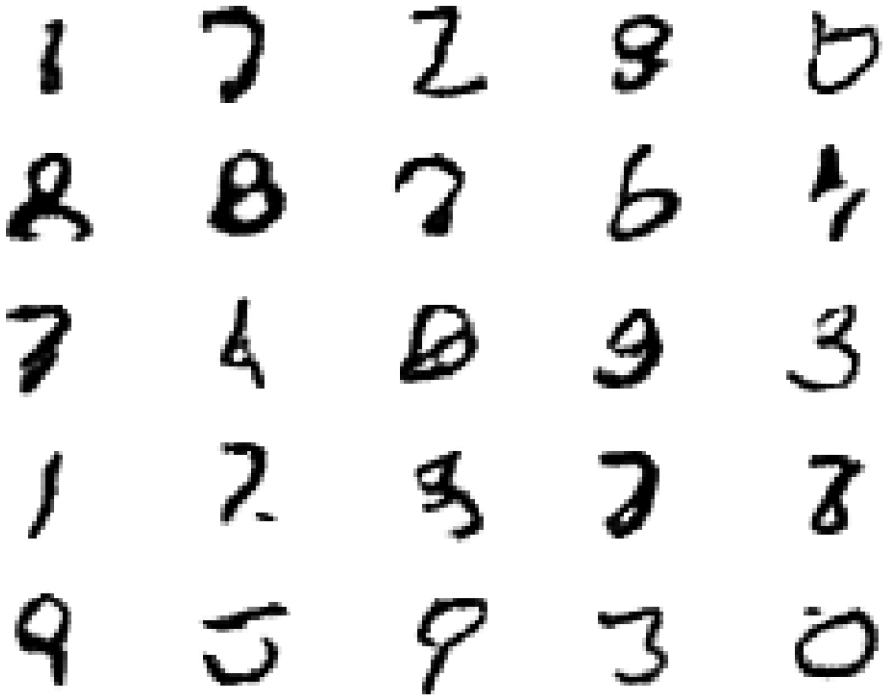

Figure 7.9: Example of 25 GAN Generated MNIST Handwritten Images.

The latent space now defines a compressed representation of MNIST handwritten digits. You can experiment with generating different points in this space and see what types of numbers they generate. The example below generates a single handwritten digit using a vector of all 0.0 values.

```
# example of generating an image for a specific point in the latent space
from keras.models import load_model
from numpy import asarray
from matplotlib import pyplot
# load model
model = load_model('generator_model_100.h5')
# all 0s
vector = asarray([[0.0 for _ in range(100)]])
# generate image
X = model.predict(vector)
# plot the result
pyplot.imshow(X[0, :, :, 0], cmap='gray_r')
pyplot.show()
```

Listing 7.41: Complete example of using the saved model to generate a single image.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, a vector of all zeros results in a handwritten 9 or maybe an 8. You can then try navigating the space and see if you can generate a range of similar, but different handwritten digits.

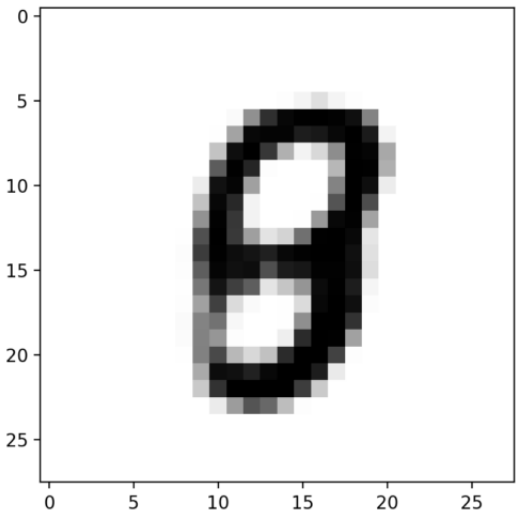

Figure 7.10: Example of a GAN Generated MNIST Handwritten Digit for a Vector of Zeros.

7.9 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **TanH Activation and Scaling**. Update the example to use the Tanh activation function in the generator and scale all pixel values to the range [-1, 1].
- **Change Latent Space**. Update the example to use a larger or smaller latent space and compare the quality of the results and speed of training.
- **Batch Normalization**. Update the discriminator and/or the generator to make use of batch normalization, recommended for DCGAN models.
- **Label Smoothing**. Update the example to use one-sided label smoothing when training the discriminator, specifically change the target label of real examples from 1.0 to 0.9, and review the effects on image quality and speed of training.
- **Model Configuration**. Update the model configuration to use deeper or more shallow discriminator and/or generator models, perhaps experiment with the `UpSampling2D` layers in the generator.

If you explore any of these extensions, I'd love to know.

7.10 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

7.10.1 APIs

- Keras API.
  https://keras.io/
- How can I "freeze" Keras layers?.
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers
- MatplotLib API.
  https://matplotlib.org/api/
- `numpy.random.rand` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html
- `numpy.random.randn` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html
- `numpy.zeros` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html
- `numpy.ones` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.ones.html
- `numpy.hstack` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.hstack.html

7.10.2 Articles

- MNIST Dataset, Wikipedia.
  https://en.wikipedia.org/wiki/MNIST_database

7.11 Summary

In this tutorial, you discovered how to develop a generative adversarial network with deep convolutional networks for generating handwritten digits. Specifically, you learned:

- How to define and train the standalone discriminator model for learning the difference between real and fake images.
- How to define the standalone generator model and train the composite generator and discriminator model.
- How to evaluate the performance of the GAN and use the final standalone generator model to generate new images.

7.11.1 Next

In the next tutorial, you will develop a deep convolutional GAN for small color photos in the CIFAR-10 dataset.

Chapter 8

How to Develop a DCGAN for Small Color Photographs

Generative Adversarial Networks, or GANs, are an architecture for training generative models, such as deep convolutional neural networks for generating images. Developing a GAN for generating images requires both a discriminator convolutional neural network model for classifying whether a given image is real or generated and a generator model that uses inverse convolutional layers to transform an input to a full two-dimensional image of pixel values.

It can be challenging to understand both how GANs work and how deep convolutional neural network models can be trained in a GAN architecture for image generation. A good starting point for beginners is to practice developing and using GANs on standard image datasets used in the field of computer vision, such as the CIFAR small object photograph dataset. Using small and well-understood datasets means that smaller models can be developed and trained quickly, allowing focus to be put on the model architecture and image generation process itself. In this tutorial, you will discover how to develop a generative adversarial network with deep convolutional networks for generating small photographs of objects. After completing this tutorial, you will know:

- How to define and train the standalone discriminator model for learning the difference between real and fake images.
- How to define the standalone generator model and train the composite generator and discriminator model.
- How to evaluate the performance of the GAN and use the final standalone generator model to generate new images.

Let's get started.

8.1 Tutorial Overview

This tutorial is divided into seven parts; they are:

1. CIFAR-10 Small Object Photograph Dataset
2. How to Define and Train the Discriminator Model

3. How to Define and Use the Generator Model
4. How to Train the Generator Model
5. How to Evaluate GAN Model Performance
6. Complete Example of GAN for CIFAR-10
7. How to Use the Final Generator Model

8.2 CIFAR-10 Small Object Photograph Dataset

CIFAR is an acronym that stands for the Canadian Institute For Advanced Research and the CIFAR-10 dataset was developed along with the CIFAR-100 dataset (covered in the next section) by researchers at the CIFAR institute. The dataset is comprised of 60,000 32×32 pixel color photographs of objects from 10 classes, such as frogs, birds, cats, ships, airplanes, etc. These are very small images, much smaller than a typical photograph, and the dataset is intended for computer vision research. Keras provides access to the CIFAR-10 dataset via the `cifar10.load_dataset()` function. It returns two tuples, one with the input and output elements for the standard training dataset, and another with the input and output elements for the standard test dataset. The example below loads the dataset and summarizes the shape of the loaded dataset.

Note: the first time you load the dataset, Keras will automatically download a compressed version of the images and save them under your home directory in `~/.keras/datasets/`. The download is fast as the dataset is only about 163 megabytes in its compressed form.

```
# example of loading the cifar10 dataset
from keras.datasets.cifar10 import load_data
# load the images into memory
(trainX, trainy), (testX, testy) = load_data()
# summarize the shape of the dataset
print('Train', trainX.shape, trainy.shape)
print('Test', testX.shape, testy.shape)
```

Listing 8.1: Example of loading and summarizing the CIFAR-10 dataset.

Running the example loads the dataset and prints the shape of the input and output components of the train and test splits of images. We can see that there are 50K examples in the training set and 10K in the test set and that each image is a square of 32 by 32 pixels.

```
Train (50000, 32, 32, 3) (50000, 1)
Test (10000, 32, 32, 3) (10000, 1)
```

Listing 8.2: Example output from loading and summarizing the CIFAR-10 dataset.

The images are color with the object centered in the middle of the frame. We can plot some of the images from the training dataset with the Matplotlib library using the `imshow()` function.

```
...
# plot raw pixel data
pyplot.imshow(trainX[i])
```

Listing 8.3: Example of plotting a single image.

The example below plots the first 49 images from the training dataset in a 7 by 7 square.

```
# example of loading and plotting the cifar10 dataset
from keras.datasets.cifar10 import load_data
from matplotlib import pyplot
# load the images into memory
(trainX, trainy), (testX, testy) = load_data()
# plot images from the training dataset
for i in range(49):
  # define subplot
  pyplot.subplot(7, 7, 1 + i)
  # turn off axis
  pyplot.axis('off')
  # plot raw pixel data
  pyplot.imshow(trainX[i])
pyplot.show()
```

Listing 8.4: Example of plotting images from the CIFAR-10 dataset.

Running the example creates a figure with a plot of 49 images from the CIFAR-10 training dataset, arranged in a 7×7 square. In the plot, you can see small photographs of planes, trucks, horses, cars, frogs, and so on.

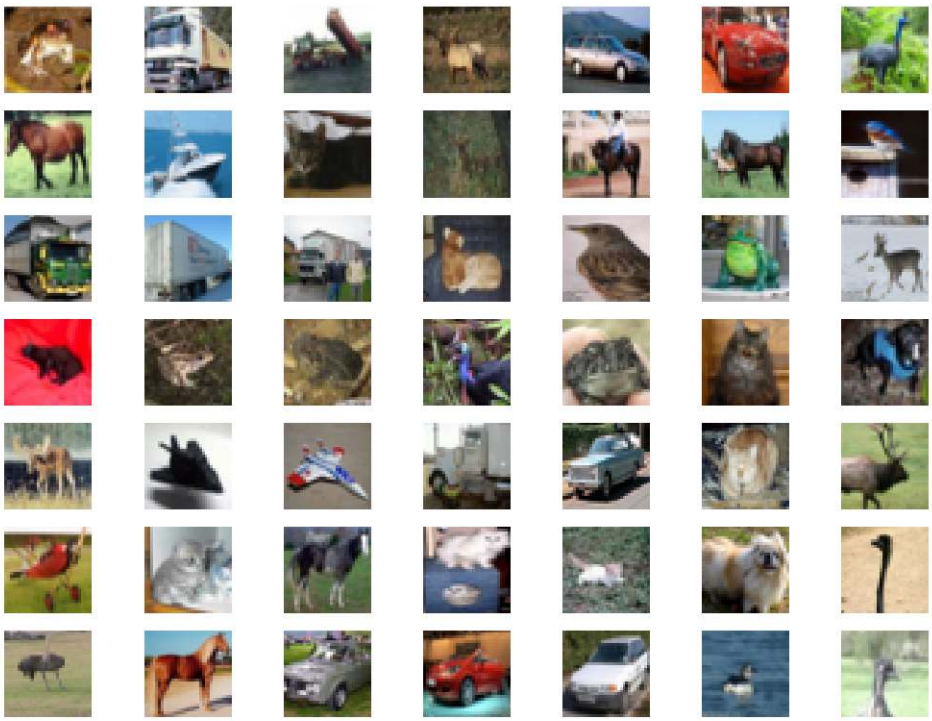

Figure 8.1: Plot of the First 49 Small Object Photographs From the CIFAR-10 Dataset.

We will use the images in the training dataset as the basis for training a Generative

Adversarial Network. Specifically, the generator model will learn how to generate new plausible photographs of objects using a discriminator that will try to distinguish between real images from the CIFAR-10 training dataset and new images output by the generator model.

8.3 How to Define and Train the Discriminator Model

The first step is to define the discriminator model. The model must take a sample image from our dataset as input and output a classification prediction as to whether the sample is real or fake. This is a binary classification problem.

- **Inputs**: Image with three color channel and 32×32 pixels in size.
- **Outputs**: Binary classification, likelihood the sample is real (or fake).

The discriminator model has a normal convolutional layer followed by three convolutional layers using a stride of 2×2 to downsample the input image. The model has no pooling layers and a single node in the output layer with the sigmoid activation function to predict whether the input sample is real or fake. The model is trained to minimize the binary cross-entropy loss function, appropriate for binary classification. We will use some best practices in defining the discriminator model, such as the use of `LeakyReLU` instead of `ReLU`, using `Dropout`, and using the Adam version of stochastic gradient descent with a learning rate of 0.0002 and a momentum of 0.5. The `define_discriminator()` function below defines the discriminator model and parametrizes the size of the input image.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(32,32,3)):
  model = Sequential()
  # normal
  model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

Listing 8.5: Example of a function for defining the discriminator model.

We can use this function to define the discriminator model and summarize it. The complete example is listed below.

```
# example of defining the discriminator model
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(32,32,3)):
  model = Sequential()
  # normal
  model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define model
model = define_discriminator()
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='discriminator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 8.6: Example of defining and summarizing the discriminator model.

Running the example first summarizes the model architecture, showing the output shape for each layer. We can see that the aggressive 2×2 stride acts to downsample the input image, first from 32×32 to 16×16, then to 8×8 and more before the model makes an output prediction. This pattern is by design as we do not use pooling layers and use the large stride to achieve a similar downsampling effect. We will see a similar pattern, but in reverse in the generator model in the next section.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 64)        1792
_________________________________________________________________
```

```
leaky_re_lu_1 (LeakyReLU) (None, 32, 32, 64)      0
_________________________________________________________________
conv2d_2 (Conv2D)         (None, 16, 16, 128)     73856
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU) (None, 16, 16, 128)     0
_________________________________________________________________
conv2d_3 (Conv2D)         (None, 8, 8, 128)       147584
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU) (None, 8, 8, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)         (None, 4, 4, 256)       295168
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU) (None, 4, 4, 256)       0
_________________________________________________________________
flatten_1 (Flatten)       (None, 4096)            0
_________________________________________________________________
dropout_1 (Dropout)       (None, 4096)            0
_________________________________________________________________
dense_1 (Dense)           (None, 1)               4097
=================================================================
Total params: 522,497
Trainable params: 522,497
Non-trainable params: 0
_________________________________________________________________
```

Listing 8.7: Example output from defining and summarizing the discriminator model.

A plot of the model is also created and we can see that the model expects two inputs and will predict a single output.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

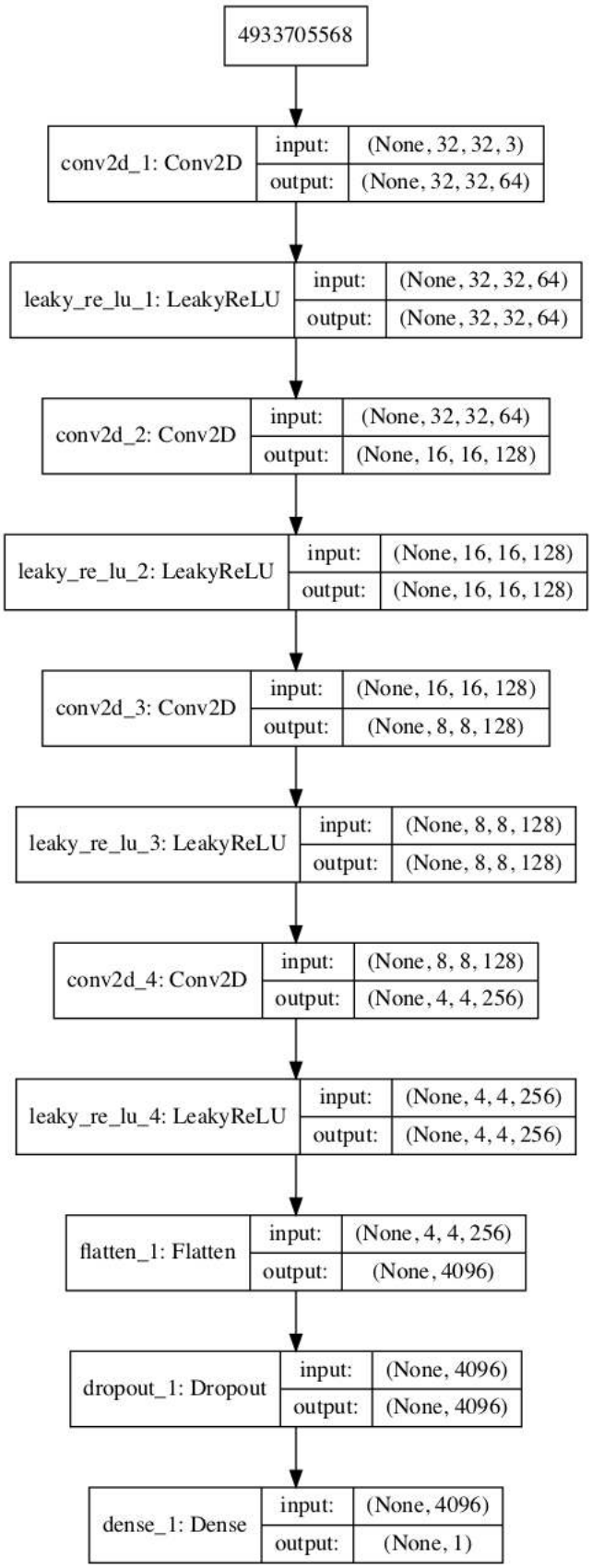

Figure 8.2: Plot of the Discriminator Model in the CIFAR-10 Generative Adversarial Network.

We could start training this model now with real examples with a class label of one and randomly generate samples with a class label of zero. The development of these elements will be useful later, and it helps to see that the discriminator is just a normal neural network model for binary classification. First, we need a function to load and prepare the dataset of real images. We will use the `cifar.load_data()` function to load the CIFAR-10 dataset and just use the input part of the training dataset as the real images.

```
...
# load cifar10 dataset
(trainX, _), (_, _) = load_data()
```

Listing 8.8: Example of loading the CIFAR-10 training dataset.

We must scale the pixel values from the range of unsigned integers in [0,255] to the normalized range of [-1,1]. The generator model will generate images with pixel values in the range [-1,1] as it will use the Tanh activation function, a best practice. It is also a good practice for the real images to be scaled to the same range.

```
...
# convert from unsigned ints to floats
X = trainX.astype('float32')
# scale from [0,255] to [-1,1]
X = (X - 127.5) / 127.5
```

Listing 8.9: Example of scaling the pixel values of the loaded dataset.

The `load_real_samples()` function below implements the loading and scaling of real CIFAR-10 photographs.

```
# load and prepare cifar10 training images
def load_real_samples():
  # load cifar10 dataset
  (trainX, _), (_, _) = load_data()
  # convert from unsigned ints to floats
  X = trainX.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X
```

Listing 8.10: Example of a function for loading and scaling the pixel values of the CIFAR-10 training dataset.

The discriminator model will be updated in batches, specifically with a collection of real samples and a collection of generated samples. On training, an epoch is defined as one pass through the entire training dataset. We could systematically enumerate all samples in the training dataset, and that is a good approach, but good training via stochastic gradient descent requires that the training dataset be shuffled prior to each epoch. A simpler approach is to select random samples of images from the training dataset. The `generate_real_samples()` function below will take the training dataset as an argument and will select a random subsample of images; it will also return class labels for the sample, specifically a class label of 1, to indicate real images.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
```

```
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y
```

Listing 8.11: Example of a function for sampling images in the loaded dataset.

Now, we need a source of fake images. We don't have a generator model yet, so instead, we can generate images comprised of random pixel values, specifically random pixel values in the range [0,1], then scaled to the range [-1, 1] like our scaled real images. The `generate_fake_samples()` function below implements this behavior and generates images of random pixel values and their associated class label of 0, for fake.

```
# generate n fake samples with class labels
def generate_fake_samples(n_samples):
  # generate uniform random numbers in [0,1]
  X = rand(32 * 32 * 3 * n_samples)
  # update to have the range [-1, 1]
  X = -1 + X * 2
  # reshape into a batch of color images
  X = X.reshape((n_samples, 32, 32, 3))
  # generate 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y
```

Listing 8.12: Example of a function for generating images with random pixel values.

Finally, we need to train the discriminator model. This involves repeatedly retrieving samples of real images and samples of generated images and updating the model for a fixed number of iterations. We will ignore the idea of epochs for now (e.g. complete passes through the training dataset) and fit the discriminator model for a fixed number of batches. The model will learn to discriminate between real and fake (randomly generated) images rapidly, therefore not many batches will be required before it learns to discriminate perfectly.

The `train_discriminator()` function implements this, using a batch size of 128 images, where 64 are real and 64 are fake each iteration. We update the discriminator separately for real and fake examples so that we can calculate the accuracy of the model on each sample prior to the update. This gives insight into how the discriminator model is performing over time.

```
# train the discriminator model
def train_discriminator(model, dataset, n_iter=20, n_batch=128):
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_iter):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator on real samples
    _, real_acc = model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update discriminator on fake samples
    _, fake_acc = model.train_on_batch(X_fake, y_fake)
    # summarize performance
    print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))
```

Listing 8.13: Example of a function for training the discriminator model.

Tying all of this together, the complete example of training an instance of the discriminator model on real and randomly generated (fake) images is listed below.

```
# example of training the discriminator model on real and random cifar10 images
from numpy import ones
from numpy import zeros
from numpy.random import rand
from numpy.random import randint
from keras.datasets.cifar10 import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LeakyReLU

# define the standalone discriminator model
def define_discriminator(in_shape=(32,32,3)):
  model = Sequential()
  # normal
  model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# load and prepare cifar10 training images
def load_real_samples():
  # load cifar10 dataset
  (trainX, _), (_, _) = load_data()
  # convert from unsigned ints to floats
  X = trainX.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
```

```
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y

# generate n fake samples with class labels
def generate_fake_samples(n_samples):
  # generate uniform random numbers in [0,1]
  X = rand(32 * 32 * 3 * n_samples)
  # update to have the range [-1, 1]
  X = -1 + X * 2
  # reshape into a batch of color images
  X = X.reshape((n_samples, 32, 32, 3))
  # generate 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# train the discriminator model
def train_discriminator(model, dataset, n_iter=20, n_batch=128):
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_iter):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator on real samples
    _, real_acc = model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update discriminator on fake samples
    _, fake_acc = model.train_on_batch(X_fake, y_fake)
    # summarize performance
    print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))

# define the discriminator model
model = define_discriminator()
# load image data
dataset = load_real_samples()
# fit the model
train_discriminator(model, dataset)
```

Listing 8.14: Example of defining and training the discriminator model.

Running the example first defines the model, loads the CIFAR-10 dataset, then trains the discriminator model.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the discriminator model learns to tell the difference between real and randomly generated CIFAR-10 images very quickly, in about 20 batches.

```
...
>16 real=100% fake=100%
```

```
>17 real=100% fake=100%
>18 real=98% fake=100%
>19 real=100% fake=100%
>20 real=100% fake=100%
```

Listing 8.15: Example output from defining and training the discriminator model.

Now that we know how to define and train the discriminator model, we need to look at developing the generator model.

8.4 How to Define and Use the Generator Model

The generator model is responsible for creating new, fake, but plausible small photographs of objects. It does this by taking a point from the latent space as input and outputting a square color image. The latent space is an arbitrarily defined vector space of Gaussian-distributed values, e.g. 100 dimensions. It has no meaning, but by drawing points from this space randomly and providing them to the generator model during training, the generator model will assign meaning to the latent points and, in turn, the latent space, until, at the end of training, the latent vector space represents a compressed representation of the output space, CIFAR-10 images, that only the generator knows how to turn into plausible CIFAR-10 images.

- **Inputs**: Point in latent space, e.g. a 100-element vector of Gaussian random numbers.
- **Outputs**: Two-dimensional square color image (3 channels) of 32×32 pixels with pixel values in [-1,1].

We don't have to use a 100 element vector as input; it is a round number and widely used, but I would expect that 10, 50, or 500 would work just as well. Developing a generator model requires that we transform a vector from the latent space with, 100 dimensions to a 2D array with $32 \times 32 \times 3$, or 3,072 values. There are a number of ways to achieve this, but there is one approach that has proven effective on deep convolutional generative adversarial networks. It involves two main elements. The first is a `Dense` layer as the first hidden layer that has enough nodes to represent a low-resolution version of the output image. Specifically, an image half the size (one quarter the area) of the output image would be 16x16x3, or 768 nodes, and an image one quarter the size (one eighth the area) would be $8 \times 8 \times 3$, or 192 nodes.

With some experimentation, I have found that a smaller low-resolution version of the image works better. Therefore, we will use $4 \times 4 \times 3$, or 48 nodes. We don't just want one low-resolution version of the image; we want many parallel versions or interpretations of the input. This is a pattern in convolutional neural networks where we have many parallel filters resulting in multiple parallel activation maps, called feature maps, with different interpretations of the input. We want the same thing in reverse: many parallel versions of our output with different learned features that can be collapsed in the output layer into a final image. The model needs space to invent, create, or generate. Therefore, the first hidden layer, the `Dense` layer, needs enough nodes for multiple versions of our output image, such as 256.

```
...
# foundation for 4x4 image
n_nodes = 256 * 4 * 4
model.add(Dense(n_nodes, input_dim=latent_dim))
```

```
model.add(LeakyReLU(alpha=0.2))
```

Listing 8.16: Example of defining the foundation activations for the generator model.

The activations from these nodes can then be reshaped into something image-like to pass into a convolutional layer, such as 256 different 4×4 feature maps.

```
...
model.add(Reshape((4, 4, 256)))
```

Listing 8.17: Example of reshaping the foundation activations for the generator model.

The next major architectural innovation involves upsampling the low-resolution image to a higher resolution version of the image. There are two common ways to do this upsampling process, sometimes called deconvolution. One way is to use an `UpSampling2D` layer (like a reverse pooling layer) followed by a normal `Conv2D` layer. The other and perhaps more modern way is to combine these two operations into a single layer, called a `Conv2DTranspose`. We will use this latter approach for our generator. The `Conv2DTranspose` layer can be configured with a stride of (2×2) that will quadruple the area of the input feature maps (double their width and height dimensions). It is also good practice to use a kernel size that is a factor of the stride (e.g. double) to avoid a checkerboard pattern that can sometimes be observed when upsampling.

```
...
# upsample to 8x8
model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
model.add(LeakyReLU(alpha=0.2))
```

Listing 8.18: Example of adding a transpose convolutional layer to the generator.

This can be repeated two more times to arrive at our required 32×32 output image. Again, we will use the `LeakyReLU` with a default slope of 0.2, reported as a best practice when training GAN models. The output layer of the model is a `Conv2D` with three filters for the three required channels and a kernel size of 3×3 and '`same`' padding, designed to create a single feature map and preserve its dimensions at $32 \times 32 \times 3$ pixels. A Tanh activation is used to ensure output values are in the desired range of [-1,1], a current best practice. The `define_generator()` function below implements this and defines the generator model. The generator model is not compiled and does not specify a loss function or optimization algorithm. This is because the generator is not trained directly. We will learn more about this in the next section.

```
# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 4x4 image
  n_nodes = 256 * 4 * 4
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((4, 4, 256)))
  # upsample to 8x8
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 16x16
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 32x32
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
```

```
  model.add(LeakyReLU(alpha=0.2))
  # output layer
  model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
  return model
```

Listing 8.19: Example of a function for defining the generator model.

We can summarize the model to help better understand the input and output shapes. The complete example is listed below.

```
# example of defining the generator model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 4x4 image
  n_nodes = 256 * 4 * 4
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((4, 4, 256)))
  # upsample to 8x8
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 16x16
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 32x32
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # output layer
  model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
  return model

# define the size of the latent space
latent_dim = 100
# define the generator model
model = define_generator(latent_dim)
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='generator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 8.20: Example of defining and summarizing the generator model.

Running the example summarizes the layers of the model and their output shape. We can see that, as designed, the first hidden layer has 4,096 parameters or $256 \times 4 \times 4$, the activations of which are reshaped into 256 4×4 feature maps. The feature maps are then upscaled via the three `Conv2DTranspose` layers to the desired output shape of 32×32, until the output layer where three filter maps (channels) are created.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 4096)              413696
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 4096)              0
_________________________________________________________________
reshape_1 (Reshape)          (None, 4, 4, 256)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 8, 8, 128)         524416
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 16, 16, 128)       262272
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 32, 32, 128)       262272
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 32, 32, 128)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 3)         3459
=================================================================
Total params: 1,466,115
Trainable params: 1,466,115
Non-trainable params: 0
_________________________________________________________________
```

Listing 8.21: Example output from defining and summarizing the generator model.

A plot of the model is also created and we can see that the model expects a 100-element point from the latent space as input and will predict a two-element vector as output.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

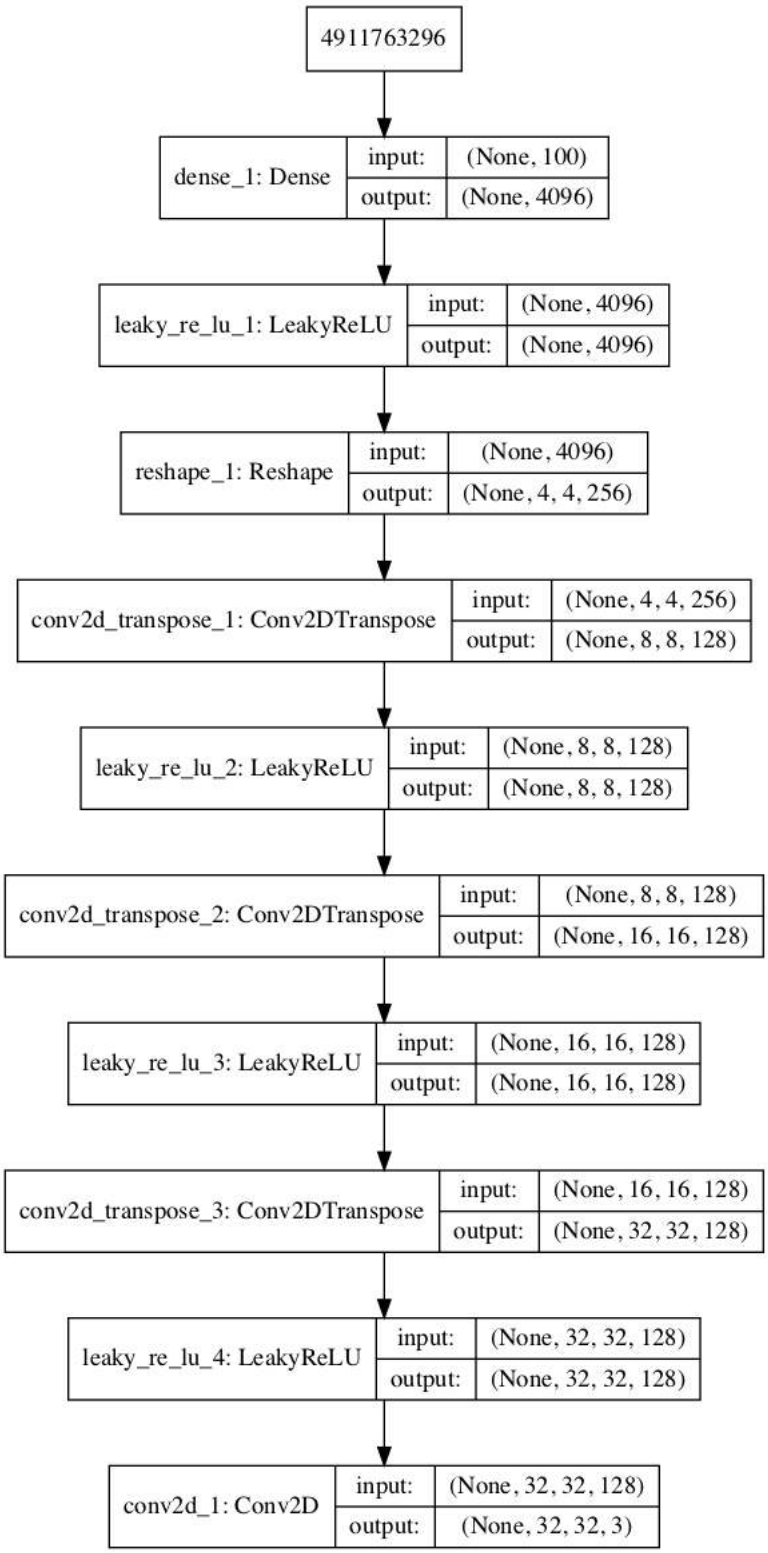

Figure 8.3: Plot of the Generator Model in the CIFAR-10 Generative Adversarial Network.

This model cannot do much at the moment. Nevertheless, we can demonstrate how to use it to generate samples. This is a helpful demonstration to understand the generator as just another model, and some of these elements will be useful later. The first step is to generate

new points in the latent space. We can achieve this by calling the `randn()` NumPy function for generating arrays of random numbers drawn from a standard Gaussian. The array of random numbers can then be reshaped into samples, that is n rows with 100 elements per row. The `generate_latent_points()` function below implements this and generates the desired number of points in the latent space that can be used as input to the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

Listing 8.22: Example of a function for generating random points in the latent space.

Next, we can use the generated points as input to the generator model to generate new samples, then plot the samples. We can update the `generate_fake_samples()` function from the previous section to take the generator model as an argument and use it to generate the desired number of samples by first calling the `generate_latent_points()` function to generate the required number of points in latent space as input to the model. The updated `generate_fake_samples()` function is listed below and returns both the generated samples and the associated class labels.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y
```

Listing 8.23: Example of a function for creating images with the generator.

We can then plot the generated samples as we did the real CIFAR-10 examples in the first section by calling the `imshow()` function. The complete example of generating new CIFAR-10 images with the untrained generator model is listed below.

```
# example of defining and using the generator model
from numpy import zeros
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from matplotlib import pyplot

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 4x4 image
  n_nodes = 256 * 4 * 4
```

```
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((4, 4, 256)))
  # upsample to 8x8
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 16x16
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 32x32
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # output layer
  model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
  return model

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# size of the latent space
latent_dim = 100
# define the discriminator model
model = define_generator(latent_dim)
# generate samples
n_samples = 49
X, _ = generate_fake_samples(model, latent_dim, n_samples)
# scale pixel values from [-1,1] to [0,1]
X = (X + 1) / 2.0
# plot the generated samples
for i in range(n_samples):
  # define subplot
  pyplot.subplot(7, 7, 1 + i)
  # turn off axis labels
  pyplot.axis('off')
  # plot single image
  pyplot.imshow(X[i])
# show the figure
pyplot.show()
```

Listing 8.24: Example of using the untrained generator model.

Running the example generates 49 examples of fake CIFAR-10 images and visualizes them on a single plot of 7 by 7 images. As the model is not trained, the generated images are completely random pixel values in [-1, 1], rescaled to [0, 1]. As we might expect, the images look like a mess of gray.

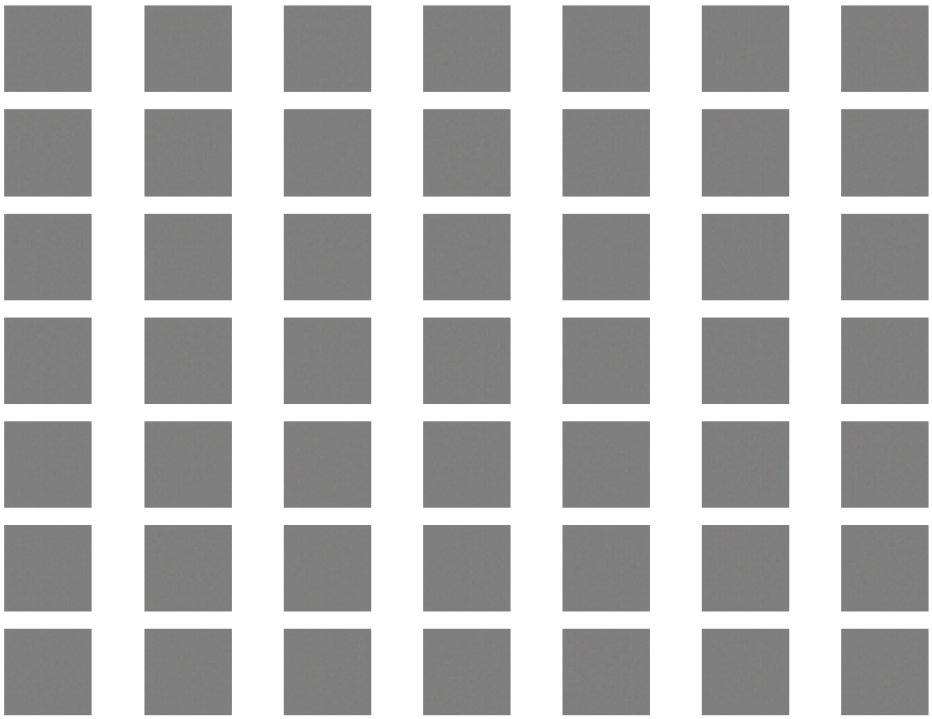

Figure 8.4: Example of 49 CIFAR-10 Images Output by the Untrained Generator Model.

Now that we know how to define and use the generator model, the next step is to train the model.

8.5 How to Train the Generator Model

The weights in the generator model are updated based on the performance of the discriminator model. When the discriminator is good at detecting fake samples, the generator is updated more, and when the discriminator model is relatively poor or confused when detecting fake samples, the generator model is updated less. This defines the zero-sum or adversarial relationship between these two models. There may be many ways to implement this using the Keras API, but perhaps the simplest approach is to create a new model that combines the generator and discriminator models.

Specifically, a new GAN model can be defined that stacks the generator and discriminator such that the generator receives as input random points in the latent space and generates

samples that are fed into the discriminator model directly, classified, and the output of this larger model can be used to update the model weights of the generator. To be clear, we are not talking about a new third model, just a new logical model that uses the already-defined layers and weights from the standalone generator and discriminator models. Only the discriminator is concerned with distinguishing between real and fake examples, therefore the discriminator model can be trained in a standalone manner on examples of each, as we did in the section on the discriminator model above.

The generator model is only concerned with the discriminator's performance on fake examples. Therefore, we will mark all of the layers in the discriminator as not trainable when it is part of the GAN model so that they cannot be updated and overtrained on fake examples. When training the generator via this logical GAN model, there is one more important change. We want the discriminator to think that the samples output by the generator are real, not fake. Therefore, when the generator is trained as part of the GAN model, we will mark the generated samples as real ($class = 1$).

Why would we want to do this? We can imagine that the discriminator will then classify the generated samples as not real ($class = 0$) or a low probability of being real (0.3 or 0.5). The backpropagation process used to update the model weights will see this as a large error and will update the model weights (i.e. only the weights in the generator) to correct for this error, in turn making the generator better at generating good fake samples. Let's make this concrete.

- **Inputs**: Point in latent space, e.g. a 100-element vector of Gaussian random numbers.
- **Outputs**: Binary classification, likelihood the sample is real (or fake).

The `define_gan()` function below takes as arguments the already-defined generator and discriminator models and creates the new, logical third model subsuming these two models. The weights in the discriminator are marked as not trainable, which only affects the weights as seen by the GAN model and not the standalone discriminator model. The GAN model then uses the same binary cross-entropy loss function as the discriminator and the efficient Adam version of stochastic gradient descent with the learning rate of 0.0002 and momentum of 0.5, recommended when training deep convolutional GANs.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model
```

Listing 8.25: Example of a function for defining the composite model for training the generator via the discriminator.

Making the discriminator not trainable is a clever trick in the Keras API. The trainable property impacts the model after it is compiled. The discriminator model was compiled with trainable layers, therefore the model weights in those layers will be updated when the standalone model is updated via calls to the `train_on_batch()` function. The discriminator model was then marked as not trainable, added to the GAN model, and compiled. In this model, the model weights of the discriminator model are not trainable and cannot be changed when the GAN model is updated via calls to the `train_on_batch()` function. This change in the trainable property does not impact the training of the standalone discriminator model. The complete example of creating the discriminator, generator and composite model is listed below.

```
# demonstrate creating the three models in the gan
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(32,32,3)):
  model = Sequential()
  # normal
  model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 4x4 image
  n_nodes = 256 * 4 * 4
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((4, 4, 256)))
  # upsample to 8x8
```

```
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 16x16
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 32x32
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # output layer
  model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# summarize gan model
gan_model.summary()
# plot gan model
plot_model(gan_model, to_file='gan_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 8.26: Example of defining and summarizing the composite model.

Running the example first creates a summary of the composite model, which is pretty uninteresting. We can see that the model expects CIFAR-10 images as input and predicts a single value as output.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_2 (Sequential) (None, 32, 32, 3)       1466115
_________________________________________________________________
sequential_1 (Sequential) (None, 1)               522497
=================================================================
Total params: 1,988,612
Trainable params: 1,466,115
Non-trainable params: 522,497
```

```
_________________________________________________________________
```

Listing 8.27: Example output from defining and summarizing the composite model.

A plot of the model is also created and we can see that the model expects a 100-element point in latent space as input and will predict a single output classification label.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

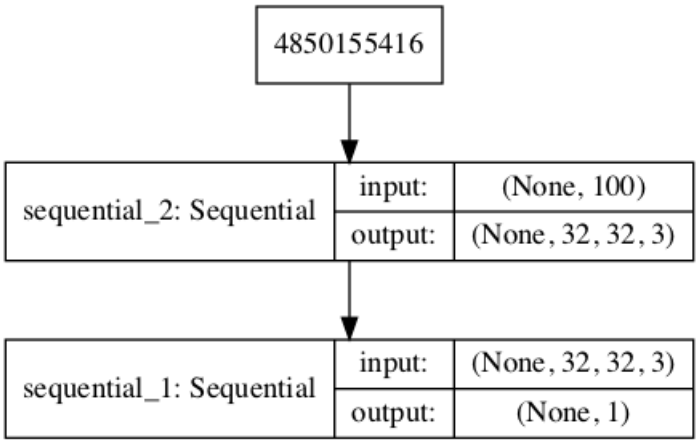

Figure 8.5: Plot of the Composite Generator and Discriminator Model in the CIFAR-10 Generative Adversarial Network.

Training the composite model involves generating a batch worth of points in the latent space via the `generate_latent_points()` function in the previous section, and $class = 1$ labels and calling the `train_on_batch()` function. The `train_gan()` function below demonstrates this, although it is pretty simple as only the generator will be updated each epoch, leaving the discriminator with default model weights.

```
# train the composite model
def train_gan(gan_model, latent_dim, n_epochs=200, n_batch=128):
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    gan_model.train_on_batch(x_gan, y_gan)
```

Listing 8.28: Example of a function for training the generator model via the discriminator model.

Instead, what is required is that we first update the discriminator model with real and fake samples, then update the generator via the composite model. This requires combining elements from the `train_discriminator()` function defined in the discriminator section above and the `train_gan()` function defined above. It also requires that we enumerate over both epochs and

batches within in an epoch. The complete train function for updating the discriminator model and the generator (via the composite model) is listed below.

There are a few things to note in this model training function. First, the number of batches within an epoch is defined by how many times the batch size divides into the training dataset. We have a dataset size of 50K samples and a batch size of 128, so with rounding down, there are $\frac{50000}{128}$ or 390 batches per epoch. The discriminator model is updated twice per batch, once with real samples and once with fake samples, which is a best practice as opposed to combining the samples and performing a single update. Finally, we report the loss each batch. It is critical to keep an eye on the loss over batches. The reason for this is that a crash in the discriminator loss indicates that the generator model has started generating rubbish examples that the discriminator can easily discriminate.

Monitor the discriminator loss and expect it to hover around 0.5 to 0.8 per batch. The generator loss is less critical and may hover between 0.5 and 2 or higher. A clever programmer might even attempt to detect the crashing loss of the discriminator, halt, and then restart the training process.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=200, n_batch=128):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # update discriminator model weights
      d_loss1, _ = d_model.train_on_batch(X_real, y_real)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update discriminator model weights
      d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
        (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
```

Listing 8.29: Example of a function for training the GAN.

We almost have everything we need to develop a GAN for the CIFAR-10 photographs of objects dataset. One remaining aspect is the evaluation of the model.

8.6 How to Evaluate GAN Model Performance

Generally, there are no objective ways to evaluate the performance of a GAN model. We cannot calculate this objective error score for generated images. Instead, images must be subjectively evaluated for quality by a human operator. This means that we cannot know when to stop

training without looking at examples of generated images. In turn, the adversarial nature of the training process means that the generator is changing after every batch, meaning that once *good enough* images can be generated, the subjective quality of the images may then begin to vary, improve, or even degrade with subsequent updates. There are three ways to handle this complex training situation.

1. Periodically evaluate the classification accuracy of the discriminator on real and fake images.
2. Periodically generate many images and save them to file for subjective review.
3. Periodically save the generator model.

All three of these actions can be performed at the same time for a given training epoch, such as every 10 training epochs. The result will be a saved generator model for which we have a way of subjectively assessing the quality of its output and objectively knowing how well the discriminator was fooled at the time the model was saved. Training the GAN over many epochs, such as hundreds or thousands of epochs, will result in many snapshots of the model that can be inspected, and from which specific outputs and models can be cherry-picked for later use.

First, we can define a function called `summarize_performance()` that will summarize the performance of the discriminator model. It does this by retrieving a sample of real CIFAR-10 images, as well as generating the same number of fake CIFAR-10 images with the generator model, then evaluating the classification accuracy of the discriminator model on each sample, and reporting these scores.

```
# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=150):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
```

Listing 8.30: Example of a function for summarizing the model's performance.

This function can be called from the `train()` function based on the current epoch number, such as every 10 epochs.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=200, n_batch=128):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
  ...
  # evaluate the model performance, sometimes
  if (i+1) % 10 == 0:
    summarize_performance(i, g_model, d_model, dataset, latent_dim)
```

Listing 8.31: Example of a summarizing the model's performance from the train function.

Next, we can update the `summarize_performance()` function to both save the model and to create and save a plot generated examples. The generator model can be saved by calling the `save()` function on the generator model and providing a unique filename based on the training epoch number.

```
...
# save the generator model tile file
filename = 'generator_model_%03d.h5' % (epoch+1)
g_model.save(filename)
```

Listing 8.32: Example of saving the generator model to file.

We can develop a function to create a plot of the generated samples. As we are evaluating the discriminator on 100 generated CIFAR-10 images, we can plot about half, or 49, as a 7 by 7 grid. The `save_plot()` function below implements this, again saving the resulting plot with a unique filename based on the epoch number.

```
# create and save a plot of generated images
def save_plot(examples, epoch, n=7):
  # scale from [-1,1] to [0,1]
  examples = (examples + 1) / 2.0
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i])
  # save plot to file
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()
```

Listing 8.33: Example of a function for saving a plot of generated images and the generator model.

The updated `summarize_performance()` function with these additions is listed below.

```
# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=150):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
  # save plot
  save_plot(x_fake, epoch)
  # save the generator model tile file
  filename = 'generator_model_%03d.h5' % (epoch+1)
  g_model.save(filename)
```

Listing 8.34: Example of the updated function for summarizing the model's performance.

8.7 Complete Example of GAN for CIFAR-10

We now have everything we need to train and evaluate a GAN on the CIFAR-10 photographs of small objects dataset. The complete example is listed below.

```
# example of a dcgan on cifar10
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.cifar10 import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(32,32,3)):
  model = Sequential()
  # normal
  model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 4x4 image
  n_nodes = 256 * 4 * 4
```

```
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((4, 4, 256)))
  # upsample to 8x8
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 16x16
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 32x32
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # output layer
  model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# load and prepare cifar10 training images
def load_real_samples():
  # load cifar10 dataset
  (trainX, _), (_, _) = load_data()
  # convert from unsigned ints to floats
  X = trainX.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
```

```
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# create and save a plot of generated images
def save_plot(examples, epoch, n=7):
  # scale from [-1,1] to [0,1]
  examples = (examples + 1) / 2.0
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i])
  # save plot to file
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()

# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=150):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
  # save plot
  save_plot(x_fake, epoch)
  # save the generator model tile file
  filename = 'generator_model_%03d.h5' % (epoch+1)
  g_model.save(filename)

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=200, n_batch=128):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
```

```
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # update discriminator model weights
      d_loss1, _ = d_model.train_on_batch(X_real, y_real)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update discriminator model weights
      d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
        (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
    # evaluate the model performance, sometimes
    if (i+1) % 10 == 0:
      summarize_performance(i, g_model, d_model, dataset, latent_dim)

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim)
```

Listing 8.35: Complete example of training a GAN on the CIFAR-10 training dataset.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

The chosen configuration results in the stable training of both the generative and discriminative model. The model performance is reported every batch, including the loss of both the discriminative (d) and generative (g) models.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the loss remains stable over the course of training. The discriminator loss on the real and generated examples sits around 0.5, whereas the loss for the generator trained via the discriminator sits around 1.5 for much of the training process.

```
>1, 1/390, d1=0.720, d2=0.695 g=0.692
>1, 2/390, d1=0.658, d2=0.697 g=0.691
>1, 3/390, d1=0.604, d2=0.700 g=0.687
```

```
>1, 4/390, d1=0.522, d2=0.709 g=0.680
>1, 5/390, d1=0.417, d2=0.731 g=0.662
...
>200, 386/390, d1=0.499, d2=0.401 g=1.565
>200, 387/390, d1=0.459, d2=0.623 g=1.481
>200, 388/390, d1=0.588, d2=0.556 g=1.700
>200, 389/390, d1=0.579, d2=0.288 g=1.555
>200, 390/390, d1=0.620, d2=0.453 g=1.466
```

Listing 8.36: Example output of loss from training a GAN on the CIFAR-10 training dataset.

The generator is evaluated every 10 epochs, resulting in 20 evaluations, 20 plots of generated images, and 20 saved models. In this case, we can see that the accuracy fluctuates over training. When viewing the discriminator model's accuracy score in concert with generated images, we can see that the accuracy on fake examples does not correlate well with the subjective quality of images, but the accuracy for real examples may. It is a crude and possibly unreliable metric of GAN performance, along with loss.

```
>Accuracy real: 55%, fake: 89%
>Accuracy real: 50%, fake: 75%
>Accuracy real: 49%, fake: 86%
>Accuracy real: 60%, fake: 79%
>Accuracy real: 49%, fake: 87%
...
```

Listing 8.37: Example output of accuracy from training a GAN on the CIFAR-10 training dataset.

More training, beyond some point, does not mean better quality generated images. In this case, the results after 10 epochs are low quality, although we can see some difference between background and foreground with a blob in the middle of each image.

Figure 8.6: Plot of 49 GAN Generated CIFAR-10 Photographs After 10 Epochs.

After 90 or 100 epochs, we are starting to see plausible photographs with blobs that look like birds, dogs, cats, and horses. The objects are familiar and CIFAR-10-like, but many of them are not clearly one of the 10 specified classes.

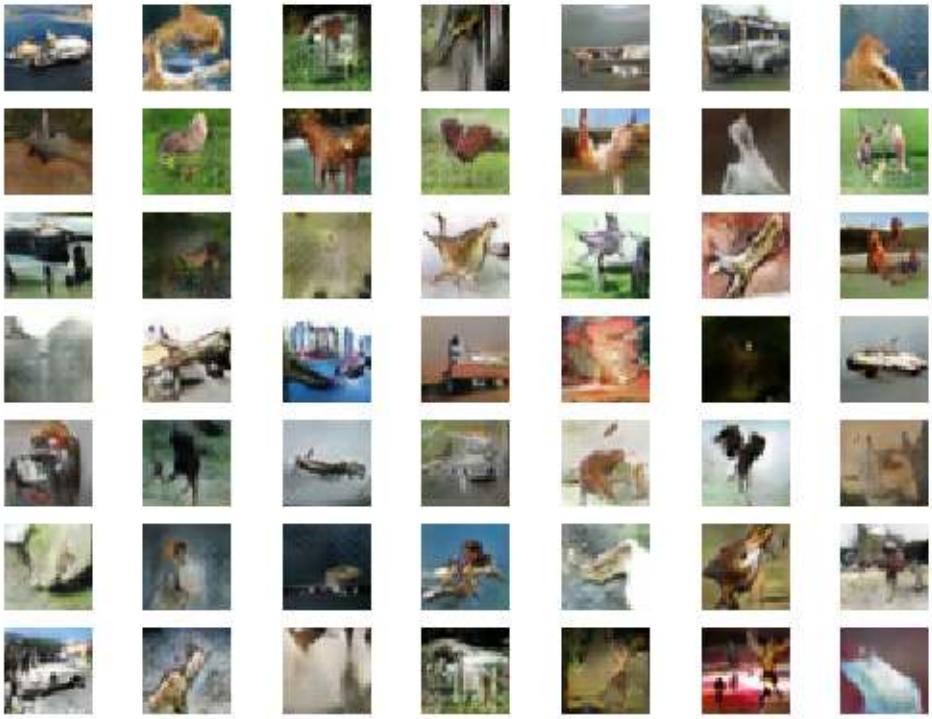

Figure 8.7: Plot of 49 GAN Generated CIFAR-10 Photographs After 90 Epochs.

Figure 8.8: Plot of 49 GAN Generated CIFAR-10 Photographs After 100 Epochs.

The model remains stable over the next 100 epochs, with little improvement in the generated images. The small photos remain vaguely CIFAR-10 like and focused on animals like dogs, cats, and birds.

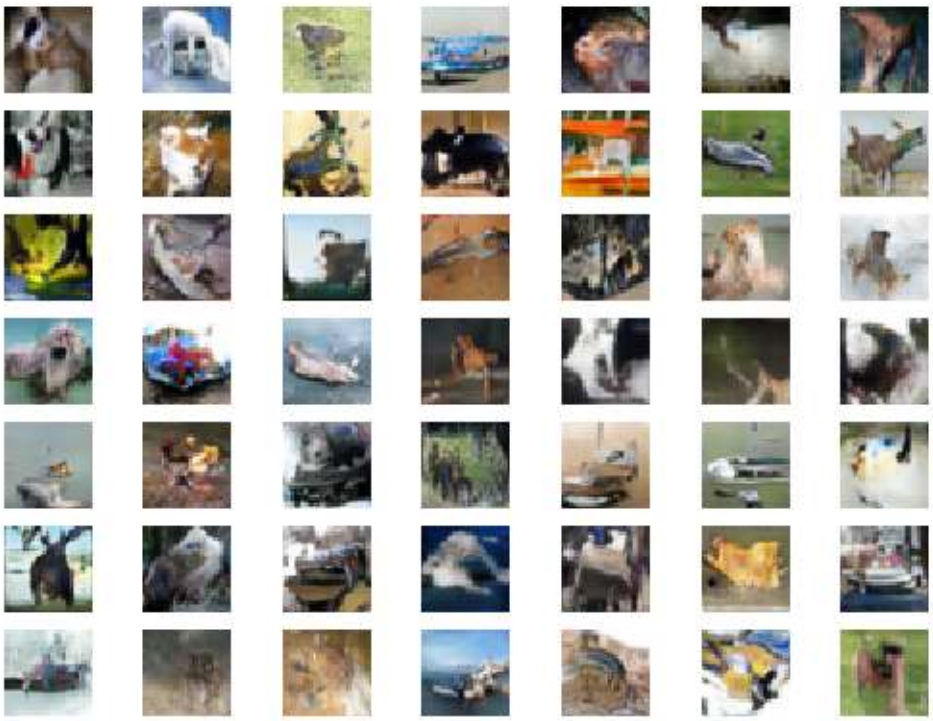

Figure 8.9: Plot of 49 GAN Generated CIFAR-10 Photographs After 200 Epochs.

8.8 How to Use the Final Generator Model

Once a final generator model is selected, it can be used in a standalone manner for your application. This involves first loading the model from file, then using it to generate images. The generation of each image requires a point in the latent space as input. The complete example of loading the saved model and generating images is listed below. In this case, we will use the model saved after 200 training epochs, but the model saved after 100 epochs would work just as well.

```
# example of loading the generator model and generating images
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

```
# create and save a plot of generated images
def save_plot(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :])
  pyplot.show()

# load model
model = load_model('generator_model_200.h5')
# generate images
latent_points = generate_latent_points(100, 100)
# generate images
X = model.predict(latent_points)
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
# plot the result
save_plot(X, 10)
```

Listing 8.38: Example of loading the saved generator model and outputting synthetic images.

Running the example first loads the model, samples 100 random points in the latent space, generates 100 images, then plots the results as a single image. We can see that most of the images are plausible, or plausible pieces of small objects. I can see dogs, cats, horses, birds, frogs, and perhaps planes.

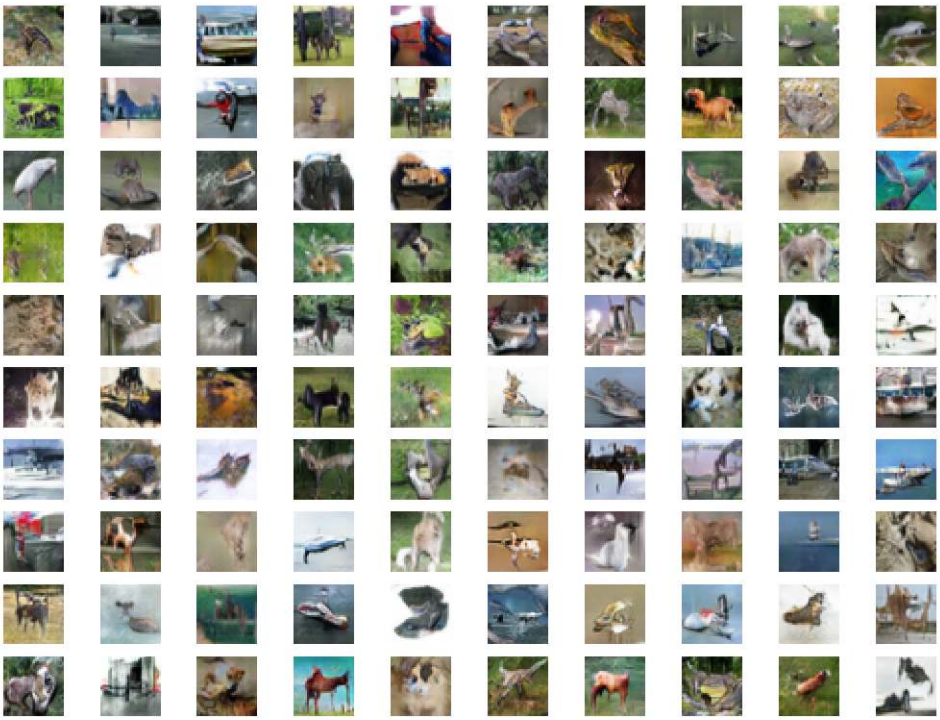

Figure 8.10: Example of 100 GAN Generated CIFAR-10 Small Object Photographs.

The latent space now defines a compressed representation of CIFAR-10 photos. You can experiment with generating different points in this space and see what types of images they generate. The example below generates a single image using a vector of all 0.75 values.

```
# example of generating an image for a specific point in the latent space
from keras.models import load_model
from numpy import asarray
from matplotlib import pyplot
# load model
model = load_model('generator_model_200.h5')
# all 0s
vector = asarray([[0.75 for _ in range(100)]])
# generate image
X = model.predict(vector)
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
# plot the result
pyplot.imshow(X[0, :, :])
pyplot.show()
```

Listing 8.39: Example of loading the saved generator model and outputting a single image.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, a vector of all 0.75 results in a deer or perhaps deer-horse-looking animal in a green field.

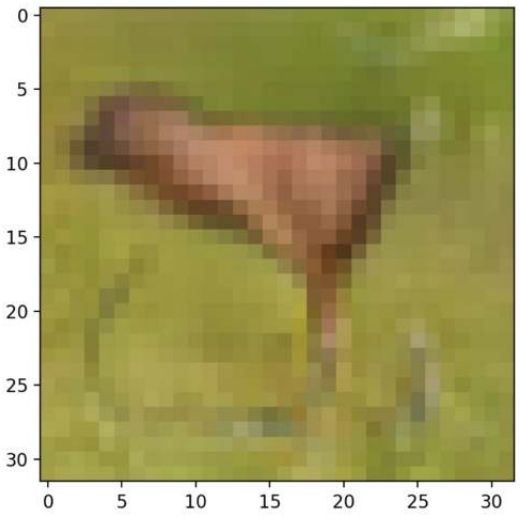

Figure 8.11: Example of a GAN Generated CIFAR Small Object Photo for a Specific Point in the Latent Space.

8.9 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Change Latent Space**. Update the example to use a larger or smaller latent space and compare the quality of the results and speed of training.
- **Batch Normalization**. Update the discriminator and/or the generator to make use of batch normalization, recommended for DCGAN models.
- **Label Smoothing**. Update the example to use one-sided label smoothing when training the discriminator, specifically change the target label of real examples from 1.0 to 0.9 and add random noise, then review the effects on image quality and speed of training.
- **Model Configuration**. Update the model configuration to use deeper or more shallow discriminator and/or generator models, perhaps experiment with the `UpSampling2D` layers in the generator.

If you explore any of these extensions, I'd love to know.

8.10 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

8.10.1 APIs

- Keras API.
  https://keras.io/
- How can I "freeze" Keras layers?.
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers
- MatplotLib API.
  https://matplotlib.org/api/
- `numpy.random.rand` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html
- `numpy.random.randn` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html
- `numpy.zeros` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html
- `numpy.ones` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.ones.html
- `numpy.hstack` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.hstack.html

8.10.2 Articles

- CIFAR-10, Wikipedia.
  https://en.wikipedia.org/wiki/CIFAR-10
- The CIFAR-10 dataset and CIFAR-100 datasets.
  https://www.cs.toronto.edu/~kriz/cifar.html

8.11 Summary

In this tutorial, you discovered how to develop a generative adversarial network with deep convolutional networks for generating small photographs of objects. Specifically, you learned:

- How to define and train the standalone discriminator model for learning the difference between real and fake images.
- How to define the standalone generator model and train the composite generator and discriminator model.
- How to evaluate the performance of the GAN and use the final standalone generator model to generate new images.

8.11.1 Next

In the next tutorial, you will develop a DCGAN for generating faces and then explore the latent space using interpolation and vector arithmetic.

Chapter 9

How to Explore the Latent Space When Generating Faces

Generative Adversarial Networks, or GANs, are an architecture for training generative models, such as deep convolutional neural networks for generating images. The generative model in the GAN architecture learns to map points in the latent space to generated images. The latent space has no meaning other than the meaning applied to it via the generative model. Yet, the latent space has structure that can be explored, such as by interpolating between points and performing vector arithmetic between points in latent space which have meaningful and targeted effects on the generated images. In this tutorial, you will discover how to develop a generative adversarial network for face generation and explore the structure of the latent space and the effect on generated faces. After completing this tutorial, you will know:

- How to develop a generative adversarial network for generating faces.
- How to interpolate between points in latent space and generate images that morph from one face to another.
- How to perform vector arithmetic in latent space and achieve targeted effects in the resulting generated faces.

Let's get started.

9.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Vector Arithmetic in Latent Space
2. Large-Scale CelebFaces Dataset (CelebA)
3. How to Prepare the CelebA Faces Dataset
4. How to Develop a GAN for CelebA
5. How to Explore the Latent Space for Faces

9.2 Vector Arithmetic in Latent Space

The generator model in the GAN architecture takes a point from the latent space as input and generates a new image. The latent space itself has no meaning. Typically it is a 100-dimensional hypersphere with each variable drawn from a Gaussian distribution with a mean of zero and a standard deviation of one. Through training, the generator learns to map points onto the latent space with specific output images and this mapping will be different each time the model is trained. The latent space has structure when interpreted by the generator model, and this structure can be queried and navigated for a given model.

Typically, new images are generated using random points in the latent space. Taken a step further, points in the latent space can be constructed (e.g. all 0s, all 0.5s, or all 1s) and used as input or a query to generate a specific image. A series of points can be created on a linear path between two points in the latent space, such as two generated images. These points can be used to generate a series of images that show a transition between the two generated images. Finally, the points in the latent space can be kept and used in simple vector arithmetic to create new points in the latent space that, in turn, can be used to generate images. This is an interesting idea, as it allows for the intuitive and targeted generation of images.

The important 2015 paper by Alec Radford, et al. titled *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks* introduced a stable model configuration for training deep convolutional neural network models as part of the GAN architecture. In the paper, the authors explored the latent space for GANs fit on a number of different training datasets, most notably a dataset of celebrity faces. They demonstrated two interesting aspects. The first was the vector arithmetic with faces. For example, a face of a smiling woman minus the face of a neutral woman plus the face of a neutral man resulted in the face of a smiling man.

$$\text{smiling woman} - \text{neutral woman} + \text{neutral man} = \text{smiling man} \tag{9.1}$$

Specifically, the arithmetic was performed on the points in the latent space for the resulting faces. Actually on the average of multiple faces with a given characteristic, to provide a more robust result.

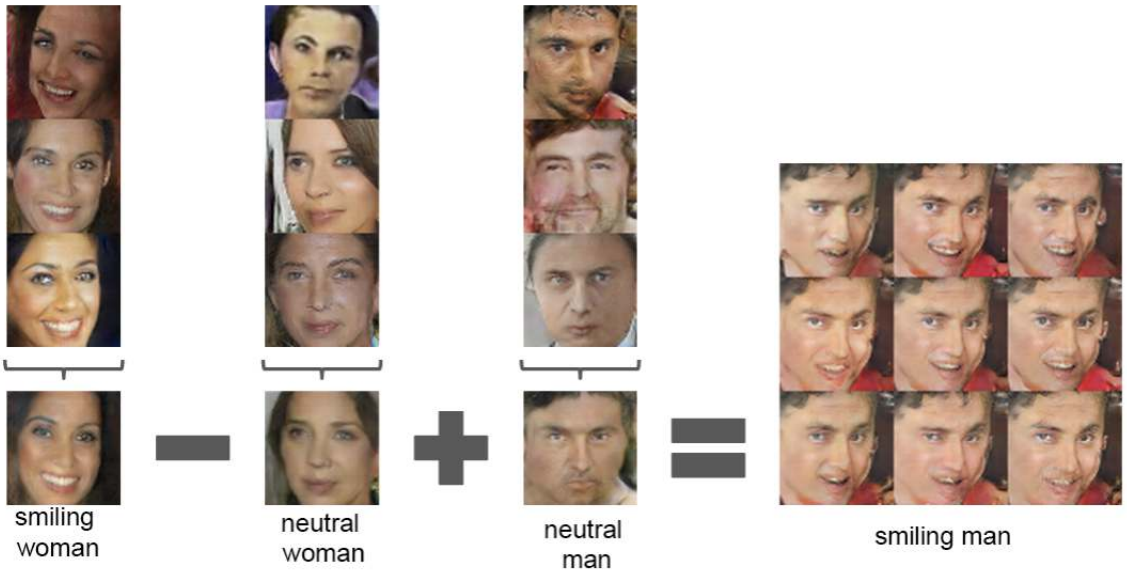

Figure 9.1: Example of Vector Arithmetic on Points in the Latent Space for Generating Faces With a GAN. Taken from: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

The second demonstration was the transition between two generated faces, specifically by creating a linear path through the latent dimension between the points that generated two faces and then generating all of the faces for the points along the path.

Figure 9.2: Example of Faces on a Path Between Two GAN Generated Faces. Taken from Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

Exploring the structure of the latent space for a GAN model is both interesting for the problem domain and helps to develop an intuition for what has been learned by the generator model. In this tutorial, we will develop a GAN for generating photos of faces, then explore the latent space for the model with vector arithmetic.

9.3 Large-Scale CelebFaces Dataset (CelebA)

The first step is to select a dataset of faces. In this tutorial, we will use the Large-scale CelebFaces Attributes Dataset, referred to as CelebA. This dataset was developed and published by Ziwei Liu, et al. for their 2015 paper titled *From Facial Parts Responses to Face Detection: A Deep Learning Approach*. The dataset provides about 200,000 photographs of celebrity faces along with annotations for what appears in given photos, such as glasses, face shape, hats, hair type, etc. As part of the dataset, the authors provide a version of each photo centered on the face and cropped to the portrait with varying sizes around 150 pixels wide and 200 pixels tall. We will use this as the basis for developing our GAN model. The dataset can be easily downloaded from the Kaggle webpage. You will require an account with Kaggle.

- CelebFaces Attributes (CelebA) Dataset. [1]

Specifically, download the file `img_align_celeba.zip` which is about 1.3 gigabytes. To do this, click on the filename on the Kaggle website and then click the download icon. The download might take a while depending on the speed of your internet connection. After downloading, unzip the archive. This will create a new directory named `img_align_celeba/` that contains all of the images with filenames like `202599.jpg` and `202598.jpg`. Next, we can look at preparing the raw images for modeling.

9.4 How to Prepare the CelebA Faces Dataset

The first step is to develop code to load the images. We can use the Pillow library to load a given image file, convert it to RGB format (if needed) and return an array of pixel data. The `load_image()` function below implements this.

```
# load an image as an rgb array
def load_image(filename):
  # load image from file
  image = Image.open(filename)
  # convert to RGB, if needed
  image = image.convert('RGB')
  # convert to array
  pixels = asarray(image)
  return pixels
```

Listing 9.1: Example of a function for loading an image as a NumPy array.

Next, we can enumerate the directory of images, load each as an array of pixels in turn, and return an array with all of the images. There are 200K images in the dataset, which is probably more than we need so we can also limit the number of images to load with an argument. The `load_faces()` function below implements this.

[1] https://www.kaggle.com/jessicali9530/celeba-dataset

```
# load images and extract faces for all images in a directory
def load_faces(directory, n_faces):
  faces = list()
  # enumerate files
  for filename in listdir(directory):
    # load the image
    pixels = load_image(directory + filename)
    # store
    faces.append(pixels)
    # stop once we have enough
    if len(faces) >= n_faces:
      break
  return asarray(faces)
```

Listing 9.2: Example of a loading all images in a directory.

Finally, once the images are loaded, we can plot them using the `imshow()` function from the Matplotlib library. The `plot_faces()` function below does this, plotting images arranged into in a square.

```
# plot a list of loaded faces
def plot_faces(faces, n):
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(faces[i])
  pyplot.show()
```

Listing 9.3: Example of a function for plotting loaded faces.

Tying this together, the complete example is listed below.

```
# load and plot faces
from os import listdir
from numpy import asarray
from PIL import Image
from matplotlib import pyplot

# load an image as an rgb numpy array
def load_image(filename):
  # load image from file
  image = Image.open(filename)
  # convert to RGB, if needed
  image = image.convert('RGB')
  # convert to array
  pixels = asarray(image)
  return pixels

# load images and extract faces for all images in a directory
def load_faces(directory, n_faces):
  faces = list()
  # enumerate files
  for filename in listdir(directory):
```

```
    # load the image
    pixels = load_image(directory + filename)
    # store
    faces.append(pixels)
    # stop once we have enough
    if len(faces) >= n_faces:
      break
  return asarray(faces)

# plot a list of loaded faces
def plot_faces(faces, n):
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(faces[i])
  pyplot.show()

# directory that contains all images
directory = 'img_align_celeba/'
# load and extract all faces
faces = load_faces(directory, 25)
print('Loaded: ', faces.shape)
# plot faces
plot_faces(faces, 5)
```

Listing 9.4: Example of loading and plotting faces from the dataset.

Running the example loads a total of 25 images from the directory, then summarizes the size of the returned array.

```
Loaded: (25, 218, 178, 3)
```

Listing 9.5: Example output from loading and summarizing the faces dataset.

Finally, the 25 images are plotted in a 5×5 square.

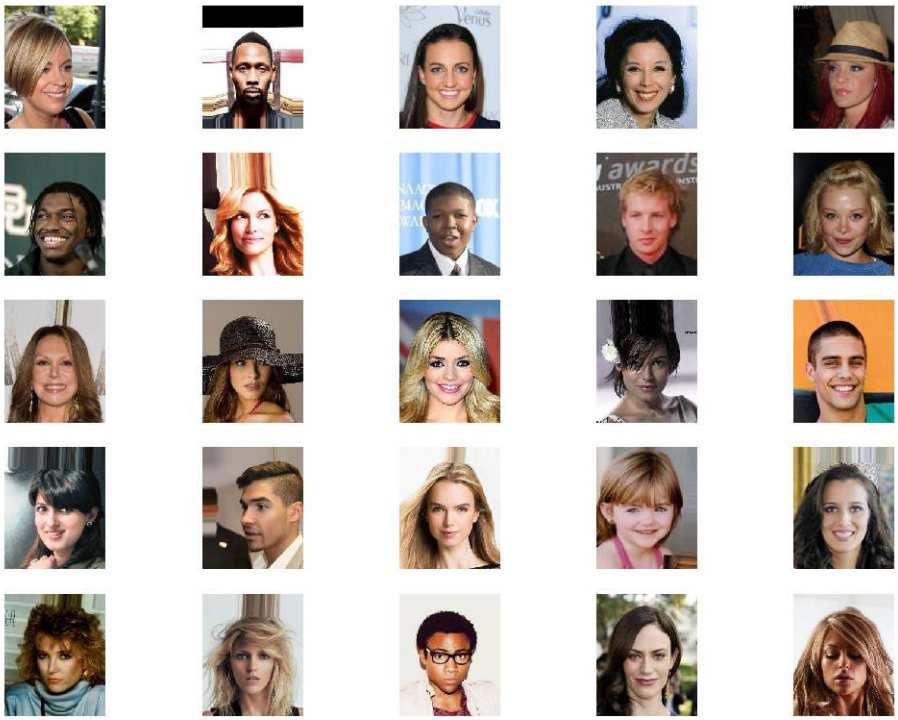

Figure 9.3: Plot of a Sample of 25 Faces from the Celebrity Faces Dataset.

When working with a GAN, it is easier to model a dataset if all of the images are small and square in shape. Further, as we are only interested in the face in each photo, and not the background, we can perform face detection and extract only the face before resizing the result to a fixed size. There are many ways to perform face detection. In this case, we will use a pre-trained Multi-Task Cascaded Convolutional Neural Network, or MTCNN. This is a state-of-the-art deep learning model for face detection, described in the 2016 paper titled *Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks*. We will use the implementation provided by Ivan de Paz Centeno in the `ipazc/mtcnn` project. This library can be installed via `pip` as follows:

```
sudo pip install mtcnn
```

Listing 9.6: Example of installing the mtcnn library with `pip`.

We can confirm that the library was installed correctly by importing the library and printing the version; for example:

```
# confirm mtcnn was installed correctly
import mtcnn
# show version
print(mtcnn.__version__)
```

Listing 9.7: Example of checking that the mtcnn library is installed correctly.

Running the example prints the current version of the library.

```
0.1.0
```

Listing 9.8: Example output from checking that the mtcnn library is installed correctly.

The MTCNN model is very easy to use. First, an instance of the MTCNN model is created, then the `detect_faces()` function can be called passing in the pixel data for one image. The result is a list of detected faces, with a bounding box defined in pixel offset values.

```
...
# prepare model
model = MTCNN()
# detect face in the image
faces = model.detect_faces(pixels)
# extract details of the face
x1, y1, width, height = faces[0]['box']
```

Listing 9.9: Example of using a MTCNN model to extract a face from a photograph.

We can update our example to extract the face from each loaded photo and resize the extracted face pixels to a fixed size. In this case, we will use the square shape of 80×80 pixels. The `extract_face()` function below implements this, taking the MTCNN model and pixel values for a single photograph as arguments and returning an $80 \times 80 \times 3$ array of pixel values with just the face, or `None` if no face was detected (which can happen rarely).

```
# extract the face from a loaded image and resize
def extract_face(model, pixels, required_size=(80, 80)):
  # detect face in the image
  faces = model.detect_faces(pixels)
  # skip cases where we could not detect a face
  if len(faces) == 0:
    return None
  # extract details of the face
  x1, y1, width, height = faces[0]['box']
  # force detected pixel values to be positive (bug fix)
  x1, y1 = abs(x1), abs(y1)
  # convert into coordinates
  x2, y2 = x1 + width, y1 + height
  # retrieve face pixels
  face_pixels = pixels[y1:y2, x1:x2]
  # resize pixels to the model size
  image = Image.fromarray(face_pixels)
  image = image.resize(required_size)
  face_array = asarray(image)
  return face_array
```

Listing 9.10: Example of a function for extracting a face from a loaded image.

We can now update the `load_faces()` function to extract the face from the loaded photo and store that in the list of faces returned.

```
# load images and extract faces for all images in a directory
def load_faces(directory, n_faces):
  # prepare model
  model = MTCNN()
  faces = list()
```

```
  # enumerate files
  for filename in listdir(directory):
    # load the image
    pixels = load_image(directory + filename)
    # get face
    face = extract_face(model, pixels)
    if face is None:
      continue
    # store
    faces.append(face)
    print(len(faces), face.shape)
    # stop once we have enough
    if len(faces) >= n_faces:
      break
  return asarray(faces)
```

Listing 9.11: Example of loading photographs and extracting faces for images in a directory.

Tying this together, the complete example is listed below. In this case, we increase the total number of loaded faces to 50,000 to provide a good training dataset for our GAN model.

```
# example of extracting and resizing faces into a new dataset
from os import listdir
from numpy import asarray
from numpy import savez_compressed
from PIL import Image
from mtcnn.mtcnn import MTCNN

# load an image as an rgb numpy array
def load_image(filename):
  # load image from file
  image = Image.open(filename)
  # convert to RGB, if needed
  image = image.convert('RGB')
  # convert to array
  pixels = asarray(image)
  return pixels

# extract the face from a loaded image and resize
def extract_face(model, pixels, required_size=(80, 80)):
  # detect face in the image
  faces = model.detect_faces(pixels)
  # skip cases where we could not detect a face
  if len(faces) == 0:
    return None
  # extract details of the face
  x1, y1, width, height = faces[0]['box']
  # force detected pixel values to be positive (bug fix)
  x1, y1 = abs(x1), abs(y1)
  # convert into coordinates
  x2, y2 = x1 + width, y1 + height
  # retrieve face pixels
  face_pixels = pixels[y1:y2, x1:x2]
  # resize pixels to the model size
  image = Image.fromarray(face_pixels)
  image = image.resize(required_size)
  face_array = asarray(image)
```

```
  return face_array

# load images and extract faces for all images in a directory
def load_faces(directory, n_faces):
  # prepare model
  model = MTCNN()
  faces = list()
  # enumerate files
  for filename in listdir(directory):
    # load the image
    pixels = load_image(directory + filename)
    # get face
    face = extract_face(model, pixels)
    if face is None:
      continue
    # store
    faces.append(face)
    print(len(faces), face.shape)
    # stop once we have enough
    if len(faces) >= n_faces:
      break
  return asarray(faces)

# directory that contains all images
directory = 'img_align_celeba/'
# load and extract all faces
all_faces = load_faces(directory, 50000)
print('Loaded: ', all_faces.shape)
# save in compressed format
savez_compressed('img_align_celeba.npz', all_faces)
```

Listing 9.12: Example of preparing the celebrity faces dataset for modeling.

Running the example may take a few minutes given the large number of faces to be loaded. At the end of the run, the array of extracted and resized faces is saved as a compressed NumPy array with the filename `img_align_celeba.npz`. The prepared dataset can then be loaded any time, as follows.

```
# load the prepared dataset
from numpy import load
# load the face dataset
data = load('img_align_celeba.npz')
faces = data['arr_0']
print('Loaded: ', faces.shape)
```

Listing 9.13: Example of loading and summarizing the prepared and saved dataset.

Loading the dataset summarizes the shape of the array, showing 50K images with the size of 80×80 pixels and three color channels.

```
Loaded: (50000, 80, 80, 3)
```

Listing 9.14: Example output from loading and summarizing the prepared and saved dataset.

We are now ready to develop a GAN model to generate faces using this dataset.

9.5 How to Develop a GAN for CelebA

In this section, we will develop a GAN for the faces dataset that we have prepared. The first step is to define the models. The discriminator model takes as input one 80×80 color image and outputs a binary prediction as to whether the image is real ($class = 1$) or fake ($class = 0$). It is implemented as a modest convolutional neural network using best practices for GAN design such as using the `LeakyReLU` activation function with a slope of 0.2, using a 2×2 stride to downsample, and the Adam version of stochastic gradient descent with a learning rate of 0.0002 and a momentum of 0.5 The `define_discriminator()` function below implements this, defining and compiling the discriminator model and returning it. The input shape of the image is parameterized as a default function argument in case you want to re-use the function for your own image data later.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(80,80,3)):
  model = Sequential()
  # normal
  model.add(Conv2D(128, (5,5), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 40x40
  model.add(Conv2D(128, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 20x30
  model.add(Conv2D(128, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 10x10
  model.add(Conv2D(128, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 5x5
  model.add(Conv2D(128, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

Listing 9.15: Example of a function for defining the discriminator model.

The generator model takes as input a point in the latent space and outputs a single 80×80 color image. This is achieved by using a fully connected layer to interpret the point in the latent space and provide sufficient activations that can be reshaped into many different (in this case 128) of a low-resolution version of the output image (e.g. 5×5). This is then upsampled four times, doubling the size and quadrupling the area of the activations each time using transpose convolutional layers. The model uses best practices such as the `LeakyReLU` activation, a kernel size that is a factor of the stride size, and a hyperbolic tangent (Tanh) activation function in the output layer.

The `define_generator()` function below defines the generator model but intentionally does not compile it as it is not trained directly, then returns the model. The size of the latent space is parameterized as a function argument.

```
# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 5x5 feature maps
  n_nodes = 128 * 5 * 5
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((5, 5, 128)))
  # upsample to 10x10
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 20x20
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 40x40
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 80x80
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # output layer 80x80x3
  model.add(Conv2D(3, (5,5), activation='tanh', padding='same'))
  return model
```

Listing 9.16: Example of a function for defining the generator model.

Next, a GAN model can be defined that combines both the generator model and the discriminator model into one larger model. This larger model will be used to train the model weights in the generator, using the output and error calculated by the discriminator model. The discriminator model is trained separately, and as such, the model weights are marked as not trainable in this larger GAN model to ensure that only the weights of the generator model are updated. This change to the trainability of the discriminator weights only has an effect when training the combined GAN model, not when training the discriminator standalone.

This larger GAN model takes as input a point in the latent space, uses the generator model to generate an image, which is fed as input to the discriminator model, then output or classified as real or fake. The `define_gan()` function below implements this, taking the already-defined generator and discriminator models as input.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model
```

Listing 9.17: Example of a function for defining the composite model.

Now that we have defined the GAN model, we need to train it. But, before we can train the model, we require input data. The first step is to load and scale the pre-processed faces dataset. The saved NumPy array can be loaded, as we did in the previous section, then the pixel values must be scaled to the range [-1,1] to match the output of the generator model. The `load_real_samples()` function below implements this, returning the loaded and scaled image data ready for modeling.

```
# load and prepare training images
def load_real_samples():
  # load the face dataset
  data = load('img_align_celeba.npz')
  X = data['arr_0']
  # convert from unsigned ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X
```

Listing 9.18: Example of a function for loading and scaling the prepared celebrity faces dataset.

We will require one batch (or a half batch) of real images from the dataset each update to the GAN model. A simple way to achieve this is to select a random sample of images from the dataset each time. The `generate_real_samples()` function below implements this, taking the prepared dataset as an argument, selecting and returning a random sample of face images and their corresponding class label for the discriminator, specifically $class = 1$, indicating that they are real images.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y
```

Listing 9.19: Example of a function selecting a random sample of real images.

Next, we need inputs for the generator model. These are random points from the latent space, specifically Gaussian distributed random variables. The `generate_latent_points()` function implements this, taking the size of the latent space as an argument and the number of points required and returning them as a batch of input samples for the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

Listing 9.20: Example of a function for generating random points in the latent space.

Next, we need to use the points in the latent space as input to the generator in order to generate new images. The `generate_fake_samples()` function below implements this, taking

the generator model and size of the latent space as arguments, then generating points in the latent space and using them as input to the generator model. The function returns the generated images and their corresponding class label for the discriminator model, specifically $class = 0$ to indicate they are fake or generated.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y
```

Listing 9.21: Example of a function for generating synthetic images.

We are now ready to fit the GAN models. The model is fit for 100 training epochs, which is arbitrary, as the model begins generating plausible faces after perhaps the first few epochs. A batch size of 128 samples is used, and each training epoch involves $\frac{50000}{128}$ or about 390 batches of real and fake samples and updates to the model. First, the discriminator model is updated for a half batch of real samples, then a half batch of fake samples, together forming one batch of weight updates. The generator is then updated via the combined GAN model. Importantly, the class label is set to 1 or real for the fake samples. This has the effect of updating the generator toward getting better at generating real samples on the next batch.

The `train()` function below implements this, taking the defined models, dataset, and size of the latent dimension as arguments and parameterizing the number of epochs and batch size with default arguments.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=128):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # update discriminator model weights
      d_loss1, _ = d_model.train_on_batch(X_real, y_real)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update discriminator model weights
      d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
        (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
    # evaluate the model performance, sometimes
```

```
    if (i+1) % 10 == 0:
      summarize_performance(i, g_model, d_model, dataset, latent_dim)
```

Listing 9.22: Example of a function for training the GAN models.

You will note that every 10 training epochs, the `summarize_performance()` function is called. There is currently no reliable way to automatically evaluate the quality of generated images. Therefore, we must generate images periodically during training and save the model at these times. This both provides a checkpoint that we can later load and use to generate images, and a way to safeguard against the training process failing, which can happen. Below defines the `summarize_performance()` and `save_plot()` functions. The `summarize_performance()` function generates samples and evaluates the performance of the discriminator on real and fake samples. The classification accuracy is reported and might provide insight into model performance. The `save_plot()` is called to create and save a plot of the generated images, and then the model is saved to a file.

```
# create and save a plot of generated images
def save_plot(examples, epoch, n=10):
  # scale from [-1,1] to [0,1]
  examples = (examples + 1) / 2.0
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i])
  # save plot to file
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()

# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
  # save plot
  save_plot(x_fake, epoch)
  # save the generator model tile file
  filename = 'generator_model_%03d.h5' % (epoch+1)
  g_model.save(filename)
```

Listing 9.23: Example of functions for summarizing the performance of the GAN models.

We can then define the size of the latent space, define all three models, and train them on the loaded face dataset.

```
...
# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim)
```

Listing 9.24: Example of configuring and running the GAN training process.

Tying all of this together, the complete example is listed below.

```
# example of a gan for generating faces
from numpy import load
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(80,80,3)):
  model = Sequential()
  # normal
  model.add(Conv2D(128, (5,5), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 40x40
  model.add(Conv2D(128, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 20x30
  model.add(Conv2D(128, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 10x10
  model.add(Conv2D(128, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 5x5
  model.add(Conv2D(128, (5,5), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
```

```
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 5x5 feature maps
  n_nodes = 128 * 5 * 5
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((5, 5, 128)))
  # upsample to 10x10
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 20x20
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 40x40
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 80x80
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # output layer 80x80x3
  model.add(Conv2D(3, (5,5), activation='tanh', padding='same'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# load and prepare training images
def load_real_samples():
  # load the face dataset
  data = load('img_align_celeba.npz')
  X = data['arr_0']
  # convert from unsigned ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X
```

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# create and save a plot of generated images
def save_plot(examples, epoch, n=10):
  # scale from [-1,1] to [0,1]
  examples = (examples + 1) / 2.0
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i])
  # save plot to file
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()

# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
```

```
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
  # save plot
  save_plot(x_fake, epoch)
  # save the generator model tile file
  filename = 'generator_model_%03d.h5' % (epoch+1)
  g_model.save(filename)

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=128):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # update discriminator model weights
      d_loss1, _ = d_model.train_on_batch(X_real, y_real)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update discriminator model weights
      d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
        (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
    # evaluate the model performance, sometimes
    if (i+1) % 10 == 0:
      summarize_performance(i, g_model, d_model, dataset, latent_dim)

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim)
```

Listing 9.25: Example training the GAN models on the prepared celebrity faces dataset.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

The loss for the discriminator on real and fake samples, as well as the loss for the generator, is reported after each batch.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
>1, 1/390, d1=0.699, d2=0.696 g=0.692
>1, 2/390, d1=0.541, d2=0.702 g=0.686
>1, 3/390, d1=0.213, d2=0.742 g=0.656
>1, 4/390, d1=0.013, d2=0.806 g=0.656
>1, 5/390, d1=0.012, d2=0.772 g=0.682
...
```

Listing 9.26: Example output of loss from training the GAN models on the prepared celebrity faces dataset.

The discriminator loss may crash down to values of 0.0 for real and generated samples. If this happens, it is an example of a training failure from which the model is likely not likely to recover and you should restart the training process.

```
...
>34, 130/390, d1=0.844, d2=8.434 g=3.450
>34, 131/390, d1=1.233, d2=12.021 g=3.541
>34, 132/390, d1=1.183, d2=15.759 g=0.000
>34, 133/390, d1=0.000, d2=15.942 g=0.006
>34, 134/390, d1=0.081, d2=15.942 g=0.000
>34, 135/390, d1=0.000, d2=15.942 g=0.000
...
```

Listing 9.27: Example output of loss indicating a possible failure mode.

Review the generated plots and select a model based on the best quality images. The model should begin to generate faces after about 30 training epochs. The faces are not completely clear, but it is obvious that they are faces, with all the right things (hair, eyes, nose, mouth) in roughly the right places.

Figure 9.4: Example of Celebrity Faces Generated by a Generative Adversarial Network.

9.6 How to Explore the Latent Space for Generated Faces

In this section, we will use our trained GAN model as the basis for exploring the latent space.

9.6.1 How to Load the Model and Generate Faces

The first step is to load the saved model and confirm that it can generate plausible faces. The model can be loaded using the `load_model()` function in the Keras API. We can then generate a number of random points in the latent space and use them as input to the loaded model to generate new faces. The faces can then be plotted. The complete example is listed below.

```
# example of loading the generator model and generating images
from numpy.random import randn
from keras.models import load_model
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
```

```
  z_input = x_input.reshape(n_samples, latent_dim)
  return z_input

# create a plot of generated images
def plot_generated(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :])
  pyplot.show()

# load model
model = load_model('generator_model_030.h5')
# generate images
latent_points = generate_latent_points(100, 25)
# generate images
X  = model.predict(latent_points)
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
# plot the result
plot_generated(X, 5)
```

Listing 9.28: Example loading the saved generator and generating multiple faces.

Running the example first loads the saved model. Then, 25 random points in the 100-dimensional latent space are created and provided to the generator model to create 25 images of faces, which are then plotted in a 5×5 grid.

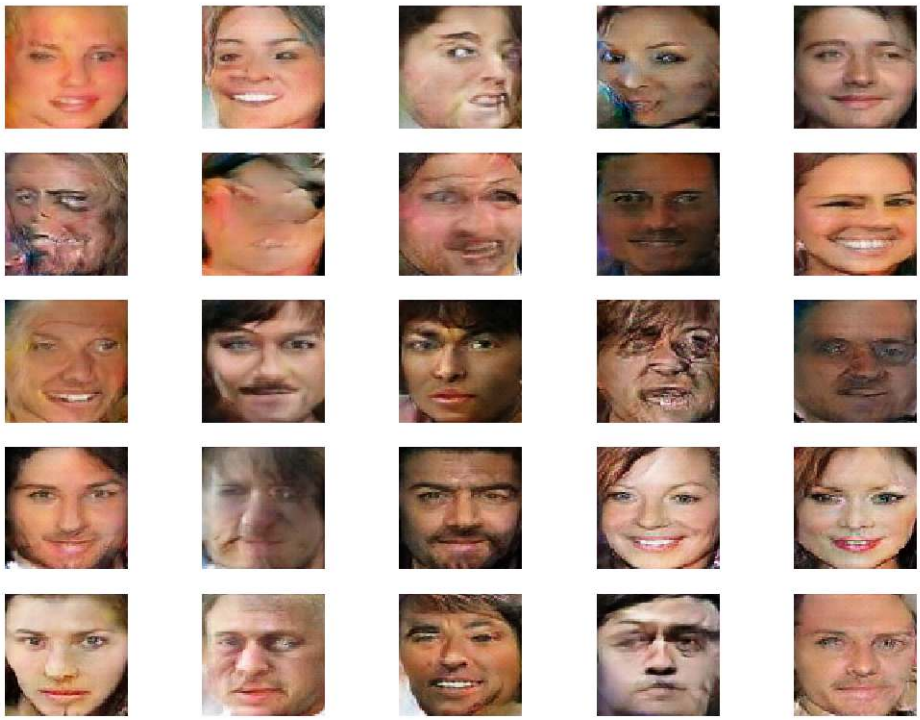

Figure 9.5: Plot of Randomly Generated Faces Using the Loaded GAN Model.

9.6.2 How to Interpolate Between Generated Faces

Next, we can create an interpolation path between two points in the latent space and generate faces along this path. The simplest interpolation we can use is a linear or uniform interpolation between two points in the latent space. We can achieve this using the `linspace()` NumPy function to calculate ratios of the contribution from two points, then enumerate these ratios and construct a vector for each ratio. The `interpolate_points()` function below implements this and returns a series of linearly interpolated vectors between two points in latent space, including the first and last point.

```
# uniform interpolation between two points in latent space
def interpolate_points(p1, p2, n_steps=10):
  # interpolate ratios between the points
  ratios = linspace(0, 1, num=n_steps)
  # linear interpolate vectors
  vectors = list()
  for ratio in ratios:
    v = (1.0 - ratio) * p1 + ratio * p2
    vectors.append(v)
  return asarray(vectors)
```

Listing 9.29: Example of a function for interpolating between points in latent space.

We can then generate two points in the latent space, perform the interpolation, then generate an image for each interpolated vector. The result will be a series of images that transition between the two original images. The example below demonstrates this for two faces.

```
# example of interpolating between generated faces
from numpy import asarray
from numpy.random import randn
from numpy import linspace
from keras.models import load_model
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  return z_input

# uniform interpolation between two points in latent space
def interpolate_points(p1, p2, n_steps=10):
  # interpolate ratios between the points
  ratios = linspace(0, 1, num=n_steps)
  # linear interpolate vectors
  vectors = list()
  for ratio in ratios:
    v = (1.0 - ratio) * p1 + ratio * p2
    vectors.append(v)
  return asarray(vectors)

# create a plot of generated images
def plot_generated(examples, n):
  # plot images
  for i in range(n):
    # define subplot
    pyplot.subplot(1, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :])
  pyplot.show()

# load model
model = load_model('generator_model_030.h5')
# generate points in latent space
pts = generate_latent_points(100, 2)
# interpolate points in latent space
interpolated = interpolate_points(pts[0], pts[1])
# generate images
X = model.predict(interpolated)
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
# plot the result
plot_generated(X, len(interpolated))
```

Listing 9.30: Example linearly interpolating between two points in latent space.

Running the example calculates the interpolation path between the two points in latent space, generates images for each, and plots the result. You can see the clear linear progression in ten steps from the first face on the left to the final face on the right.

Figure 9.6: Plot Showing the Linear Interpolation Between Two GAN Generated Faces.

We can update the example to repeat this process multiple times so we can see the transition between multiple generated faces on a single plot. The complete example is listed below.

```
# example of interpolating between generated faces
from numpy import asarray
from numpy import vstack
from numpy.random import randn
from numpy import linspace
from keras.models import load_model
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  return z_input

# uniform interpolation between two points in latent space
def interpolate_points(p1, p2, n_steps=10):
  # interpolate ratios between the points
  ratios = linspace(0, 1, num=n_steps)
  # linear interpolate vectors
  vectors = list()
  for ratio in ratios:
    v = (1.0 - ratio) * p1 + ratio * p2
    vectors.append(v)
  return asarray(vectors)

# create a plot of generated images
def plot_generated(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :])
  pyplot.show()

# load model
model = load_model('generator_model_030.h5')
# generate points in latent space
```

```
n = 20
pts = generate_latent_points(100, n)
# interpolate pairs
results = None
for i in range(0, n, 2):
  # interpolate points in latent space
  interpolated = interpolate_points(pts[i], pts[i+1])
  # generate images
  X = model.predict(interpolated)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  if results is None:
    results = X
  else:
    results = vstack((results, X))
# plot the result
plot_generated(results, 10)
```

Listing 9.31: Example of multiple cases of linearly interpolating between two points in latent space.

Running the example creates 10 different face starting points and 10 matching face endpoints, and the linear interpolation between each.

Figure 9.7: Plot Showing Multiple Linear Interpolations Between Two GAN Generated Faces.

In these cases, we have performed a linear interpolation which assumes that the latent space is a uniformly distributed hypercube. Technically, our chosen latent space is a 100-dimension hypersphere or multimodal Gaussian distribution. There is a mathematical function called the spherical linear interpolation function, or *Slerp*, that should be used when interpolating this space to ensure the curvature of the space is taken into account. For more details, I recommend reading the Issue on Linear Interpolation in Soumith Chintala's `dcgan.torch` project[2]. In that project, an implementation of the Slerp function for Python is provided that we can use as the basis for our own Slerp function, provided below:

```
# spherical linear interpolation (slerp)
def slerp(val, low, high):
  omega = arccos(clip(dot(low/norm(low), high/norm(high)), -1, 1))
  so = sin(omega)
  if so == 0:
    # L'Hopital's rule/LERP
    return (1.0-val) * low + val * high
  return sin((1.0-val)*omega) / so * low + sin(val*omega) / so * high
```

Listing 9.32: Example of a function for performing spherical linear interpolation.

This function can be called from our `interpolate_points()` function instead of performing the manual linear interpolation. The complete example with this change is listed below.

```
# example of interpolating between generated faces
from numpy import asarray
from numpy import vstack
from numpy.random import randn
from numpy import arccos
from numpy import clip
from numpy import dot
from numpy import sin
from numpy import linspace
from numpy.linalg import norm
from keras.models import load_model
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  return z_input

# spherical linear interpolation (slerp)
def slerp(val, low, high):
  omega = arccos(clip(dot(low/norm(low), high/norm(high)), -1, 1))
  so = sin(omega)
  if so == 0:
    # L'Hopital's rule/LERP
    return (1.0-val) * low + val * high
  return sin((1.0-val)*omega) / so * low + sin(val*omega) / so * high

# uniform interpolation between two points in latent space
```

[2] https://github.com/soumith/dcgan.torch/issues/14

```
def interpolate_points(p1, p2, n_steps=10):
  # interpolate ratios between the points
  ratios = linspace(0, 1, num=n_steps)
  # linear interpolate vectors
  vectors = list()
  for ratio in ratios:
    v = slerp(ratio, p1, p2)
    vectors.append(v)
  return asarray(vectors)

# create a plot of generated images
def plot_generated(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :])
  pyplot.show()

# load model
model = load_model('generator_model_030.h5')
# generate points in latent space
n = 20
pts = generate_latent_points(100, n)
# interpolate pairs
results = None
for i in range(0, n, 2):
  # interpolate points in latent space
  interpolated = interpolate_points(pts[i], pts[i+1])
  # generate images
  X = model.predict(interpolated)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  if results is None:
    results = X
  else:
    results = vstack((results, X))
# plot the result
plot_generated(results, 10)
```

Listing 9.33: Example of multiple cases of using Slerp between two points in latent space.

The result is 10 more transitions between generated faces, this time using the correct Slerp interpolation method. The difference is subtle but somehow visually more correct.

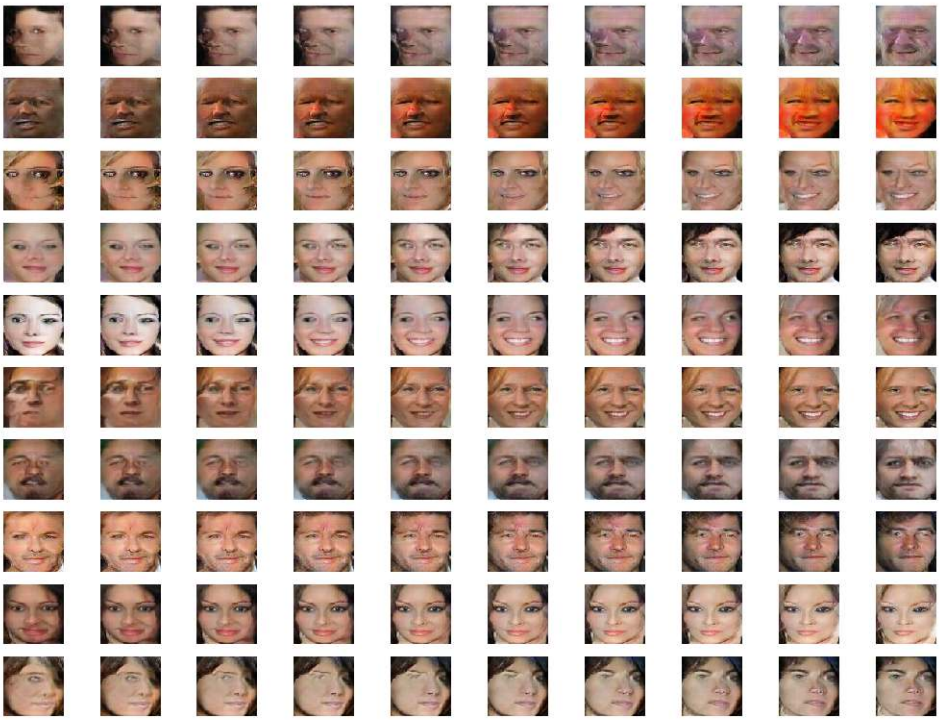

Figure 9.8: Plot Showing Multiple Spherically Linear Interpolations Between Two GAN Generated Faces.

9.6.3 How to Explore the Latent Space for Faces

Finally, we can explore the latent space by performing vector arithmetic with the generated faces. First, we must generate a large number of faces and save both the faces and their corresponding latent vectors. We can then review the plot of generated faces and select faces with features we're interested in, note their index (number), and retrieve their latent space vectors for manipulation. The example below will load the GAN model and use it to generate 100 random faces.

```
# example of loading the generator model and generating images
from numpy.random import randn
from keras.models import load_model
from matplotlib import pyplot
from numpy import savez_compressed

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  return z_input
```

```
# create a plot of generated images
def plot_generated(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :])
  pyplot.savefig('generated_faces.png')
  pyplot.close()

# load model
model = load_model('generator_model_030.h5')
# generate points in latent space
latent_points = generate_latent_points(100, 100)
# save points
savez_compressed('latent_points.npz', latent_points)
# generate images
X  = model.predict(latent_points)
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
# save plot
plot_generated(X, 10)
```

Listing 9.34: Example of generating random faces for later use.

Running the example loads the model, generates faces, and saves the latent vectors and generated faces. The latent vectors are saved to a compressed NumPy array with the filename `latent_points.npz`. The 100 generated faces are plotted in a 10×10 grid and saved in a file named `generated_faces.png`. In this case, we have a good collection of faces to work with. Each face has an index that we can use to retrieve the latent vector. For example, the first face is 1, which corresponds to the first vector in the saved array (index 0). We will perform the operation:

$$\text{smiling woman} - \text{neutral woman} + \text{neutral man} = \text{smiling man} \tag{9.2}$$

Therefore, we need three faces for each of smiling woman, neutral woman, and neutral man. In this case, we will use the following indexes in the image:

- Smiling Woman: 92, 98, 99
- Neutral Woman: 9, 21, 79
- Neutral Man: 10, 30, 45

Figure 9.9: Plot of 100 Generated Faces Used as the Basis for Vector Arithmetic with Faces.

Now that we have latent vectors to work with and a target arithmetic, we can get started. First, we can specify our preferred images and load the saved NumPy array of latent points.

```
# retrieve specific points
smiling_woman_ix = [92, 98, 99]
neutral_woman_ix = [9, 21, 79]
neutral_man_ix = [10, 30, 45]
# load the saved latent points
data = load('latent_points.npz')
points = data['arr_0']
```

Listing 9.35: Example of loading specific pre-generated latent points.

Next, we can retrieve each vector and calculate the average for each vector type (e.g. smiling woman). We could perform vector arithmetic with single images directly, but we will get a more robust result if we work with an average of a few faces with the desired property. The `average_points()` function below takes the loaded array of latent space points, retrieves each, calculates the average, and returns all of the vectors.

```
# average list of latent space vectors
def average_points(points, ix):
  # convert to zero offset points
  zero_ix = [i-1 for i in ix]
  # retrieve required points
```

```
  vectors = points[zero_ix]
  # average the vectors
  avg_vector = mean(vectors, axis=0)
  # combine original and avg vectors
  all_vectors = vstack((vectors, avg_vector))
  return all_vectors
```

Listing 9.36: Example of a function for averaging multiple pre-generated latent points.

We can now use this function to retrieve all of the required points in latent space and generate images.

```
# average vectors
smiling_woman = average_points(points, smiling_woman_ix)
neutral_woman = average_points(points, neutral_woman_ix)
neutral_man = average_points(points, neutral_man_ix)
# combine all vectors
all_vectors = vstack((smiling_woman, neutral_woman, neutral_man))
# generate images
images = model.predict(all_vectors)
# scale pixel values
images = (images + 1) / 2.0
plot_generated(images, 3, 4)
```

Listing 9.37: Example of generating faces for averaged latent points.

Finally, we can use the average vectors to perform vector arithmetic in latent space and plot the result.

```
# smiling woman - neutral woman + neutral man = smiling man
result_vector = smiling_woman[-1] - neutral_woman[-1] + neutral_man[-1]
# generate image
result_vector = expand_dims(result_vector, 0)
result_image = model.predict(result_vector)
# scale pixel values
result_image = (result_image + 1) / 2.0
pyplot.imshow(result_image[0])
pyplot.show()
```

Listing 9.38: Example of performing arithmetic in vector space and generating a face from the result.

Tying this together, the complete example is listed below.

```
# example of loading the generator model and generating images
from keras.models import load_model
from matplotlib import pyplot
from numpy import load
from numpy import mean
from numpy import vstack
from numpy import expand_dims

# average list of latent space vectors
def average_points(points, ix):
  # convert to zero offset points
  zero_ix = [i-1 for i in ix]
  # retrieve required points
  vectors = points[zero_ix]
```

```
  # average the vectors
  avg_vector = mean(vectors, axis=0)
  # combine original and avg vectors
  all_vectors = vstack((vectors, avg_vector))
  return all_vectors

# create a plot of generated images
def plot_generated(examples, rows, cols):
  # plot images
  for i in range(rows * cols):
    # define subplot
    pyplot.subplot(rows, cols, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :])
  pyplot.show()

# load model
model = load_model('generator_model_030.h5')
# retrieve specific points
smiling_woman_ix = [92, 98, 99]
neutral_woman_ix = [9, 21, 79]
neutral_man_ix = [10, 30, 45]
# load the saved latent points
data = load('latent_points.npz')
points = data['arr_0']
# average vectors
smiling_woman = average_points(points, smiling_woman_ix)
neutral_woman = average_points(points, neutral_woman_ix)
neutral_man = average_points(points, neutral_man_ix)
# combine all vectors
all_vectors = vstack((smiling_woman, neutral_woman, neutral_man))
# generate images
images = model.predict(all_vectors)
# scale pixel values
images = (images + 1) / 2.0
plot_generated(images, 3, 4)
# smiling woman - neutral woman + neutral man = smiling man
result_vector = smiling_woman[-1] - neutral_woman[-1] + neutral_man[-1]
# generate image
result_vector = expand_dims(result_vector, 0)
result_image = model.predict(result_vector)
# scale pixel values
result_image = (result_image + 1) / 2.0
pyplot.imshow(result_image[0])
pyplot.show()
```

Listing 9.39: Example of performing vector arithmetic in vector space.

Running the example first loads the points in latent space for our specific images, calculates the average of the points, and generates the faces for the points. We can see that, indeed, our selected faces were retrieved correctly and that the average of the points in the vector space captures the salient feature we are going for on each line (e.g. smiling woman, neutral woman, etc.).

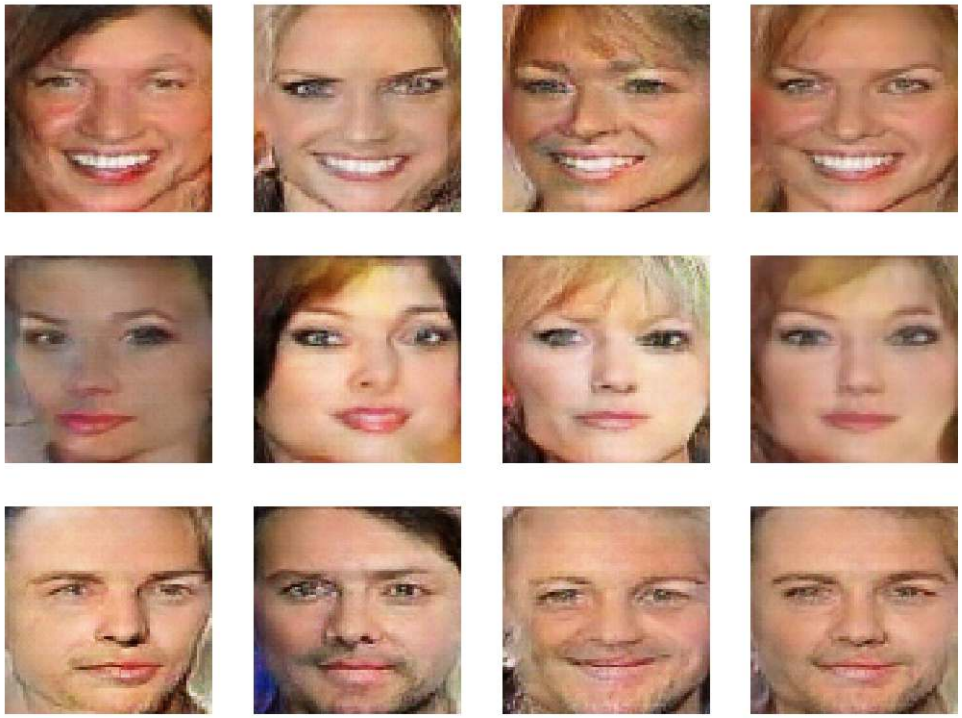

Figure 9.10: Plot of Selected Generated Faces and the Average Generated Face for Each Row.

Next, vector arithmetic is performed and the result is a smiling man, as we would expect.

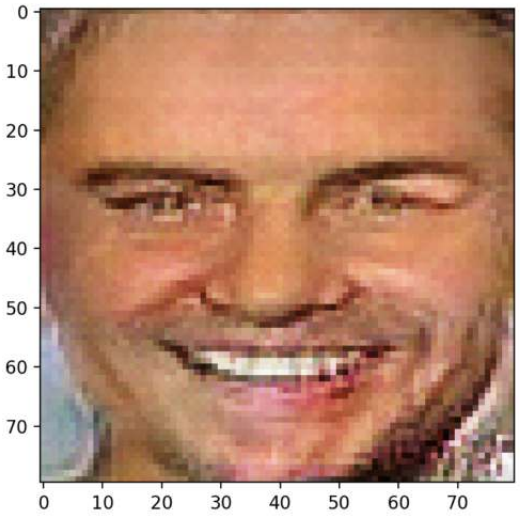

Figure 9.11: Plot of the Resulting Generated Face Based on Vector Arithmetic in Latent Space.

9.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Additional Arithmetic**. Try arithmetic with different image features or different arithmetic and review the results of the generated faces.
- **Additional Interpolation**. Try interpolating between three or more points in latent space and review the results of the generated faces.
- **Tune Model**. Update the GAN model configuration so that training is more stable and better quality faces can be generated.

If you explore any of these extensions, I'd love to know.

9.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

9.8.1 Papers

- Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks, 2016.
  https://arxiv.org/abs/1604.02878
- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.
  https://arxiv.org/abs/1511.06434
- Sampling Generative Networks, 2016.
  https://arxiv.org/abs/1609.04468

9.8.2 APIs

- Keras API.
  https://keras.io/
- How can I "freeze" Keras layers?.
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers
- MatplotLib API.
  https://matplotlib.org/api/
- `numpy.random.rand` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html
- `numpy.random.randn` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html

- `numpy.zeros` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html
- `numpy.ones` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.ones.html
- `numpy.hstack` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.hstack.html

9.8.3 Articles

- Large-scale CelebFaces Attributes (CelebA) Dataset.
  http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
- CelebFaces Attributes (CelebA) Dataset, Kaggle.
  https://www.kaggle.com/jessicali9530/celeba-dataset
- MTCNN Face Detection Project, GitHub.
  https://github.com/ipazc/mtcnn
- linear interpolation?, dcgan.torch Project, GitHub.
  https://github.com/soumith/dcgan.torch/issues/14

9.9 Summary

In this tutorial, you discovered how to develop a generative adversarial network for face generation and explore the structure of latent space and the effect on generated faces. Specifically, you learned:

- How to develop a generative adversarial network for generating faces.
- How to interpolate between points in latent space and generate images that morph from one face to another.
- How to perform vector arithmetic in latent space and achieve targeted effects in the resulting generated faces.

9.9.1 Next

In the next tutorial, you will explore common failure modes seen when training GAN models.

Chapter 10

How to Identify and Diagnose GAN Failure Modes

GANs are difficult to train. The reason they are difficult to train is that both the generator model and the discriminator model are trained simultaneously in a zero sum game. This means that improvements to one model come at the expense of the other model. The goal of training two models involves finding a point of equilibrium between the two competing concerns. It also means that every time the parameters of one of the models are updated, the nature of the optimization problem that is being solved is changed. This has the effect of creating a dynamic system. In neural network terms, the technical challenge of training two competing neural networks at the same time is that they can fail to converge.

It is important to develop an intuition for both the normal convergence of a GAN model and unusual convergence of GAN models, sometimes called failure modes. In this tutorial, we will first develop a stable GAN model for a simple image generation task in order to establish what normal convergence looks like and what to expect more generally. We will then impair the GAN models in different ways and explore a range of failure modes that you may encounter when training GAN models. These scenarios will help you to develop an intuition for what to look for or expect when a GAN model is failing to train, and ideas for what you could do about it.

After completing this tutorial, you will know:

- How to identify a stable GAN training process from the generator and discriminator loss over time.
- How to identify a mode collapse by reviewing both learning curves and generated images.
- How to identify a convergence failure by reviewing learning curves of generator and discriminator loss over time.

Let's get started.

10.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. How To Identify a Stable GAN

2. How To Identify a Mode Collapse
3. How To Identify Convergence Failure

10.2 How To Train a Stable GAN

In this section, we will train a stable GAN to generate images of a handwritten digit. Specifically, we will use the digit *'8'* from the MNIST handwritten digit dataset. The results of this model will establish both a stable GAN that can be used for later experimentation and a profile for what generated images and learning curves look like for a stable GAN training process.

The first step is to define the models. The discriminator model takes as input one 28×28 grayscale image and outputs a binary prediction as to whether the image is real ($class = 1$) or fake ($class = 0$). It is implemented as a modest convolutional neural network using best practices for GAN design such as using the `LeakyReLU` activation function with a slope of 0.2, a 2×2 stride to downsample, and the adam version of stochastic gradient descent with a learning rate of 0.0002 and a momentum of 0.5 The `define_discriminator()` function below implements this, defining and compiling the discriminator model and returning it. The input shape of the image is parameterized as a default function argument to make it clear.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # downsample to 14x14
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 7x7
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

Listing 10.1: Example of a function for defining the discriminator model.

The generator model takes as input a point in the latent space and outputs a single 28×28 grayscale image. This is achieved by using a fully connected layer to interpret the point in the latent space and provide sufficient activations that can be reshaped into many copies (in this case, 128) of a low-resolution version of the output image (e.g. 7×7). This is then upsampled two times, doubling the size and quadrupling the area of the activations each time using transpose convolutional layers. The model uses best practices such as the `LeakyReLU` activation, a kernel size that is a factor of the stride size, and a hyperbolic tangent (Tanh) activation function in the output layer.

The `define_generator()` function below defines the generator model, but intentionally does not compile it as it is not trained directly, then returns the model. The size of the latent space is parameterized as a function argument.

```
# define the standalone generator model
def define_generator(latent_dim):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # output 28x28x1
  model.add(Conv2D(1, (7,7), activation='tanh', padding='same', kernel_initializer=init))
  return model
```

Listing 10.2: Example of a function for defining the generator model.

Next, a GAN model can be defined that combines both the generator model and the discriminator model into one larger model. This larger model will be used to train the model weights in the generator, using the output and error calculated by the discriminator model. The discriminator model is trained separately, and as such, the model weights are marked as not trainable in this larger GAN model to ensure that only the weights of the generator model are updated. This change to the trainability of the discriminator weights only has an effect when training the combined GAN model, not when training the discriminator standalone.

This larger GAN model takes as input a point in the latent space, uses the generator model to generate an image, which is fed as input to the discriminator model, then output or classified as real or fake. The `define_gan()` function below implements this, taking the already defined generator and discriminator models as input.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model
```

Listing 10.3: Example of a function for defining composite model for training the generator.

Now that we have defined the GAN model, we need to train it. But, before we can train the model, we require input data. The first step is to load and scale the MNIST dataset. The whole dataset is loaded via a call to the `load_data()` Keras function, then a subset of the images are selected (about 5,000) that belong to class 8, e.g. are a handwritten depiction of the number eight. Then the pixel values must be scaled to the range [-1,1] to match the output of the generator model. The `load_real_samples()` function below implements this, returning the loaded and scaled subset of the MNIST training dataset ready for modeling.

```
# load mnist images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # select all of the examples for a given class
  selected_ix = trainy == 8
  X = X[selected_ix]
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X
```

Listing 10.4: Example of a function for loading and preparing the MNIST training dataset.

We will require one (or a half) batch of real images from the dataset each update to the GAN model. A simple way to achieve this is to select a random sample of images from the dataset each time. The `generate_real_samples()` function below implements this, taking the prepared dataset as an argument, selecting and returning a random sample of digit images, and their corresponding class label for the discriminator, specifically $class = 1$ indicating that they are real images.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y
```

Listing 10.5: Example of a function for selecting a random sample of real images.

Next, we need inputs for the generator model. These are random points from the latent space, specifically Gaussian distributed random variables. The `generate_latent_points()` function implements this, taking the size of the latent space as an argument and the number of points required, and returning them as a batch of input samples for the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
```

```
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

Listing 10.6: Example of a function for generating random points in latent space.

Next, we need to use the points in the latent space as input to the generator in order to generate new images. The `generate_fake_samples()` function below implements this, taking the generator model and size of the latent space as arguments, then generating points in the latent space and using them as input to the generator model. The function returns the generated images and their corresponding class label for the discriminator model, specifically $class = 0$ to indicate they are fake or generated.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n_samples, 1))
  return X, y
```

Listing 10.7: Example of a function for generating fake of synthesized images.

We need to record the performance of the model. Perhaps the most reliable way to evaluate the performance of a GAN is to use the generator to generate images, and then review and subjectively evaluate them. The `summarize_performance()` function below takes the generator model at a given point during training and uses it to generate 100 images in a 10×10 grid that are then plotted and saved to file. The model is also saved to file at this time, in case we would like to use it later to generate more images.

```
# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(10 * 10):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  pyplot.savefig('results_baseline/generated_plot_%03d.png' % (step+1))
  pyplot.close()
  # save the generator model
  g_model.save('results_baseline/model_%03d.h5' % (step+1))
```

Listing 10.8: Example of a function for summarizing model performance.

In addition to image quality, it is a good idea to keep track of the loss and accuracy of the model over time. The loss and classification accuracy for the discriminator for real and fake samples can be tracked for each model update, as can the loss for the generator for each update. These can then be used to create line plots of loss and accuracy at the end of the training run. The `plot_history()` function below implements this and saves the results to file.

```
# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist):
  # plot loss
  pyplot.subplot(2, 1, 1)
  pyplot.plot(d1_hist, label='d-real')
  pyplot.plot(d2_hist, label='d-fake')
  pyplot.plot(g_hist, label='gen')
  pyplot.legend()
  # plot discriminator accuracy
  pyplot.subplot(2, 1, 2)
  pyplot.plot(a1_hist, label='acc-real')
  pyplot.plot(a2_hist, label='acc-fake')
  pyplot.legend()
  # save plot to file
  pyplot.savefig('results_baseline/plot_line_plot_loss.png')
  pyplot.close()
```

Listing 10.9: Example of a function for creating and saving a plot of model loss.

We are now ready to fit the GAN model. The model is fit for 10 training epochs, which is arbitrary, as the model begins generating plausible number-8 digits after perhaps the first few epochs. A batch size of 128 samples is used, and each training epoch involves $\frac{5851}{128}$ or about 45 batches of real and fake samples and updates to the model. The model is therefore trained for 10 epochs of 45 batches, or 450 iterations. First, the discriminator model is updated for a half batch of real samples, then a half batch of fake samples, together forming one batch of weight updates. The generator is then updated via the composite GAN model. Importantly, the class label is set to 1, or real, for the fake samples. This has the effect of updating the generator toward getting better at generating real samples on the next batch.

The `train()` function below implements this, taking the defined models, dataset, and size of the latent dimension as arguments and parameterizing the number of epochs and batch size with default arguments. The generator model is saved at the end of training. The performance of the discriminator and generator models is reported each iteration. Sample images are generated and saved every epoch, and line plots of model performance are created and saved at the end of the run.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=128):
  # calculate the number of batches per epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the total iterations based on batch and epoch
  n_steps = bat_per_epo * n_epochs
  # calculate the number of samples in half a batch
  half_batch = int(n_batch / 2)
  # prepare lists for storing stats each iteration
  d1_hist, d2_hist, g_hist, a1_hist, a2_hist = list(), list(), list(), list(), list()
  # manually enumerate epochs
  for i in range(n_steps):
    # get randomly selected 'real' samples
```

```
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator model weights
    d_loss1, d_acc1 = d_model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator model weights
    d_loss2, d_acc2 = d_model.train_on_batch(X_fake, y_fake)
    # prepare points in latent space as input for the generator
    X_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    g_loss = gan_model.train_on_batch(X_gan, y_gan)
    # summarize loss on this batch
    print('>%d, d1=%.3f, d2=%.3f g=%.3f, a1=%d, a2=%d' %
      (i+1, d_loss1, d_loss2, g_loss, int(100*d_acc1), int(100*d_acc2)))
    # record history
    d1_hist.append(d_loss1)
    d2_hist.append(d_loss2)
    g_hist.append(g_loss)
    a1_hist.append(d_acc1)
    a2_hist.append(d_acc2)
    # evaluate the model performance every 'epoch'
    if (i+1) % bat_per_epo == 0:
      summarize_performance(i, g_model, latent_dim)
  plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist)
```

Listing 10.10: Example of a function for training the GAN models.

Now that all of the functions have been defined, we can create the directory where images and models will be stored (in this case `results_baseline`), create the models, load the dataset, and begin the training process.

```
...
# make folder for results
makedirs('results_baseline', exist_ok=True)
# size of the latent space
latent_dim = 50
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
print(dataset.shape)
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Listing 10.11: Example of a configuring and running the training process.

Tying all of this together, the complete example is listed below.

```
# example of training a stable gan for generating a handwritten digit
from os import makedirs
from numpy import expand_dims
```

```
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.initializers import RandomNormal
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # downsample to 14x14
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 7x7
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
```

```
  # output 28x28x1
  model.add(Conv2D(1, (7,7), activation='tanh', padding='same', kernel_initializer=init))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# load mnist images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # select all of the examples for a given class
  selected_ix = trainy == 8
  X = X[selected_ix]
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
```

```
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n_samples, 1))
  return X, y

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(10 * 10):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  pyplot.savefig('results_baseline/generated_plot_%03d.png' % (step+1))
  pyplot.close()
  # save the generator model
  g_model.save('results_baseline/model_%03d.h5' % (step+1))

# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist):
  # plot loss
  pyplot.subplot(2, 1, 1)
  pyplot.plot(d1_hist, label='d-real')
  pyplot.plot(d2_hist, label='d-fake')
  pyplot.plot(g_hist, label='gen')
  pyplot.legend()
  # plot discriminator accuracy
  pyplot.subplot(2, 1, 2)
  pyplot.plot(a1_hist, label='acc-real')
  pyplot.plot(a2_hist, label='acc-fake')
  pyplot.legend()
  # save plot to file
  pyplot.savefig('results_baseline/plot_line_plot_loss.png')
  pyplot.close()

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=128):
  # calculate the number of batches per epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the total iterations based on batch and epoch
  n_steps = bat_per_epo * n_epochs
  # calculate the number of samples in half a batch
  half_batch = int(n_batch / 2)
  # prepare lists for storing stats each iteration
  d1_hist, d2_hist, g_hist, a1_hist, a2_hist = list(), list(), list(), list(), list()
  # manually enumerate epochs
  for i in range(n_steps):
    # get randomly selected 'real' samples
```

```
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator model weights
    d_loss1, d_acc1 = d_model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator model weights
    d_loss2, d_acc2 = d_model.train_on_batch(X_fake, y_fake)
    # prepare points in latent space as input for the generator
    X_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    g_loss = gan_model.train_on_batch(X_gan, y_gan)
    # summarize loss on this batch
    print('>%d, d1=%.3f, d2=%.3f g=%.3f, a1=%d, a2=%d' %
      (i+1, d_loss1, d_loss2, g_loss, int(100*d_acc1), int(100*d_acc2)))
    # record history
    d1_hist.append(d_loss1)
    d2_hist.append(d_loss2)
    g_hist.append(g_loss)
    a1_hist.append(d_acc1)
    a2_hist.append(d_acc2)
    # evaluate the model performance every 'epoch'
    if (i+1) % bat_per_epo == 0:
      summarize_performance(i, g_model, latent_dim)
  plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist)

# make folder for results
makedirs('results_baseline', exist_ok=True)
# size of the latent space
latent_dim = 50
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
print(dataset.shape)
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Listing 10.12: Example of fitting a GAN model on part of the MNIST handwritten digit dataset.

Running the example is quick, taking approximately 10 minutes on modern hardware without a GPU.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

First, the loss and accuracy of the discriminator and loss for the generator model are reported to the console each iteration of the training loop. This is important. A stable GAN will have a discriminator loss around 0.5, typically between 0.5 and maybe as high as 0.7 or 0.8. The generator loss is typically higher and may hover around 1.0, 1.5, 2.0, or even higher.

The accuracy of the discriminator on both real and generated (fake) images will not be 50%, but should typically hover around 70% to 80%. For both the discriminator and generator, behaviors are likely to start off erratic and move around a lot before the model converges to a stable equilibrium.

```
>1, d1=0.859, d2=0.664 g=0.872, a1=37, a2=59
>2, d1=0.190, d2=1.429 g=0.555, a1=100, a2=10
>3, d1=0.094, d2=1.467 g=0.597, a1=100, a2=4
>4, d1=0.097, d2=1.315 g=0.686, a1=100, a2=9
>5, d1=0.100, d2=1.241 g=0.714, a1=100, a2=9
...
>446, d1=0.593, d2=0.546 g=1.330, a1=76, a2=82
>447, d1=0.551, d2=0.739 g=0.981, a1=82, a2=39
>448, d1=0.628, d2=0.505 g=1.420, a1=79, a2=89
>449, d1=0.641, d2=0.533 g=1.381, a1=60, a2=85
>450, d1=0.550, d2=0.731 g=1.100, a1=76, a2=42
```

Listing 10.13: Example output from fitting a GAN model on part of the MNIST handwritten digit dataset.

Line plots for loss and accuracy are created and saved at the end of the run. The figure contains two subplots. The top subplot shows line plots for the discriminator loss for real images (blue), discriminator loss for generated fake images (orange), and the generator loss for generated fake images (green). We can see that all three losses are somewhat erratic early in the run before stabilizing around epoch 100 to epoch 300. Losses remain stable after that, although the variance increases. This is an example of the normal or expected loss during training. Namely, discriminator loss for real and fake samples is about the same at or around 0.5, and loss for the generator is slightly higher between 0.5 and 2.0. If the generator model is capable of generating plausible images, then the expectation is that those images would have been generated between epochs 100 and 300 and likely between 300 and 450 as well.

The bottom subplot shows a line plot of the discriminator accuracy on real (blue) and fake (orange) images during training. We see a similar structure as the subplot of loss, namely that accuracy starts off quite different between the two image types, then stabilizes between epochs 100 to 300 at around 70% to 80%, and remains stable beyond that, although with increased variance. The time scales (e.g. number of iterations or training epochs) for these patterns and absolute values will vary across problems and types of GAN models, although the plot provides a good baseline for what to expect when training a stable GAN model.

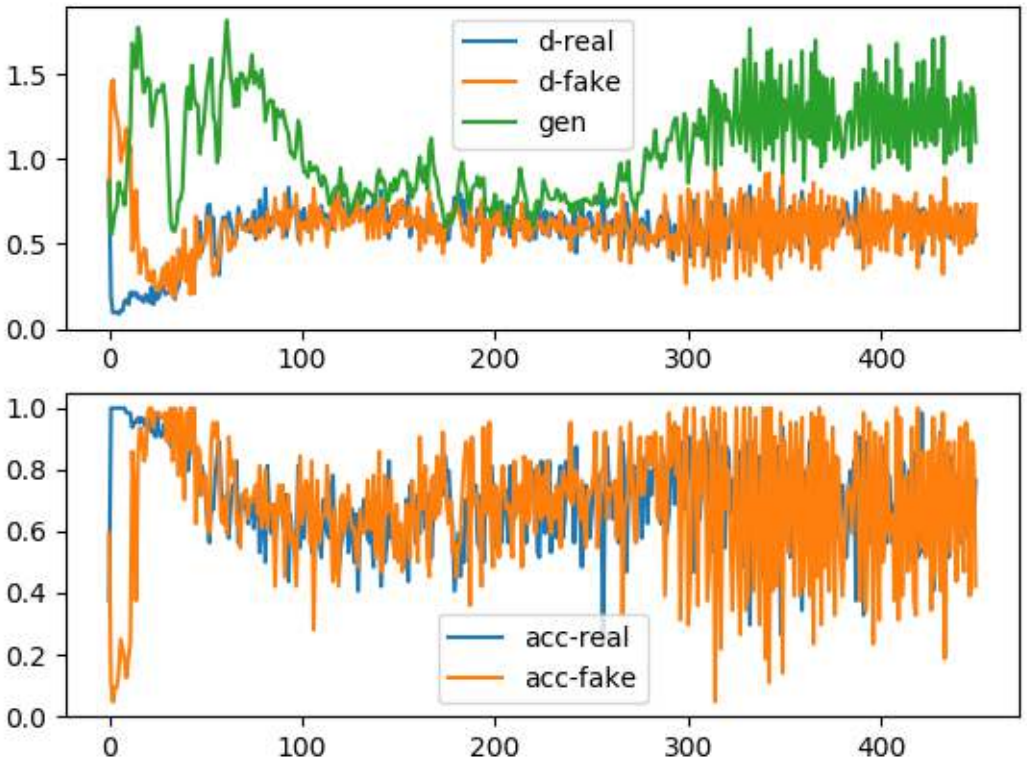

Figure 10.1: Line Plots of Loss and Accuracy for a Stable Generative Adversarial Network.

Finally, we can review samples of generated images. We are generating images using a reverse grayscale color map, meaning that the normal white figure on a background is inverted to a black figure on a white background. This was done to make the generated figures easier to review. As we might expect, samples of images generated before epoch 100 are relatively poor in quality.

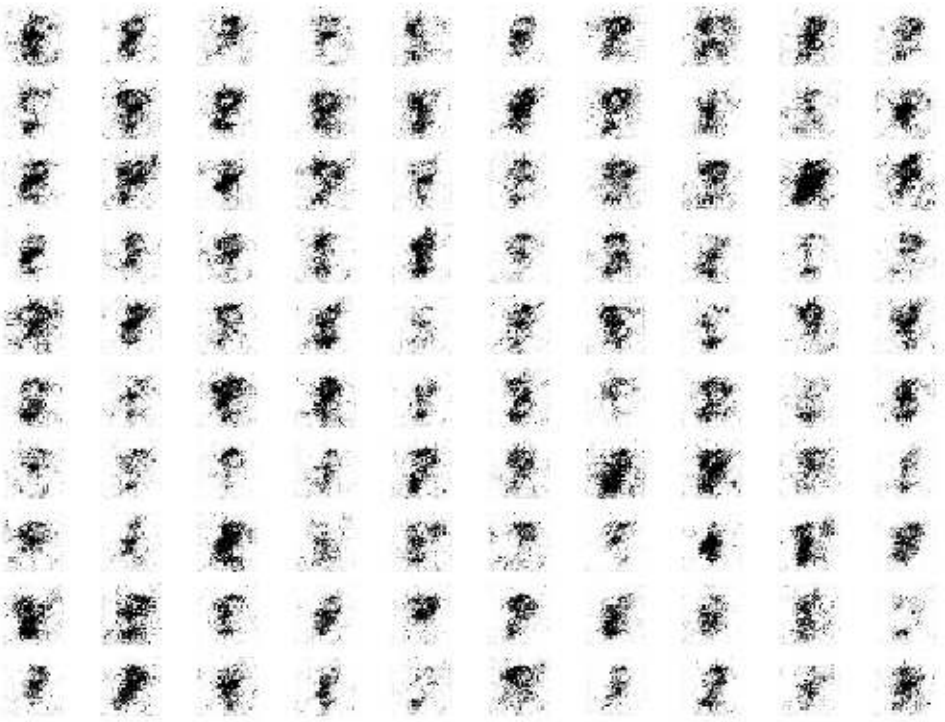

Figure 10.2: Sample of 100 Generated Images of a Handwritten Number 8 at Epoch 45 From a Stable GAN.

Samples of images generated between epochs 100 and 300 are plausible, and perhaps the best quality.

Figure 10.3: Sample of 100 Generated Images of a Handwritten Number 8 at Epoch 180 From a Stable GAN.

And samples of generated images after epoch 300 remain plausible, although perhaps have more noise, e.g. background noise.

Figure 10.4: Sample of 100 Generated Images of a Handwritten Number 8 at Epoch 450 From a Stable GAN.

These results are important, as it highlights that the quality generated can and does vary across the run, even after the training process becomes stable. More training iterations, beyond some point of training stability may or may not result in higher quality images. We can summarize these observations for stable GAN training as follows:

- Discriminator loss on real and fake images is expected to sit around 0.5.
- Generator loss on fake images is expected to sit between 0.5 and perhaps 2.0.
- Discriminator accuracy on real and fake images is expected to sit around 80%.
- Variance of generator and discriminator loss is expected to remain modest.
- The generator is expected to produce its highest quality images during a period of stability.
- Training stability may degenerate into periods of high-variance loss and corresponding lower quality generated images.

Now that we have a stable GAN model, we can look into modifying it to produce some specific failure cases. There are two failure cases that are common to see when training GAN models on new problems; they are mode collapse and convergence failure.

10.3 How To Identify a Mode Collapse

A mode collapse refers to a generator model that is only capable of generating one or a small subset of different outcomes, or modes. Here, mode refers to an output distribution, e.g. a multi-modal function refers to a function with more than one peak or optima. With a GAN generator model, a mode failure means that the vast number of points in the input latent space (e.g. hypersphere of 100 dimensions in many cases) result in one or a small subset of generated images.

> Mode collapse, also known as the scenario, is a problem that occurs when the generator learns to map several different input z values to the same output point.
>
> — *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

A mode collapse can be identified when reviewing a large sample of generated images. The images will show low diversity, with the same identical image or same small subset of identical images repeating many times. A mode collapse can also be identified by reviewing the line plot of model loss. The line plot will show oscillations in the loss over time, most notably in the generator model, as the generator model is updated and jumps from generating one mode to another model that has different loss. We can impair our stable GAN to suffer mode collapse a number of ways. Perhaps the most reliable is to restrict the size of the latent dimension directly, forcing the model to only generate a small subset of plausible outputs. Specifically, the `latent_dim` variable can be changed from 100 to 1, and the experiment re-run.

```
...
# size of the latent space
latent_dim = 1
...
```

Listing 10.14: Example of reducing the size of the latent space.

The full code listing is provided below for completeness.

```
# example of training an unstable gan for generating a handwritten digit
from os import makedirs
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.initializers import RandomNormal
from matplotlib import pyplot

# define the standalone discriminator model
```

```
def define_discriminator(in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # downsample to 14x14
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 7x7
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # output 28x28x1
  model.add(Conv2D(1, (7,7), activation='tanh', padding='same', kernel_initializer=init))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
```

```
  return model

# load mnist images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # select all of the examples for a given class
  selected_ix = trainy == 8
  X = X[selected_ix]
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X

# # select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n_samples, 1))
  return X, y

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(10 * 10):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
```

```
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  pyplot.savefig('results_collapse/generated_plot_%03d.png' % (step+1))
  pyplot.close()
  # save the generator model
  g_model.save('results_collapse/model_%03d.h5' % (step+1))

# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist):
  # plot loss
  pyplot.subplot(2, 1, 1)
  pyplot.plot(d1_hist, label='d-real')
  pyplot.plot(d2_hist, label='d-fake')
  pyplot.plot(g_hist, label='gen')
  pyplot.legend()
  # plot discriminator accuracy
  pyplot.subplot(2, 1, 2)
  pyplot.plot(a1_hist, label='acc-real')
  pyplot.plot(a2_hist, label='acc-fake')
  pyplot.legend()
  # save plot to file
  pyplot.savefig('results_collapse/plot_line_plot_loss.png')
  pyplot.close()

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=128):
  # calculate the number of batches per epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the total iterations based on batch and epoch
  n_steps = bat_per_epo * n_epochs
  # calculate the number of samples in half a batch
  half_batch = int(n_batch / 2)
  # prepare lists for storing stats each iteration
  d1_hist, d2_hist, g_hist, a1_hist, a2_hist = list(), list(), list(), list(), list()
  # manually enumerate epochs
  for i in range(n_steps):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator model weights
    d_loss1, d_acc1 = d_model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator model weights
    d_loss2, d_acc2 = d_model.train_on_batch(X_fake, y_fake)
    # prepare points in latent space as input for the generator
    X_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    g_loss = gan_model.train_on_batch(X_gan, y_gan)
    # summarize loss on this batch
    print('>%d, d1=%.3f, d2=%.3f g=%.3f, a1=%d, a2=%d' %
      (i+1, d_loss1, d_loss2, g_loss, int(100*d_acc1), int(100*d_acc2)))
    # record history
```

```
    d1_hist.append(d_loss1)
    d2_hist.append(d_loss2)
    g_hist.append(g_loss)
    a1_hist.append(d_acc1)
    a2_hist.append(d_acc2)
    # evaluate the model performance every 'epoch'
    if (i+1) % bat_per_epo == 0:
      summarize_performance(i, g_model, latent_dim)
  plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist)

# make folder for results
makedirs('results_collapse', exist_ok=True)
# size of the latent space
latent_dim = 1
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
print(dataset.shape)
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Listing 10.15: Example of fitting a GAN model with a mode collapse.

Running the example will report the loss and accuracy each step of training, as before. In this case, the loss for the discriminator sits in a sensible range, although the loss for the generator jumps up and down. The accuracy for the discriminator also shows higher values, many around 100%, meaning that for many batches, it has perfect skill at identifying real or fake examples, a bad sign for image quality or diversity.

```
>1, d1=0.963, d2=0.699 g=0.614, a1=28, a2=54
>2, d1=0.185, d2=5.084 g=0.097, a1=96, a2=0
>3, d1=0.088, d2=4.861 g=0.065, a1=100, a2=0
>4, d1=0.077, d2=4.202 g=0.090, a1=100, a2=0
>5, d1=0.062, d2=3.533 g=0.128, a1=100, a2=0
...
>446, d1=0.277, d2=0.261 g=0.684, a1=95, a2=100
>447, d1=0.201, d2=0.247 g=0.713, a1=96, a2=100
>448, d1=0.285, d2=0.285 g=0.728, a1=89, a2=100
>449, d1=0.351, d2=0.467 g=1.184, a1=92, a2=81
>450, d1=0.492, d2=0.388 g=1.351, a1=76, a2=100
```

Listing 10.16: Example output from fitting a GAN model with a mode collapse.

The figure with learning curve and accuracy line plots is created and saved. In the top subplot, we can see the loss for the generator (green) oscillating from sensible to high values over time, with a period of about 25 model updates (batches). We can also see some small oscillations in the loss for the discriminator on real and fake samples (orange and blue). In the bottom subplot, we can see that the discriminator's classification accuracy for identifying fake images remains high throughout the run. This suggests that the generator is poor at generating examples in some consistent way that makes it easy for the discriminator to identify the fake

images.

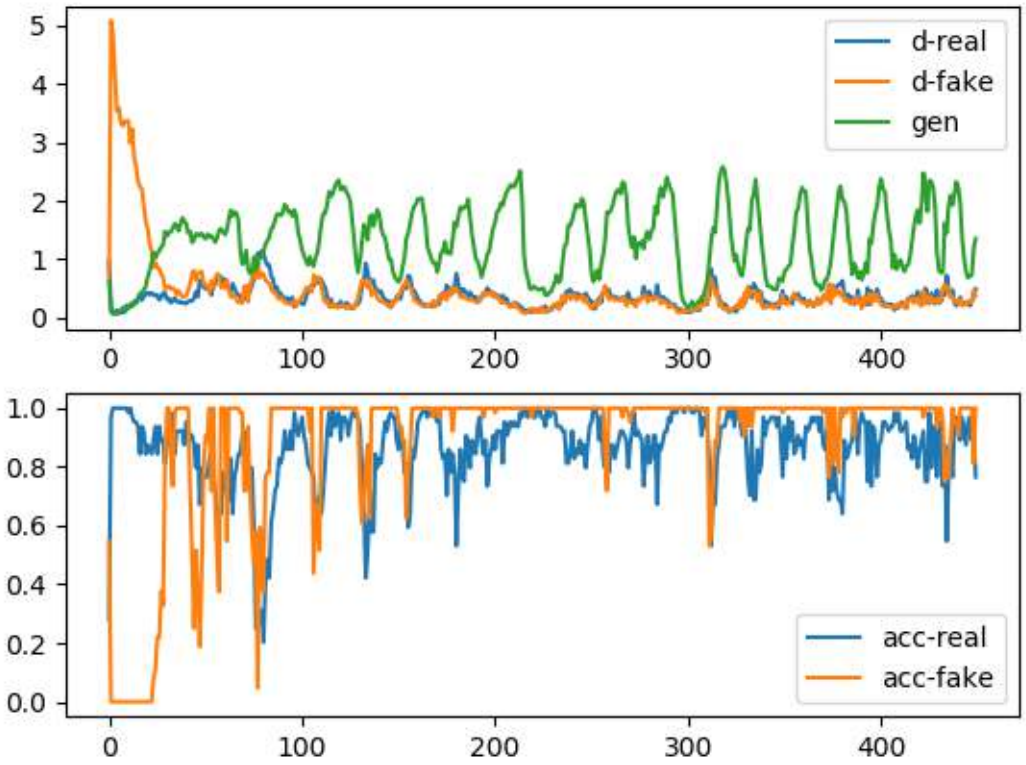

Figure 10.5: Line Plots of Loss and Accuracy for a Generative Adversarial Network With Mode Collapse.

Reviewing generated images shows the expected feature of mode collapse, namely many nearly identical generated examples, regardless of the input point in the latent space. It just so happens that we have changed the dimensionality of the latent space to be dramatically small to force this effect. I have chosen an example of generated images that helps to make this clear. There appear to be only a few types of figure-eights in the image, one leaning left, one leaning right, and one sitting up with a blur. I have drawn boxes around some of the similar examples in the image below to make this clearer.

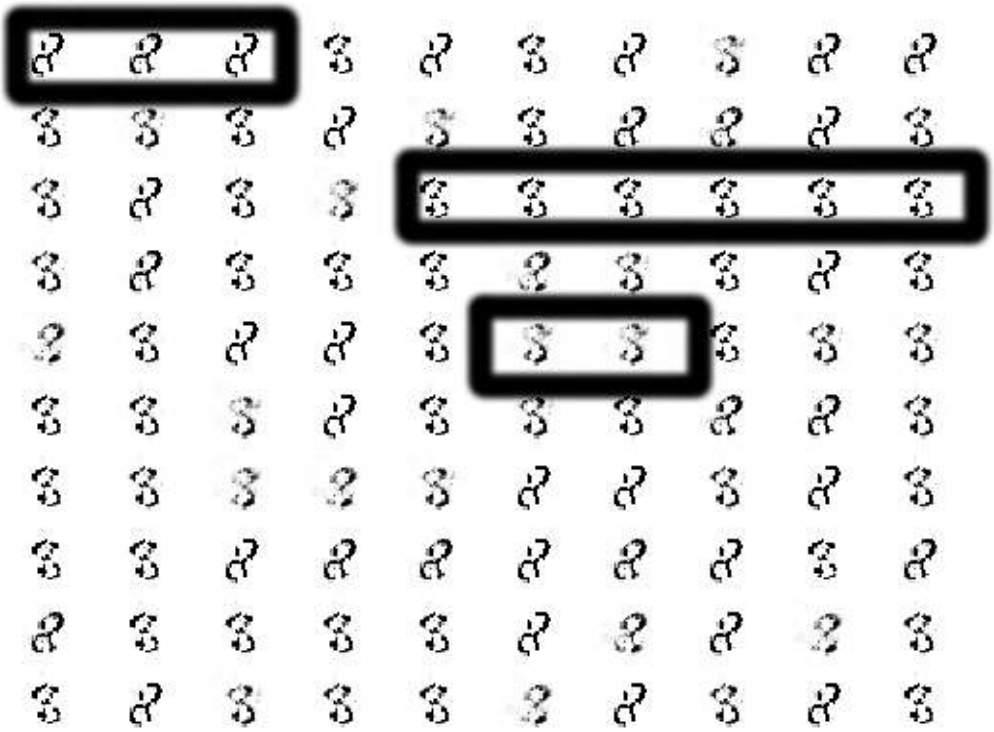

Figure 10.6: Sample of 100 Generated Images of a Handwritten Number 8 at Epoch 315 From a GAN That Has Suffered Mode Collapse.

A mode collapse is less common during training given the findings from the DCGAN model architecture and training configuration. In summary, you can identify a mode collapse as follows:

- The loss for the generator, and probably the discriminator, is expected to oscillate over time.
- The generator model is expected to generate identical output images from different points in the latent space.

10.4 How To Identify Convergence Failure

Perhaps the most common failure when training a GAN is a failure to converge. Typically, a neural network fails to converge when the model loss does not settle down during the training process. In the case of a GAN, a failure to converge refers to not finding an equilibrium between the discriminator and the generator. The likely way that you will identify this type of failure is that the loss for the discriminator has gone to zero or close to zero. In some cases, the loss of the generator may also rise and continue to rise over the same period.

This type of loss is most commonly caused by the generator outputting garbage images that the discriminator can easily identify. This type of failure might happen at the beginning of the

run and continue throughout training, at which point you should halt the training process. For some unstable GANs, it is possible for the GAN to fall into this failure mode for a number of batch updates, or even a number of epochs, and then recover. There are many ways to impair our stable GAN to achieve a convergence failure, such as changing one or both models to have insufficient capacity, changing the Adam optimization algorithm to be too aggressive, and using very large or very small kernel sizes in the models.

In this case, we will update the example to combine the real and fake samples when updating the discriminator. This simple change will cause the model to fail to converge. This change is as simple as using the `vstack()` NumPy function to combine the real and fake samples and then calling the `train_on_batch()` function to update the discriminator model. The result is also a single loss and accuracy scores, meaning that the reporting of model performance, must also be updated. The full code listing with these changes is provided below for completeness.

```
# example of training an unstable gan for generating a handwritten digit
from os import makedirs
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy import vstack
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.initializers import RandomNormal
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # downsample to 14x14
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 7x7
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

```
# define the standalone generator model
def define_generator(latent_dim):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # output 28x28x1
  model.add(Conv2D(1, (7,7), activation='tanh', padding='same', kernel_initializer=init))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# load mnist images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # select all of the examples for a given class
  selected_ix = trainy == 8
  X = X[selected_ix]
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X

# # select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
```

```
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n_samples, 1))
  return X, y

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(10 * 10):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  pyplot.savefig('results_convergence/generated_plot_%03d.png' % (step+1))
  pyplot.close()
  # save the generator model
  g_model.save('results_convergence/model_%03d.h5' % (step+1))

# create a line plot of loss for the gan and save to file
def plot_history(d_hist, g_hist, a_hist):
  # plot loss
  pyplot.subplot(2, 1, 1)
  pyplot.plot(d_hist, label='dis')
  pyplot.plot(g_hist, label='gen')
  pyplot.legend()
  # plot discriminator accuracy
  pyplot.subplot(2, 1, 2)
  pyplot.plot(a_hist, label='acc')
  pyplot.legend()
```

```
  # save plot to file
  pyplot.savefig('results_convergence/plot_line_plot_loss.png')
  pyplot.close()

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=128):
  # calculate the number of batches per epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the total iterations based on batch and epoch
  n_steps = bat_per_epo * n_epochs
  # calculate the number of samples in half a batch
  half_batch = int(n_batch / 2)
  # prepare lists for storing stats each iteration
  d_hist, g_hist, a_hist = list(), list(), list()
  # manually enumerate epochs
  for i in range(n_steps):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # combine into one batch
    X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))
    # update discriminator model weights
    d_loss, d_acc = d_model.train_on_batch(X, y)
    # prepare points in latent space as input for the generator
    X_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    g_loss = gan_model.train_on_batch(X_gan, y_gan)
    # summarize loss on this batch
    print('>%d, d=%.3f, g=%.3f, a=%d' % (i+1, d_loss, g_loss, int(100*d_acc)))
    # record history
    d_hist.append(d_loss)
    g_hist.append(g_loss)
    a_hist.append(d_acc)
    # evaluate the model performance every 'epoch'
    if (i+1) % bat_per_epo == 0:
      summarize_performance(i, g_model, latent_dim)
  plot_history(d_hist, g_hist, a_hist)

# make folder for results
makedirs('results_convergence', exist_ok=True)
# size of the latent space
latent_dim = 50
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
print(dataset.shape)
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Listing 10.17: Example of fitting a GAN model with a convergence failure.

Running the example reports loss and accuracy for each model update. A clear sign of this type of failure is the rapid drop of the discriminator loss towards zero, where it remains. This is what we see in this case.

```
>1, d=0.514, g=0.969, a=80
>2, d=0.475, g=0.395, a=74
>3, d=0.452, g=0.223, a=69
>4, d=0.302, g=0.220, a=85
>5, d=0.177, g=0.195, a=100
>6, d=0.122, g=0.200, a=100
>7, d=0.088, g=0.179, a=100
>8, d=0.075, g=0.159, a=100
>9, d=0.071, g=0.167, a=100
>10, d=0.102, g=0.127, a=100
...
>446, d=0.000, g=0.001, a=100
>447, d=0.000, g=0.001, a=100
>448, d=0.000, g=0.001, a=100
>449, d=0.000, g=0.001, a=100
>450, d=0.000, g=0.001, a=100
```

Listing 10.18: Example output from fitting a GAN model with a convergence failure.

Line plots of learning curves and classification accuracy are created. The top subplot shows the loss for the discriminator (blue) and generator (orange) and clearly shows the drop of both values down towards zero over the first 20 to 30 iterations, where it remains for the rest of the run. The bottom subplot shows the discriminator classification accuracy sitting on 100% for the same period, meaning the model is perfect at identifying real and fake images. The expectation is that there is something about fake images that makes them very easy for the discriminator to identify.

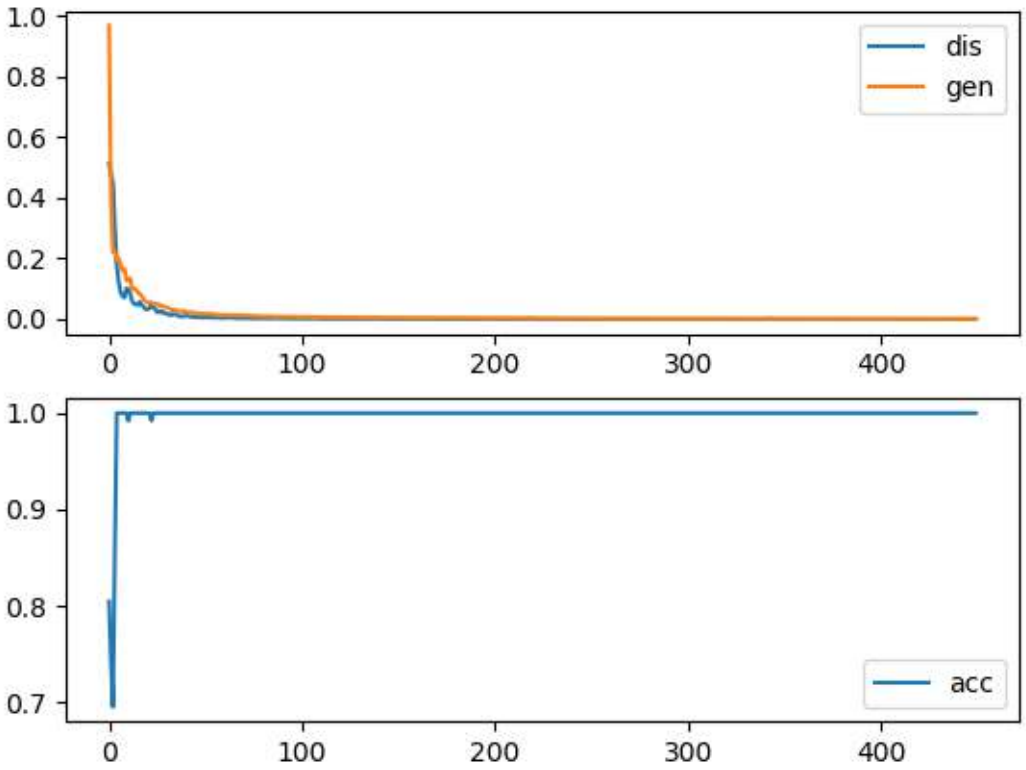

Figure 10.7: Line Plots of Loss and Accuracy for a Generative Adversarial Network With a Convergence Failure.

Finally, reviewing samples of generated images makes it clear why the discriminator is so successful. Samples of images generated at each epoch are all very low quality, showing static, perhaps with a faint figure eight in the background.

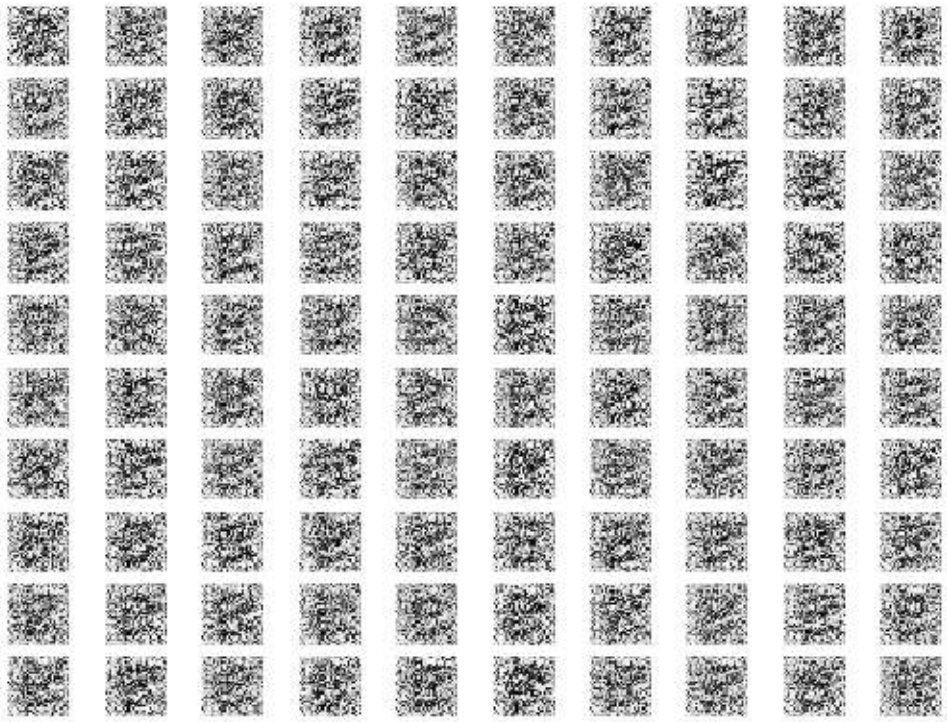

Figure 10.8: Sample of 100 Generated Images of a Handwritten Number 8 at Epoch 450 From a GAN That Has a Convergence Failure via Combined Updates to the Discriminator.

It is useful to see another example of this type of failure. In this case, the configuration of the Adam optimization algorithm can be modified to use the defaults, which in turn makes the updates to the models aggressive and causes a failure for the training process to find a point of equilibrium between training the two models. For example, the discriminator can be compiled as follows:

```
...
# compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 10.19: Example of using an aggressive configuration for stochastic gradient descent in the discriminator.

And the composite GAN model can be compiled as follows:

```
...
# compile model
model.compile(loss='binary_crossentropy', optimizer='adam')
```

Listing 10.20: Example of using an aggressive configuration for stochastic gradient descent in the composite model.

The full code listing is provided below for completeness.

```
# example of training an unstable gan for generating a handwritten digit
from os import makedirs
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.initializers import RandomNormal
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # downsample to 14x14
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 7x7
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
```

```
      kernel_initializer=init))
  model.add(LeakyReLU(alpha=0.2))
  # output 28x28x1
  model.add(Conv2D(1, (7,7), activation='tanh', padding='same', kernel_initializer=init))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  model.compile(loss='binary_crossentropy', optimizer='adam')
  return model

# load mnist images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # select all of the examples for a given class
  selected_ix = trainy == 8
  X = X[selected_ix]
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
```

```
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n_samples, 1))
  return X, y

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(10 * 10):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  pyplot.savefig('results_opt/generated_plot_%03d.png' % (step+1))
  pyplot.close()
  # save the generator model
  g_model.save('results_opt/model_%03d.h5' % (step+1))

# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist):
  # plot loss
  pyplot.subplot(2, 1, 1)
  pyplot.plot(d1_hist, label='d-real')
  pyplot.plot(d2_hist, label='d-fake')
  pyplot.plot(g_hist, label='gen')
  pyplot.legend()
  # plot discriminator accuracy
  pyplot.subplot(2, 1, 2)
  pyplot.plot(a1_hist, label='acc-real')
  pyplot.plot(a2_hist, label='acc-fake')
  pyplot.legend()
  # save plot to file
  pyplot.savefig('results_opt/plot_line_plot_loss.png')
  pyplot.close()

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=128):
  # calculate the number of batches per epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the total iterations based on batch and epoch
  n_steps = bat_per_epo * n_epochs
  # calculate the number of samples in half a batch
  half_batch = int(n_batch / 2)
  # prepare lists for storing stats each iteration
  d1_hist, d2_hist, g_hist, a1_hist, a2_hist = list(), list(), list(), list(), list()
  # manually enumerate epochs
  for i in range(n_steps):
```

```
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator model weights
    d_loss1, d_acc1 = d_model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator model weights
    d_loss2, d_acc2 = d_model.train_on_batch(X_fake, y_fake)
    # prepare points in latent space as input for the generator
    X_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    g_loss = gan_model.train_on_batch(X_gan, y_gan)
    # summarize loss on this batch
    print('>%d, d1=%.3f, d2=%.3f g=%.3f, a1=%d, a2=%d' %
      (i+1, d_loss1, d_loss2, g_loss, int(100*d_acc1), int(100*d_acc2)))
    # record history
    d1_hist.append(d_loss1)
    d2_hist.append(d_loss2)
    g_hist.append(g_loss)
    a1_hist.append(d_acc1)
    a2_hist.append(d_acc2)
    # evaluate the model performance every 'epoch'
    if (i+1) % bat_per_epo == 0:
      summarize_performance(i, g_model, latent_dim)
  plot_history(d1_hist, d2_hist, g_hist, a1_hist, a2_hist)

# make folder for results
makedirs('results_opt', exist_ok=True)
# size of the latent space
latent_dim = 50
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
print(dataset.shape)
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Listing 10.21: Example of fitting a GAN model with a different type of convergence failure.

Running the example reports the loss and accuracy for each step during training, as before. As we expected, the loss for the discriminator rapidly falls to a value close to zero, where it remains, and classification accuracy for the discriminator on real and fake examples remains at 100%.

```
>1, d1=0.728, d2=0.902 g=0.763, a1=54, a2=12
>2, d1=0.001, d2=4.509 g=0.033, a1=100, a2=0
>3, d1=0.000, d2=0.486 g=0.542, a1=100, a2=76
>4, d1=0.000, d2=0.446 g=0.733, a1=100, a2=82
>5, d1=0.002, d2=0.855 g=0.649, a1=100, a2=46
```

```
...
>446, d1=0.000, d2=0.000 g=10.410, a1=100, a2=100
>447, d1=0.000, d2=0.000 g=10.414, a1=100, a2=100
>448, d1=0.000, d2=0.000 g=10.419, a1=100, a2=100
>449, d1=0.000, d2=0.000 g=10.424, a1=100, a2=100
>450, d1=0.000, d2=0.000 g=10.427, a1=100, a2=100
```

Listing 10.22: Example output from fitting a GAN model with a different type of convergence failure.

A plot of the learning curves and accuracy from training the model with this single change is created. The plot shows that this change causes the loss for the discriminator to crash down to a value close to zero and remain there. An important difference for this case is that the loss for the generator rises quickly and continues to rise for the duration of training.

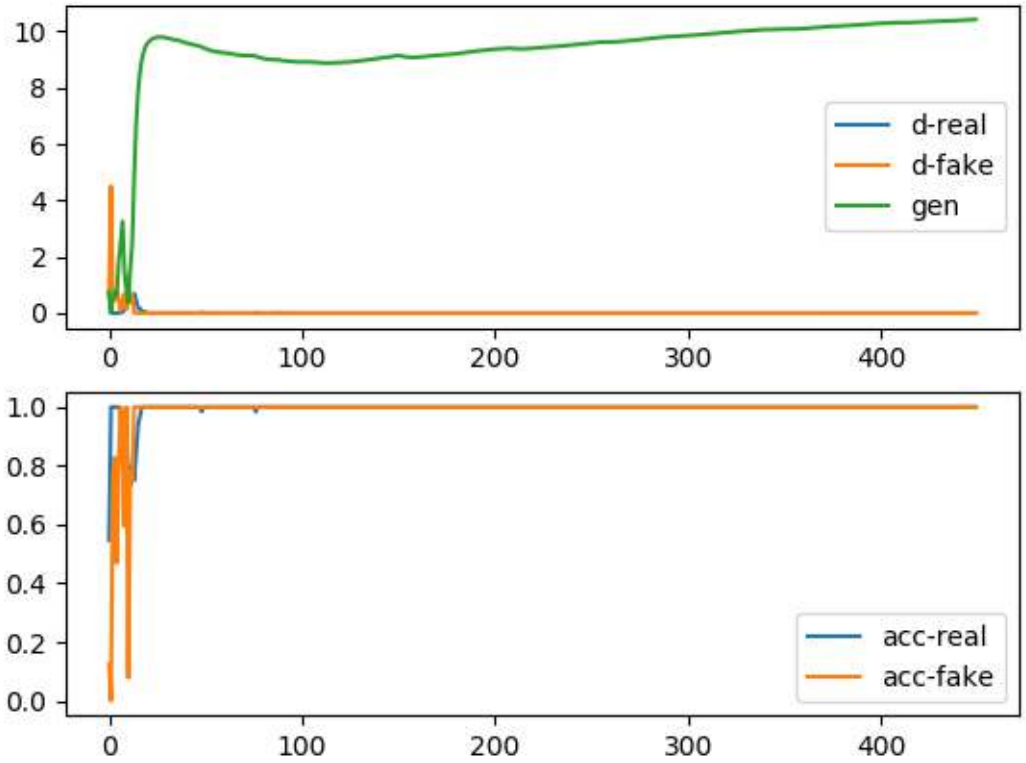

Figure 10.9: Line Plots of Loss and Accuracy for a Generative Adversarial Network With a Convergence Failure Due To Aggressive Optimization.

We can review the properties of a convergence failure as follows:

- The loss for the discriminator is expected to rapidly decrease to a value close to zero where it remains during training.
- The loss for the generator is expected to either decrease to zero or continually increase during training.

- The generator is expected to produce extremely low-quality images that are easily identified as fake by the discriminator.

10.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

10.5.1 Papers

- Generative Adversarial Networks, 2014.
https://arxiv.org/abs/1406.2661
- Tutorial: Generative Adversarial Networks, NIPS, 2016.
https://arxiv.org/abs/1701.00160
- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.
https://arxiv.org/abs/1511.06434

10.5.2 Articles

- How to Train a GAN? Tips and tricks to make GANs work.
https://github.com/soumith/ganhacks

10.6 Summary

In this tutorial, you discovered how to identify stable and unstable GAN training by reviewing examples of generated images and plots of metrics recorded during training. Specifically, you learned:

- How to identify a stable GAN training process from the generator and discriminator loss over time.
- How to identify a mode collapse by reviewing both learning curves and generated images.
- How to identify a convergence failure by reviewing learning curves of generator and discriminator loss over time.

10.6.1 Next

This was the final tutorial in this part. In the next part, you will explore techniques for evaluating GAN models.

Part III

GAN Evaluation

Overview

In this part you will discover how to evaluate generative adversarial networks based on the images that they generate. After reading the chapters in this part, you will know:

- How to use qualitative and quantitative methods to evaluate GAN models (Chapter 11).
- How to implement and interpret the inception score for evaluating GAN models (Chapter 12).
- How to implement and interpret the Frechet Inception Distance score for evaluating GAN models (Chapter 13).

Chapter 11

How to Evaluate Generative Adversarial Networks

Generative adversarial networks, or GANs for short, are an effective deep learning approach for developing generative models. Unlike other deep learning neural network models that are trained with a loss function until convergence, a GAN generator model is trained using a second model called a discriminator that learns to classify images as real or generated. Both the generator and discriminator model are trained together to maintain an equilibrium. As such, there is no objective loss function used to train the GAN generator models and no way to objectively assess the progress of the training and the relative or absolute quality of the model from loss alone.

Instead, a suite of qualitative and quantitative techniques have been developed to assess the performance of a GAN model based on the quality and diversity of the generated synthetic images. In this tutorial, you will discover techniques for evaluating generative adversarial network models based on generated synthetic images. After reading this tutorial, you will know:

- There is no objective function used when training GAN generator models, meaning models must be evaluated using the quality of the generated synthetic images.
- Manual inspection of generated images is a good starting point when getting started.
- Quantitative measures, such as the inception score and the Frechet inception distance, can be combined with qualitative assessment to provide a robust assessment of GAN models.

Let's get started.

11.1 Overview

This tutorial is divided into five parts; they are:

1. Problem with Evaluating Generator Models
2. Manual GAN Generator Evaluation
3. Qualitative GAN Generator Evaluation
4. Quantitative GAN Generator Evaluation
5. Which GAN Evaluation Scheme to Use

11.2 Problem with Evaluating Generator Models

Generative adversarial networks are a type of deep-learning-based generative model. GANs have proved to be remarkably effective at generating high-quality synthetic images in a range of problem domains. Instead of being trained directly, the generator models are trained by a second model, called the discriminator, that learns to differentiate real images from fake or generated images. As such, there is no objective function or objective measure for the generator model.

> Generative adversarial networks lack an objective function, which makes it difficult to compare performance of different models.
>
> — *Improved Techniques for Training GANs*, 2016.

This means that there is no generally agreed upon way of evaluating a given GAN generator model. This is a problem for the research and use of GANs; for example, when:

- Choosing a final GAN generator model during a training run.
- Choosing generated images to demonstrate the capability of a GAN generator model.
- Comparing GAN model architectures.
- Comparing GAN model configurations.

The objective evaluation of GAN generator models remains an open problem.

> While several measures have been introduced, as of yet, there is no consensus as to which measure best captures strengths and limitations of models and should be used for fair model comparison.
>
> — *Pros and Cons of GAN Evaluation Measures*, 2018.

As such, GAN generator models are evaluated based on the quality of the images generated, often in the context of the target problem domain.

11.3 Manual GAN Generator Evaluation

Many GAN practitioners fall back to the evaluation of GAN generators via the manual assessment of images synthesized by a generator model. This involves using the generator model to create a batch of synthetic images, then evaluating the quality and diversity of the images in relation to the target domain. This may be performed by the researcher or practitioner themselves.

> Visual examination of samples by humans is one of the common and most intuitive ways to evaluate GANs.
>
> — *Pros and Cons of GAN Evaluation Measures*, 2018.

The generator model is trained iteratively over many training epochs. As there is no objective measure of model performance, we cannot know when the training process should stop and when a final model should be saved for later use. Therefore, it is common to use the current state of the model during training to generate a large number of synthetic images and to save the current state of the generator used to generate the images. This allows for the post-hoc evaluation of each saved generator model via its generated images. One training epoch refers to one cycle through the images in the training dataset used to update the model. Models may be saved systematically across training epochs, such as every one, five, ten, or more training epochs. Although manual inspection is the simplest method of model evaluation, it has many limitations, including:

- It is subjective, including biases of the reviewer about the model, its configuration, and the project objective.
- It requires knowledge of what is realistic and what is not for the target domain.
- It is limited to the number of images that can be reviewed in a reasonable time.

> ... evaluating the quality of generated images with human vision is expensive and cumbersome, biased [...] difficult to reproduce, and does not fully reflect the capacity of models.
>
> — *Pros and Cons of GAN Evaluation Measures*, 2018.

The subjective nature almost certainty leads to biased model selection and cherry picking and should not be used for final model selection on non-trivial projects. Nevertheless, it is a starting point for practitioners when getting familiar with the technique. Thankfully, more sophisticated GAN generator evaluation methods have been proposed and adopted. For a thorough survey, see the 2018 paper titled *Pros and Cons of GAN Evaluation Measures*. This paper divides GAN generator model evaluation into qualitative and quantitative measures, and we will review some of them in the following sections using this division.

11.4 Qualitative GAN Generator Evaluation

Qualitative measures are those measures that are not numerical and often involve human subjective evaluation or evaluation via comparison. Five qualitative techniques for evaluating GAN generator models are listed below.

1. Nearest Neighbors.
2. Rapid Scene Categorization.
3. Rating and Preference Judgment.
4. Evaluating Mode Drop and Mode Collapse.
5. Investigating and Visualizing the Internals of Networks.

Perhaps the most used qualitative GAN generator model is an extension of the manual inspection of images referred to as *Rating and Preference Judgment*.

> These types of experiments ask subjects to rate models in terms of the fidelity of their generated images.
>
> — *Pros and Cons of GAN Evaluation Measures*, 2018.

This is where human judges are asked to rank or compare examples of real and generated images from the domain. The *Rapid Scene Categorization* method is generally the same, although images are presented to human judges for a very limited amount of time, such as a fraction of a second, and classified as real or fake. Images are often presented in pairs and the human judge is asked which image they prefer, e.g. which image is more realistic. A score or rating is determined based on the number of times a specific model generated images that won such tournaments. Variance in the judging is reduced by averaging the ratings across multiple different human judges. This is a labor-intensive exercise, although costs can be lowered by using a crowdsourcing platform like Amazon's Mechanical Turk, and efficiency can be increased by using a web interface.

> One intuitive metric of performance can be obtained by having human annotators judge the visual quality of samples. We automate this process using Amazon Mechanical Turk [...] using the web interface [...] which we use to ask annotators to distinguish between generated data and real data.
>
> — *Improved Techniques for Training GANs*, 2016.

A major downside of the approach is that the performance of human judges is not fixed and can improve over time. This is especially the case if they are given feedback, such as clues on how to detect generated images.

> By learning from such feedback, annotators are better able to point out the flaws in generated images, giving a more pessimistic quality assessment.
>
> — *Improved Techniques for Training GANs*, 2016.

Another popular approach for subjectively summarizing generator performance is *Nearest Neighbors*. This involves selecting examples of real images from the domain and locating one or more most similar generated images for comparison. Distance measures, such as Euclidean distance between the image pixel data, is often used for selecting the most similar generated images. The nearest neighbor approach is useful to give context for evaluating how realistic the generated images happen to be.

11.5 Quantitative GAN Generator Evaluation

Quantitative GAN generator evaluation refers to the calculation of specific numerical scores used to summarize the quality of generated images. Twenty-four quantitative techniques for evaluating GAN generator models are listed below.

1. Average Log-likelihood
2. Coverage Metric
3. Inception Score (IS)
4. Modified Inception Score (m-IS)
5. Mode Score
6. AM Score
7. Frechet Inception Distance (FID)
8. Maximum Mean Discrepancy (MMD)
9. The Wasserstein Critic
10. Birthday Paradox Test
11. Classifier Two-sample Tests (C2ST)
12. Classification Performance
13. Boundary Distortion
14. Number of Statistically-Different Bins (NDB)
15. Image Retrieval Performance
16. Generative Adversarial Metric (GAM)
17. Tournament Win Rate and Skill Rating
18. Normalized Relative Discriminative Score (NRDS)
19. Adversarial Accuracy and Adversarial Divergence
20. Geometry Score
21. Reconstruction Error
22. Image Quality Measures (SSIM, PSNR and Sharpness Difference)
23. Low-level Image Statistics
24. Precision, Recall and F1 Score

The original 2014 GAN paper by Goodfellow, et al. titled *Generative Adversarial Networks* used the *Average Log-likelihood* method, also referred to as kernel estimation or Parzen density estimation, to summarize the quality of the generated images. This involves the challenging approach of estimating how well the generator captures the probability distribution of images in the domain and has generally been found not to be effective for evaluating GANs.

> Parzen windows estimation of likelihood favors trivial models and is irrelevant to visual fidelity of samples. Further, it fails to approximate the true likelihood in high dimensional spaces or to rank models
>
> — *Pros and Cons of GAN Evaluation Measures*, 2018.

Two widely adopted metrics for evaluating generated images are the Inception Score and the Frechet Inception Distance. The inception score was proposed by Tim Salimans, et al. in their 2016 paper titled *Improved Techniques for Training GANs*.

> Inception Score (IS) [...] is perhaps the most widely adopted score for GAN evaluation.
>
> — *Pros and Cons of GAN Evaluation Measures*, 2018.

Calculating the inception score involves using a pre-trained deep learning neural network model for image classification to classify the generated images. Specifically, the Inception v3 model described by Christian Szegedy, et al. in their 2015 paper titled *Rethinking the Inception Architecture for Computer Vision*. The reliance on the inception model gives the inception score its name. A large number of generated images are classified using the model. Specifically, the probability of the image belonging to each class is predicted. The probabilities are then summarized in the score to both capture how much each image looks like a known class and how diverse the set of images are across the known classes. A higher inception score indicates better-quality generated images. The Frechet Inception Distance, or FID, score was proposed and used by Martin Heusel, et al. in their 2017 paper titled *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. The score was proposed as an improvement over the existing Inception Score.

> FID performs well in terms of discriminability, robustness and computational efficiency. [...] It has been shown that FID is consistent with human judgments and is more robust to noise than IS.
>
> — *Pros and Cons of GAN Evaluation Measures*, 2018.

Like the inception score, the FID score uses the inception v3 model. Specifically, the coding layer of the model (the last pooling layer prior to the output classification of images) is used to capture computer vision specific features of an input image. These activations are calculated for a collection of real and generated images. The activations for each real and generated image are summarized as a multivariate Gaussian and the distance between these two distributions is then calculated using the Frechet distance, also called the Wasserstein-2 distance. A lower FID score indicates more realistic images that match the statistical properties of real images.

11.6 Which GAN Evaluation Scheme to Use

When getting started, it is a good idea to start with the manual inspection of generated images in order to evaluate and select generator models.

- Manual Image Inspection

Developing GAN models is complex enough for beginners. Manual inspection can get you a long way while refining your model implementation and testing model configurations. Once your confidence in developing GAN models improves, both the Inception Score and the Frechet Inception Distance can be used to quantitatively summarize the quality of generated images. There is no single best and agreed upon measure, although, these two measures come close.

> As of yet, there is no consensus regarding the best score. Different scores assess various aspects of the image generation process, and it is unlikely that a single score can cover all aspects. Nevertheless, some measures seem more plausible than others (e.g. FID score).
>
> — *Pros and Cons of GAN Evaluation Measures*, 2018.

These measures capture the quality and diversity of generated images, both alone (former) and compared to real images (latter) and are widely used.

- Inception Score (see Chapter 12).
- Frechet Inception Distance (see Chapter 13).

Both measures are easy to implement and calculate on batches of generated images. As such, the practice of systematically generating images and saving models during training can and should continue to be used to allow post-hoc model selection. The nearest neighbor method can be used to qualitatively summarize generated images. Human-based ratings and preference judgments can also be used if needed via a crowdsourcing platform.

- Nearest Neighbors
- Rating and Preference Judgment

11.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Generative Adversarial Networks, 2014.
  https://arxiv.org/abs/1406.2661
- Pros and Cons of GAN Evaluation Measures, 2018.
  https://arxiv.org/abs/1802.03446
- Improved Techniques for Training GANs, 2016.
  https://arxiv.org/abs/1606.03498
- GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium, 2017.
  https://arxiv.org/abs/1706.08500
- Are GANs Created Equal? A Large-Scale Study, 2017.
  https://arxiv.org/abs/1711.10337

11.8 Summary

In this tutorial, you discovered techniques for evaluating generative adversarial network models based on generated synthetic images. Specifically, you learned:

- There is no objective function used when training GAN generator models, meaning models must be evaluated using the quality of the generated synthetic images.
- Manual inspection of generated images is a good starting point when getting started.
- Quantitative measures, such as the inception score and the Frechet inception distance, can be combined with qualitative assessment to provide a robust assessment of GAN models.

11.8.1 Next

In the next tutorial, you will discover the inception score and how to implement it from scratch and interpret the results.

Chapter 12

How to Implement the Inception Score

Generative Adversarial Networks, or GANs for short, is a deep learning neural network architecture for training a generator model for generating synthetic images. A problem with generative models is that there is no objective way to evaluate the quality of the generated images. As such, it is common to periodically generate and save images during the model training process and use subjective human evaluation of the generated images in order to both evaluate the quality of the generated images and to select a final generator model. Many attempts have been made to establish an objective measure of generated image quality. An early and somewhat widely adopted example of an objective evaluation method for generated images is the Inception Score, or IS. In this tutorial, you will discover the inception score for evaluating the quality of generated images. After completing this tutorial, you will know:

- How to calculate the inception score and the intuition behind what it measures.
- How to implement the inception score in Python with NumPy and the Keras deep learning library.
- How to calculate the inception score for small images such as those in the CIFAR-10 dataset.

Let's get started.

12.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What Is the Inception Score?
2. How to Calculate the Inception Score
3. How to Implement the Inception Score With NumPy
4. How to Implement the Inception Score With Keras
5. Problems With the Inception Score

12.2 What Is the Inception Score?

The Inception Score, or IS for short, is an objective metric for evaluating the quality of generated images, specifically synthetic images output by generative adversarial network models. The inception score was proposed by Tim Salimans, et al. in their 2016 paper titled *Improved Techniques for Training GANs*. In the paper, the authors use a crowd-sourcing platform (Amazon Mechanical Turk) to evaluate a large number of GAN generated images. They developed the inception score as an attempt to remove the subjective human evaluation of images. The authors discover that their scores correlated well with the subjective evaluation.

> As an alternative to human annotators, we propose an automatic method to evaluate samples, which we find to correlate well with human evaluation ...
>
> — *Improved Techniques for Training GANs*, 2016.

The inception score involves using a pre-trained deep learning neural network model for image classification to classify the generated images. Specifically, the Inception v3 model described by Christian Szegedy, et al. in their 2015 paper titled *Rethinking the Inception Architecture for Computer Vision*. The reliance on the inception model gives the inception score its name. A large number of generated images are classified using the model. Specifically, the probability of the image belonging to each class is predicted. These predictions are then summarized into the inception score. The score seeks to capture two properties of a collection of generated images:

- **Image Quality**. Do images look like a specific object?
- **Image Diversity**. Is a wide range of objects generated?

The inception score has a lowest value of 1.0 and a highest value of the number of classes supported by the classification model; in this case, the Inception v3 model supports the 1,000 classes of the ILSVRC 2012 dataset, and as such, the highest inception score on this dataset is 1,000. The CIFAR-10 dataset is a collection of 50,000 images divided into 10 classes of objects. The original paper that introduces the inception calculated the score on the real CIFAR-10 training dataset, achieving a result of 11.24 +/- 0.12. Using the GAN model also introduced in their paper, they achieved an inception score of 8.09 +/- .07 when generating synthetic images for this dataset.

12.3 How to Calculate the Inception Score

The inception score is calculated by first using a pre-trained Inception v3 model to predict the class probabilities for each generated image. These are conditional probabilities, e.g. class label conditional on the generated image. Images that are classified strongly as one class over all other classes indicate a high quality. As such, the conditional probability of all generated images in the collection should have a low entropy.

> Images that contain meaningful objects should have a conditional label distribution $p(y|x)$ with low entropy.
>
> — *Improved Techniques for Training GANs*, 2016.

The entropy is calculated as the negative sum of each observed probability multiplied by the log of the probability. The intuition here is that large probabilities have less information than small probabilities.

$$entropy = -\sum p_i \times \log(p_i) \tag{12.1}$$

The conditional probability captures our interest in image quality. To capture our interest in a variety of images, we use the marginal probability. This is the probability distribution of all generated images. We, therefore, would prefer the integral of the marginal probability distribution to have a high entropy.

> Moreover, we expect the model to generate varied images, so the marginal integral $p(y|x = G(z))dz$ should have high entropy.
>
> — *Improved Techniques for Training GANs*, 2016.

These elements are combined by calculating the Kullback-Leibler divergence, or KL divergence (relative entropy), between the conditional and marginal probability distributions. Calculating the divergence between two distributions is written using the || operator, therefore we can say we are interested in the KL divergence between C for conditional and M for marginal distributions or:

$$KL(C||M) \tag{12.2}$$

Specifically, we are interested in the average of the KL divergence for all generated images.

> Combining these two requirements, the metric that we propose is: $exp(E_x KL(p(y|x)||p(y)))$.
>
> — *Improved Techniques for Training GANs*, 2016.

We don't need to translate the calculation of the inception score. Thankfully, the authors of the paper also provide source code on GitHub that includes an implementation of the inception score. The calculation of the score assumes a large number of images for a range of objects, such as 50,000. The images are split into 10 groups, e.g 5,000 images per group, and the inception score is calculated on each group of images, then the average and standard deviation of the score is reported.

The calculation of the inception score on a group of images involves first using the inception v3 model to calculate the conditional probability for each image ($p(y|x)$). The marginal probability is then calculated as the average of the conditional probabilities for the images in the group ($p(y)$). The KL divergence is then calculated for each image as the conditional probability multiplied by the log of the conditional probability minus the log of the marginal probability.

$$\text{KL divergence} = p(y|x) \times (\log(p(y|x)) - \log(p(y))) \tag{12.3}$$

The KL divergence is then summed over all images and averaged over all classes and the exponent of the result is calculated to give the final score. This defines the official inception score implementation used when reported in most papers that use the score, although variations on how to calculate the score do exist.

12.4 How to Implement the Inception Score With NumPy

Implementing the calculation of the inception score in Python with NumPy arrays is straightforward. First, let's define a function that will take a collection of conditional probabilities and calculate the inception score. The `calculate_inception_score()` function listed below implements the procedure. One small change is the introduction of an epsilon (a tiny number close to zero) when calculating the log probabilities to avoid blowing up when trying to calculate the log of a zero probability. This is probably not needed in practice (e.g. with real generated images) but is useful here and good practice when working with log probabilities.

```
# calculate the inception score for p(y|x)
def calculate_inception_score(p_yx, eps=1E-16):
  # calculate p(y)
  p_y = expand_dims(p_yx.mean(axis=0), 0)
  # kl divergence for each image
  kl_d = p_yx * (log(p_yx + eps) - log(p_y + eps))
  # sum over classes
  sum_kl_d = kl_d.sum(axis=1)
  # average over images
  avg_kl_d = mean(sum_kl_d)
  # undo the logs
  is_score = exp(avg_kl_d)
  return is_score
```

Listing 12.1: Example of a function for calculating the inception score for probabilities.

We can then test out this function to calculate the inception score for some contrived conditional probabilities. We can imagine the case of three classes of image and a perfect confident prediction for each class for three images.

```
# conditional probabilities for high quality images
p_yx = asarray([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])
```

Listing 12.2: Example of confident conditional probabilities.

We would expect the inception score for this case to be 3.0 (or very close to it). This is because we have the same number of images for each image class (one image for each of the three classes) and each conditional probability is maximally confident. The complete example for calculating the inception score for these probabilities is listed below.

```
# calculate inception score in numpy
from numpy import asarray
from numpy import expand_dims
from numpy import log
from numpy import mean
from numpy import exp

# calculate the inception score for p(y|x)
def calculate_inception_score(p_yx, eps=1E-16):
  # calculate p(y)
  p_y = expand_dims(p_yx.mean(axis=0), 0)
  # kl divergence for each image
  kl_d = p_yx * (log(p_yx + eps) - log(p_y + eps))
  # sum over classes
  sum_kl_d = kl_d.sum(axis=1)
  # average over images
```

```
  avg_kl_d = mean(sum_kl_d)
  # undo the logs
  is_score = exp(avg_kl_d)
  return is_score

# conditional probabilities for high quality images
p_yx = asarray([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])
score = calculate_inception_score(p_yx)
print(score)
```

Listing 12.3: Example of calculating the inception score for confident probabilities.

Running the example gives the expected score of 3.0 (or a number extremely close).

```
2.999999999999999
```

Listing 12.4: Example output from calculating the inception score for confident probabilities.

We can also try the worst case. This is where we still have the same number of images for each class (one for each of the three classes), but the objects are unknown, giving a uniform predicted probability distribution across each class.

```
# conditional probabilities for low quality images
p_yx = asarray([[0.33, 0.33, 0.33], [0.33, 0.33, 0.33], [0.33, 0.33, 0.33]])
score = calculate_inception_score(p_yx)
print(score)
```

Listing 12.5: Example of uniform conditional probabilities.

In this case, we would expect the inception score to be the worst possible where there is no difference between the conditional and marginal distributions, e.g. an inception score of 1.0. Tying this together, the complete example is listed below.

```
# calculate inception score in numpy
from numpy import asarray
from numpy import expand_dims
from numpy import log
from numpy import mean
from numpy import exp

# calculate the inception score for p(y|x)
def calculate_inception_score(p_yx, eps=1E-16):
  # calculate p(y)
  p_y = expand_dims(p_yx.mean(axis=0), 0)
  # kl divergence for each image
  kl_d = p_yx * (log(p_yx + eps) - log(p_y + eps))
  # sum over classes
  sum_kl_d = kl_d.sum(axis=1)
  # average over images
  avg_kl_d = mean(sum_kl_d)
  # undo the logs
  is_score = exp(avg_kl_d)
  return is_score

# conditional probabilities for low quality images
p_yx = asarray([[0.33, 0.33, 0.33], [0.33, 0.33, 0.33], [0.33, 0.33, 0.33]])
score = calculate_inception_score(p_yx)
```

```
print(score)
```

Listing 12.6: Example of calculating the inception score for uniform probabilities.

Running the example reports the expected inception score of 1.0.

```
1.0
```

Listing 12.7: Example output from calculating the inception score for uniform probabilities.

You may want to experiment with the calculation of the inception score and test other pathological cases.

12.5 How to Implement the Inception Score With Keras

Now that we know how to calculate the inception score and to implement it in Python, we can develop an implementation in Keras. This involves using the real Inception v3 model to classify images and to average the calculation of the score across multiple splits of a collection of images. First, we can load the Inception v3 model in Keras directly.

```
...
# load inception v3 model
model = InceptionV3()
```

Listing 12.8: Example of loading the inception v3 model.

The model expects images to be color and to have the shape 299×299 pixels. Additionally, the pixel values must be scaled in the same way as the training data images, before they can be classified. This can be achieved by converting the pixel values from integers to floating point values and then calling the `preprocess_input()` function for the images.

```
...
# convert from uint8 to float32
processed = images.astype('float32')
# pre-process raw images for inception v3 model
processed = preprocess_input(processed)
```

Listing 12.9: Example of pre-processing images for the inception v3 model.

Then the conditional probabilities for each of the 1,000 image classes can be predicted for the images.

```
...
# predict class probabilities for images
yhat = model.predict(images)
```

Listing 12.10: Example of making a prediction with the inception v3 model.

The inception score can then be calculated directly on the NumPy array of probabilities as we did in the previous section. Before we do that, we must split the conditional probabilities into groups, controlled by a `n_split` argument and set to the default of 10 as was used in the original paper.

```
...
n_part = floor(images.shape[0] / n_split)
```

Listing 12.11: Example of calculating the number of images in each split.

We can then enumerate over the conditional probabilities in blocks of `n_part` images or predictions and calculate the inception score.

```
...
# retrieve p(y|x)
ix_start, ix_end = i * n_part, (i+1) * n_part
p_yx = yhat[ix_start:ix_end]
```

Listing 12.12: Example of retrieving the conditional probabilities for the images in a given split.

After calculating the scores for each split of conditional probabilities, we can calculate and return the average and standard deviation inception scores.

```
...
# average across images
is_avg, is_std = mean(scores), std(scores)
```

Listing 12.13: Example of calculating summary statistics on the calculated inception scores across splits.

Tying all of this together, the `calculate_inception_score()` function below takes an array of images with the expected size and pixel values in [0,255] and calculates the average and standard deviation inception scores using the inception v3 model in Keras.

```
# assumes images have the shape 299x299x3, pixels in [0,255]
def calculate_inception_score(images, n_split=10, eps=1E-16):
  # load inception v3 model
  model = InceptionV3()
  # convert from uint8 to float32
  processed = images.astype('float32')
  # pre-process raw images for inception v3 model
  processed = preprocess_input(processed)
  # predict class probabilities for images
  yhat = model.predict(processed)
  # enumerate splits of images/predictions
  scores = list()
  n_part = floor(images.shape[0] / n_split)
  for i in range(n_split):
    # retrieve p(y|x)
    ix_start, ix_end = i * n_part, i * n_part + n_part
    p_yx = yhat[ix_start:ix_end]
    # calculate p(y)
    p_y = expand_dims(p_yx.mean(axis=0), 0)
    # calculate KL divergence using log probabilities
    kl_d = p_yx * (log(p_yx + eps) - log(p_y + eps))
    # sum over classes
    sum_kl_d = kl_d.sum(axis=1)
    # average over images
    avg_kl_d = mean(sum_kl_d)
    # undo the log
    is_score = exp(avg_kl_d)
    # store
    scores.append(is_score)
  # average across images
  is_avg, is_std = mean(scores), std(scores)
  return is_avg, is_std
```

Listing 12.14: Example of a function for calculating the conditional probabilities for images and calculating the inception score.

We can test this function with 50 artificial images with the value 1.0 for all pixels.

```
...
# pretend to load images
images = ones((50, 299, 299, 3))
print('loaded', images.shape)
```

Listing 12.15: Example of a sample of contrived images.

This will calculate the score for each group of five images and the low quality would suggest that an average inception score of 1.0 will be reported. The complete example is listed below.

```
# calculate inception score with Keras
from math import floor
from numpy import ones
from numpy import expand_dims
from numpy import log
from numpy import mean
from numpy import std
from numpy import exp
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input

# assumes images have the shape 299x299x3, pixels in [0,255]
def calculate_inception_score(images, n_split=10, eps=1E-16):
  # load inception v3 model
  model = InceptionV3()
  # convert from uint8 to float32
  processed = images.astype('float32')
  # pre-process raw images for inception v3 model
  processed = preprocess_input(processed)
  # predict class probabilities for images
  yhat = model.predict(processed)
  # enumerate splits of images/predictions
  scores = list()
  n_part = floor(images.shape[0] / n_split)
  for i in range(n_split):
    # retrieve p(y|x)
    ix_start, ix_end = i * n_part, i * n_part + n_part
    p_yx = yhat[ix_start:ix_end]
    # calculate p(y)
    p_y = expand_dims(p_yx.mean(axis=0), 0)
    # calculate KL divergence using log probabilities
    kl_d = p_yx * (log(p_yx + eps) - log(p_y + eps))
    # sum over classes
    sum_kl_d = kl_d.sum(axis=1)
    # average over images
    avg_kl_d = mean(sum_kl_d)
    # undo the log
    is_score = exp(avg_kl_d)
    # store
    scores.append(is_score)
```

```
  # average across images
  is_avg, is_std = mean(scores), std(scores)
  return is_avg, is_std

# pretend to load images
images = ones((50, 299, 299, 3))
print('loaded', images.shape)
# calculate inception score
is_avg, is_std = calculate_inception_score(images)
print('score', is_avg, is_std)
```

Listing 12.16: Example of calculating conditional probabilities with the Inception model and calculating the inception score.

Running the example first defines the 50 fake images, then calculates the inception score on each batch and reports the expected inception score of 1.0, with a standard deviation of 0.0.

Note: the first time the InceptionV3 model is used, Keras will download the model weights and save them into the ∼/.keras/models/ directory on your workstation. The weights are about 100 megabytes and may take a moment to download depending on the speed of your internet connection.

```
loaded (50, 299, 299, 3)
score 1.0 0.0
```

Listing 12.17: Example output from calculating conditional probabilities with the Inception model and calculating the inception score.

We can test the calculation of the inception score on some real images. The Keras API provides access to the CIFAR-10 dataset (described in Section 8.2). These are color photos with the small size of 32×32 pixels. First, we can split the images into groups, then upsample the images to the expected size of 299×299, pre-process the pixel values, predict the class probabilities, then calculate the inception score. This will be a useful example if you intend to calculate the inception score on your own generated images, as you may have to either scale the images to the expected size for the inception v3 model or change the model to perform the upsampling for you. First, the images can be loaded and shuffled to ensure each split covers a diverse set of classes.

```
...
# load cifar10 images
(images, _), (_, _) = cifar10.load_data()
# shuffle images
shuffle(images)
```

Listing 12.18: Example of loading and shuffling the CIFAR-10 training dataset.

Next, we need a way to scale the images. We will use the scikit-image library to resize the NumPy array of pixel values to the required size. The `scale_images()` function below implements this.

```
# scale an array of images to a new size
def scale_images(images, new_shape):
  images_list = list()
  for image in images:
```

```
    # resize with nearest neighbor interpolation
    new_image = resize(image, new_shape, 0)
    # store
    images_list.append(new_image)
  return asarray(images_list)
```

Listing 12.19: Example of a function for scaling images to a new size.

You may have to install the scikit-image library if it is not already installed. This can be achieved as follows:

```
sudo pip install scikit-image
```

Listing 12.20: Example installing the scikit-image library with `pip`.

We can then enumerate the number of splits, select a subset of the images, scale them, pre-process them, and use the model to predict the conditional class probabilities.

```
...
# retrieve images
ix_start, ix_end = i * n_part, (i+1) * n_part
subset = images[ix_start:ix_end]
# convert from uint8 to float32
subset = subset.astype('float32')
# scale images to the required size
subset = scale_images(subset, (299,299,3))
# pre-process images, scale to [-1,1]
subset = preprocess_input(subset)
# predict p(y|x)
p_yx = model.predict(subset)
```

Listing 12.21: Example of preparing a split of the CIFAR-10 images and predicting the conditional probabilities.

The rest of the calculation of the inception score is the same. Tying this all together, the complete example for calculating the inception score on the real CIFAR-10 training dataset is listed below. Based on the similar calculation reported in the original inception score paper, we would expect the reported score on this dataset to be approximately 11. Interestingly, the best inception score for CIFAR-10 with generated images is about 8.8 at the time of writing using a progressive growing GAN.

```
# calculate inception score for cifar-10 in Keras
from math import floor
from numpy import expand_dims
from numpy import log
from numpy import mean
from numpy import std
from numpy import exp
from numpy.random import shuffle
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input
from keras.datasets import cifar10
from skimage.transform import resize
from numpy import asarray

# scale an array of images to a new size
def scale_images(images, new_shape):
```

```
  images_list = list()
  for image in images:
    # resize with nearest neighbor interpolation
    new_image = resize(image, new_shape, 0)
    # store
    images_list.append(new_image)
  return asarray(images_list)

# assumes images have any shape and pixels in [0,255]
def calculate_inception_score(images, n_split=10, eps=1E-16):
  # load inception v3 model
  model = InceptionV3()
  # enumerate splits of images/predictions
  scores = list()
  n_part = floor(images.shape[0] / n_split)
  for i in range(n_split):
    # retrieve images
    ix_start, ix_end = i * n_part, (i+1) * n_part
    subset = images[ix_start:ix_end]
    # convert from uint8 to float32
    subset = subset.astype('float32')
    # scale images to the required size
    subset = scale_images(subset, (299,299,3))
    # pre-process images, scale to [-1,1]
    subset = preprocess_input(subset)
    # predict p(y|x)
    p_yx = model.predict(subset)
    # calculate p(y)
    p_y = expand_dims(p_yx.mean(axis=0), 0)
    # calculate KL divergence using log probabilities
    kl_d = p_yx * (log(p_yx + eps) - log(p_y + eps))
    # sum over classes
    sum_kl_d = kl_d.sum(axis=1)
    # average over images
    avg_kl_d = mean(sum_kl_d)
    # undo the log
    is_score = exp(avg_kl_d)
    # store
    scores.append(is_score)
  # average across images
  is_avg, is_std = mean(scores), std(scores)
  return is_avg, is_std

# load cifar10 images
(images, _), (_, _) = cifar10.load_data()
# shuffle images
shuffle(images)
print('loaded', images.shape)
# calculate inception score
is_avg, is_std = calculate_inception_score(images)
print('score', is_avg, is_std)
```

Listing 12.22: Example of calculating the inception score for the CIFAR-10 training dataset.

Running the example may take some time depending on the speed of your workstation. It may also require a large amount of RAM. If you have trouble with the example, try reducing

the number of images in the training dataset. Running the example loads the dataset, prepares the model, and calculates the inception score on the CIFAR-10 training dataset. We can see that the score is 11.3, which is close to the expected score of 11.24.

```
loaded (50000, 32, 32, 3)
score 11.317895 0.14821531
```

Listing 12.23: Example output from calculating the inception score for the CIFAR-10 training dataset.

12.6 Problems With the Inception Score

The inception score is effective, but it is not perfect. Generally, the inception score is appropriate for generated images of objects known to the model used to calculate the conditional class probabilities. In this case, because the inception v3 model is used, this means that it is most suitable for 1,000 object types used in the ILSVRC 2012 dataset[1]. This is a lot of classes, but not all objects that may interest us.

It also requires that the images are square and have the relatively small size of about 300×300 pixels, including any scaling required to get your generated images to that size. A good score also requires having a good distribution of generated images across the possible objects supported by the model, and close to an even number of examples for each class. This can be hard to control for many GAN models that don't offer controls over the types of objects generated. Shane Barratt and Rishi Sharma take a closer look at the inception score and list a number of technical issues and edge cases in their 2018 paper titled *A Note on the Inception Score*. This is a good reference if you wish to dive deeper.

12.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

12.7.1 Papers

- Improved Techniques for Training GANs, 2016.
  https://arxiv.org/abs/1606.03498
- A Note on the Inception Score, 2018.
  https://arxiv.org/abs/1801.01973
- Rethinking the Inception Architecture for Computer Vision, 2015.
  https://arxiv.org/abs/1512.00567

12.7.2 Projects

- Code for the paper Improved Techniques for Training GANs.
  https://github.com/openai/improved-gan

[1] http://image-net.org/challenges/LSVRC/2012/browse-synsets

- Large Scale Visual Recognition Challenge 2012 (ILSVRC2012).
  http://image-net.org/challenges/LSVRC/2012/

12.7.3 API

- Keras Inception v3 Model.
  https://keras.io/applications/#inceptionv3
- scikit-image Library.
  https://scikit-image.org/

12.7.4 Articles

- Image Generation on CIFAR-10
  https://paperswithcode.com/sota/image-generation-on-cifar-10
- Inception Score calculation, GitHub, 2017.
  https://github.com/openai/improved-gan/issues/29
- Kullback-Leibler divergence, Wikipedia.
  https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence
- Entropy (information theory), Wikipedia.
  https://en.wikipedia.org/wiki/Entropy_(information_theory)

12.8 Summary

In this tutorial, you discovered the inception score for evaluating the quality of generated images. Specifically, you learned:

- How to calculate the inception score and the intuition behind what it measures.
- How to implement the inception score in Python with NumPy and the Keras deep learning library.
- How to calculate the inception score for small images such as those in the CIFAR-10 dataset.

12.8.1 Next

In the next tutorial, you will discover the FID score and how to implement it from scratch and interpret the results.

Chapter 13

How to Implement the Frechet Inception Distance

The Frechet Inception Distance score, or FID for short, is a metric that calculates the distance between feature vectors calculated for real and generated images. The score summarizes how similar the two groups are in terms of statistics on computer vision features of the raw images calculated using the inception v3 model used for image classification. Lower scores indicate the two groups of images are more similar, or have more similar statistics, with a perfect score being 0.0 indicating that the two groups of images are identical.

The FID score is used to evaluate the quality of images generated by generative adversarial networks, and lower scores have been shown to correlate well with higher quality images. In this tutorial, you will discover how to implement the Frechet Inception Distance for evaluating generated images. After completing this tutorial, you will know:

- The Frechet Inception Distance summarizes the distance between the Inception feature vectors for real and generated images in the same domain.
- How to calculate the FID score and implement the calculation from scratch in NumPy.
- How to implement the FID score using the Keras deep learning library and calculate it with real images.

Let's get started.

13.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What Is the Frechet Inception Distance?
2. How to Calculate the FID
3. How to Implement the FID With NumPy
4. How to Implement the FID With Keras
5. How to Calculate the FID for Real Images

13.2 What Is the Frechet Inception Distance?

The Frechet Inception Distance, or FID for short, is a metric for evaluating the quality of generated images and specifically developed to evaluate the performance of generative adversarial networks. The FID score was proposed and used by Martin Heusel, et al. in their 2017 paper titled *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. The score was proposed as an improvement over the existing Inception Score, or IS.

> For the evaluation of the performance of GANs at image generation, we introduce the "Frechet Inception Distance" (FID) which captures the similarity of generated images to real ones better than the Inception Score.

— *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*, 2017.

The inception score estimates the quality of a collection of synthetic images based on how well the top-performing image classification model Inception v3 classifies them as one of 1,000 known objects. The scores combine both the confidence of the conditional class predictions for each synthetic image (quality) and the integral of the marginal probability of the predicted classes (diversity). The inception score does not capture how synthetic images compare to real images. The goal in developing the FID score was to evaluate synthetic images based on the statistics of a collection of synthetic images compared to the statistics of a collection of real images from the target domain.

> Drawback of the Inception Score is that the statistics of real world samples are not used and compared to the statistics of synthetic samples.

— *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*, 2017.

Like the inception score, the FID score uses the inception v3 model. Specifically, the coding layer of the model (the last pooling layer prior to the output classification of images) is used to capture computer-vision-specific features of an input image. These activations are calculated for a collection of real and generated images. The activations are summarized as a multivariate Gaussian by calculating the mean and covariance of the images. These statistics are then calculated for the activations across the collection of real and generated images. The distance between these two distributions is then calculated using the Frechet distance, also called the Wasserstein-2 distance.

> The difference of two Gaussians (synthetic and real-world images) is measured by the Frechet distance also known as Wasserstein-2 distance.

— *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*, 2017.

The use of activations from the Inception v3 model to summarize each image gives the score its name of *Frechet Inception Distance*. A lower FID indicates better-quality images; conversely, a higher score indicates a lower-quality image and the relationship may be linear. The authors of the score show that lower FID scores correlate with better-quality images when systematic distortions were applied such as the addition of random noise and blur.

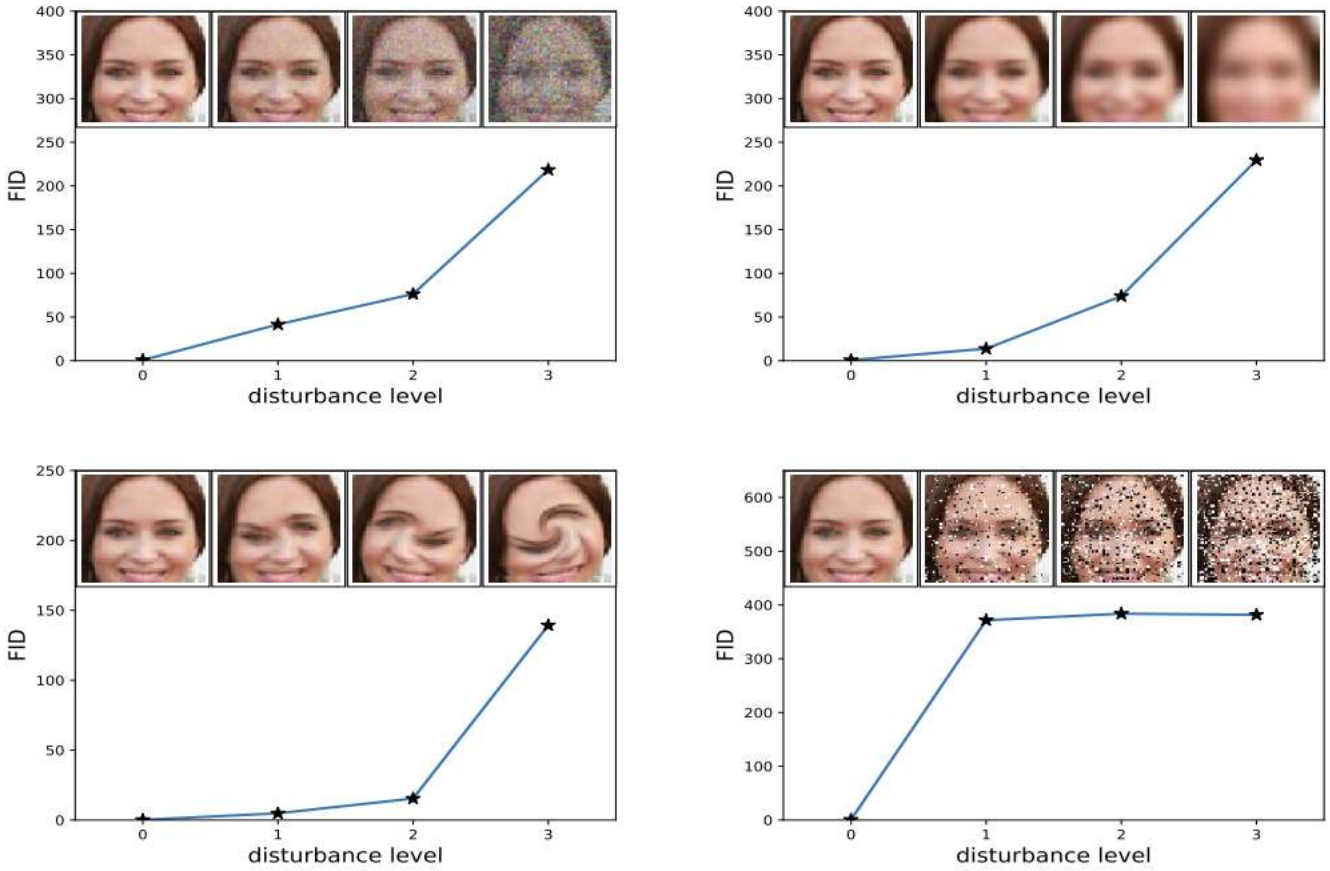

Figure 13.1: Example of How Increased Distortion of an Image Correlates with High FID Score. Taken from: GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium.

13.3 How to Calculate the FID

The FID score is calculated by first loading a pre-trained Inception v3 model. The output layer of the model is removed and the output is taken as the activations from the last pooling layer, a global spatial pooling layer. This output layer has 2,048 activations, therefore, each image is predicted as 2,048 activation features. This is called the coding vector or feature vector for the image.

A 2,048 feature vector is then predicted for a collection of real images from the problem domain to provide a reference for how real images are represented. Feature vectors can then be calculated for synthetic images. The result will be two collections of 2,048 feature vectors for real and generated images. The FID score is then calculated using the following equation taken from the paper:

$$d^2 = ||mu_1 - mu_2||^2 + Tr(C_1 + C_2 - 2 \times \sqrt{C_1 \times C_2}) \tag{13.1}$$

The score is referred to as d^2, showing that it is a distance and has squared units. The mu_1

and mu_2 refer to the feature-wise mean of the real and generated images, e.g. 2,048 element vectors where each element is the mean feature observed across the images. The C_1 and C_2 are the covariance matrix for the real and generated feature vectors, often referred to as sigma. The $||mu_1 - mu_2||^2$ refers to the sum squared difference between the two mean vectors. Tr refers to the trace linear algebra operation, e.g. the sum of the elements along the main diagonal of the square matrix. The square root of a matrix is often also written as $M^{\frac{1}{2}}$, e.g. the matrix to the power of one half, which has the same effect. This operation can fail depending on the values in the matrix because the operation is solved using numerical methods. Commonly, some elements in the resulting matrix may be imaginary, which often can be detected and removed.

13.4 How to Implement the FID With NumPy

Implementing the calculation of the FID score in Python with NumPy arrays is straightforward. First, let's define a function that will take a collection of activations for real and generated images and return the FID score. The `calculate_fid()` function listed below implements the procedure. Here, we implement the FID calculation almost directly. It is worth noting that the official implementation in TensorFlow implements elements of the calculation in a slightly different order, likely for efficiency, and introduces additional checks around the matrix square root to handle possible numerical instabilities. I recommend reviewing the official implementation and extending the implementation below to add these checks if you experience problems calculating the FID on your own datasets.

```
# calculate frechet inception distance
def calculate_fid(act1, act2):
  # calculate mean and covariance statistics
  mu1, sigma1 = act1.mean(axis=0), cov(act1, rowvar=False)
  mu2, sigma2 = act2.mean(axis=0), cov(act2, rowvar=False)
  # calculate sum squared difference between means
  ssdiff = numpy.sum((mu1 - mu2)**2.0)
  # calculate sqrt of product between cov
  covmean = sqrtm(sigma1.dot(sigma2))
  # check and correct imaginary numbers from sqrt
  if iscomplexobj(covmean):
    covmean = covmean.real
  # calculate score
  fid = ssdiff + trace(sigma1 + sigma2 - 2.0 * covmean)
  return fid
```

Listing 13.1: Example of a function for calculating the Frechet inception distance between two sets of activations.

We can then test out this function to calculate the inception score for some contrived feature vectors. Feature vectors will probably contain small positive values and will have a length of 2,048 elements. We can construct two lots of 10 images worth of feature vectors with small random numbers as follows:

```
...
# define two collections of activations
act1 = random(10*2048)
act1 = act1.reshape((10,2048))
act2 = random(10*2048)
```

```
act2 = act2.reshape((10,2048))
```

Listing 13.2: Example of defining some random activations.

One test would be to calculate the FID between a set of activations and itself, which we would expect to have a score of 0.0. We can then calculate the distance between the two sets of random activations, which we would expect to be a large number.

```
...
# fid between act1 and act1
fid = calculate_fid(act1, act1)
print('FID (same): %.3f' % fid)
# fid between act1 and act2
fid = calculate_fid(act1, act2)
print('FID (different): %.3f' % fid)
```

Listing 13.3: Example of calculating the FID between sets of activations.

Tying this all together, the complete example is listed below.

```
# example of calculating the frechet inception distance
import numpy
from numpy import cov
from numpy import trace
from numpy import iscomplexobj
from numpy.random import random
from scipy.linalg import sqrtm

# calculate frechet inception distance
def calculate_fid(act1, act2):
  # calculate mean and covariance statistics
  mu1, sigma1 = act1.mean(axis=0), cov(act1, rowvar=False)
  mu2, sigma2 = act2.mean(axis=0), cov(act2, rowvar=False)
  # calculate sum squared difference between means
  ssdiff = numpy.sum((mu1 - mu2)**2.0)
  # calculate sqrt of product between cov
  covmean = sqrtm(sigma1.dot(sigma2))
  # check and correct imaginary numbers from sqrt
  if iscomplexobj(covmean):
    covmean = covmean.real
  # calculate score
  fid = ssdiff + trace(sigma1 + sigma2 - 2.0 * covmean)
  return fid

# define two collections of activations
act1 = random(10*2048)
act1 = act1.reshape((10,2048))
act2 = random(10*2048)
act2 = act2.reshape((10,2048))
# fid between act1 and act1
fid = calculate_fid(act1, act1)
print('FID (same): %.3f' % fid)
# fid between act1 and act2
fid = calculate_fid(act1, act2)
print('FID (different): %.3f' % fid)
```

Listing 13.4: Example of calculating the FID in NumPy between two sets of activations.

Running the example first reports the FID between the act1 activations and itself, which is 0.0 as we expect (note that the sign of the score can be ignored). The distance between the two collections of random activations is also as we expect: a large number, which in this case was about 359.

```
FID (same): -0.000
FID (different): 358.927
```

Listing 13.5: Example output from calculating the FID in NumPy between two sets of activations.

You may want to experiment with the calculation of the FID score and test other pathological cases.

13.5 How to Implement the FID With Keras

Now that we know how to calculate the FID score and to implement it in NumPy, we can develop an implementation in Keras. This involves the preparation of the image data and using a pre-trained Inception v3 model to calculate the activations or feature vectors for each image. First, we can load the Inception v3 model in Keras directly.

```
...
# load inception v3 model
model = InceptionV3()
```

Listing 13.6: Example of loading the inception v3 model.

This will prepare a version of the inception model for classifying images as one of 1,000 known classes. We can remove the output (the top) of the model via the `include_top=False` argument. Painfully, this also removes the global average pooling layer that we require, but we can add it back via specifying the `pooling='avg'` argument. When the output layer of the model is removed, we must specify the shape of the input images, which is $299 \times 299 \times 3$ pixels, e.g. the `input_shape=(299,299,3)` argument. Therefore, the inception model can be loaded as follows:

```
...
# prepare the inception v3 model
model = InceptionV3(include_top=False, pooling='avg', input_shape=(299,299,3))
```

Listing 13.7: Example of loading the inception v3 model to output an embedding for an image.

This model can then be used to predict the feature vector for one or more images. Our images are likely to not have the required shape. We will use the scikit-image library to resize the NumPy array of pixel values to the required size. The `scale_images()` function below implements this.

```
# scale an array of images to a new size
def scale_images(images, new_shape):
  images_list = list()
  for image in images:
    # resize with nearest neighbor interpolation
    new_image = resize(image, new_shape, 0)
    # store
    images_list.append(new_image)
  return asarray(images_list)
```

Listing 13.8: Example of a function for scaling images to a new size.

You may have to install the scikit-image library if it is not already installed. This can be achieved as follows:

```
sudo pip install scikit-image
```

Listing 13.9: Example installing the scikit-image library with `pip`.

Once resized, the image pixel values will also need to be scaled to meet the expectations for inputs to the inception model. This can be achieved by calling the `preprocess_input()` function. We can update our `calculate_fid()` function defined in the previous section to take the loaded inception model and two NumPy arrays of image data as arguments, instead of activations. The function will then calculate the activations before calculating the FID score as before. The updated version of the `calculate_fid()` function is listed below.

```
# calculate frechet inception distance
def calculate_fid(model, images1, images2):
  # calculate activations
  act1 = model.predict(images1)
  act2 = model.predict(images2)
  # calculate mean and covariance statistics
  mu1, sigma1 = act1.mean(axis=0), cov(act1, rowvar=False)
  mu2, sigma2 = act2.mean(axis=0), cov(act2, rowvar=False)
  # calculate sum squared difference between means
  ssdiff = numpy.sum((mu1 - mu2)**2.0)
  # calculate sqrt of product between cov
  covmean = sqrtm(sigma1.dot(sigma2))
  # check and correct imaginary numbers from sqrt
  if iscomplexobj(covmean):
    covmean = covmean.real
  # calculate score
  fid = ssdiff + trace(sigma1 + sigma2 - 2.0 * covmean)
  return fid
```

Listing 13.10: Example of a function for predicting the embedding for two images and calculating the FID between them.

We can then test this function with some contrived collections of images, in this case, 10 32×32 images with random pixel values in the range [0,255].

```
...
# define two fake collections of images
images1 = randint(0, 255, 10*32*32*3)
images1 = images1.reshape((10,32,32,3))
images2 = randint(0, 255, 10*32*32*3)
images2 = images2.reshape((10,32,32,3))
```

Listing 13.11: Example of defining images with random pixel values.

We can then convert the integer pixel values to floating point values and scale them to the required size of 299×299 pixels.

```
...
# convert integer to floating point values
```

```
images1 = images1.astype('float32')
images2 = images2.astype('float32')
# resize images
images1 = scale_images(images1, (299,299,3))
images2 = scale_images(images2, (299,299,3))
```

Listing 13.12: Example of scaling the random images to the required size.

Then the pixel values can be scaled to meet the expectations of the Inception v3 model.

```
...
# pre-process images
images1 = preprocess_input(images1)
images2 = preprocess_input(images2)
```

Listing 13.13: Example of scaling pixel values to the required bounds.

Then calculate the FID scores, first between a collection of images and itself, then between the two collections of images.

```
...
# fid between images1 and images1
fid = calculate_fid(model, images1, images1)
print('FID (same): %.3f' % fid)
# fid between images1 and images2
fid = calculate_fid(model, images1, images2)
print('FID (different): %.3f' % fid)
```

Listing 13.14: Example of calculating the FID between two sets of prepared images.

Tying all of this together, the complete example is listed below.

```
# example of calculating the frechet inception distance in Keras
import numpy
from numpy import cov
from numpy import trace
from numpy import iscomplexobj
from numpy import asarray
from numpy.random import randint
from scipy.linalg import sqrtm
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input
from skimage.transform import resize

# scale an array of images to a new size
def scale_images(images, new_shape):
  images_list = list()
  for image in images:
    # resize with nearest neighbor interpolation
    new_image = resize(image, new_shape, 0)
    # store
    images_list.append(new_image)
  return asarray(images_list)

# calculate frechet inception distance
def calculate_fid(model, images1, images2):
  # calculate activations
  act1 = model.predict(images1)
```

```
  act2 = model.predict(images2)
  # calculate mean and covariance statistics
  mu1, sigma1 = act1.mean(axis=0), cov(act1, rowvar=False)
  mu2, sigma2 = act2.mean(axis=0), cov(act2, rowvar=False)
  # calculate sum squared difference between means
  ssdiff = numpy.sum((mu1 - mu2)**2.0)
  # calculate sqrt of product between cov
  covmean = sqrtm(sigma1.dot(sigma2))
  # check and correct imaginary numbers from sqrt
  if iscomplexobj(covmean):
    covmean = covmean.real
  # calculate score
  fid = ssdiff + trace(sigma1 + sigma2 - 2.0 * covmean)
  return fid

# prepare the inception v3 model
model = InceptionV3(include_top=False, pooling='avg', input_shape=(299,299,3))
# define two fake collections of images
images1 = randint(0, 255, 10*32*32*3)
images1 = images1.reshape((10,32,32,3))
images2 = randint(0, 255, 10*32*32*3)
images2 = images2.reshape((10,32,32,3))
print('Prepared', images1.shape, images2.shape)
# convert integer to floating point values
images1 = images1.astype('float32')
images2 = images2.astype('float32')
# resize images
images1 = scale_images(images1, (299,299,3))
images2 = scale_images(images2, (299,299,3))
print('Scaled', images1.shape, images2.shape)
# pre-process images
images1 = preprocess_input(images1)
images2 = preprocess_input(images2)
# fid between images1 and images1
fid = calculate_fid(model, images1, images1)
print('FID (same): %.3f' % fid)
# fid between images1 and images2
fid = calculate_fid(model, images1, images2)
print('FID (different): %.3f' % fid)
```

Listing 13.15: Example of calculating the FID in Keras between two sets of random images.

Running the example first summarizes the shapes of the fabricated images and their rescaled versions, matching our expectations.

Note: the first time the InceptionV3 model is used, Keras will download the model weights and save them into the ~/.keras/models/ directory on your workstation. The weights are about 100 megabytes and may take a moment to download depending on the speed of your internet connection.

The FID score between a given set of images and itself is 0.0, as we expect, and the distance between the two collections of random images is about 35.

```
Prepared (10, 32, 32, 3) (10, 32, 32, 3)
Scaled (10, 299, 299, 3) (10, 299, 299, 3)
```

```
FID (same): -0.000
FID (different): 35.495
```

Listing 13.16: Example output from calculating the FID in Keras between two sets of random images.

13.6 How to Calculate the FID for Real Images

It may be useful to calculate the FID score between two collections of real images. The Keras library provides a number of computer vision datasets, including the CIFAR-10 dataset (described in Section 8.2). These are color photos with the small size of 32×32 pixels and is split into train and test elements and can be loaded as follows:

```
...
# load cifar-10 images
(images, _), (_, _) = cifar10.load_data()
```

Listing 13.17: Example of loading the CIFAR-10 training dataset.

The training dataset has 50,000 images, whereas the test dataset has only 10,000 images. It may be interesting to calculate the FID score between these two datasets to get an idea of how representative the test dataset is of the training dataset. Scaling and scoring 50K images takes a long time, therefore, we can reduce the *training set* to a 10K random sample as follows:

```
...
shuffle(images1)
images1 = images1[:10000]
```

Listing 13.18: Example of shuffling and selecting a subset of the image dataset.

Tying this all together, we can calculate the FID score between a sample of the train and the test dataset as follows.

```
# example of calculating the frechet inception distance in Keras for cifar10
import numpy
from numpy import cov
from numpy import trace
from numpy import iscomplexobj
from numpy import asarray
from numpy.random import shuffle
from scipy.linalg import sqrtm
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input
from skimage.transform import resize
from keras.datasets import cifar10

# scale an array of images to a new size
def scale_images(images, new_shape):
  images_list = list()
  for image in images:
    # resize with nearest neighbor interpolation
    new_image = resize(image, new_shape, 0)
    # store
    images_list.append(new_image)
```

```
  return asarray(images_list)

# calculate frechet inception distance
def calculate_fid(model, images1, images2):
  # calculate activations
  act1 = model.predict(images1)
  act2 = model.predict(images2)
  # calculate mean and covariance statistics
  mu1, sigma1 = act1.mean(axis=0), cov(act1, rowvar=False)
  mu2, sigma2 = act2.mean(axis=0), cov(act2, rowvar=False)
  # calculate sum squared difference between means
  ssdiff = numpy.sum((mu1 - mu2)**2.0)
  # calculate sqrt of product between cov
  covmean = sqrtm(sigma1.dot(sigma2))
  # check and correct imaginary numbers from sqrt
  if iscomplexobj(covmean):
    covmean = covmean.real
  # calculate score
  fid = ssdiff + trace(sigma1 + sigma2 - 2.0 * covmean)
  return fid

# prepare the inception v3 model
model = InceptionV3(include_top=False, pooling='avg', input_shape=(299,299,3))
# load cifar10 images
(images1, _), (images2, _) = cifar10.load_data()
shuffle(images1)
images1 = images1[:10000]
print('Loaded', images1.shape, images2.shape)
# convert integer to floating point values
images1 = images1.astype('float32')
images2 = images2.astype('float32')
# resize images
images1 = scale_images(images1, (299,299,3))
images2 = scale_images(images2, (299,299,3))
print('Scaled', images1.shape, images2.shape)
# pre-process images
images1 = preprocess_input(images1)
images2 = preprocess_input(images2)
# calculate fid
fid = calculate_fid(model, images1, images2)
print('FID: %.3f' % fid)
```

Listing 13.19: Example of calculating the FID in Keras for a sample if the CIFAR-10 dataset.

Running the example may take some time depending on the speed of your workstation. It may also require a large amount of RAM. If you have trouble with the example, try halving the number of samples to 5,000. At the end of the run, we can see that the FID score between the train and test datasets is about five. Top-performing models at the time of writing are capable of achieving a FID score of about 15 or 16[1].

```
Loaded (10000, 32, 32, 3) (10000, 32, 32, 3)
Scaled (10000, 299, 299, 3) (10000, 299, 299, 3)
FID: 5.492
```

[1] https://paperswithcode.com/sota/image-generation-on-cifar-10

Listing 13.20: Example output from calculating the FID in Keras for a sample if the CIFAR-10 dataset.

13.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

13.7.1 Papers

- GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium, 2017.
  https://arxiv.org/abs/1706.08500
- Are GANs Created Equal? A Large-Scale Study, 2017.
  https://arxiv.org/abs/1711.10337
- Pros and Cons of GAN Evaluation Measures, 2018.
  https://arxiv.org/abs/1802.03446

13.7.2 Code Projects

- Official Implementation in TensorFlow, GitHub.
  https://github.com/bioinf-jku/TTUR
- Frechet Inception Distance (FID score) in PyTorch, GitHub.
  https://github.com/mseitzer/pytorch-fid

13.7.3 API

- `numpy.trace` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.trace.html
- `numpy.cov` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.trace.html
- `numpy.iscomplexobj` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.iscomplexobj.html
- Keras Inception v3 Model
  https://keras.io/applications/#inceptionv3
- scikit-image Library
  https://scikit-image.org/

13.7.4 Articles

- Frechet distance, Wikipedia.
  https://en.wikipedia.org/wiki/Fr%C3%A9chet_distance
- Covariance matrix, Wikipedia.
  https://en.wikipedia.org/wiki/Covariance_matrix
- Square root of a matrix, Wikipedia.
  https://en.wikipedia.org/wiki/Square_root_of_a_matrix

13.8 Summary

In this tutorial, you discovered how to implement the Frechet Inception Distance for evaluating generated images. Specifically, you learned:

- The Frechet Inception Distance summarizes the distance between the Inception feature vectors for real and generated images in the same domain.
- How to calculate the FID score and implement the calculation from scratch in NumPy.
- How to implement the FID score using the Keras deep learning library and calculate it with real images.

13.8.1 Next

This was the final tutorial in this part. In the next part, you will discover alternate loss functions to use when training GAN models.

Part IV

GAN Loss

Overview

In this part you will discover loss functions used to train generative adversarial networks. After reading the chapters in this part, you will know:

- Details of standard and alternate GAN loss functions and their differences when compared directly(Chapter 14).
- How to implement and evaluate the least squares GAN loss function (Chapter 15).
- How to implement and evaluate the Wasserstein GAN loss function (Chapter 16).

Chapter 14

How to Use Different GAN Loss Functions

The GAN architecture is relatively straightforward, although one aspect that remains challenging for beginners is the topic of GAN loss functions. The main reason is that the architecture involves the simultaneous training of two models: the generator and the discriminator. The discriminator model is updated like any other deep learning neural network, although the generator uses the discriminator as the loss function, meaning that the loss function for the generator is implicit and learned during training. In this tutorial, you will discover an introduction to loss functions for generative adversarial networks. After reading this tutorial, you will know:

- The GAN architecture is defined with the minimax GAN loss, although it is typically implemented using the non-saturating loss function.
- Common alternate loss functions used in modern GANs include the least squares and Wasserstein loss functions.
- Large-scale evaluation of GAN loss functions suggests little difference when other concerns, such as computational budget and model hyperparameters, are held constant.

Let's get started.

14.1 Overview

This tutorial is divided into four parts; they are:

1. Challenge of GAN Loss
2. Standard GAN Loss Functions
3. Alternate GAN Loss Functions
4. Effect of Different GAN Loss Functions

14.2 Challenge of GAN Loss

The generative adversarial network, or GAN for short, is a deep learning architecture for training a generative model for image synthesis. They have proven very effective, achieving impressive results in generating photorealistic faces, scenes, and more. The GAN architecture is relatively straightforward, although one aspect that remains challenging for beginners is the topic of GAN loss functions. The GAN architecture is comprised of two models: a discriminator and a generator. The discriminator is trained directly on real and generated images and is responsible for classifying images as real or fake (generated). The generator is not trained directly and instead is trained via the discriminator model. Specifically, the discriminator is learned to provide the loss function for the generator.

The two models compete in a two-player game, where simultaneous improvements are made to both generator and discriminator models that compete. In a non-GAN model, we typically seek convergence of the model on a training dataset observed as the minimization of the chosen loss function on the training dataset. In a GAN, convergence signals the end of the two player game. Instead, equilibrium between generator and discriminator loss is sought. We will take a closer look at the official GAN loss function used to train the generator and discriminator models and some alternate popular loss functions that may be used instead.

14.3 Standard GAN Loss Functions

The GAN architecture was described by Ian Goodfellow, et al. in their 2014 paper titled *Generative Adversarial Networks*. The approach was introduced with two loss functions: the first that has become known as the Minimax GAN Loss and the second that has become known as the Non-Saturating GAN Loss.

14.3.1 Discriminator Loss

Under both schemes, the discriminator loss is the same. The discriminator seeks to maximize the probability assigned to real and fake images.

> We train D to maximize the probability of assigning the correct label to both training examples and samples from G.
>
> — *Generative Adversarial Networks*, 2014.

Described mathematically, the discriminator seeks to maximize the average of the log probability for real images and the log of the inverted probabilities of fake images.

$$\max \log(D(x)) + \log(1 - D(G(z))) \tag{14.1}$$

If implemented directly, this would require changes be made to model weights using stochastic ascent rather than stochastic descent. It is more commonly implemented as a traditional binary classification problem with labels 0 and 1 for generated and real images respectively. The model is fit seeking to minimize the average binary cross-entropy, also called log loss.

$$\min y_{true} \times -\log(y_{predicted}) + (1 - y_{true}) \times -\log(1 - y_{predicted}) \tag{14.2}$$

14.3.2 Minimax GAN Loss

Minimax GAN loss refers to the minimax simultaneous optimization of the discriminator and generator models. Minimax refers to an optimization strategy in two-player turn-based games for minimizing the loss or cost for the worst case of the other player. For the GAN, the generator and discriminator are the two players and take turns involving updates to their model weights. The min and max refer to the minimization of the generator loss and the maximization of the discriminator's loss.

$$\min \max(D, G) \tag{14.3}$$

As stated above, the discriminator seeks to maximize the average of the log probability of real images and the log of the inverse probability for fake images.

$$\text{discriminator: } \max \log D(x) + log(1 - D(G(z))) \tag{14.4}$$

> Here the generator learns to generate samples that have a low probability of being fake.
>
> — *Are GANs Created Equal? A Large-Scale Study*, 2018.

This framing of the loss for the GAN was found to be useful in the analysis of the model as a minimax game, but in practice, it was found that this loss function for the generator saturates. This means that if it cannot learn as quickly as the discriminator, the discriminator wins, the game ends, and the model cannot be trained effectively.

> In practice, [the loss function] may not provide sufficient gradient for G to learn well. Early in learning, when G is poor, D can reject samples with high confidence because they are clearly different from the training data.
>
> — *Generative Adversarial Networks*, 2014.

14.3.3 Non-Saturating GAN Loss

The Non-Saturating GAN Loss is a modification to the generator loss to overcome the saturation problem. It is a subtle change that involves the generator maximizing the log of the discriminator probabilities for generated images instead of minimizing the log of the inverted discriminator probabilities for generated images.

$$\text{generator: } \max \log(D(G(z))) \tag{14.5}$$

This is a change in the framing of the problem. In the previous case, the generator sought to minimize the probability of images being predicted as fake. Here, the generator seeks to maximize the probability of images being predicted as real.

> To improve the gradient signal, the authors also propose the non-saturating loss, where the generator instead aims to maximize the probability of generated samples being real.
>
> — *Are GANs Created Equal? A Large-Scale Study*, 2018.

The result is better gradient information when updating the weights of the generator and a more stable training process.

> This objective function results in the same fixed point of the dynamics of G and D but provides much stronger gradients early in learning.
>
> — *Generative Adversarial Networks*, 2014.

In practice, this is also implemented as a binary classification problem, like the discriminator. Instead of maximizing the loss, we can flip the labels for real and fake images and minimize the cross-entropy.

> ... one approach is to continue to use cross-entropy minimization for the generator. Instead of flipping the sign on the discriminator's cost to obtain a cost for the generator, we flip the target used to construct the cross-entropy cost.
>
> — *NIPS 2016 Tutorial: Generative Adversarial Networks*, 2016.

14.4 Alternate GAN Loss Functions

The choice of loss function is a hot research topic and many alternate loss functions have been proposed and evaluated. Two popular alternate loss functions used in many GAN implementations are the least squares loss and the Wasserstein loss.

14.4.1 Least Squares GAN Loss

The least squares loss was proposed by Xudong Mao, et al. in their 2016 paper titled *Least Squares Generative Adversarial Networks*. Their approach was based on the observation of the limitations for using binary cross-entropy loss when generated images are very different from real images, which can lead to very small or vanishing gradients, and in turn, little or no update to the model.

> ... this loss function, however, will lead to the problem of vanishing gradients when updating the generator using the fake samples that are on the correct side of the decision boundary, but are still far from the real data.
>
> — *Least Squares Generative Adversarial Networks*, 2016.

The discriminator seeks to minimize the sum squared difference between predicted and expected values for real and fake images.

$$\text{discriminator: } \min(D(x) - 1)^2 + (D(G(z)))^2 \tag{14.6}$$

The generator seeks to minimize the sum squared difference between predicted and expected values as though the generated images were real.

$$\text{generator: } \min(D(G(z)) - 1)^2 \tag{14.7}$$

In practice, this involves maintaining the class labels of 0 and 1 for fake and real images respectively, minimizing the least squares, also called mean squared error or L2 loss.

$$\text{l2 loss} = \sum (y_{predicted} - y_{true})^2 \tag{14.8}$$

The benefit of the least squares loss is that it gives more penalty to larger errors, in turn resulting in a large correction rather than a vanishing gradient and no model update.

> ... the least squares loss function is able to move the fake samples toward the decision boundary, because the least squares loss function penalizes samples that lie in a long way on the correct side of the decision boundary.
>
> — *Least Squares Generative Adversarial Networks*, 2016.

For more details on the LSGAN, see Chapter 15.

14.4.2 Wasserstein GAN Loss

The Wasserstein loss was proposed by Martin Arjovsky, et al. in their 2017 paper titled *Wasserstein GAN*. The Wasserstein loss is informed by the observation that the traditional GAN is motivated to minimize the distance between the actual and predicted probability distributions for real and generated images, the so-called Kullback-Leibler divergence, or the Jensen-Shannon divergence. Instead, they propose modeling the problem on the Earth-Mover's distance, also referred to as the Wasserstein-1 distance. The Earth-Mover's distance calculates the distance between two probability distributions in terms of the cost of turning one distribution (pile of earth) into another.

The GAN using Wasserstein loss involves changing the notion of the discriminator into a critic that is updated more often (e.g. five times more often) than the generator model. The critic scores images with a real value instead of predicting a probability. It also requires that model weights be kept small, e.g. clipped to a hypercube of `[-0.01, 0.01]`. The score is calculated such that the distance between scores for real and fake images are maximally separate. The loss function can be implemented by calculating the average predicted score across real and fake images and multiplying the average score by 1 and -1 respectively. This has the desired effect of driving the scores for real and fake images apart.

The benefit of Wasserstein loss is that it provides a useful gradient almost everywhere, allowing for the continued training of the models. It also means that a lower Wasserstein loss correlates with better generator image quality, meaning that we are explicitly seeking a minimization of generator loss.

> To our knowledge, this is the first time in GAN literature that such a property is shown, where the loss of the GAN shows properties of convergence.
>
> — *Wasserstein GAN*, 2017.

For more details on the WGAN, see Chapter 16.

14.5 Effect of Different GAN Loss Functions

Many loss functions have been developed and evaluated in an effort to improve the stability of training GAN models. The most common is the non-saturating loss, generally, and the Least Squares and Wasserstein loss in larger and more recent GAN models. As such, there is much interest in whether one loss function is truly better than another for a given model implementation. This question motivated a large study of GAN loss functions by Mario Lucic, et al. in their 2018 paper titled *Are GANs Created Equal? A Large-Scale Study*.

> Despite a very rich research activity leading to numerous interesting GAN algorithms, it is still very hard to assess which algorithm(s) perform better than others. We conduct a neutral, multi-faceted large-scale empirical study on state-of-the-art models and evaluation measures.
>
> — *Are GANs Created Equal? A Large-Scale Study*, 2018.

They fix the computational budget and hyperparameter configuration for models and look at a suite of seven loss functions. This includes the Minimax loss (MM GAN), Non-Saturating loss (NS GAN), Wasserstein loss (WGAN), and Least-Squares loss (LS GAN) described above. The study also includes an extension of Wasserstein loss to remove the weight clipping called Wasserstein Gradient Penalty loss (WGAN GP) and two others, DRAGAN and BEGAN. The table below, taken from the paper, provides a useful summary of the different loss functions for both the discriminator and generator.

GAN	Discriminator Loss	Generator Loss
MM GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{GAN}} = -\mathbb{E}_{x\sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x}\sim p_g}[\log(1 - D(\hat{x}))]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{GAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\log(1 - D(\hat{x}))]$
NS GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{NSGAN}} = -\mathbb{E}_{x\sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x}\sim p_g}[\log(1 - D(\hat{x}))]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{NSGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[\log(D(\hat{x}))]$
WGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{WGAN}} = -\mathbb{E}_{x\sim p_d}[D(x)] + \mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{WGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$
WGAN GP	$\mathcal{L}_{\mathrm{D}}^{\mathrm{WGANGP}} = \mathcal{L}_{\mathrm{D}}^{\mathrm{WGAN}} + \lambda\mathbb{E}_{\hat{x}\sim p_g}[(\|\nabla D(\alpha x + (1-\alpha\hat{x})\|_2 - 1)^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{WGANGP}} = -\mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})]$
LS GAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{LSGAN}} = -\mathbb{E}_{x\sim p_d}[(D(x) - 1)^2] + \mathbb{E}_{\hat{x}\sim p_g}[D(\hat{x})^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{LSGAN}} = -\mathbb{E}_{\hat{x}\sim p_g}[(D(\hat{x} - 1))^2]$
DRAGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{DRAGAN}} = \mathcal{L}_{\mathrm{D}}^{\mathrm{GAN}} + \lambda\mathbb{E}_{\hat{x}\sim p_d+\mathcal{N}(0,c)}[(\|\nabla D(\hat{x})\|_2 - 1)^2]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{DRAGAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\log(1 - D(\hat{x}))]$
BEGAN	$\mathcal{L}_{\mathrm{D}}^{\mathrm{BEGAN}} = \mathbb{E}_{x\sim p_d}[\|x - \mathrm{AE}(x)\|_1] - k_t\mathbb{E}_{\hat{x}\sim p_g}[\|\hat{x} - \mathrm{AE}(\hat{x})\|_1]$	$\mathcal{L}_{\mathrm{G}}^{\mathrm{BEGAN}} = \mathbb{E}_{\hat{x}\sim p_g}[\|\hat{x} - \mathrm{AE}(\hat{x})\|_1]$

Figure 14.1: Summary of Different GAN Loss Functions. Taken from: Are GANs Created Equal? A Large-Scale Study.

The models were evaluated systematically using a range of GAN evaluation metrics, including the popular Frechet Inception Distance, or FID. Surprisingly, they discovered that all evaluated loss functions performed approximately the same when all other elements were held constant.

> We provide a fair and comprehensive comparison of the state-of-the-art GANs, and empirically demonstrate that nearly all of them can reach similar values of FID, given a high enough computational budget.
>
> — *Are GANs Created Equal? A Large-Scale Study*, 2018.

This does not mean that the choice of loss does not matter for specific problems and model configurations. Instead, the result suggests that the difference in the choice of loss function disappears when the other concerns of the model are held constant, such as computational budget and model configuration.

14.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

14.6.1 Papers

- Generative Adversarial Networks, 2014.
  https://arxiv.org/abs/1406.2661
- NIPS 2016 Tutorial: Generative Adversarial Networks, 2016.
  https://arxiv.org/abs/1701.00160
- Least Squares Generative Adversarial Networks, 2016.
  https://arxiv.org/abs/1611.04076
- Wasserstein GAN, 2017.
  https://arxiv.org/abs/1701.07875
- Improved Training of Wasserstein GANs, 2017.
  https://arxiv.org/abs/1704.00028
- Are GANs Created Equal? A Large-Scale Study, 2018.
  https://arxiv.org/abs/1711.10337

14.6.2 Articles

- Minimax, Wikipedia.
  https://en.wikipedia.org/wiki/Minimax
- Earth mover's distance, Wikipedia.
  https://en.wikipedia.org/wiki/Earth_mover%27s_distance

14.7 Summary

In this tutorial, you discovered an introduction to loss functions for generative adversarial networks. Specifically, you learned:

- The GAN architecture is defined with the minimax GAN loss, although it is typically implemented using the non-saturating loss function.
- Common alternate loss functions used in modern GANs include the least squares and Wasserstein loss functions.
- Large-scale evaluation of GAN loss functions suggests little difference when other concerns, such as computational budget and model hyperparameters, are held constant.

14.7.1 Next

In the next tutorial, you will discover the least-squares GAN and how to implement it from scratch.

Chapter 15

How to Develop a Least Squares GAN (LSGAN)

The Least Squares Generative Adversarial Network, or LSGAN for short, is an extension to the GAN architecture that addresses the problem of vanishing gradients and loss saturation. It is motivated by the desire to provide a signal to the generator about fake samples that are far from the discriminator model's decision boundary for classifying them as real or fake. The further the generated images are from the decision boundary, the larger the error signal provided to the generator, encouraging the generation of more realistic images. The LSGAN can be implemented with a minor change to the output layer of the discriminator layer and the adoption of the least squares, or L2, loss function. In this tutorial, you will discover how to develop a least squares generative adversarial network. After completing this tutorial, you will know:

- The LSGAN addresses vanishing gradients and loss saturation of the deep convolutional GAN.
- The LSGAN can be implemented by a mean squared error or L2 loss function for the discriminator model.
- How to implement the LSGAN model for generating handwritten digits for the MNIST dataset.

Let's get started.

15.1 Tutorial Overview

This tutorial is divided into three parts; they are:

- What Is Least Squares GAN
- How to Develop an LSGAN for MNIST
- How to Generate Images With LSGAN

15.2 What Is Least Squares GAN

The standard Generative Adversarial Network, or GAN for short, is an effective architecture for training an unsupervised generative model. The architecture involves training a discriminator model to tell the difference between real (from the dataset) and fake (generated) images, and using the discriminator, in turn, to train the generator model. The generator is updated in such a way that it is encouraged to generate images that are more likely to fool the discriminator. The discriminator is a binary classifier and is trained using binary cross-entropy loss function. A limitation of this loss function is that it is primarily concerned with whether the predictions are correct or not, and less so with how correct or incorrect they might be.

> ... when we use the fake samples to update the generator by making the discriminator believe they are from real data, it will cause almost no error because they are on the correct side, i.e., the real data side, of the decision boundary
>
> — *Least Squares Generative Adversarial Networks*, 2016.

This can be conceptualized in two dimensions as a line or decision boundary separating dots that represent real and fake images. The discriminator is responsible for devising the decision boundary to best separate real and fake images and the generator is responsible for creating new points that look like real points, confusing the discriminator. The choice of cross-entropy loss means that points generated far from the boundary are right or wrong, but provide very little gradient information to the generator on how to generate better images. This small gradient for generated images far from the decision boundary is referred to as a vanishing gradient problem or a loss saturation. The loss function is unable to give a strong signal as to how to best update the model.

The Least Squares Generative Adversarial Network, or LSGAN for short, is an extension to the GAN architecture proposed by Xudong Mao, et al. in their 2016 paper titled *Least Squares Generative Adversarial Networks*. The LSGAN is a modification to the GAN architecture that changes the loss function for the discriminator from binary cross-entropy to a least squares loss. The motivation for this change is that the least squares loss will penalize generated images based on their distance from the decision boundary. This will provide a strong gradient signal for generated images that are very different or far from the existing data and address the problem of saturated loss.

> ... minimizing the objective function of regular GAN suffers from vanishing gradients, which makes it hard to update the generator. LSGANs can relieve this problem because LSGANs penalize samples based on their distances to the decision boundary, which generates more gradients to update the generator.
>
> — *Least Squares Generative Adversarial Networks*, 2016.

This can be conceptualized with a plot, below, taken from the paper, that shows on the left the sigmoid decision boundary (blue) and generated fake points far from the decision boundary (pink), and on the right the least squares decision boundary (red) and the points far from the boundary (pink) given a gradient that moves them closer to the boundary.

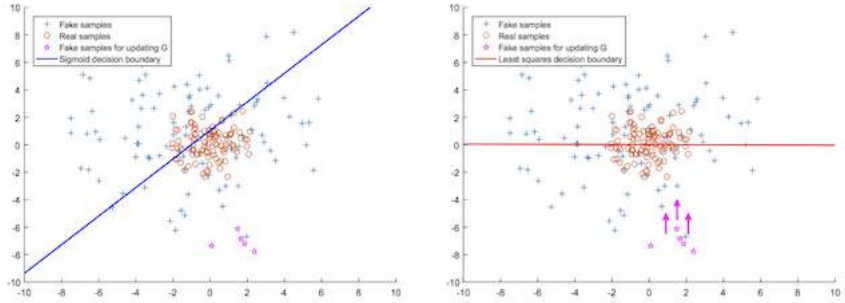

Figure 15.1: Plot of the Sigmoid Decision Boundary vs. the Least Squared Decision Boundary for Updating the Generator. Taken from: Least Squares Generative Adversarial Networks.

In addition to avoiding loss saturation, the LSGAN also results in a more stable training process and the generation of higher quality and larger images than the traditional deep convolutional GAN.

> First, LSGANs are able to generate higher quality images than regular GANs. Second, LSGANs perform more stable during the learning process.
>
> — *Least Squares Generative Adversarial Networks*, 2016.

The LSGAN can be implemented by using the target values of 1.0 for real and 0.0 for fake images and optimizing the model using the mean squared error (MSE) loss function, e.g. L2 loss. The output layer of the discriminator model must be a linear activation function. The authors propose a generator and discriminator model architecture, inspired by the VGG model architecture, and use interleaving upsampling and normal convolutional layers in the generator model, seen on the left in the image below.

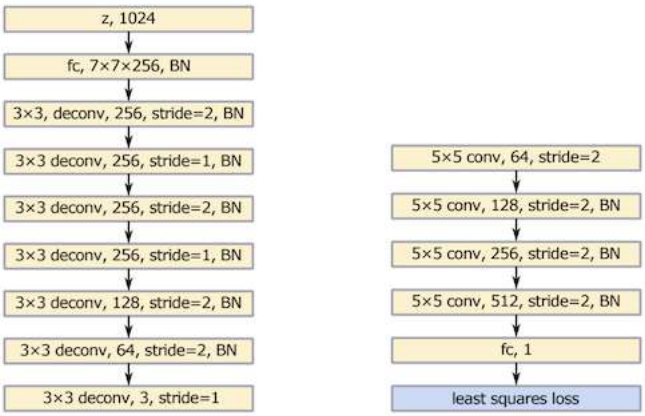

Figure 15.2: Summary of the Generator (left) and Discriminator (right) Model Architectures used in LSGAN Experiments. Taken from: Least Squares Generative Adversarial Networks.

15.3 How to Develop an LSGAN for MNIST

In this section, we will develop an LSGAN for the MNIST handwritten digit dataset (described in Section 7.2). The first step is to define the models. Both the discriminator and the generator will be based on the Deep Convolutional GAN, or DCGAN, architecture. This involves the use of Convolution-BatchNorm-Activation layer blocks with the use of 2×2 stride for downsampling and transpose convolutional layers for upsampling. `LeakyReLU` activation layers are used in the discriminator and ReLU activation layers are used in the generator. The discriminator expects grayscale input images with the shape 28×28, the shape of images in the MNIST dataset, and the output layer is a single node with a linear activation function. The model is optimized using the mean squared error (MSE) loss function as per the LSGAN. The `define_discriminator()` function below defines the discriminator model.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # downsample to 14x14
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      input_shape=in_shape))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 7x7
  model.add(Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dense(1, activation='linear', kernel_initializer=init))
  # compile model with L2 loss
  model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5))
  return model
```

Listing 15.1: Example of a function for defining the discriminator model.

The generator model takes a point in latent space as input and outputs a grayscale image with the shape 28×28 pixels, where pixel values are in the range [-1,1] via the Tanh activation function on the output layer. The `define_generator()` function below defines the generator model. This model is not compiled as it is not trained in a standalone manner.

```
# define the standalone generator model
def define_generator(latent_dim):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 256 * 7 * 7
  model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  model.add(Reshape((7, 7, 256)))
  # upsample to 14x14
```

```
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  # upsample to 28x28
  model.add(Conv2DTranspose(64, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  # output 28x28x1
  model.add(Conv2D(1, (7,7), padding='same', kernel_initializer=init))
  model.add(Activation('tanh'))
  return model
```

Listing 15.2: Example of a function for defining the generator model.

The generator model is updated via the discriminator model. This is achieved by creating a composite model that stacks the generator on top of the discriminator so that error signals can flow back through the discriminator to the generator. The weights of the discriminator are marked as not trainable when used in this composite model. Updates via the composite model involve using the generator to create new images by providing random points in the latent space as input. The generated images are passed to the discriminator, which will classify them as real or fake. The weights are updated as though the generated images are real (e.g. target of 1.0), allowing the generator to be updated toward generating more realistic images. The `define_gan()` function defines and compiles the composite model for updating the generator model via the discriminator, again optimized via mean squared error as per the LSGAN.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  for layer in discriminator.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model with L2 loss
  model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5))
  return model
```

Listing 15.3: Example of a function for defining the composite model for training the generator.

Next, we can define a function to load the MNIST handwritten digit dataset and scale the pixel values to the range [-1,1] to match the images output by the generator model. Only the training part of the MNIST dataset is used, which contains 60,000 centered grayscale images of digits zero through nine.

```
# load mnist images
def load_real_samples():
  # load dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels
```

```
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X
```

Listing 15.4: Example of a function for loading and preparing the MNIST dataset.

We can then define a function to retrieve a batch of randomly selected images from the training dataset. The real images are returned with corresponding target values for the discriminator model, e.g. y=1.0, to indicate they are real.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y
```

Listing 15.5: Example of a function for selecting a random sample of real images.

Next, we can develop the corresponding functions for the generator. First, a function for generating random points in the latent space to use as input for generating images via the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

Listing 15.6: Example of a function for generating random points in latent space.

Next, a function that will use the generator model to generate a batch of fake images for updating the discriminator model, along with the target value (y=0) to indicate the images are fake.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n_samples, 1))
  return X, y
```

Listing 15.7: Example of a function for generating synthetic images.

We need to use the generator periodically during training to generate images that we can subjectively inspect and use as the basis for choosing a final generator model. The

summarize_performance() function below can be called during training to generate and save a plot of images and save the generator model. Images are plotted using a reverse grayscale color map to make the digits black on a white background.

```
# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(10 * 10):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename1 = 'generated_plot_%06d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # save the generator model
  filename2 = 'model_%06d.h5' % (step+1)
  g_model.save(filename2)
  print('Saved %s and %s' % (filename1, filename2))
```

Listing 15.8: Example of a function for evaluating the generator model.

We are also interested in the behavior of loss during training. As such, we can record loss in lists across each training iteration, then create and save a line plot of the learning dynamics of the models. Creating and saving the plot of learning curves is implemented in the plot_history() function.

```
# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist):
  pyplot.plot(d1_hist, label='dloss1')
  pyplot.plot(d2_hist, label='dloss2')
  pyplot.plot(g_hist, label='gloss')
  pyplot.legend()
  filename = 'plot_line_plot_loss.png'
  pyplot.savefig(filename)
  pyplot.close()
  print('Saved %s' % (filename))
```

Listing 15.9: Example of a function for creating and saving plots of learning curves.

Finally, we can define the main training loop via the train() function. The function takes the defined models and dataset as arguments and parameterizes the number of training epochs and batch size as default function arguments. Each training loop involves first generating a half-batch of real and fake samples and using them to create one batch worth of weight updates to the discriminator. Next, the generator is updated via the composite model, providing the real (y=1) target as the expected output for the model. The loss is reported each training iteration, and the model performance is summarized in terms of a plot of generated images at the end of every epoch. The plot of learning curves is created and saved at the end of the run.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=20, n_batch=64):
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  # lists for storing loss, for plotting later
  d1_hist, d2_hist, g_hist = list(), list(), list()
  # manually enumerate epochs
  for i in range(n_steps):
    # prepare real and fake samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator model
    d_loss1 = d_model.train_on_batch(X_real, y_real)
    d_loss2 = d_model.train_on_batch(X_fake, y_fake)
    # update the generator via the discriminator's error
    z_input = generate_latent_points(latent_dim, n_batch)
    y_real2 = ones((n_batch, 1))
    g_loss = gan_model.train_on_batch(z_input, y_real2)
    # summarize loss on this batch
    print('>%d, d1=%.3f, d2=%.3f g=%.3f' % (i+1, d_loss1, d_loss2, g_loss))
    # record history
    d1_hist.append(d_loss1)
    d2_hist.append(d_loss2)
    g_hist.append(g_loss)
    # evaluate the model performance every 'epoch'
    if (i+1) % (bat_per_epo * 1) == 0:
      summarize_performance(i, g_model, latent_dim)
  # create line plot of training history
  plot_history(d1_hist, d2_hist, g_hist)
```

Listing 15.10: Example of a function for training the GAN models.

Tying all of this together, the complete code example of training an LSGAN on the MNIST handwritten digit dataset is listed below.

```
# example of lsgan for mnist
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import Activation
from keras.layers import LeakyReLU
from keras.layers import BatchNormalization
```

```
from keras.initializers import RandomNormal
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # downsample to 14x14
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      input_shape=in_shape))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 7x7
  model.add(Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dense(1, activation='linear', kernel_initializer=init))
  # compile model with L2 loss
  model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5))
  return model

# define the standalone generator model
def define_generator(latent_dim):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 256 * 7 * 7
  model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  model.add(Reshape((7, 7, 256)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  # upsample to 28x28
  model.add(Conv2DTranspose(64, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  # output 28x28x1
  model.add(Conv2D(1, (7,7), padding='same', kernel_initializer=init))
  model.add(Activation('tanh'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  for layer in discriminator.layers:
```

```
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model with L2 loss
  model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5))
  return model

# load mnist images
def load_real_samples():
  # load dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X

# # select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n_samples, 1))
  return X, y

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
```

```
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(10 * 10):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename1 = 'generated_plot_%06d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # save the generator model
  filename2 = 'model_%06d.h5' % (step+1)
  g_model.save(filename2)
  print('Saved %s and %s' % (filename1, filename2))

# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist):
  pyplot.plot(d1_hist, label='dloss1')
  pyplot.plot(d2_hist, label='dloss2')
  pyplot.plot(g_hist, label='gloss')
  pyplot.legend()
  filename = 'plot_line_plot_loss.png'
  pyplot.savefig(filename)
  pyplot.close()
  print('Saved %s' % (filename))

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=20, n_batch=64):
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  # lists for storing loss, for plotting later
  d1_hist, d2_hist, g_hist = list(), list(), list()
  # manually enumerate epochs
  for i in range(n_steps):
    # prepare real and fake samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator model
    d_loss1 = d_model.train_on_batch(X_real, y_real)
    d_loss2 = d_model.train_on_batch(X_fake, y_fake)
    # update the generator via the discriminator's error
    z_input = generate_latent_points(latent_dim, n_batch)
    y_real2 = ones((n_batch, 1))
    g_loss = gan_model.train_on_batch(z_input, y_real2)
    # summarize loss on this batch
    print('>%d, d1=%.3f, d2=%.3f g=%.3f' % (i+1, d_loss1, d_loss2, g_loss))
    # record history
    d1_hist.append(d_loss1)
```

```
    d2_hist.append(d_loss2)
    g_hist.append(g_loss)
    # evaluate the model performance every 'epoch'
    if (i+1) % (bat_per_epo * 1) == 0:
      summarize_performance(i, g_model, latent_dim)
  # create line plot of training history
  plot_history(d1_hist, d2_hist, g_hist)

# size of the latent space
latent_dim = 100
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
print(dataset.shape)
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Listing 15.11: Example of training the LSGAN on the MNIST training dataset.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

Running the example will report the loss of the discriminator on real ($d1$) and fake ($d2$) examples and the loss of the generator via the discriminator on generated examples presented as real (g). These scores are printed at the end of each training run and are expected to remain small values throughout the training process. Values of zero for an extended period may indicate a failure mode and the training process should be restarted.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
>1, d1=9.292, d2=0.153 g=2.530
>2, d1=1.173, d2=2.057 g=0.903
>3, d1=1.347, d2=1.922 g=2.215
>4, d1=0.604, d2=0.545 g=1.846
>5, d1=0.643, d2=0.734 g=1.619
...
```

Listing 15.12: Example output from training the LSGAN on the MNIST training dataset.

Plots of generated images are created at the end of every epoch. The generated images at the beginning of the run are rough.

Figure 15.3: Example of 100 LSGAN Generated Handwritten Digits after 1 Training Epoch.

After a handful of training epochs, the generated images begin to look crisp and realistic. Recall that more training epochs may or may not correspond to a generator that outputs higher quality images. Review the generated plots and choose a final model with the best quality images.

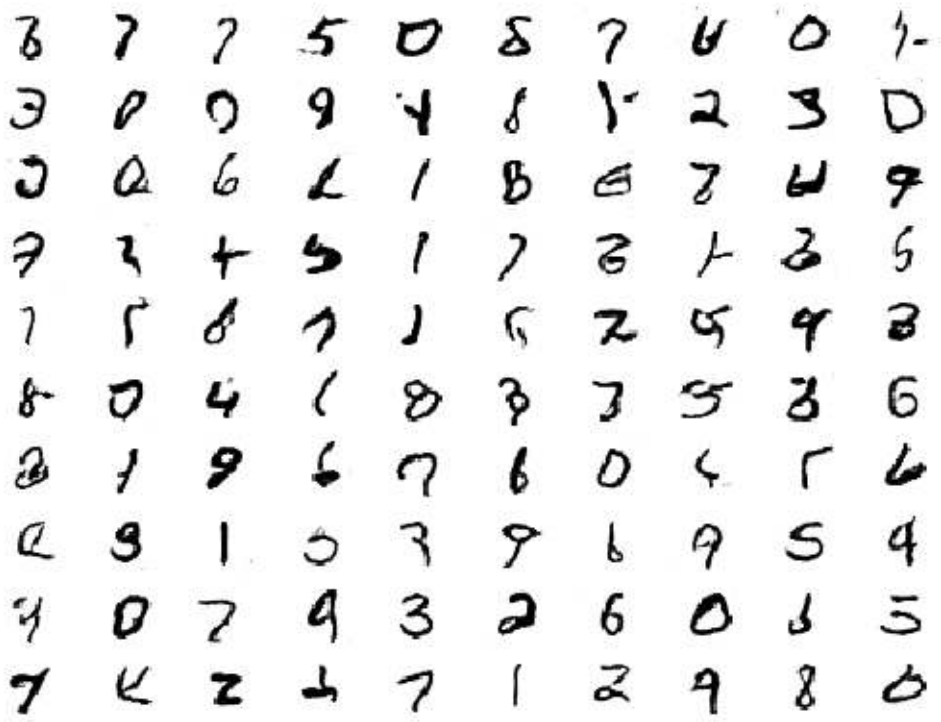

Figure 15.4: Example of 100 LSGAN Generated Handwritten Digits After 20 Training Epochs.

At the end of the training run, a plot of learning curves is created for the discriminator and generator. In this case, we can see that training remains somewhat stable throughout the run, with some very large peaks observed, which wash out the scale of the plot.

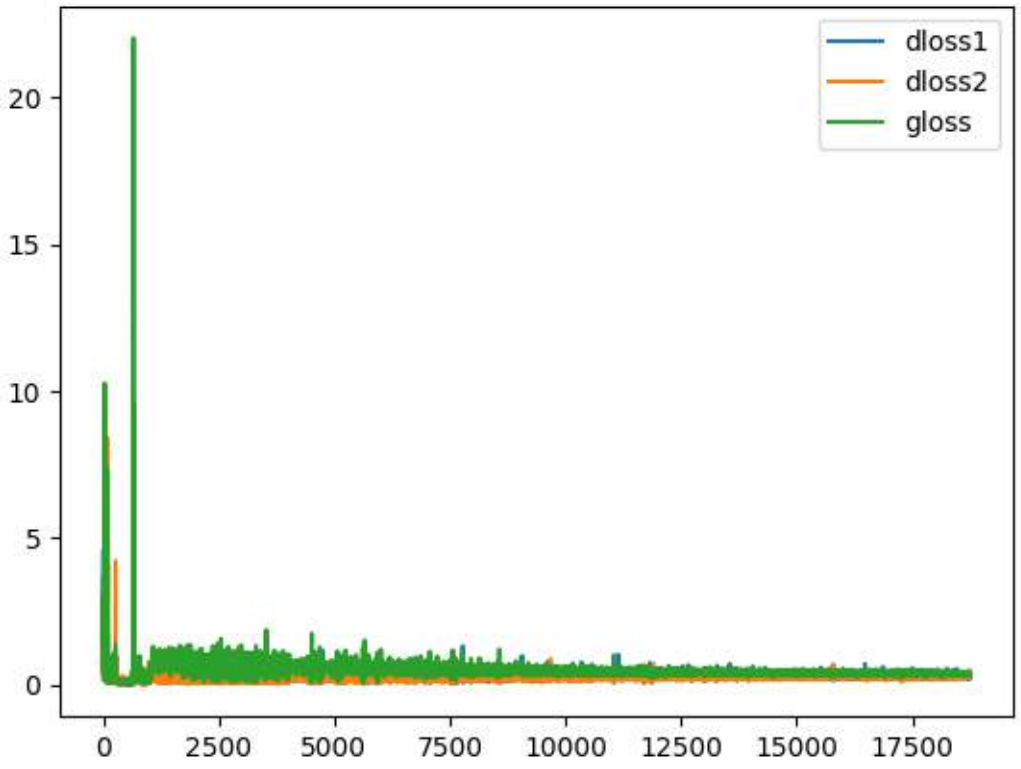

Figure 15.5: Plot of Learning Curves for the Generator and Discriminator in the LSGAN During Training.

15.4 How to Generate Images With LSGAN

We can use the saved generator model to create new images on demand. This can be achieved by first selecting a final model based on image quality, then loading it and providing new points from the latent space as input in order to generate new plausible images from the domain. In this case, we will use the model saved after 20 epochs, or 18,740 ($\frac{60000}{64}$ or 937 batches per epoch $\times$20 epochs) training iterations.

```
# example of loading the generator model and generating images
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

```
# create a plot of generated images (reversed grayscale)
def plot_generated(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  pyplot.show()

# load model
model = load_model('model_018740.h5')
# generate images
latent_points = generate_latent_points(100, 100)
# generate images
X = model.predict(latent_points)
# plot the result
plot_generated(X, 10)
```

Listing 15.13: Example of loading the saved LSGAN generator model and using it to generate images.

Running the example generates a plot of 10×10, or 100, new and plausible handwritten digits.

Figure 15.6: Plot of 100 LSGAN Generated Plausible Handwritten Digits.

15.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

15.5.1 Papers

- Least Squares Generative Adversarial Networks, 2016.
  https://arxiv.org/abs/1611.04076

15.5.2 API

- Keras Datasets API..
  https://keras.io/datasets/
- Keras Sequential Model API.
  https://keras.io/models/sequential/
- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/

- How can I "freeze" Keras layers?.
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers
- MatplotLib API.
  https://matplotlib.org/api/
- NumPy Random sampling (numpy.random) API.
  https://docs.scipy.org/doc/numpy/reference/routines.random.html
- NumPy Array manipulation routines.
  https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html

15.5.3 Articles

- Least Squares GAN, 2017.
  https://wiseodd.github.io/techblog/2017/03/02/least-squares-gan/
- LSGAN Project (Official), GitHub.
  https://github.com/xudonmao/LSGAN
- Keras-GAN Project, GitHub.
  https://github.com/eriklindernoren/Keras-GAN

15.6 Summary

In this tutorial, you discovered how to develop a least squares generative adversarial network. Specifically, you learned:

- The LSGAN addresses vanishing gradients and loss saturation of the deep convolutional GAN.
- The LSGAN can be implemented by a mean squared error or L2 loss function for the discriminator model.
- How to implement the LSGAN model for generating handwritten digits for the MNIST dataset.

15.6.1 Next

In the next tutorial, you will discover the Wasserstein loss function and the WGAN and how to implement it from scratch.

Chapter 16

How to Develop a Wasserstein GAN (WGAN)

The Wasserstein Generative Adversarial Network, or Wasserstein GAN, is an extension to the generative adversarial network that both improves the stability when training the model and provides a loss function that correlates with the quality of generated images. The development of the WGAN has a dense mathematical motivation, although in practice requires only a few minor modifications to the established standard deep convolutional generative adversarial network, or DCGAN. In this tutorial, you will discover how to implement the Wasserstein generative adversarial network from scratch. After completing this tutorial, you will know:

- The differences between the standard DCGAN and the new Wasserstein GAN.
- How to implement the specific details of the Wasserstein GAN from scratch.
- How to develop a WGAN for image generation and interpret model behavior.

Let's get started.

16.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Wasserstein Generative Adversarial Network
2. How to Implement Wasserstein Loss
3. Wasserstein GAN Implementation Details
4. How to Train a Wasserstein GAN Model
5. How to Generate Images With WGAN

16.2 What Is a Wasserstein GAN?

The Wasserstein GAN, or WGAN for short, was introduced by Martin Arjovsky, et al. in their 2017 paper titled *Wasserstein GAN*. It is an extension of the GAN that seeks an alternate way of training the generator model to better approximate the distribution of data observed in a given training dataset. Instead of using a discriminator to classify or predict the probability of generated images as being real or fake, the WGAN changes or replaces the discriminator model with a critic that scores the realness or fakeness of a given image. This change is motivated by a mathematical argument that training the generator should seek a minimization of the distance between the distribution of the data observed in the training dataset and the distribution observed in generated examples. The argument contrasts different distribution distance measures, such as Kullback-Leibler (KL) divergence, Jensen-Shannon (JS) divergence, and the Earth-Mover (EM) distance, the latter also referred to as Wasserstein distance.

> The most fundamental difference between such distances is their impact on the convergence of sequences of probability distributions.
>
> — *Wasserstein GAN*, 2017.

They demonstrate that a critic neural network can be trained to approximate the Wasserstein distance, and, in turn, used to effectively train a generator model. Importantly, the Wasserstein distance has the properties that it is continuous and differentiable and continues to provide a linear gradient, even after the critic is well trained.

> The fact that the EM distance is continuous and differentiable a.e. means that we can (and should) train the critic till optimality. [...] the more we train the critic, the more reliable gradient of the Wasserstein we get, which is actually useful by the fact that Wasserstein is differentiable almost everywhere.
>
> — *Wasserstein GAN*, 2017.

This is unlike the discriminator model of the DCGAN that, once trained, may fail to provide useful gradient information for updating the generator model. The benefit of the WGAN is that the training process is more stable and less sensitive to model architecture and choice of hyperparameter configurations.

> ... training WGANs does not require maintaining a careful balance in training of the discriminator and the generator, and does not require a careful design of the network architecture either. The mode dropping phenomenon that is typical in GANs is also drastically reduced.
>
> — *Wasserstein GAN*, 2017.

Perhaps most importantly, the loss of the discriminator appears to relate to the quality of images created by the generator. Specifically, the lower the loss of the critic when evaluating generated images, the higher the expected quality of the generated images. This is important as unlike other GANs that seek stability in terms of finding an equilibrium between two models, the WGAN seeks convergence, lowering generator loss.

> To our knowledge, this is the first time in GAN literature that such a property is shown, where the loss of the GAN shows properties of convergence. This property is extremely useful when doing research in adversarial networks as one does not need to stare at the generated samples to figure out failure modes and to gain information on which models are doing better over others.
>
> — *Wasserstein GAN*, 2017.

16.3 How to Implement Wasserstein Loss

The Wasserstein loss function seeks to increase the gap between the scores for real and generated images. We can summarize the function as it is described in the paper as follows:

- Critic Loss = [average critic score on real images] - [average critic score on fake images]
- Generator Loss = -[average critic score on fake images]

Where the average scores are calculated across a minibatch of samples. This is precisely how the loss is implemented for graph-based deep learning frameworks such as PyTorch and TensorFlow. The calculations are straightforward to interpret once we recall that stochastic gradient descent seeks to minimize loss. In the case of the generator, a larger score from the critic will result in a smaller loss for the generator, encouraging the critic to output larger scores for fake images. For example, an average score of 10 becomes -10, an average score of 50 becomes -50, which is smaller, and so on.

In the case of the critic, a larger score for real images results in a larger resulting loss for the critic, penalizing the model. This encourages the critic to output smaller scores for real images. For example, an average score of 20 for real images and 50 for fake images results in a loss of -30; an average score of 10 for real images and 50 for fake images results in a loss of -40, which is better, and so on. The sign of the loss does not matter in this case, as long as loss for real images is a small number and the loss for fake images is a large number. The Wasserstein loss encourages the critic to separate these numbers. We can also reverse the situation and encourage the critic to output a large score for real images and a small score for fake images and achieve the same result. Some implementations make this change.

In the Keras deep learning library (and some others), we cannot implement the Wasserstein loss function directly as described in the paper and as implemented in PyTorch and TensorFlow. Instead, we can achieve the same effect without having the calculation of the loss for the critic dependent upon the loss calculated for real and fake images. A good way to think about this is a negative score for real images and a positive score for fake images, although this negative/positive split of scores learned during training is not required; just larger and smaller is sufficient.

- Small Critic Score (e.g.$<$ 0): Real
- Large Critic Score (e.g. $>$0): Fake

We can multiply the average predicted score by -1 in the case of real images so that larger averages become smaller averages and the gradient is in the correct direction, i.e. minimizing loss. For example, average scores on real images of [0.5, 0.8, and 1.0] across three batches of real images would become [-0.5, -0.8, and -1.0] when calculating weight updates.

- Loss For Real Images = -1 × Average Critic Score

No change is needed for the case of fake scores, as we want to encourage smaller average scores for real images.

- Loss For Fake Images = Average Critic Score

This can be implemented consistently by assigning an expected outcome target of -1 for real images and 1 for fake images and implementing the loss function as the expected label multiplied by the average score. The -1 label will be multiplied by the average score for real images and encourage a larger predicted average, and the 1 label will be multiplied by the average score for fake images and have no effect, encouraging a smaller predicted average.

- Wasserstein Loss = Label × Average Critic Score

Or

- Wasserstein Loss(Fake Images) = 1 × Average Predicted Score
- Wasserstein Loss(Real Images) = -1 × Average Predicted Score

We can implement this in Keras by assigning the expected labels of -1 and 1 for real and fake images respectively. The inverse labels could be used to the same effect, e.g. -1 for fake and 1 for real to encourage small scores for real images and large scores for fake images. Some developers do implement the WGAN in this alternate way, which is just as correct. Now that we know how to implement the Wasserstein loss function in Keras, let's clarify one common point of misunderstanding.

16.3.1 Common Point of Confusion With Expected Labels

Recall we are using the expected labels of -1 for real images and 1 for fake images. A common point of confusion is that a perfect critic model will output -1 for every real image and 1 for every fake image. This is incorrect.

Again, recall we are using stochastic gradient descent to find the set of weights in the critic (and generator) models that minimize the loss function. We have established that we want the critic model to output larger scores on average for fake images and smaller scores on average for real images. We then designed a loss function to encourage this outcome. This is the key point about loss functions used to train neural network models. They encourage a desired model behavior, and they do not have to achieve this by providing the expected outcomes. In this case, we defined our Wasserstein loss function to interpret the average score predicted by the critic model and used labels for the real and fake cases to help with this interpretation. So what is a good loss for real and fake images under Wasserstein loss?

Wasserstein is not an absolute and comparable loss for comparing across GAN models. Instead, it is relative and depends on your model configuration and dataset. What is important is that it is consistent for a given critic model and convergence of the generator (better loss) does correlate with better generated image quality. It could be negative scores for real images and positive scores for fake images, but this is not required. All scores could be positive or all scores could be negative. The loss function only encourages a separation between scores for fake and real images as larger and smaller, not necessarily positive and negative.

16.4 Wasserstein GAN Implementation Details

Although the theoretical grounding for the WGAN is dense, the implementation of a WGAN requires a few minor changes to the standard Deep Convolutional GAN, or DCGAN. The image below provides a summary of the main training loop for training a WGAN, taken from the paper. Take note of the listing of recommended hyperparameters used in the model.

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.
while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Figure 16.1: Algorithm for the Wasserstein Generative Adversarial Networks. Taken from: Wasserstein GAN.

The differences in implementation for the WGAN are as follows:

1. Use a linear activation function in the output layer of the critic model (instead of sigmoid).
2. Use -1 labels for real images and 1 labels for fake images (instead of 1 and 0).
3. Use Wasserstein loss to train the critic and generator models.
4. Constrain critic model weights to a limited range after each minibatch update (e.g. [-0.01,0.01]).
5. Update the critic model more times than the generator each iteration (e.g. 5).
6. Use the RMSProp version of gradient descent with a small learning rate and no momentum (e.g. 0.00005).

Using the standard DCGAN model as a starting point, let's take a look at each of these implementation details in turn.

16.4.1 Linear Activation in Critic Output Layer

The DCGAN uses the sigmoid activation function in the output layer of the discriminator to predict the likelihood of a given image being real. In the WGAN, the critic model requires a linear activation to predict the score of *realness* for a given image. This can be achieved by setting the `activation` argument to `'linear'` in the output layer of the critic model.

```
# define output layer of the critic model
...
model.add(Dense(1, activation='linear'))
```

Listing 16.1: Example of the linear activation function in the output layer of the discriminator.

The linear activation is the default activation for a layer, so we can, in fact, leave the activation unspecified to achieve the same result.

```
# define output layer of the critic model
...
model.add(Dense(1))
```

Listing 16.2: Example of the linear activation function in the output layer of the discriminator.

16.4.2 Class Labels for Real and Fake Images

The DCGAN uses the class 0 for fake images and class 1 for real images, and these class labels are used to train the GAN. In the DCGAN, these are precise labels that the discriminator is expected to achieve. The WGAN does not have precise labels for the critic. Instead, it encourages the critic to output scores that are different for real and fake images. This is achieved via the Wasserstein function that cleverly makes use of positive and negative class labels. The WGAN can be implemented where -1 class labels are used for real images and 1 class labels are used for fake or generated images. This can be achieved using the `ones()` NumPy function. For example:

```
...
# generate class labels, -1 for 'real'
y = -ones((n_samples, 1))
...
# create class labels with 1.0 for 'fake'
y = ones((n_samples, 1))
```

Listing 16.3: Example of defining the target values for real and fake images.

16.4.3 Wasserstein Loss Function

We can implement the Wasserstein loss as a custom function in Keras that calculates the average score for real or fake images. The score is maximizing for real examples and minimizing for fake examples. Given that stochastic gradient descent is a minimization algorithm, we can multiply the class label by the mean score (e.g. -1 for real and 1 for fake which as no effect), which ensures that the loss for real and fake images is minimizing to the network. An efficient implementation of this loss function for Keras is listed below.

```
from keras import backend

# implementation of wasserstein loss
def wasserstein_loss(y_true, y_pred):
  return backend.mean(y_true * y_pred)
```

Listing 16.4: Example of function for calculating Wasserstein loss.

This loss function can be used to train a Keras model by specifying the function name when compiling the model. For example:

```
...
# compile the model
model.compile(loss=wasserstein_loss, ...)
```

Listing 16.5: Example of compiling a model to use Wasserstein loss.

16.4.4 Critic Weight Clipping

The DCGAN does not use any gradient clipping, although the WGAN requires gradient clipping for the critic model. We can implement weight clipping as a Keras constraint. This is a class that must extend the Constraint class and define an implementation of the `__call__()` function for applying the operation and the `get_config()` function for returning any configuration. We can also define an `__init__()` function to set the configuration, in this case, the symmetrical size of the bounding box for the weight hypercube, e.g. 0.01. The `ClipConstraint` class is defined below.

```
# clip model weights to a given hypercube
class ClipConstraint(Constraint):
  # set clip value when initialized
  def __init__(self, clip_value):
    self.clip_value = clip_value

  # clip model weights to hypercube
  def __call__(self, weights):
    return backend.clip(weights, -self.clip_value, self.clip_value)

  # get the config
  def get_config(self):
    return {'clip_value': self.clip_value}
```

Listing 16.6: Example of a custom weight constraint class.

To use the constraint, the class can be constructed, then used in a layer by setting the `kernel_constraint` argument; for example:

```
...
# define the constraint
const = ClipConstraint(0.01)
...
# use the constraint in a layer
model.add(Conv2D(..., kernel_constraint=const))
```

Listing 16.7: Example of using the custom weight constraint in a layer.

The constraint is only required when updating the critic model.

16.4.5 Update Critic More Than Generator

In the DCGAN, the generator and the discriminator model must be updated in equal amounts. Specifically, the discriminator is updated with a half batch of real and a half batch of fake samples each iteration, whereas the generator is updated with a single batch of generated samples (described in Chapter 4). For example:

```
...
# main gan training loop
for i in range(n_steps):

  # update the discriminator

  # get randomly selected 'real' samples
  X_real, y_real = generate_real_samples(dataset, half_batch)
  # update critic model weights
  c_loss1 = c_model.train_on_batch(X_real, y_real)
  # generate 'fake' examples
  X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
  # update critic model weights
  c_loss2 = c_model.train_on_batch(X_fake, y_fake)

  # update generator

  # prepare points in latent space as input for the generator
  X_gan = generate_latent_points(latent_dim, n_batch)
  # create inverted labels for the fake samples
  y_gan = ones((n_batch, 1))
  # update the generator via the critic's error
  g_loss = gan_model.train_on_batch(X_gan, y_gan)
```

Listing 16.8: Example of the DCGAN training algorithm.

In the WGAN model, the critic model must be updated more than the generator model. Specifically, a new hyperparameter is defined to control the number of times that the critic is updated for each update to the generator model, called `n_critic`, and is set to 5. This can be implemented as a new loop within the main GAN update loop; for example:

```
...
# main gan training loop
for i in range(n_steps):

  # update the critic
  for _ in range(n_critic):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update critic model weights
    c_loss1 = c_model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update critic model weights
    c_loss2 = c_model.train_on_batch(X_fake, y_fake)

  # update generator

  # prepare points in latent space as input for the generator
  X_gan = generate_latent_points(latent_dim, n_batch)
  # create inverted labels for the fake samples
  y_gan = ones((n_batch, 1))
  # update the generator via the critic's error
  g_loss = gan_model.train_on_batch(X_gan, y_gan)
```

Listing 16.9: Example of the WGAN training algorithm.

16.4.6 Use RMSProp Stochastic Gradient Descent

The DCGAN uses the Adam version of stochastic gradient descent with a small learning rate and modest momentum. The WGAN recommends the use of RMSProp instead, with a small learning rate of 0.00005. This can be implemented in Keras when the model is compiled. For example:

```
...
# compile model
opt = RMSprop(lr=0.00005)
model.compile(loss=wasserstein_loss, optimizer=opt)
```

Listing 16.10: Example of compiling a model to use RMSProp stochastic gradient descent.

16.5 How to Train a Wasserstein GAN Model

Now that we know the specific implementation details for the WGAN, we can implement the model for image generation. In this section, we will develop a WGAN to generate a single handwritten digit (the number seven) from the MNIST dataset (described in Section 7.2). This is a good test problem for the WGAN as it is a small dataset requiring a modest mode that is quick to train. The first step is to define the models.

The critic model takes as input one 28×28 grayscale image and outputs a score for the realness or fakeness of the image. It is implemented as a modest convolutional neural network using best practices for DCGAN design such as using the `LeakyReLU` activation function with a slope of 0.2, batch normalization, and using a 2×2 stride to downsample. The critic model makes use of the new `ClipConstraint` weight constraint to clip model weights after minibatch updates and is optimized using the custom `wasserstein_loss()` function, and the RMSProp version of stochastic gradient descent with a learning rate of 0.00005. The `define_critic()` function below implements this, defining and compiling the critic model and returning it. The input shape of the image is parameterized as a default function argument to make it clear.

```
# define the standalone critic model
def define_critic(in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # weight constraint
  const = ClipConstraint(0.01)
  # define model
  model = Sequential()
  # downsample to 14x14
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      kernel_constraint=const, input_shape=in_shape))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 7x7
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      kernel_constraint=const))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # scoring, linear activation
  model.add(Flatten())
  model.add(Dense(1))
```

```
  # compile model
  opt = RMSprop(lr=0.00005)
  model.compile(loss=wasserstein_loss, optimizer=opt)
  return model
```

Listing 16.11: Example of a function for defining the critic model.

The generator model takes as input a point in the latent space and outputs a single 28×28 grayscale image. This is achieved by using a fully connected layer to interpret the point in the latent space and provide sufficient activations that can be reshaped into many copies (in this case, 128) of a low-resolution version of the output image (e.g. 7×7). This is then upsampled two times, doubling the size and quadrupling the area of the activations each time using transpose convolutional layers. The model uses best practices such as the `LeakyReLU` activation, a kernel size that is a factor of the stride size, and a hyperbolic tangent (Tanh) activation function in the output layer.

The `define_generator()` function below defines the generator model but intentionally does not compile it as it is not trained directly, then returns the model. The size of the latent space is parameterized as a function argument.

```
# define the standalone generator model
def define_generator(latent_dim):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # output 28x28x1
  model.add(Conv2D(1, (7,7), activation='tanh', padding='same', kernel_initializer=init))
  return model
```

Listing 16.12: Example of a function for defining the generator model.

Next, a GAN model can be defined that combines both the generator model and the critic model into one larger model. This larger model will be used to train the model weights in the generator, using the output and error calculated by the critic model. The critic model is trained separately, and as such, the model weights are marked as not trainable in this larger GAN model to ensure that only the weights of the generator model are updated. This change to the trainability of the critic weights only has an effect when training the combined GAN model, not when training the critic standalone.

This larger GAN model takes as input a point in the latent space, uses the generator model to generate an image, which is fed as input to the critic model, then scored as real or fake. The model is fit using RMSProp with the custom `wasserstein_loss()` function. The `define_gan()` function below implements this, taking the already defined generator and critic models as input.

```
# define the combined generator and critic model, for updating the generator
def define_gan(generator, critic):
  # make weights in the critic not trainable
  for layer in critic.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the critic
  model.add(critic)
  # compile model
  opt = RMSprop(lr=0.00005)
  model.compile(loss=wasserstein_loss, optimizer=opt)
  return model
```

Listing 16.13: Example of a function for defining the composite model for training the generator.

Now that we have defined the GAN model, we need to train it. But, before we can train the model, we require input data. The first step is to load and scale the MNIST dataset. The whole dataset is loaded via a call to the `load_data()` Keras function, then a subset of the images is selected (about 5,000) that belongs to class 7, e.g. are a handwritten depiction of the number seven. Then the pixel values must be scaled to the range [-1,1] to match the output of the generator model. The `load_real_samples()` function below implements this, returning the loaded and scaled subset of the MNIST training dataset ready for modeling.

```
# load images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # select all of the examples for a given class
  selected_ix = trainy == 7
  X = trainX[selected_ix]
  # expand to 3d, e.g. add channels
  X = expand_dims(X, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X
```

Listing 16.14: Example of a function for loading and preparing a subset of the MNIST training dataset.

We will require one batch (or a half batch) of real images from the dataset each update to the GAN model. A simple way to achieve this is to select a random sample of images from the dataset each time. The `generate_real_samples()` function below implements this, taking the prepared dataset as an argument, selecting and returning a random sample of images and their corresponding label for the critic, specifically $target = -1$ indicating that they are real images.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels, -1 for 'real'
  y = -ones((n_samples, 1))
  return X, y
```

Listing 16.15: Example of a function for selecting a random subset of real images.

Next, we need inputs for the generator model. These are random points from the latent space, specifically Gaussian distributed random variables. The `generate_latent_points()` function implements this, taking the size of the latent space as an argument and the number of points required, and returning them as a batch of input samples for the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

Listing 16.16: Example of a function for generating random points in latent space.

Next, we need to use the points in the latent space as input to the generator in order to generate new images. The `generate_fake_samples()` function below implements this, taking the generator model and size of the latent space as arguments, then generating points in the latent space and using them as input to the generator model. The function returns the generated images and their corresponding label for the critic model, specifically $target = 1$ to indicate they are fake or generated.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels with 1.0 for 'fake'
  y = ones((n_samples, 1))
  return X, y
```

Listing 16.17: Example of a function for generating synthetic images with the generator model.

We need to record the performance of the model. Perhaps the most reliable way to evaluate the performance of a GAN is to use the generator to generate images, and then review and subjectively evaluate them. The `summarize_performance()` function below takes the generator model at a given point during training and uses it to generate 100 images in a 10×10 grid, that are then plotted and saved to file. The model is also saved to file at this time, in case we would like to use it later to generate more images.

```
# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
```

```
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(10 * 10):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename1 = 'generated_plot_%04d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # save the generator model
  filename2 = 'model_%04d.h5' % (step+1)
  g_model.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))
```

Listing 16.18: Example of a function for summarizing the performance of the model.

In addition to image quality, it is a good idea to keep track of the loss and accuracy of the model over time. The loss for the critic for real and fake samples can be tracked for each model update, as can the loss for the generator for each update. These can then be used to create line plots of loss at the end of the training run. The `plot_history()` function below implements this and saves the results to file.

```
# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist):
  # plot history
  pyplot.plot(d1_hist, label='crit_real')
  pyplot.plot(d2_hist, label='crit_fake')
  pyplot.plot(g_hist, label='gen')
  pyplot.legend()
  pyplot.savefig('plot_line_plot_loss.png')
  pyplot.close()
```

Listing 16.19: Example of a function for generating and saving plots of the model learning curves.

We are now ready to fit the GAN model. The model is fit for 10 training epochs, which is arbitrary, as the model begins generating plausible number-7 digits after perhaps the first few epochs. A batch size of 64 samples is used, and each training epoch involves $\frac{6265}{64}$, or about 97, batches of real and fake samples and updates to the model. The model is therefore trained for 10 epochs of 97 batches, or 970 iterations. First, the critic model is updated for a half batch of real samples, then a half batch of fake samples, together forming one batch of weight updates. This is then repeated `n_critic` (5) times as required by the WGAN algorithm. The generator is then updated via the composite GAN model. Importantly, the target label is set to -1 or real for the generated samples. This has the effect of updating the generator toward getting better at generating real samples on the next batch.

The `train()` function below implements this, taking the defined models, dataset, and size of the latent dimension as arguments and parameterizing the number of epochs and batch size

with default arguments. The generator model is saved at the end of training. The performance of the critic and generator models is reported each iteration. Sample images are generated and saved every epoch, and line plots of model performance are created and saved at the end of the run.

```
# train the generator and critic
def train(g_model, c_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=64,
    n_critic=5):
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  # lists for keeping track of loss
  c1_hist, c2_hist, g_hist = list(), list(), list()
  # manually enumerate epochs
  for i in range(n_steps):
    # update the critic more than the generator
    c1_tmp, c2_tmp = list(), list()
    for _ in range(n_critic):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # update critic model weights
      c_loss1 = c_model.train_on_batch(X_real, y_real)
      c1_tmp.append(c_loss1)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update critic model weights
      c_loss2 = c_model.train_on_batch(X_fake, y_fake)
      c2_tmp.append(c_loss2)
    # store critic loss
    c1_hist.append(mean(c1_tmp))
    c2_hist.append(mean(c2_tmp))
    # prepare points in latent space as input for the generator
    X_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = -ones((n_batch, 1))
    # update the generator via the critic's error
    g_loss = gan_model.train_on_batch(X_gan, y_gan)
    g_hist.append(g_loss)
    # summarize loss on this batch
    print('>%d, c1=%.3f, c2=%.3f g=%.3f' % (i+1, c1_hist[-1], c2_hist[-1], g_loss))
    # evaluate the model performance every 'epoch'
    if (i+1) % bat_per_epo == 0:
      summarize_performance(i, g_model, latent_dim)
  # line plots of loss
  plot_history(c1_hist, c2_hist, g_hist)
```

Listing 16.20: Example of a function for training the WGAN models.

Now that all of the functions have been defined, we can create the models, load the dataset, and begin the training process.

```
...
# size of the latent space
```

```
latent_dim = 50
# create the critic
critic = define_critic()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, critic)
# load image data
dataset = load_real_samples()
print(dataset.shape)
# train model
train(generator, critic, gan_model, dataset, latent_dim)
```

Listing 16.21: Example of configuring and starting the training process.

Tying all of this together, the complete example is listed below.

```
# example of a wgan for generating handwritten digits
from numpy import expand_dims
from numpy import mean
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras import backend
from keras.optimizers import RMSprop
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import BatchNormalization
from keras.initializers import RandomNormal
from keras.constraints import Constraint
from matplotlib import pyplot

# clip model weights to a given hypercube
class ClipConstraint(Constraint):
  # set clip value when initialized
  def __init__(self, clip_value):
    self.clip_value = clip_value

  # clip model weights to hypercube
  def __call__(self, weights):
    return backend.clip(weights, -self.clip_value, self.clip_value)

  # get the config
  def get_config(self):
    return {'clip_value': self.clip_value}

# calculate wasserstein loss
def wasserstein_loss(y_true, y_pred):
  return backend.mean(y_true * y_pred)

# define the standalone critic model
```

```
def define_critic(in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # weight constraint
  const = ClipConstraint(0.01)
  # define model
  model = Sequential()
  # downsample to 14x14
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      kernel_constraint=const, input_shape=in_shape))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # downsample to 7x7
  model.add(Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init,
      kernel_constraint=const))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # scoring, linear activation
  model.add(Flatten())
  model.add(Dense(1))
  # compile model
  opt = RMSprop(lr=0.00005)
  model.compile(loss=wasserstein_loss, optimizer=opt)
  return model

# define the standalone generator model
def define_generator(latent_dim):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, kernel_initializer=init, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # output 28x28x1
  model.add(Conv2D(1, (7,7), activation='tanh', padding='same', kernel_initializer=init))
  return model

# define the combined generator and critic model, for updating the generator
def define_gan(generator, critic):
  # make weights in the critic not trainable
  for layer in critic.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
```

```
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the critic
  model.add(critic)
  # compile model
  opt = RMSprop(lr=0.00005)
  model.compile(loss=wasserstein_loss, optimizer=opt)
  return model

# load images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # select all of the examples for a given class
  selected_ix = trainy == 7
  X = trainX[selected_ix]
  # expand to 3d, e.g. add channels
  X = expand_dims(X, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels, -1 for 'real'
  y = -ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels with 1.0 for 'fake'
  y = ones((n_samples, 1))
  return X, y

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
```

```
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(10 * 10):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename1 = 'generated_plot_%04d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # save the generator model
  filename2 = 'model_%04d.h5' % (step+1)
  g_model.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))

# create a line plot of loss for the gan and save to file
def plot_history(d1_hist, d2_hist, g_hist):
  # plot history
  pyplot.plot(d1_hist, label='crit_real')
  pyplot.plot(d2_hist, label='crit_fake')
  pyplot.plot(g_hist, label='gen')
  pyplot.legend()
  pyplot.savefig('plot_line_plot_loss.png')
  pyplot.close()

# train the generator and critic
def train(g_model, c_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=64,
    n_critic=5):
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  # lists for keeping track of loss
  c1_hist, c2_hist, g_hist = list(), list(), list()
  # manually enumerate epochs
  for i in range(n_steps):
    # update the critic more than the generator
    c1_tmp, c2_tmp = list(), list()
    for _ in range(n_critic):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # update critic model weights
      c_loss1 = c_model.train_on_batch(X_real, y_real)
      c1_tmp.append(c_loss1)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update critic model weights
      c_loss2 = c_model.train_on_batch(X_fake, y_fake)
```

```
      c2_tmp.append(c_loss2)
    # store critic loss
    c1_hist.append(mean(c1_tmp))
    c2_hist.append(mean(c2_tmp))
    # prepare points in latent space as input for the generator
    X_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = -ones((n_batch, 1))
    # update the generator via the critic's error
    g_loss = gan_model.train_on_batch(X_gan, y_gan)
    g_hist.append(g_loss)
    # summarize loss on this batch
    print('>%d, c1=%.3f, c2=%.3f g=%.3f' % (i+1, c1_hist[-1], c2_hist[-1], g_loss))
    # evaluate the model performance every 'epoch'
    if (i+1) % bat_per_epo == 0:
      summarize_performance(i, g_model, latent_dim)
  # line plots of loss
  plot_history(c1_hist, c2_hist, g_hist)

# size of the latent space
latent_dim = 50
# create the critic
critic = define_critic()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, critic)
# load image data
dataset = load_real_samples()
print(dataset.shape)
# train model
train(generator, critic, gan_model, dataset, latent_dim)
```

Listing 16.22: Example of training the WGAN on the MNIST training dataset.

Running the example is quick, taking approximately 10 minutes on modern hardware without a GPU. First, the loss of the critic and generator models is reported to the console each iteration of the training loop. Specifically, $c1$ is the loss of the critic on real examples, $c2$ is the loss of the critic in generated samples, and g is the loss of the generator trained via the critic. The $c1$ scores are inverted as part of the loss function; this means if they are reported as negative, then they are really positive, and if they are reported as positive, they are really negative. The sign of the $c2$ scores is unchanged.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

Recall that the Wasserstein loss seeks scores for real and fake that are more different during training. We can see this towards the end of the run, such as the final epoch where the $c1$ loss for real examples is 5.338 (really -5.338) and the $c2$ loss for fake examples is -14.260, and this separation of about 10 units is consistent at least for the prior few iterations. We can also see that in this case, the model is scoring the loss of the generator at around 20. Again, recall that we update the generator via the critic model and treat the generated examples as real with the target of -1, therefore the score can be interpreted as a value around -20, close to the loss for fake samples.

```
...
>961, c1=5.110, c2=-15.388 g=19.579
>962, c1=6.116, c2=-15.222 g=20.054
>963, c1=4.982, c2=-15.192 g=21.048
>964, c1=4.238, c2=-14.689 g=23.911
>965, c1=5.585, c2=-14.126 g=19.578
>966, c1=4.807, c2=-14.755 g=20.034
>967, c1=6.307, c2=-16.538 g=19.572
>968, c1=4.298, c2=-14.178 g=17.956
>969, c1=4.283, c2=-13.398 g=17.326
>970, c1=5.338, c2=-14.260 g=19.927
```

Listing 16.23: Example output from training the WGAN on the MNIST training dataset.

Line plots for loss are created and saved at the end of the run. The plot shows the loss for the critic on real samples (blue), the loss for the critic on fake samples (orange), and the loss for the critic when updating the generator with fake samples (green). There is one important factor when reviewing learning curves for the WGAN and that is the trend. The benefit of the WGAN is that the loss correlates with generated image quality. Lower loss means better quality images, for a stable training process. In this case, lower loss specifically refers to lower Wasserstein loss for generated images as reported by the critic (orange line). This sign of this loss is not inverted by the target label (e.g. the target label is 1.0), therefore, a well-performing WGAN should show this line trending down as the image quality of the generated model is increased.

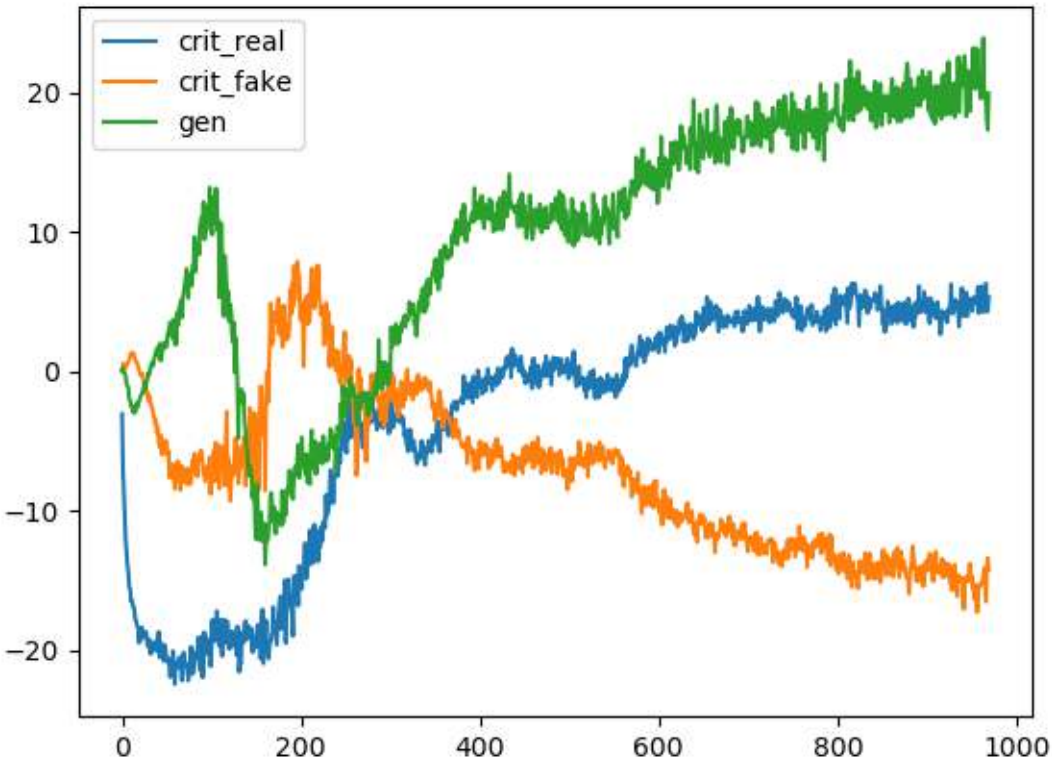

Figure 16.2: Line Plots of Loss and Accuracy for a Wasserstein Generative Adversarial Network.

In this case, more training seems to result in better quality generated images, with a major hurdle occurring around epoch 200-300 after which quality remains pretty good for the model. Before and around this hurdle, image quality is poor; for example:

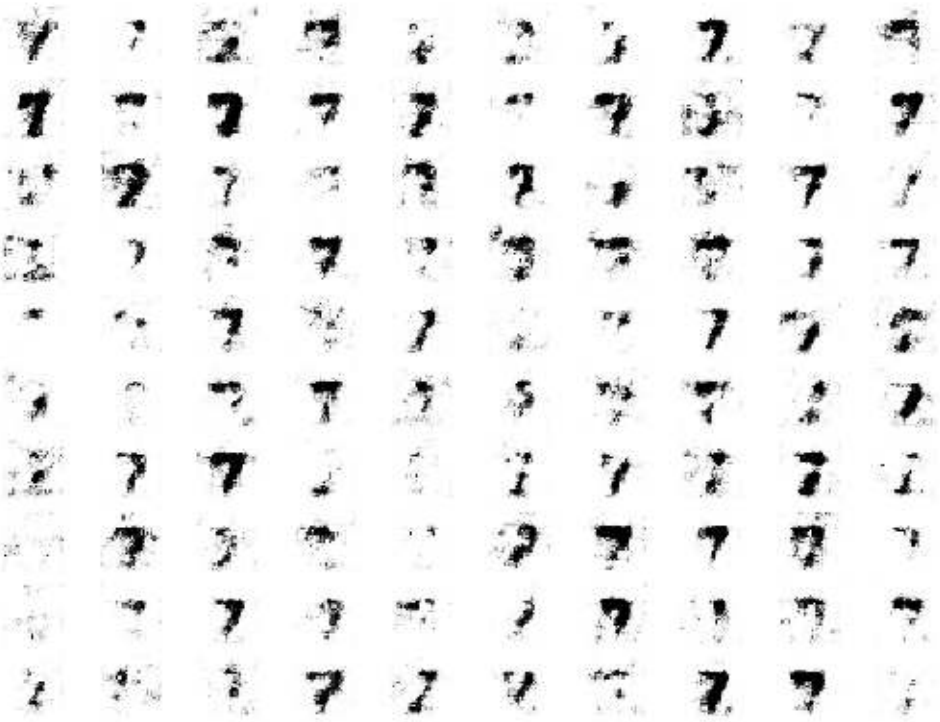

Figure 16.3: Sample of 100 Generated Images of a Handwritten Number 7 at Epoch 97 from a Wasserstein GAN.

After this epoch, the WGAN begins to generate plausible handwritten digits.

Figure 16.4: Sample of 100 Generated Images of a Handwritten Number 7 at Epoch 970 from a Wasserstein GAN.

16.6 How to Generate Images With WGAN

We can use the saved generator model to create new images on demand. This can be achieved by first selecting a final model based on image quality, then loading it and providing new points from the latent space as input in order to generate new plausible images from the domain. In this case, we will use the model saved after 10 epochs, or 970 training iterations.

```
# example of loading the generator model and generating images
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# create a plot of generated images (reversed grayscale)
```

```
def plot_generated(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  pyplot.show()

# load model
model = load_model('model_0970.h5')
# generate images
latent_points = generate_latent_points(50, 25)
# generate images
X = model.predict(latent_points)
# plot the result
plot_generated(X, 5)
```

Listing 16.24: Example of loading the saved WGAN generator model and using it to generate images.

Running the example generates a plot of 5×5, or 25, new and plausible handwritten number seven digits.

Figure 16.5: Plot of 25 WGAN Generated Plausible Handwritten Number Seven Digits.

16.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

16.7.1 Papers

- Wasserstein GAN, 2017.
  https://arxiv.org/abs/1701.07875
- Improved Training of Wasserstein GANs, 2017.
  https://arxiv.org/abs/1704.00028

16.7.2 API

- Keras Datasets API..
  https://keras.io/datasets/
- Keras Sequential Model API.
  https://keras.io/models/sequential/
- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/
- How can I "freeze" Keras layers?.
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers
- MatplotLib API.
  https://matplotlib.org/api/
- NumPy Random sampling (numpy.random) API.
  https://docs.scipy.org/doc/numpy/reference/routines.random.html
- NumPy Array manipulation routines.
  https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html

16.7.3 Articles

- Wasserstein GAN, GitHub.
  https://github.com/martinarjovsky/WassersteinGAN
- Wasserstein Generative Adversarial Networks (WGANS) Project, GitHub.
  https://github.com/kpandey008/wasserstein-gans
- Keras-GAN: Keras implementations of Generative Adversarial Networks, GitHub.
  https://github.com/eriklindernoren/Keras-GAN
- Improved WGAN, `keras-contrib` Project, GitHub.
  https://github.com/keras-team/keras-contrib/blob/master/examples/improved_wgan.py

16.8 Summary

In this tutorial, you discovered how to implement the Wasserstein generative adversarial network from scratch. Specifically, you learned:

- The differences between the standard DCGAN and the new Wasserstein GAN.
- How to implement the specific details of the Wasserstein GAN from scratch.
- How to develop a WGAN for image generation and interpret model behavior.

16.8.1 Next

This was the final tutorial in this part. In the next part, you will explore a suite of different conditional GAN models.

Part V

Conditional GANs

Overview

In this part you will discover conditional GANs that add some control over the types of images generated by the model. After reading the chapters in this part, you will know:

- Discover how to generate class-conditional images with the cGAN (Chapter 17).
- Discover how to add control variables to influence image generation with the InfoGAN (Chapter 18).
- Discover how to improve the image generation process by adding an auxiliary classifier model with the AC-GAN (Chapter 19).
- Discover how to train a semi-supervised model along with the discriminator in the SGAN (Chapter 20).

Chapter 17

How to Develop a Conditional GAN (cGAN)

Generative Adversarial Networks, or GANs, are an architecture for training generative models, such as deep convolutional neural networks for generating images. Although GAN models are capable of generating new random plausible examples for a given dataset, there is no way to control the types of images that are generated other than trying to figure out the complex relationship between the latent space input to the generator and the generated images. The conditional generative adversarial network, or cGAN for short, is a type of GAN that involves the conditional generation of images by a generator model. Image generation can be conditional on a class label, if available, allowing the targeted generated of images of a given type. In this tutorial, you will discover how to develop a conditional generative adversarial network for the targeted generation of items of clothing. After completing this tutorial, you will know:

- The limitations of generating random samples with a GAN that can be overcome with a conditional generative adversarial network.
- How to develop and evaluate an unconditional generative adversarial network for generating photos of items of clothing.
- How to develop and evaluate a conditional generative adversarial network for generating photos of items of clothing.

Let's get started.

17.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Conditional Generative Adversarial Networks
2. Fashion-MNIST Clothing Photograph Dataset
3. Unconditional GAN for Fashion-MNIST
4. Conditional GAN for Fashion-MNIST
5. Conditional Clothing Generation

17.2 Conditional Generative Adversarial Networks

A generative adversarial network, or GAN for short, is an architecture for training deep learning-based generative models. The architecture is comprised of a generator and a discriminator model. The generator model is responsible for generating new plausible examples that ideally are indistinguishable from real examples in the dataset. The discriminator model is responsible for classifying a given image as either real (drawn from the dataset) or fake (generated). The models are trained together in a zero-sum or adversarial manner, such that improvements in the discriminator come at the cost of a reduced capability of the generator, and vice versa.

GANs are effective at image synthesis, that is, generating new examples of images for a target dataset. Some datasets have additional information, such as a class label, and it is desirable to make use of this information. For example, the MNIST handwritten digit dataset has class labels of the corresponding integers, the CIFAR-10 small object photograph dataset has class labels for the corresponding objects in the photographs, and the Fashion-MNIST clothing dataset has class labels for the corresponding items of clothing. There are two motivations for making use of the class label information in a GAN model.

1. Improve the GAN.
2. Targeted Image Generation.

Additional information that is correlated with the input images, such as class labels, can be used to improve the GAN. This improvement may come in the form of more stable training, faster training, and/or generated images that have better quality. Class labels can also be used for the deliberate or targeted generation of images of a given type. A limitation of a GAN model is that it may generate a random image from the domain. There is a relationship between points in the latent space to the generated images, but this relationship is complex and hard to map. Alternately, a GAN can be trained in such a way that both the generator and the discriminator models are conditioned on the class label. This means that when the trained generator model is used as a standalone model to generate images in the domain, images of a given type, or class label, can be generated.

> Generative adversarial nets can be extended to a conditional model if both the generator and discriminator are conditioned on some extra information y. [...] We can perform the conditioning by feeding y into the both the discriminator and generator as additional input layer.
>
> — *Conditional Generative Adversarial Nets*, 2014.

For example, in the case of MNIST, specific handwritten digits can be generated, such as the number 9; in the case of CIFAR-10, specific object photographs can be generated such as *frogs*; and in the case of the Fashion-MNIST dataset, specific items of clothing can be generated, such as *dress*. This type of model is called a Conditional Generative Adversarial Network, CGAN or cGAN for short. The cGAN was first described by Mehdi Mirza and Simon Osindero in their 2014 paper titled *Conditional Generative Adversarial Nets*. In the paper, the authors motivate the approach based on the desire to direct the image generation process of the generator model.

> ... by conditioning the model on additional information it is possible to direct the data generation process. Such conditioning could be based on class labels

— *Conditional Generative Adversarial Nets*, 2014.

Their approach is demonstrated in the MNIST handwritten digit dataset where the class labels are one hot encoded and concatenated with the input to both the generator and discriminator models. The image below provides a summary of the model architecture.

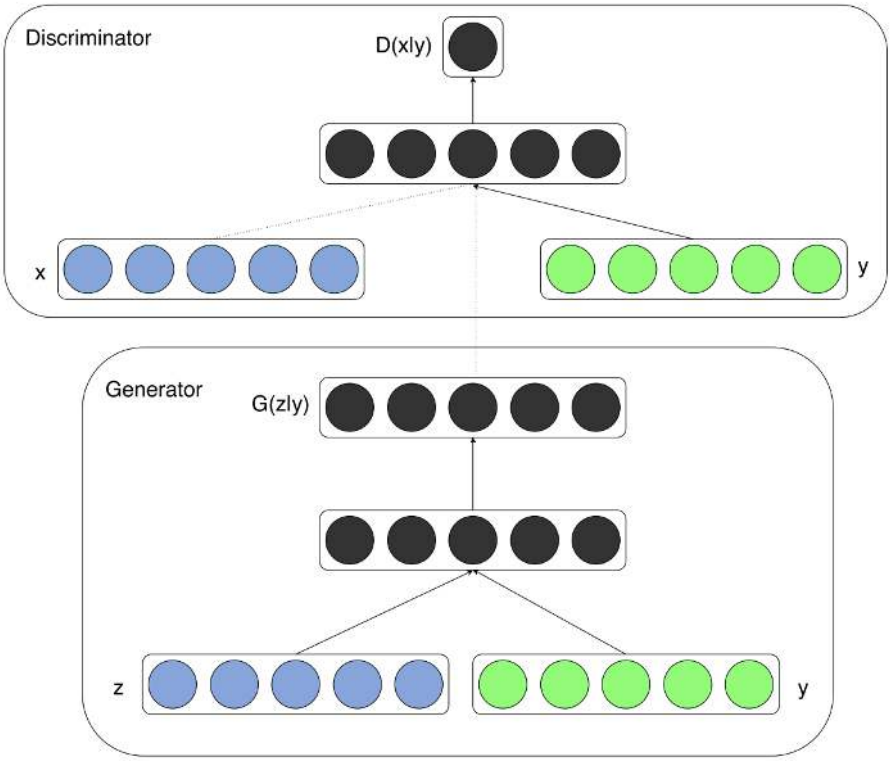

Figure 17.1: Example of a Conditional Generator and a Conditional Discriminator in a Conditional Generative Adversarial Network. Taken from Conditional Generative Adversarial Nets, 2014.

There have been many advancements in the design and training of GAN models, most notably the deep convolutional GAN, or DCGAN for short, that outlines the model configuration and training procedures that reliably result in the stable training of GAN models for a wide variety of problems. The conditional training of the DCGAN-based models may be referred to as CDCGAN or cDCGAN for short. There are many ways to encode and incorporate the class labels into the discriminator and generator models. A best practice involves using an embedding layer followed by a fully connected layer with a linear activation that scales the embedding to the size of the image before concatenating it in the model as an additional channel or feature map. A version of this recommendation was described in the 2015 paper titled *Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks*.

> ... we also explore a class conditional version of the model, where a vector c encodes the label. This is integrated into *Gk* & *Dk* by passing it through a linear layer whose output is reshaped into a single plane feature map which is then concatenated with the 1st layer maps.

— *Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks*, 2015.

This recommendation was later added to the *GAN Hacks* list of heuristic recommendations when designing and training GAN models, summarized as:

> 16: Discrete variables in Conditional GANs
>
> - Use an embedding layer
> - Add as additional channels to images
> - Keep embedding dimensionality low and upsample to match image channel size
>
> — *GAN Hacks*, 2016.

Although GANs can be conditioned on the class label, so-called class-conditional GANs, they can also be conditioned on other inputs, such as an image, in the case where a GAN is used for image-to-image translation tasks. In this tutorial, we will develop a GAN, specifically a DCGAN, then update it to use class labels in a cGAN, specifically a cDCGAN model architecture.

17.3 Fashion-MNIST Clothing Photograph Dataset

The Fashion-MNIST dataset is proposed as a more challenging replacement dataset for the MNIST dataset. It is a dataset comprised of 60,000 small square 28×28 pixel grayscale images of items of 10 types of clothing, such as shoes, t-shirts, dresses, and more. Keras provides access to the Fashion-MNIST dataset via the `fashion_mnist.load_dataset()` function. It returns two tuples, one with the input and output elements for the standard training dataset, and another with the input and output elements for the standard test dataset. The example below loads the dataset and summarizes the shape of the loaded dataset.

Note: the first time you load the dataset, Keras will automatically download a compressed version of the images and save them under your home directory in `~/.keras/datasets/`. The download is fast as the dataset is only about 25 megabytes in its compressed form.

```
# example of loading the fashion_mnist dataset
from keras.datasets.fashion_mnist import load_data
# load the images into memory
(trainX, trainy), (testX, testy) = load_data()
# summarize the shape of the dataset
print('Train', trainX.shape, trainy.shape)
print('Test', testX.shape, testy.shape)
```

Listing 17.1: Example of loading and summarizing the Fashion-MNIST dataset.

Running the example loads the dataset and prints the shape of the input and output components of the train and test splits of images. We can see that there are 60K examples in the training set and 10K in the test set and that each image is a square of 28 by 28 pixels.

```
Train (60000, 28, 28) (60000,)
Test (10000, 28, 28) (10000,)
```

Listing 17.2: Example output from loading and summarizing the Fashion-MNIST dataset.

The images are grayscale with a black background (0 pixel value) and the items of clothing are in white (pixel values near 255). This means if the images were plotted, they would be mostly black with a white item of clothing in the middle. We can plot some of the images from the training dataset using the Matplotlib library with the `imshow()` function and specify the color map via the `cmap` argument as `'gray'` to show the pixel values correctly.

```
...
# plot raw pixel data
pyplot.imshow(trainX[i], cmap='gray')
```

Listing 17.3: Example of plotting an image with a grayscale color map.

Alternately, the images are easier to review when we reverse the colors and plot the background as white and the clothing in black. They are easier to view as most of the image is now white with the area of interest in black. This can be achieved using a reverse grayscale color map, as follows:

```
...
# plot raw pixel data
pyplot.imshow(trainX[i], cmap='gray_r')
```

Listing 17.4: Example of plotting an image with a reverse grayscale color map.

The example below plots the first 100 images from the training dataset in a 10 by 10 square.

```
# example of loading the fashion_mnist dataset
from keras.datasets.fashion_mnist import load_data
from matplotlib import pyplot
# load the images into memory
(trainX, trainy), (testX, testy) = load_data()
# plot images from the training dataset
for i in range(100):
  # define subplot
  pyplot.subplot(10, 10, 1 + i)
  # turn off axis
  pyplot.axis('off')
  # plot raw pixel data
  pyplot.imshow(trainX[i], cmap='gray_r')
pyplot.show()
```

Listing 17.5: Example of loading and plotting the Fashion-MNIST dataset.

Running the example creates a figure with a plot of 100 images from the MNIST training dataset, arranged in a 10×10 square.

Figure 17.2: Plot of the First 100 Items of Clothing From the Fashion-MNIST Dataset.

We will use the images in the training dataset as the basis for training a Generative Adversarial Network. Specifically, the generator model will learn how to generate new plausible items of clothing using a discriminator that will try to distinguish between real images from the Fashion-MNIST training dataset and new images output by the generator model. This is a relatively simple problem that does not require a sophisticated generator or discriminator model, although it does require the generation of a grayscale output image.

17.4 Unconditional GAN for Fashion-MNIST

In this section, we will develop an unconditional GAN for the Fashion-MNIST dataset. The first step is to define the models. The discriminator model takes as input one 28×28 grayscale image and outputs a binary prediction as to whether the image is real ($class = 1$) or fake ($class = 0$). It is implemented as a modest convolutional neural network using best practices for GAN design such as using the `LeakyReLU` activation function with a slope of 0.2, using a 2×2 stride to downsample, and the adam version of stochastic gradient descent with a learning rate of 0.0002 and a momentum of 0.5 The `define_discriminator()` function below implements this, defining and compiling the discriminator model and returning it. The input shape of the image is parameterized as a default function argument in case you want to re-use the function for your own image data later.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

Listing 17.6: Example of a function for defining the discriminator model.

The generator model takes as input a point in the latent space and outputs a single 28×28 grayscale image. This is achieved by using a fully connected layer to interpret the point in the latent space and provide sufficient activations that can be reshaped into many copies (in this case 128) of a low-resolution version of the output image (e.g. 7×7). This is then upsampled twice, doubling the size and quadrupling the area of the activations each time using transpose convolutional layers. The model uses best practices such as the `LeakyReLU` activation, a kernel size that is a factor of the stride size, and a hyperbolic tangent (Tanh) activation function in the output layer.

The `define_generator()` function below defines the generator model, but intentionally does not compile it as it is not trained directly, then returns the model. The size of the latent space is parameterized as a function argument.

```
# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # generate
  model.add(Conv2D(1, (7,7), activation='tanh', padding='same'))
  return model
```

Listing 17.7: Example of a function for defining the generator model.

Next, a GAN model can be defined that combines both the generator model and the discriminator model into one larger model. This larger model will be used to train the model

weights in the generator, using the output and error calculated by the discriminator model. The discriminator model is trained separately, and as such, the model weights are marked as not trainable in this larger GAN model to ensure that only the weights of the generator model are updated. This change to the trainability of the discriminator weights only has an effect when training the combined GAN model, not when training the discriminator standalone.

This larger GAN model takes as input a point in the latent space, uses the generator model to generate an image which is fed as input to the discriminator model, then is output or classified as real or fake. The `define_gan()` function below implements this, taking the already-defined generator and discriminator models as input.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model
```

Listing 17.8: Example of a function for defining the composite model for training the generator.

Now that we have defined the GAN model, we need to train it. But, before we can train the model, we require input data. The first step is to load and prepare the Fashion-MNIST dataset. We only require the images in the training dataset. The images are black and white, therefore we must add an additional channel dimension to transform them to be three dimensional, as expected by the convolutional layers of our models. Finally, the pixel values must be scaled to the range [-1,1] to match the output of the generator model. The `load_real_samples()` function below implements this, returning the loaded and scaled Fashion-MNIST training dataset ready for modeling.

```
# load fashion mnist images
def load_real_samples():
  # load dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X
```

Listing 17.9: Example of a function for loading and preparing the Fashion-MNIST dataset.

We will require one batch (or a half batch) of real images from the dataset each update to the GAN model. A simple way to achieve this is to select a random sample of images from the dataset each time. The `generate_real_samples()` function below implements this, taking the prepared dataset as an argument, selecting and returning a random sample of Fashion-MNIST

images and their corresponding class label for the discriminator, specifically $class = 1$, indicating that they are real images.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y
```

Listing 17.10: Example of a function for selecting random samples of real images.

Next, we need inputs for the generator model. These are random points from the latent space, specifically Gaussian distributed random variables. The `generate_latent_points()` function implements this, taking the size of the latent space as an argument and the number of points required and returning them as a batch of input samples for the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

Listing 17.11: Example of a function for generating random points in the latent space.

Next, we need to use the points in the latent space as input to the generator in order to generate new images. The `generate_fake_samples()` function below implements this, taking the generator model and size of the latent space as arguments, then generating points in the latent space and using them as input to the generator model. The function returns the generated images and their corresponding class label for the discriminator model, specifically $class = 0$ to indicate they are fake or generated.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n_samples, 1))
  return X, y
```

Listing 17.12: Example of a function for generating synthetic images with the generator model.

We are now ready to fit the GAN models. The model is fit for 100 training epochs, which is arbitrary, as the model begins generating plausible items of clothing after perhaps 20 epochs. A batch size of 128 samples is used, and each training epoch involves $\frac{60000}{128}$, or about 468 batches of real and fake samples and updates to the model. First, the discriminator model is updated for a half batch of real samples, then a half batch of fake samples, together forming one batch of weight updates. The generator is then updated via the composite GAN model. Importantly, the

class label is set to 1 or real for the fake samples. This has the effect of updating the generator toward getting better at generating real samples on the next batch. The `train()` function below implements this, taking the defined models, dataset, and size of the latent dimension as arguments and parameterizing the number of epochs and batch size with default arguments. The generator model is saved at the end of training.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=128):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # update discriminator model weights
      d_loss1, _ = d_model.train_on_batch(X_real, y_real)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update discriminator model weights
      d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
        (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
  # save the generator model
  g_model.save('generator.h5')
```

Listing 17.13: Example of a function for training the GAN models.

We can then define the size of the latent space, define all three models, and train them on the loaded Fashion-MNIST dataset.

```
...
# size of the latent space
latent_dim = 100
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Listing 17.14: Example of configuring and starting the training process.

Tying all of this together, the complete example is listed below.

```
# example of training an unconditional gan on the fashion mnist dataset
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.fashion_mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  # downsample
  model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # classifier
  model.add(Flatten())
  model.add(Dropout(0.4))
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # generate
  model.add(Conv2D(1, (7,7), activation='tanh', padding='same'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(generator, discriminator):
```

```
  # make weights in the discriminator not trainable
  discriminator.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(generator)
  # add the discriminator
  model.add(discriminator)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# load fashion mnist images
def load_real_samples():
  # load dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = generator.predict(x_input)
  # create class labels
  y = zeros((n_samples, 1))
  return X, y

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=128):
  bat_per_epo = int(dataset.shape[0] / n_batch)
```

```
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # update discriminator model weights
      d_loss1, _ = d_model.train_on_batch(X_real, y_real)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update discriminator model weights
      d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
        (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
  # save the generator model
  g_model.save('generator.h5')

# size of the latent space
latent_dim = 100
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Listing 17.15: Example of training an unconditional GAN on the Fashion-MNIST dataset.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

The loss for the discriminator on real and fake samples, as well as the loss for the generator, is reported after each batch.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the discriminator and generator loss both sit around values of about 0.6 to 0.7 over the course of training.

```
...
>100, 464/468, d1=0.681, d2=0.685 g=0.693
>100, 465/468, d1=0.691, d2=0.700 g=0.703
>100, 466/468, d1=0.691, d2=0.703 g=0.706
>100, 467/468, d1=0.698, d2=0.699 g=0.699
>100, 468/468, d1=0.699, d2=0.695 g=0.708
```

Listing 17.16: Example output from training an unconditional GAN on the Fashion-MNIST dataset.

At the end of training, the generator model will be saved to file with the filename `generator.h5`. This model can be loaded and used to generate new random but plausible samples from the Fashion-MNIST dataset. The example below loads the saved model and generates 100 random items of clothing.

```
# example of loading the generator model and generating images
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# create and save a plot of generated images (reversed grayscale)
def show_plot(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  pyplot.show()

# load model
model = load_model('generator.h5')
# generate images
latent_points = generate_latent_points(100, 100)
# generate images
X = model.predict(latent_points)
# plot the result
show_plot(X, 10)
```

Listing 17.17: Example of loading the saved unconditional generator model and using it to generate images.

Running the example creates a plot of 100 randomly generated items of clothing arranged into a 10×10 grid.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see an assortment of clothing items such as shoes, sweaters, and pants. Most items look quite plausible and could have come from the Fashion-MNIST dataset. They are not perfect, however, as there are some sweaters with a single sleeve and shoes that look like a mess.

Figure 17.3: Example of 100 Generated items of Clothing using an Unconditional GAN.

17.5 Conditional GAN for Fashion-MNIST

In this section, we will develop a conditional GAN for the Fashion-MNIST dataset by updating the unconditional GAN developed in the previous section. The best way to design models in Keras to have multiple inputs is by using the Functional API, as opposed to the Sequential API used in the previous section. We will use the functional API to re-implement the discriminator, generator, and the composite model. Starting with the discriminator model, a new second input is defined that takes an integer for the class label of the image. This has the effect of making the input image conditional on the provided class label.

The class label is then passed through an `Embedding` layer with the size of 50. This means that each of the 10 classes for the Fashion-MNIST dataset (0 through 9) will map to a different

50-element vector representation that will be learned by the discriminator model. The output of the embedding is then passed to a fully connected layer with a linear activation. Importantly, the fully connected layer has enough activations that can be reshaped into one channel of a 28×28 image. The activations are reshaped into single 28×28 activation map and concatenated with the input image. This has the effect of looking like a two-channel input image to the next convolutional layer. The `define_discriminator()` below implements this update to the discriminator model. The parameterized shape of the input image is also used after the embedding layer to define the number of activations for the fully connected layer to reshape its output. The number of classes in the problem is also parameterized in the function and set.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1), n_classes=10):
  # label input
  in_label = Input(shape=(1,))
  # embedding for categorical input
  li = Embedding(n_classes, 50)(in_label)
  # scale up to image dimensions with linear activation
  n_nodes = in_shape[0] * in_shape[1]
  li = Dense(n_nodes)(li)
  # reshape to additional channel
  li = Reshape((in_shape[0], in_shape[1], 1))(li)
  # image input
  in_image = Input(shape=in_shape)
  # concat label as a channel
  merge = Concatenate()([in_image, li])
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(merge)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # dropout
  fe = Dropout(0.4)(fe)
  # output
  out_layer = Dense(1, activation='sigmoid')(fe)
  # define model
  model = Model([in_image, in_label], out_layer)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

Listing 17.18: Example of a function for defining the conditional discriminator model.

In order to make the architecture clear, below is a plot of the discriminator model. The plot shows the two inputs: first the class label that passes through the embedding (left) and the image (right), and their concatenation into a two-channel 28×28 image or feature map (middle). The rest of the model is the same as the discriminator designed in the previous section.

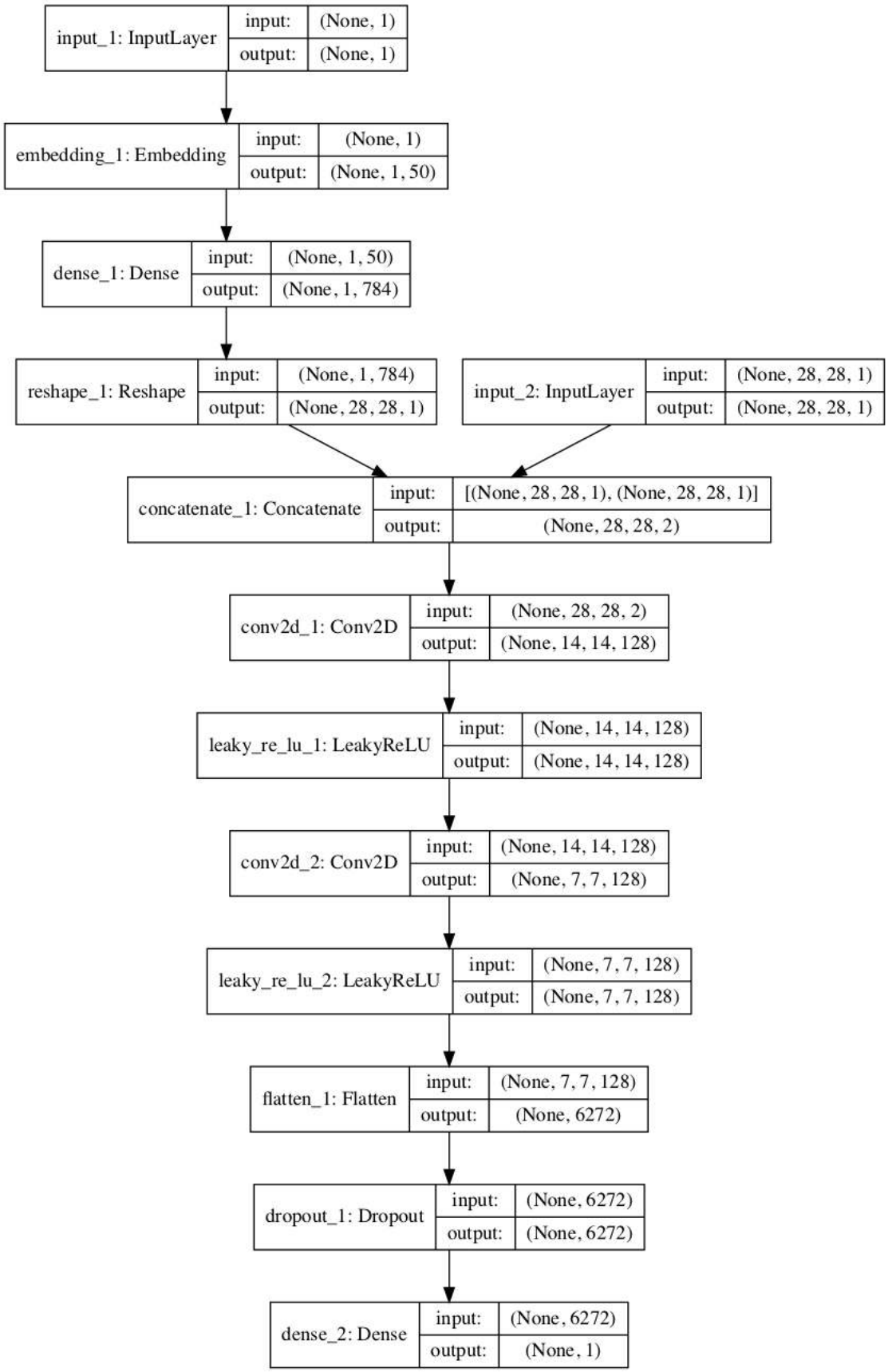

Figure 17.4: Plot of the Discriminator Model in the Conditional Generative Adversarial Network.

Next, the generator model must be updated to take the class label. This has the effect of making the point in the latent space conditional on the provided class label. As in the discriminator, the class label is passed through an embedding layer to map it to a unique 50-element vector and is then passed through a fully connected layer with a linear activation before being resized. In this case, the activations of the fully connected layer are resized into a single 7×7 feature map. This is to match the 7×7 feature map activations of the unconditional generator model. The new 7×7 feature map is added as one more channel to the existing 128, resulting in 129 feature maps that are then upsampled as in the prior model. The `define_generator()` function below implements this, again parameterizing the number of classes as we did with the discriminator model.

```
# define the standalone generator model
def define_generator(latent_dim, n_classes=10):
  # label input
  in_label = Input(shape=(1,))
  # embedding for categorical input
  li = Embedding(n_classes, 50)(in_label)
  # linear multiplication
  n_nodes = 7 * 7
  li = Dense(n_nodes)(li)
  # reshape to additional channel
  li = Reshape((7, 7, 1))(li)
  # image generator input
  in_lat = Input(shape=(latent_dim,))
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  gen = Dense(n_nodes)(in_lat)
  gen = LeakyReLU(alpha=0.2)(gen)
  gen = Reshape((7, 7, 128))(gen)
  # merge image gen and label input
  merge = Concatenate()([gen, li])
  # upsample to 14x14
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(merge)
  gen = LeakyReLU(alpha=0.2)(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
  gen = LeakyReLU(alpha=0.2)(gen)
  # output
  out_layer = Conv2D(1, (7,7), activation='tanh', padding='same')(gen)
  # define model
  model = Model([in_lat, in_label], out_layer)
  return model
```

Listing 17.19: Example of a function for defining the conditional generator model.

To help understand the new model architecture, the image below provides a plot of the new conditional generator model. In this case, you can see the 100-element point in latent space as input and subsequent resizing (left) and the new class label input and embedding layer (right), then the concatenation of the two sets of feature maps (center). The remainder of the model is the same as the unconditional case.

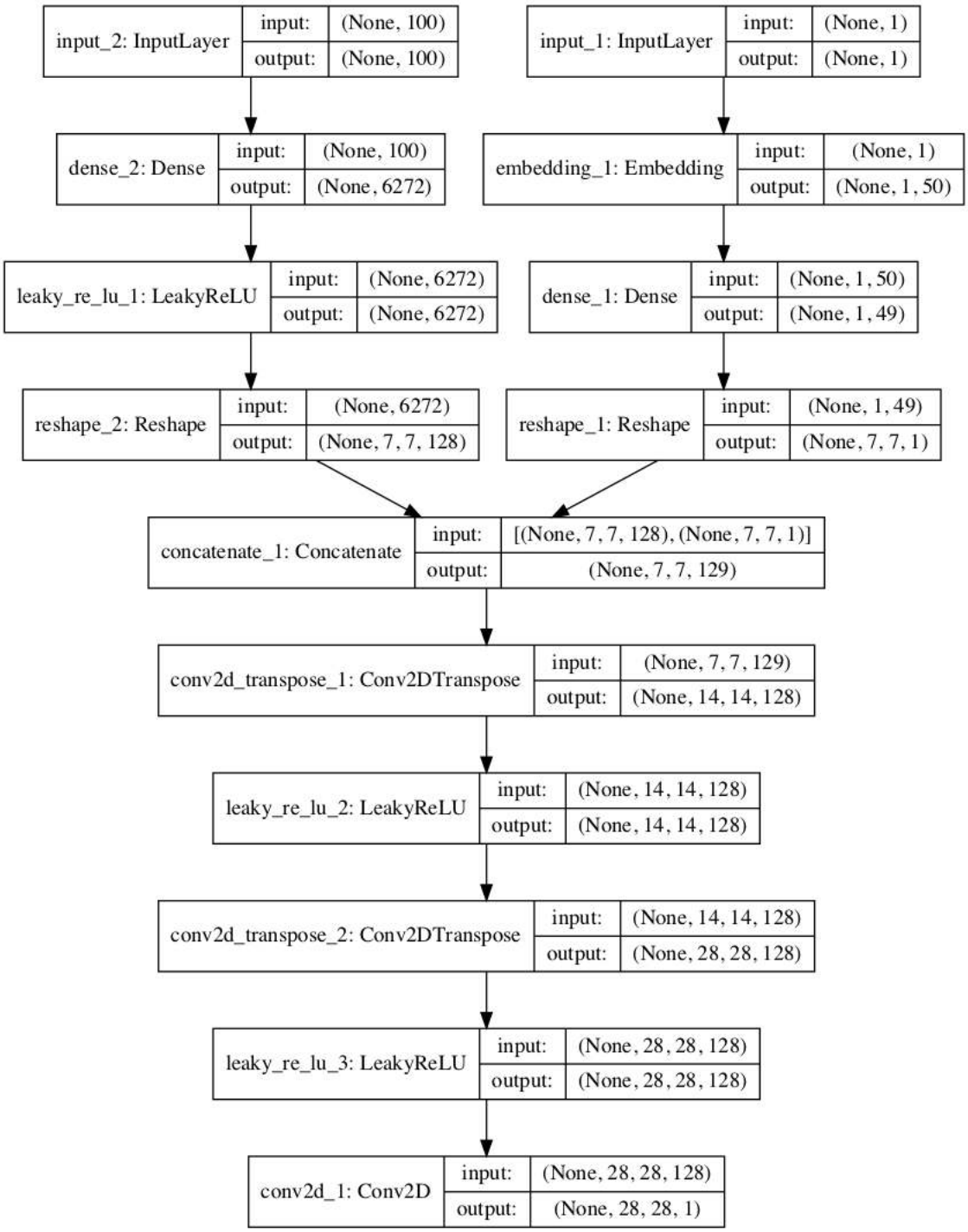

Figure 17.5: Plot of the Generator Model in the Conditional Generative Adversarial Network.

Finally, the composite GAN model requires updating. The new GAN model will take a point in latent space as input and a class label and generate a prediction of whether input was real or fake, as before. Using the functional API to design the model, it is important that we explicitly connect the image generated output from the generator as well as the class label input, both as input to the discriminator model. This allows the same class label input to flow down into the generator and down into the discriminator. The `define_gan()` function below implements the

conditional version of the GAN.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # get noise and label inputs from generator model
  gen_noise, gen_label = g_model.input
  # get image output from the generator model
  gen_output = g_model.output
  # connect image output and label input from generator as inputs to discriminator
  gan_output = d_model([gen_output, gen_label])
  # define gan model as taking noise and label and outputting a classification
  model = Model([gen_noise, gen_label], gan_output)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model
```

Listing 17.20: Example of a function for defining the conditional composite model for training the generator.

The plot below summarizes the composite GAN model. Importantly, it shows the generator model in full with the point in latent space and class label as input, and the connection of the output of the generator and the same class label as input to the discriminator model (last box at the bottom of the plot) and the output of a single class label classification of real or fake.

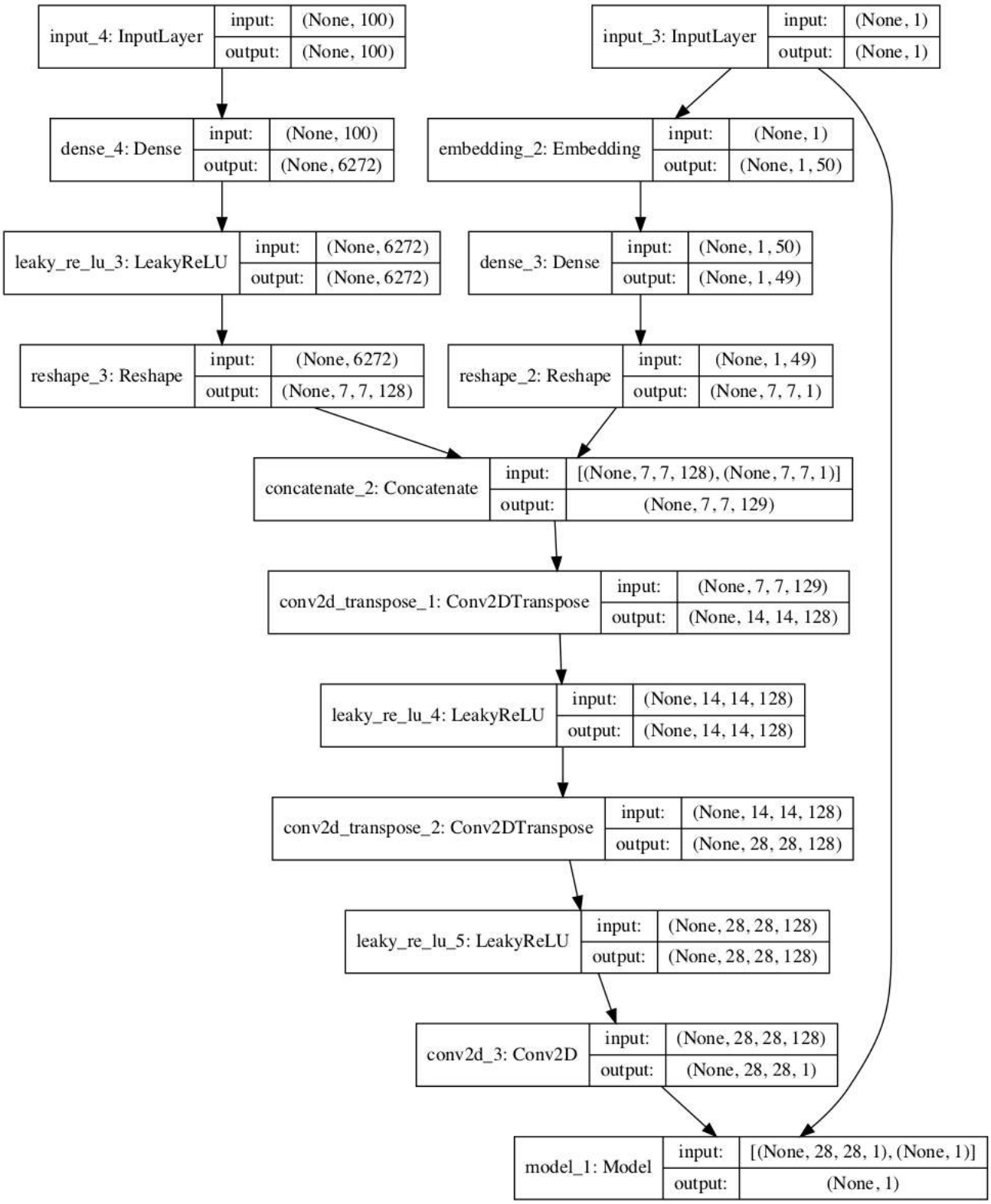

Figure 17.6: Plot of the Composite Generator and Discriminator Model in the Conditional Generative Adversarial Network.

The hard part of the conversion from unconditional to conditional GAN is done, namely

the definition and configuration of the model architecture. Next, all that remains is to update the training process to also use class labels. First, the `load_real_samples()` and `generate_real_samples()` functions for loading the dataset and selecting a batch of samples respectively must be updated to make use of the real class labels from the training dataset. Importantly, the `generate_real_samples()` function now returns images, clothing labels, and the class label for the discriminator ($class = 1$).

```
# load fashion mnist images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return [X, trainy]

# select real samples
def generate_real_samples(dataset, n_samples):
  # split into images and labels
  images, labels = dataset
  # choose random instances
  ix = randint(0, images.shape[0], n_samples)
  # select images and labels
  X, labels = images[ix], labels[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return [X, labels], y
```

Listing 17.21: Example of functions for preparing and selecting random samples of real images with target classes.

Next, the `generate_latent_points()` function must be updated to also generate an array of randomly selected integer class labels to go along with the randomly selected points in the latent space. Then the `generate_fake_samples()` function must be updated to use these randomly generated class labels as input to the generator model when generating new fake images.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples, n_classes=10):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  # generate labels
  labels = randint(0, n_classes, n_samples)
  return [z_input, labels]

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  z_input, labels_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  images = generator.predict([z_input, labels_input])
```

```
  # create class labels
  y = zeros((n_samples, 1))
  return [images, labels_input], y
```

Listing 17.22: Example of functions for generating points in latent space and synthetic images using class labels.

Finally, the `train()` function must be updated to retrieve and use the class labels in the calls to updating the discriminator and generator models.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=128):
  bat_per_epo = int(dataset[0].shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      [X_real, labels_real], y_real = generate_real_samples(dataset, half_batch)
      # update discriminator model weights
      d_loss1, _ = d_model.train_on_batch([X_real, labels_real], y_real)
      # generate 'fake' examples
      [X_fake, labels], y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update discriminator model weights
      d_loss2, _ = d_model.train_on_batch([X_fake, labels], y_fake)
      # prepare points in latent space as input for the generator
      [z_input, labels_input] = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch([z_input, labels_input], y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
        (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
  # save the generator model
  g_model.save('cgan_generator.h5')
```

Listing 17.23: Example of the updated function for training the conditional GAN models.

Tying all of this together, the complete example of a conditional deep convolutional generative adversarial network for the Fashion-MNIST dataset is listed below.

```
# example of training an conditional gan on the fashion mnist dataset
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.fashion_mnist import load_data
from keras.optimizers import Adam
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
```

```
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import Embedding
from keras.layers import Concatenate

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1), n_classes=10):
  # label input
  in_label = Input(shape=(1,))
  # embedding for categorical input
  li = Embedding(n_classes, 50)(in_label)
  # scale up to image dimensions with linear activation
  n_nodes = in_shape[0] * in_shape[1]
  li = Dense(n_nodes)(li)
  # reshape to additional channel
  li = Reshape((in_shape[0], in_shape[1], 1))(li)
  # image input
  in_image = Input(shape=in_shape)
  # concat label as a channel
  merge = Concatenate()([in_image, li])
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(merge)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # dropout
  fe = Dropout(0.4)(fe)
  # output
  out_layer = Dense(1, activation='sigmoid')(fe)
  # define model
  model = Model([in_image, in_label], out_layer)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim, n_classes=10):
  # label input
  in_label = Input(shape=(1,))
  # embedding for categorical input
  li = Embedding(n_classes, 50)(in_label)
  # linear multiplication
  n_nodes = 7 * 7
  li = Dense(n_nodes)(li)
  # reshape to additional channel
  li = Reshape((7, 7, 1))(li)
  # image generator input
  in_lat = Input(shape=(latent_dim,))
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  gen = Dense(n_nodes)(in_lat)
```

```
  gen = LeakyReLU(alpha=0.2)(gen)
  gen = Reshape((7, 7, 128))(gen)
  # merge image gen and label input
  merge = Concatenate()([gen, li])
  # upsample to 14x14
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(merge)
  gen = LeakyReLU(alpha=0.2)(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
  gen = LeakyReLU(alpha=0.2)(gen)
  # output
  out_layer = Conv2D(1, (7,7), activation='tanh', padding='same')(gen)
  # define model
  model = Model([in_lat, in_label], out_layer)
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # get noise and label inputs from generator model
  gen_noise, gen_label = g_model.input
  # get image output from the generator model
  gen_output = g_model.output
  # connect image output and label input from generator as inputs to discriminator
  gan_output = d_model([gen_output, gen_label])
  # define gan model as taking noise and label and outputting a classification
  model = Model([gen_noise, gen_label], gan_output)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# load fashion mnist images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  return [X, trainy]

# # select real samples
def generate_real_samples(dataset, n_samples):
  # split into images and labels
  images, labels = dataset
  # choose random instances
  ix = randint(0, images.shape[0], n_samples)
  # select images and labels
  X, labels = images[ix], labels[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return [X, labels], y
```

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples, n_classes=10):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  # generate labels
  labels = randint(0, n_classes, n_samples)
  return [z_input, labels]

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  z_input, labels_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  images = generator.predict([z_input, labels_input])
  # create class labels
  y = zeros((n_samples, 1))
  return [images, labels_input], y

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=128):
  bat_per_epo = int(dataset[0].shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      [X_real, labels_real], y_real = generate_real_samples(dataset, half_batch)
      # update discriminator model weights
      d_loss1, _ = d_model.train_on_batch([X_real, labels_real], y_real)
      # generate 'fake' examples
      [X_fake, labels], y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # update discriminator model weights
      d_loss2, _ = d_model.train_on_batch([X_fake, labels], y_fake)
      # prepare points in latent space as input for the generator
      [z_input, labels_input] = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch([z_input, labels_input], y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
        (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
  # save the generator model
  g_model.save('cgan_generator.h5')

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
```

```
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim)
```

Listing 17.24: Example of training the conditional GAN on the Fashion-MNIST dataset.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

At the end of the run, the model is saved to the file with name `cgan_generator.h5`.

17.6 Conditional Clothing Generation

In this section, we will use the trained generator model to conditionally generate new photos of items of clothing. We can update our code example for generating new images with the model to now generate images conditional on the class label. We can generate 10 examples for each class label in columns. The complete example is listed below.

```
# example of loading the generator model and generating images
from numpy import asarray
from numpy.random import randn
from numpy.random import randint
from keras.models import load_model
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples, n_classes=10):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  # generate labels
  labels = randint(0, n_classes, n_samples)
  return [z_input, labels]

# create and save a plot of generated images
def save_plot(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  pyplot.show()

# load model
model = load_model('cgan_generator.h5')
# generate images
```

```
latent_points, labels = generate_latent_points(100, 100)
# specify labels
labels = asarray([x for _ in range(10) for x in range(10)])
# generate images
X  = model.predict([latent_points, labels])
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
# plot the result
save_plot(X, 10)
```

Listing 17.25: Example of loading the saved conditional generator model and using it to generate images.

Running the example loads the saved conditional GAN model and uses it to generate 100 items of clothing. The clothing is organized in columns. From left to right, they are *t-shirt*, *trouser*, *pullover*, *dress*, *coat*, *sandal*, *shirt*, *sneaker*, *bag*, and *ankle boot*. We can see not only are the randomly generated items of clothing plausible, but they also match their expected category.

Figure 17.7: Example of 100 Generated items of Clothing using a Conditional GAN.

17.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Latent Space Size**. Experiment by varying the size of the latent space and review the impact on the quality of generated images.
- **Embedding Size**. Experiment by varying the size of the class label embedding, making it smaller or larger, and review the impact on the quality of generated images.
- **Alternate Architecture**. Update the model architecture to concatenate the class label elsewhere in the generator and/or discriminator model, perhaps with different dimensionality, and review the impact on the quality of generated images.

If you explore any of these extensions, I'd love to know.

17.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

17.8.1 Papers

- Generative Adversarial Networks, 2014.
https://arxiv.org/abs/1406.2661
- Tutorial: Generative Adversarial Networks, NIPS, 2016.
https://arxiv.org/abs/1701.00160
- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.
https://arxiv.org/abs/1511.06434
- Conditional Generative Adversarial Nets, 2014.
https://arxiv.org/abs/1411.1784
- Image-To-Image Translation With Conditional Adversarial Networks, 2017.
https://arxiv.org/abs/1611.07004
- Conditional Generative Adversarial Nets For Convolutional Face Generation, 2015.
https://www.foldl.me/uploads/2015/conditional-gans-face-generation/paper.pdf

17.8.2 API

- Keras Datasets API..
https://keras.io/datasets/
- Keras Sequential Model API.
https://keras.io/models/sequential/
- Keras Convolutional Layers API.
https://keras.io/layers/convolutional/
- How can I "freeze" Keras layers?.
https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers

- MatplotLib API.
  https://matplotlib.org/api/
- NumPy Random sampling (numpy.random) API.
  https://docs.scipy.org/doc/numpy/reference/routines.random.html
- NumPy Array manipulation routines.
  https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html

17.8.3 Articles

- How to Train a GAN? Tips and tricks to make GANs work.
  https://github.com/soumith/ganhacks
- Fashion-MNIST Project, GitHub.
  https://github.com/zalandoresearch/fashion-mnist

17.9 Summary

In this tutorial, you discovered how to develop a conditional generative adversarial network for the targeted generation of items of clothing. Specifically, you learned:

- The limitations of generating random samples with a GAN that can be overcome with a conditional generative adversarial network.
- How to develop and evaluate an unconditional generative adversarial network for generating photos of items of clothing.
- How to develop and evaluate a conditional generative adversarial network for generating photos of items of clothing.

17.9.1 Next

In the next tutorial, you will discover the information maximizing GAN model that adds controls over image generation.

Chapter 18

How to Develop an Information Maximizing GAN (InfoGAN)

The Generative Adversarial Network, or GAN, is an architecture for training deep convolutional models for generating synthetic images. Although remarkably effective, the default GAN provides no control over the types of images that are generated. The Information Maximizing GAN, or InfoGAN for short, is an extension to the GAN architecture that introduces control variables that are automatically learned by the architecture and allow control over the generated image, such as style, thickness, and type in the case of generating images of handwritten digits. In this tutorial, you will discover how to implement an Information Maximizing Generative Adversarial Network model from scratch. After completing this tutorial, you will know:

- The InfoGAN is motivated by the desire to disentangle and control the properties in generated images.
- The InfoGAN involves the addition of control variables to generate an auxiliary model that predicts the control variables, trained via mutual information loss function.
- How to develop and train an InfoGAN model from scratch and use the control variables to control which digit is generated by the model.

Let's get started.

18.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. What Is the Information Maximizing GAN
2. How to Implement the InfoGAN Loss Function
3. How to Develop an InfoGAN for MNIST
4. How to Use Control Codes With an InfoGAN

18.2 What Is the Information Maximizing GAN

The Generative Adversarial Network, or GAN for short, is an architecture for training a generative model, such as a model for generating synthetic images. It involves the simultaneous training of the generator model for generating images with a discriminator model that learns to classify images as either real (from the training dataset) or fake (generated). The two models compete in a zero-sum game such that convergence of the training process involves finding a balance between the generator's skill in generating convincing images and the discriminator's in being able to detect them. The generator model takes as input a random point from a latent space, typically 50 to 100 random Gaussian variables. The generator applies a unique meaning to the points in the latent space via training and maps points to specific output synthetic images. This means that although the latent space is structured by the generator model, there is no control over the generated image.

> The GAN formulation uses a simple factored continuous input noise vector z, while imposing no restrictions on the manner in which the generator may use this noise. As a result, it is possible that the noise will be used by the generator in a highly entangled way, causing the individual dimensions of z to not correspond to semantic features of the data.

— *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*, 2016.

The latent space can be explored and generated images compared in an attempt to understand the mapping that the generator model has learned. Alternatively, the generation process can be conditioned, such as via a class label, so that images of a specific type can be created on demand. This is the basis for the Conditional Generative Adversarial Network, CGAN or cGAN for short. Another approach is to provide control variables as input to the generator, along with the point in latent space (noise). The generator can be trained to use the control variables to influence specific properties of the generated images. This is the approach taken with the Information Maximizing Generative Adversarial Network, or InfoGAN for short.

> InfoGAN, an information-theoretic extension to the Generative Adversarial Network that is able to learn disentangled representations in a completely unsupervised manner.

— *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*, 2016.

The structured mapping learned by the generator during the training process is somewhat random. Although the generator model learns to spatially separate properties of generated images in the latent space, there is no control. The properties are entangled. The InfoGAN is motivated by the desire to disentangle the properties of generated images. For example, in the case of faces, the properties of generating a face can be disentangled and controlled, such as the shape of the face, hair color, hairstyle, and so on.

> For example, for a dataset of faces, a useful disentangled representation may allocate a separate set of dimensions for each of the following attributes: facial expression, eye color, hairstyle, presence or absence of eyeglasses, and the identity of the corresponding person.

— *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*, 2016.

Control variables are provided along with the noise as input to the generator and the model is trained via a mutual information loss function.

> ... we present a simple modification to the generative adversarial network objective that encourages it to learn interpretable and meaningful representations. We do so by maximizing the mutual information between a fixed small subset of the GAN's noise variables and the observations, which turns out to be relatively straightforward.

— *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*, 2016.

Mutual information refers to the amount of information learned about one variable given another variable. In this case, we are interested in the information about the control variables given the image generated using noise and the control variables. In information theory, mutual information between X and Y, $I(X;Y)$, measures the *amount of information* learned from knowledge of random variable Y about the other random variable X. The Mutual Information (MI) is calculated as the conditional entropy of the control variables (c) given the image (created by the generator (G) from the noise (z) and the control variable (c)) subtracted from the marginal entropy of the control variables (c); for example:

$$MI = Entropy(c) - Entropy(c; G(z, c)) \tag{18.1}$$

Calculating the true mutual information, in practice, is often intractable, although simplifications are adopted in the paper, referred to as Variational Information Maximization, and the entropy for the control codes is kept constant. Training the generator via mutual information is achieved through the use of a new model, referred to as Q or the auxiliary model. The new model shares all of the same weights as the discriminator model for interpreting an input image, but unlike the discriminator model that predicts whether the image is real or fake, the auxiliary model predicts the control codes that were used to generate the image.

Both models are used to update the generator model, first to improve the likelihood of generating images that fool the discriminator model, and second to improve the mutual information between the control codes used to generate an image and the auxiliary model's prediction of the control codes. The result is that the generator model is regularized via mutual information loss such that the control codes capture salient properties of the generated images and, in turn, can be used to control the image generation process.

> ... mutual information can be utilized whenever we are interested in learning a parametrized mapping from a given input X to a higher level representation Y which preserves information about the original input. [...] show that the task of maximizing mutual information is essentially equivalent to training an autoencoder to minimize reconstruction error.

— *Understanding Mutual Information and its use in InfoGAN*, 2016.

18.3 How to Implement the InfoGAN Loss Function

The InfoGAN is reasonably straightforward to implement once you are familiar with the inputs and outputs of the model. The only stumbling block might be the mutual information loss function, especially if you don't have a strong math background, like most developers. There are two main types of control variables used with the InfoGan: categorical and continuous, and continuous variables may have different data distributions, which impact how the mutual loss is calculated. The mutual loss can be calculated and summed across all control variables based on the variable type, and this is the approach used in the official InfoGAN implementation released by OpenAI for TensorFlow.

In Keras, it might be easier to simplify the control variables to categorical and either Gaussian or Uniform continuous variables and have separate outputs on the auxiliary model for each control variable type. This is so that a different loss function can be used, greatly simplifying the implementation. See the papers and posts in the further reading section for more background on the recommendations in this section.

18.3.1 Categorical Control Variables

The categorical variable may be used to control the type or class of the generated image. This is implemented as a one hot encoded vector. That is, if the class has 10 values, then the control code would be one class, e.g. 6, and the categorical control vector input to the generator model would be a 10 element vector of all zero values with a one value for class 6, for example, `[0, 0, 0, 0, 0, 0, 1, 0, 0]`. We do not need to choose the categorical control variables when training the model; instead, they are generated randomly, e.g. each selected with a uniform probability for each sample.

> ... a uniform categorical distribution on latent codes $c \sim Cat(K = 10, p = 0.1)$

— *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*, 2016.

In the auxiliary model, the output layer for the categorical variable would also be a one hot encoded vector to match the input control code, and the softmax activation function is used.

> For categorical latent code c_i , we use the natural choice of softmax nonlinearity to represent $Q(c_i|x)$.

— *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*, 2016.

Recall that the mutual information is calculated as the conditional entropy from the control variable and the output of the auxiliary model subtracted from the entropy of the control variable provided to the input variable. We can implement this directly, but it's not necessary. The entropy of the control variable is a constant and comes out to be a very small number close to zero; as such, we can remove it from our calculation. The conditional entropy can be calculated directly as the cross-entropy between the control variable input and the output from the auxiliary model. Therefore, the categorical cross-entropy loss function can be used, as we would on any multiclass classification problem. A hyperparameter, lambda, is used to scale the mutual information loss function and is set to 1, and therefore can be ignored.

> Even though InfoGAN introduces an extra hyperparameter λ, it's easy to tune and simply setting to 1 is sufficient for discrete latent codes.

— *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*, 2016.

18.3.2 Continuous Control Variables

The continuous control variables may be used to control the style of the image. The continuous variables are sampled from a uniform distribution, such as between -1 and 1, and provided as input to the generator model.

> ... continuous codes that can capture variations that are continuous in nature: $c2, c3 \sim Unif(-1, 1)$

— *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*, 2016.

The auxiliary model can implement the prediction of continuous control variables with a Gaussian distribution, where the output layer is configured to have one node for the mean, and one node for the standard deviation of the Gaussian, e.g. two outputs are required for each continuous control variable.

> For continuous latent code c_j, there are more options depending on what is the true posterior $P(c_j|x)$. In our experiments, we have found that simply treating $Q(c_j|x)$ as a factored Gaussian is sufficient.

— *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*, 2016.

Nodes that output the mean can use a linear activation function, whereas nodes that output the standard deviation must produce a positive value, therefore an activation function such as the sigmoid can be used to create a value between 0 and 1.

> For continuous latent codes, we parameterize the approximate posterior through a diagonal Gaussian distribution, and the recognition network outputs its mean and standard deviation, where the standard deviation is parameterized through an exponential transformation of the network output to ensure positivity.

— *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*, 2016.

The loss function must be calculated as the mutual information on the Gaussian control codes, meaning they must be reconstructed from the mean and standard deviation prior to calculating the loss. Calculating the entropy and conditional entropy for Gaussian distributed variables can be implemented directly, although is not necessary. Instead, the mean squared error loss can be used. Alternately, the output distribution can be simplified to a uniform distribution for each control variable, a single output node for each variable in the auxiliary model with linear activation can be used, and the model can use the mean squared error loss function.

18.4 How to Develop an InfoGAN for MNIST

In this section, we will take a closer look at the generator (g), discriminator (d), and auxiliary models (q) and how to implement them in Keras. We will develop an InfoGAN implementation for the MNIST dataset (described in Section 7.2), as was done in the InfoGAN paper. The paper explores two versions; the first uses just categorical control codes and allows the model to map one categorical variable to approximately one digit (although there is no ordering of the digits by categorical variables).

Figure 18.1: Example of Varying Generated Digit By Value of Categorical Control Code. Taken from InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets.

The paper also explores a version of the InfoGAN architecture with the one hot encoded categorical variable ($c1$) and two continuous control variables ($c2$ and $c3$). The first continuous variable is discovered to control the rotation of the digits and the second controls the thickness of the digits.

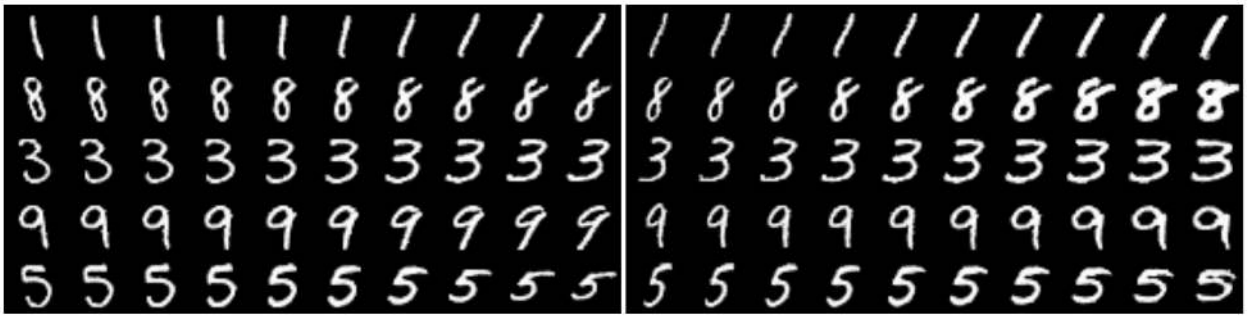

Figure 18.2: Example of Varying Generated Digit Slant and Thickness Using Continuous Control Code. Taken from: InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets.

We will focus on the simpler case of using a categorical control variable with 10 values and

encourage the model to learn to let this variable control the class of the generated digit. You may want to extend this example by either changing the cardinality of the categorical control variable or adding continuous control variables. The configuration of the GAN model used for training on the MNIST dataset was provided as an appendix to the paper, reproduced below. We will use the listed configuration as a starting point in developing our own generator (g), discriminator (d), and auxiliary (q) models.

Table 1: The discriminator and generator CNNs used for MNIST dataset.

discriminator D / recognition network Q	**generator G**
Input 28×28 Gray image	Input $\in \mathbb{R}^{74}$
4×4 conv. 64 lRELU. stride 2	FC. 1024 RELU. batchnorm
4×4 conv. 128 lRELU. stride 2. batchnorm	FC. $7 \times 7 \times 128$ RELU. batchnorm
FC. 1024 lRELU. batchnorm	4×4 upconv. 64 RELU. stride 2. batchnorm
FC. output layer for D, FC.128-batchnorm-lRELU-FC.output for Q	4×4 upconv. 1 channel

Figure 18.3: Summary of Generator, Discriminator and Auxiliary Model Configurations for the InfoGAN for Training MNIST. Taken from: InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets.

Let's start off by developing the generator model as a deep convolutional neural network (e.g. a DCGAN). The model could take the noise vector (z) and control vector (c) as separate inputs and concatenate them before using them as the basis for generating the image. Alternately, the vectors can be concatenated beforehand and provided to a single input layer in the model. The approaches are equivalent and we will use the latter in this case to keep the model simple. The `define_generator()` function below defines the generator model and takes the size of the input vector as an argument.

A fully connected layer takes the input vector and produces a sufficient number of activations to create 512 7×7 feature maps from which the activations are reshaped. These then pass through a normal convolutional layer with 1×1 stride, then two subsequent upsampling transpose convolutional layers with a 2×2 stride first to 14×14 feature maps then to the desired 1 channel 28×28 feature map output with pixel values in the range [-1,-1] via a Tanh activation function. Good generator configuration heuristics are as follows, including a random Gaussian weight initialization, ReLU activations in the hidden layers, and use of batch normalization.

```
# define the standalone generator model
def define_generator(gen_input_size):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image generator input
  in_lat = Input(shape=(gen_input_size,))
  # foundation for 7x7 image
  n_nodes = 512 * 7 * 7
  gen = Dense(n_nodes, kernel_initializer=init)(in_lat)
  gen = Activation('relu')(gen)
  gen = BatchNormalization()(gen)
  gen = Reshape((7, 7, 512))(gen)
  # normal
  gen = Conv2D(128, (4,4), padding='same', kernel_initializer=init)(gen)
```

```
  gen = Activation('relu')(gen)
  gen = BatchNormalization()(gen)
  # upsample to 14x14
  gen = Conv2DTranspose(64, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(gen)
  gen = Activation('relu')(gen)
  gen = BatchNormalization()(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(1, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(gen)
  # tanh output
  out_layer = Activation('tanh')(gen)
  # define model
  model = Model(in_lat, out_layer)
  return model
```

Listing 18.1: Example of a function for defining the generator model.

Next, we can define the discriminator and auxiliary models. The discriminator model is trained in a standalone manner on real and fake images, as per a normal GAN. Neither the generator nor the auxiliary models are fit directly; instead, they are fit as part of a composite model. Both the discriminator and auxiliary models share the same input and feature extraction layers but differ in their output layers. Therefore, it makes sense to define them both at the same time. Again, there are many ways that this architecture could be implemented, but defining the discriminator and auxiliary models as separate models first allows us later to combine them into a larger GAN model directly via the functional API.

The `define_discriminator()` function below defines the discriminator and auxiliary models and takes the cardinality of the categorical variable (e.g. number of values, such as 10) as an input. The shape of the input image is also parameterized as a function argument and set to the default value of the size of the MNIST images. The feature extraction layers involve two downsampling layers, used instead of pooling layers as a best practice. Also following best practice for DCGAN models, we use the `LeakyReLU` activation and batch normalization.

The discriminator model (d) has a single output node and predicts the probability of an input image being real via the sigmoid activation function. The model is compiled as it will be used in a standalone way, optimizing the binary cross-entropy function via the Adam version of stochastic gradient descent with best practice learning rate and momentum. The auxiliary model (q) has one node output for each value in the categorical variable and uses a softmax activation function. A fully connected layer is added between the feature extraction layers and the output layer, as was used in the InfoGAN paper. The model is not compiled as it is not for or used in a standalone manner.

```
# define the standalone discriminator model
def define_discriminator(n_cat, in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=in_shape)
  # downsample to 14x14
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
  d = LeakyReLU(alpha=0.1)(d)
  # downsample to 7x7
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
```

```
  d = LeakyReLU(alpha=0.1)(d)
  d = BatchNormalization()(d)
  # normal
  d = Conv2D(256, (4,4), padding='same', kernel_initializer=init)(d)
  d = LeakyReLU(alpha=0.1)(d)
  d = BatchNormalization()(d)
  # flatten feature maps
  d = Flatten()(d)
  # real/fake output
  out_classifier = Dense(1, activation='sigmoid')(d)
  # define d model
  d_model = Model(in_image, out_classifier)
  # compile d model
  d_model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002, beta_1=0.5))
  # create q model layers
  q = Dense(128)(d)
  q = BatchNormalization()(q)
  q = LeakyReLU(alpha=0.1)(q)
  # q model output
  out_codes = Dense(n_cat, activation='softmax')(q)
  # define q model
  q_model = Model(in_image, out_codes)
  return d_model, q_model
```

Listing 18.2: Example of a function for defining the discriminator and auxiliary models.

Next, we can define the composite GAN model. This model uses all submodels and is the basis for training the weights of the generator model. The `define_gan()` function below implements this and defines and returns the model, taking the three submodels as input. The discriminator is trained in a standalone manner as mentioned, therefore all weights of the discriminator are set as not trainable (in this context only). The output of the generator model is connected to the input of the discriminator model, and to the input of the auxiliary model.

This creates a new composite model that takes a `[noise + control]` vector as input, that then passes through the generator to produce an image. The image then passes through the discriminator model to produce a classification and through the auxiliary model to produce a prediction of the control variables. The model has two output layers that need to be trained with different loss functions. Binary cross-entropy loss is used for the discriminator output, as we did when compiling the discriminator for standalone use, and mutual information loss is used for the auxiliary model, which, in this case, can be implemented directly as categorical cross-entropy and achieve the desired result.

```
# define the combined discriminator, generator and q network model
def define_gan(g_model, d_model, q_model):
  # make weights in the discriminator (some shared with the q model) as not trainable
  for layer in d_model.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # connect g outputs to d inputs
  d_output = d_model(g_model.output)
  # connect g outputs to q inputs
  q_output = q_model(g_model.output)
  # define composite model
  model = Model(g_model.input, [d_output, q_output])
  # compile model
```

```
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'categorical_crossentropy'], optimizer=opt)
  return model
```

Listing 18.3: Example of a function for defining the composite model.

To make the GAN model architecture clearer, we can create the models and a plot of the composite model. The complete example is listed below.

```
# create and plot the infogan model for mnist
from keras.optimizers import Adam
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import BatchNormalization
from keras.layers import Activation
from keras.initializers import RandomNormal
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(n_cat, in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=in_shape)
  # downsample to 14x14
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
  d = LeakyReLU(alpha=0.1)(d)
  # downsample to 7x7
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = LeakyReLU(alpha=0.1)(d)
  d = BatchNormalization()(d)
  # normal
  d = Conv2D(256, (4,4), padding='same', kernel_initializer=init)(d)
  d = LeakyReLU(alpha=0.1)(d)
  d = BatchNormalization()(d)
  # flatten feature maps
  d = Flatten()(d)
  # real/fake output
  out_classifier = Dense(1, activation='sigmoid')(d)
  # define d model
  d_model = Model(in_image, out_classifier)
  # compile d model
  d_model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002, beta_1=0.5))
  # create q model layers
  q = Dense(128)(d)
  q = BatchNormalization()(q)
  q = LeakyReLU(alpha=0.1)(q)
  # q model output
  out_codes = Dense(n_cat, activation='softmax')(q)
  # define q model
  q_model = Model(in_image, out_codes)
```

```
  return d_model, q_model

# define the standalone generator model
def define_generator(gen_input_size):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image generator input
  in_lat = Input(shape=(gen_input_size,))
  # foundation for 7x7 image
  n_nodes = 512 * 7 * 7
  gen = Dense(n_nodes, kernel_initializer=init)(in_lat)
  gen = Activation('relu')(gen)
  gen = BatchNormalization()(gen)
  gen = Reshape((7, 7, 512))(gen)
  # normal
  gen = Conv2D(128, (4,4), padding='same', kernel_initializer=init)(gen)
  gen = Activation('relu')(gen)
  gen = BatchNormalization()(gen)
  # upsample to 14x14
  gen = Conv2DTranspose(64, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(gen)
  gen = Activation('relu')(gen)
  gen = BatchNormalization()(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(1, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(gen)
  # tanh output
  out_layer = Activation('tanh')(gen)
  # define model
  model = Model(in_lat, out_layer)
  return model

# define the combined discriminator, generator and q network model
def define_gan(g_model, d_model, q_model):
  # make weights in the discriminator (some shared with the q model) as not trainable
  for layer in d_model.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # connect g outputs to d inputs
  d_output = d_model(g_model.output)
  # connect g outputs to q inputs
  q_output = q_model(g_model.output)
  # define composite model
  model = Model(g_model.input, [d_output, q_output])
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'categorical_crossentropy'], optimizer=opt)
  return model

# number of values for the categorical control code
n_cat = 10
# size of the latent space
latent_dim = 62
# create the discriminator
d_model, q_model = define_discriminator(n_cat)
# create the generator
```

```
gen_input_size = latent_dim + n_cat
g_model = define_generator(gen_input_size)
# create the gan
gan_model = define_gan(g_model, d_model, q_model)
# plot the model
plot_model(gan_model, to_file='gan_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 18.4: Example of defining and summarizing the InfoGAN models.

Running the example creates all three models, then creates the composite GAN model and saves a plot of the model architecture.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

The plot shows all of the detail for the generator model and the compressed description of the discriminator and auxiliary models. Importantly, note the shape of the output of the discriminator as a single node for predicting whether the image is real or fake, and the 10 nodes for the auxiliary model to predict the categorical control code. Recall that this composite model will only be used to update the model weights of the generator and auxiliary models, and all weights in the discriminator model will remain untrainable, i.e. only updated when the standalone discriminator model is updated.

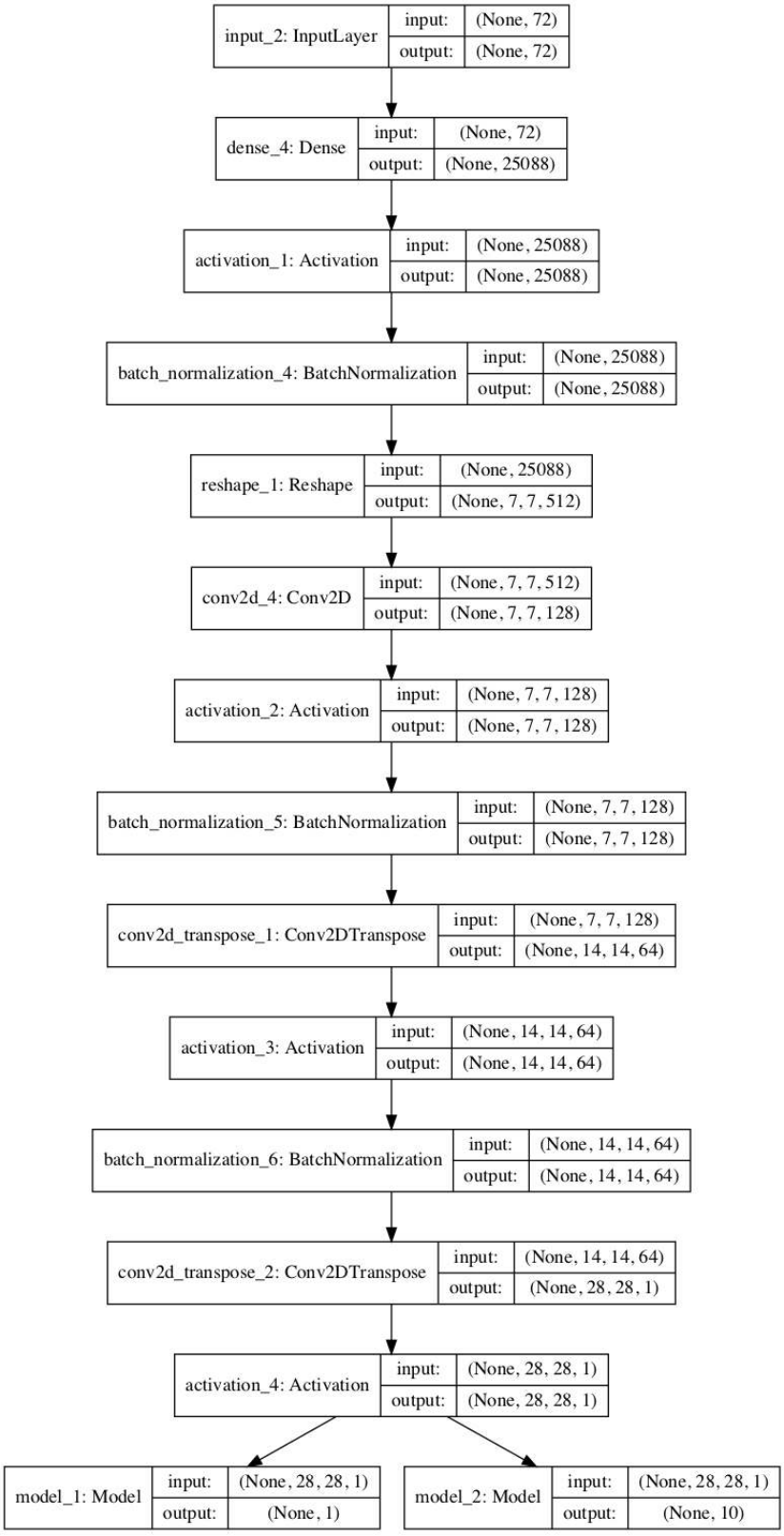

Figure 18.4: Plot of the Composite InfoGAN Model for training the Generator and Auxiliary Models.

Next, we will develop inputs for the generator. Each input will be a vector comprised of

noise and the control codes. Specifically, a vector of Gaussian random numbers and a one hot encoded randomly selected categorical value. The `generate_latent_points()` function below implements this, taking as input the size of the latent space, the number of categorical values, and the number of samples to generate as arguments. The function returns the input concatenated vectors as input for the generator model, as well as the standalone control codes. The standalone control codes will be required when updating the generator and auxiliary models via the composite GAN model, specifically for calculating the mutual information loss for the auxiliary model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_cat, n_samples):
  # generate points in the latent space
  z_latent = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_latent = z_latent.reshape(n_samples, latent_dim)
  # generate categorical codes
  cat_codes = randint(0, n_cat, n_samples)
  # one hot encode
  cat_codes = to_categorical(cat_codes, num_classes=n_cat)
  # concatenate latent points and control codes
  z_input = hstack((z_latent, cat_codes))
  return [z_input, cat_codes]
```

Listing 18.5: Example of a function for sampling random points in the latent space and control codes.

Next, we can generate real and fake examples. The MNIST dataset can be loaded, transformed into 3D input by adding an additional dimension for the grayscale images, and scaling all pixel values to the range [-1,1] to match the output from the generator model. This is implemented in the `load_real_samples()` function below. We can retrieve batches of real samples required when training the discriminator by choosing a random subset of the dataset. This is implemented in the `generate_real_samples()` function below that returns the images and the class label of 1, to indicate to the discriminator that they are real images. The discriminator also requires batches of fake samples generated via the generator, using the vectors from `generate_latent_points()` function as input. The `generate_fake_samples()` function below implements this, returning the generated images along with the class label of 0, to indicate to the discriminator that they are fake images.

```
# load images
def load_real_samples():
  # load dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  print(X.shape)
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
```

```
  ix = randint(0, dataset.shape[0], n_samples)
  # select images and labels
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_cat, n_samples):
  # generate points in latent space and control codes
  z_input, _ = generate_latent_points(latent_dim, n_cat, n_samples)
  # predict outputs
  images = generator.predict(z_input)
  # create class labels
  y = zeros((n_samples, 1))
  return images, y
```

Listing 18.6: Example of a defining functions for loading and sampling real and fake images .

Next, we need to keep track of the quality of the generated images. We will periodically use the generator to generate a sample of images and save the generator and composite models to file. We can then review the generated images at the end of training in order to choose a final generator model and load the model to start using it to generate images. The `summarize_performance()` function below implements this, first generating 100 images, scaling their pixel values back to the range [0,1], and saving them as a plot of images in a 10×10 square. The generator and composite GAN models are also saved to file, with a unique filename based on the training iteration number.

```
# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, gan_model, latent_dim, n_cat, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_cat, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(100):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename1 = 'generated_plot_%04d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # save the generator model
  filename2 = 'model_%04d.h5' % (step+1)
  g_model.save(filename2)
  # save the gan model
  filename3 = 'gan_model_%04d.h5' % (step+1)
  gan_model.save(filename3)
  print('>Saved: %s, %s, and %s' % (filename1, filename2, filename3))
```

Listing 18.7: Example of a function for summarizing the performance of the model and saving

the models.

Finally, we can train the InfoGAN. This is implemented in the `train()` function below that takes the defined models and configuration as arguments and runs the training process. The models are trained for 100 epochs and 64 samples are used in each batch. There are 60,000 images in the MNIST training dataset, therefore one epoch involves $\frac{60000}{64}$, or 937 batches or training iterations. Multiplying this by the number of epochs, or 100, means that there will be a total of 93,700 total training iterations. Each training iteration involves first updating the discriminator with half a batch of real samples and half a batch of fake samples to form one batch worth of weight updates, or 64, each iteration. Next, the composite GAN model is updated based on a batch worth of noise and control code inputs. The loss of the discriminator on real and fake images and the loss of the generator and auxiliary model is reported each training iteration.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_cat, n_epochs=100,
    n_batch=64):
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_steps):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator and q model weights
    d_loss1 = d_model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_cat, half_batch)
    # update discriminator model weights
    d_loss2 = d_model.train_on_batch(X_fake, y_fake)
    # prepare points in latent space as input for the generator
    z_input, cat_codes = generate_latent_points(latent_dim, n_cat, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the g via the d and q error
    _,g_1,g_2 = gan_model.train_on_batch(z_input, [y_gan, cat_codes])
    # summarize loss on this batch
    print('>%d, d[%.3f,%.3f], g[%.3f] q[%.3f]' % (i+1, d_loss1, d_loss2, g_1, g_2))
    # evaluate the model performance every 'epoch'
    if (i+1) % (bat_per_epo * 10) == 0:
      summarize_performance(i, g_model, gan_model, latent_dim, n_cat)
```

Listing 18.8: Example of a function for training the InfoGAN models.

We can then configure and create the models, then run the training process. We will use 10 values for the single categorical variable to match the 10 known classes in the MNIST dataset. We will use a latent space with 64 dimensions to match the InfoGAN paper, meaning, in this case, each input vector to the generator model will be 64 (random Gaussian variables) + 10 (one hot encoded control variable) or 72 elements in length.

```
...
```

```
# number of values for the categorical control code
n_cat = 10
# size of the latent space
latent_dim = 62
# create the discriminator
d_model, q_model = define_discriminator(n_cat)
# create the generator
gen_input_size = latent_dim + n_cat
g_model = define_generator(gen_input_size)
# create the gan
gan_model = define_gan(g_model, d_model, q_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim, n_cat)
```

Listing 18.9: Example of configuring and starting the training process.

Tying this together, the complete example of training an InfoGAN model on the MNIST dataset with a single categorical control variable is listed below.

```
# example of training an infogan on mnist
from numpy import zeros
from numpy import ones
from numpy import expand_dims
from numpy import hstack
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.initializers import RandomNormal
from keras.utils import to_categorical
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import BatchNormalization
from keras.layers import Activation
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(n_cat, in_shape=(28,28,1)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=in_shape)
  # downsample to 14x14
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
  d = LeakyReLU(alpha=0.1)(d)
  # downsample to 7x7
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = LeakyReLU(alpha=0.1)(d)
  d = BatchNormalization()(d)
```

```
  # normal
  d = Conv2D(256, (4,4), padding='same', kernel_initializer=init)(d)
  d = LeakyReLU(alpha=0.1)(d)
  d = BatchNormalization()(d)
  # flatten feature maps
  d = Flatten()(d)
  # real/fake output
  out_classifier = Dense(1, activation='sigmoid')(d)
  # define d model
  d_model = Model(in_image, out_classifier)
  # compile d model
  d_model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002, beta_1=0.5))
  # create q model layers
  q = Dense(128)(d)
  q = BatchNormalization()(q)
  q = LeakyReLU(alpha=0.1)(q)
  # q model output
  out_codes = Dense(n_cat, activation='softmax')(q)
  # define q model
  q_model = Model(in_image, out_codes)
  return d_model, q_model

# define the standalone generator model
def define_generator(gen_input_size):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image generator input
  in_lat = Input(shape=(gen_input_size,))
  # foundation for 7x7 image
  n_nodes = 512 * 7 * 7
  gen = Dense(n_nodes, kernel_initializer=init)(in_lat)
  gen = Activation('relu')(gen)
  gen = BatchNormalization()(gen)
  gen = Reshape((7, 7, 512))(gen)
  # normal
  gen = Conv2D(128, (4,4), padding='same', kernel_initializer=init)(gen)
  gen = Activation('relu')(gen)
  gen = BatchNormalization()(gen)
  # upsample to 14x14
  gen = Conv2DTranspose(64, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(gen)
  gen = Activation('relu')(gen)
  gen = BatchNormalization()(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(1, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(gen)
  # tanh output
  out_layer = Activation('tanh')(gen)
  # define model
  model = Model(in_lat, out_layer)
  return model

# define the combined discriminator, generator and q network model
def define_gan(g_model, d_model, q_model):
  # make weights in the discriminator (some shared with the q model) as not trainable
  for layer in d_model.layers:
```

```
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # connect g outputs to d inputs
  d_output = d_model(g_model.output)
  # connect g outputs to q inputs
  q_output = q_model(g_model.output)
  # define composite model
  model = Model(g_model.input, [d_output, q_output])
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'categorical_crossentropy'], optimizer=opt)
  return model

# load images
def load_real_samples():
  # load dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  print(X.shape)
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # select images and labels
  X = dataset[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_cat, n_samples):
  # generate points in the latent space
  z_latent = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_latent = z_latent.reshape(n_samples, latent_dim)
  # generate categorical codes
  cat_codes = randint(0, n_cat, n_samples)
  # one hot encode
  cat_codes = to_categorical(cat_codes, num_classes=n_cat)
  # concatenate latent points and control codes
  z_input = hstack((z_latent, cat_codes))
  return [z_input, cat_codes]

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_cat, n_samples):
  # generate points in latent space and control codes
  z_input, _ = generate_latent_points(latent_dim, n_cat, n_samples)
  # predict outputs
  images = generator.predict(z_input)
```

```
  # create class labels
  y = zeros((n_samples, 1))
  return images, y

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, gan_model, latent_dim, n_cat, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_cat, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(100):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename1 = 'generated_plot_%04d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # save the generator model
  filename2 = 'model_%04d.h5' % (step+1)
  g_model.save(filename2)
  # save the gan model
  filename3 = 'gan_model_%04d.h5' % (step+1)
  gan_model.save(filename3)
  print('>Saved: %s, %s, and %s' % (filename1, filename2, filename3))

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_cat, n_epochs=100,
    n_batch=64):
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_steps):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator and q model weights
    d_loss1 = d_model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_cat, half_batch)
    # update discriminator model weights
    d_loss2 = d_model.train_on_batch(X_fake, y_fake)
    # prepare points in latent space as input for the generator
    z_input, cat_codes = generate_latent_points(latent_dim, n_cat, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the g via the d and q error
    _,g_1,g_2 = gan_model.train_on_batch(z_input, [y_gan, cat_codes])
    # summarize loss on this batch
```

```
    print('>%d, d[%.3f,%.3f], g[%.3f] q[%.3f]' % (i+1, d_loss1, d_loss2, g_1, g_2))
    # evaluate the model performance every 'epoch'
    if (i+1) % (bat_per_epo * 10) == 0:
      summarize_performance(i, g_model, gan_model, latent_dim, n_cat)

# number of values for the categorical control code
n_cat = 10
# size of the latent space
latent_dim = 62
# create the discriminator
d_model, q_model = define_discriminator(n_cat)
# create the generator
gen_input_size = latent_dim + n_cat
g_model = define_generator(gen_input_size)
# create the gan
gan_model = define_gan(g_model, d_model, q_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim, n_cat)
```

Listing 18.10: Example of training the InfoGAN on the MNIST dataset.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

The loss across the models is reported each training iteration. If the loss for the discriminator remains at 0.0 or goes to 0.0 for an extended time, this may be a sign of a training failure and you may want to restart the training process. The discriminator loss may start at 0.0, but will likely rise, as it did in this specific case. The loss for the auxiliary model will likely go to zero, as it perfectly predicts the categorical variable. Loss for the generator and discriminator models is likely to hover around 1.0 eventually, to demonstrate a stable training process or equilibrium between the training of the two models.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
>1, d[0.924,0.758], g[0.448] q[2.909]
>2, d[0.000,2.699], g[0.547] q[2.704]
>3, d[0.000,1.557], g[1.175] q[2.820]
>4, d[0.000,0.941], g[1.466] q[2.813]
>5, d[0.000,1.013], g[1.908] q[2.715]
...
>93696, d[0.814,1.212], g[1.283] q[0.000]
>93697, d[1.063,0.920], g[1.132] q[0.000]
>93698, d[0.999,1.188], g[1.128] q[0.000]
>93699, d[0.935,0.985], g[1.229] q[0.000]
>93700, d[0.968,1.016], g[1.200] q[0.001]
>Saved: generated_plot_93700.png, model_93700.h5, and gan_model_93700.h5
```

Listing 18.11: Example output from training the InfoGAN on the MNIST dataset.

Plots and models are saved every 10 epochs or every 9,370 training iterations. Reviewing the plots should show poor quality images in early epochs and improved and stable quality images in later epochs. For example, the plot of images saved after the first 10 epochs is below showing low-quality generated images.

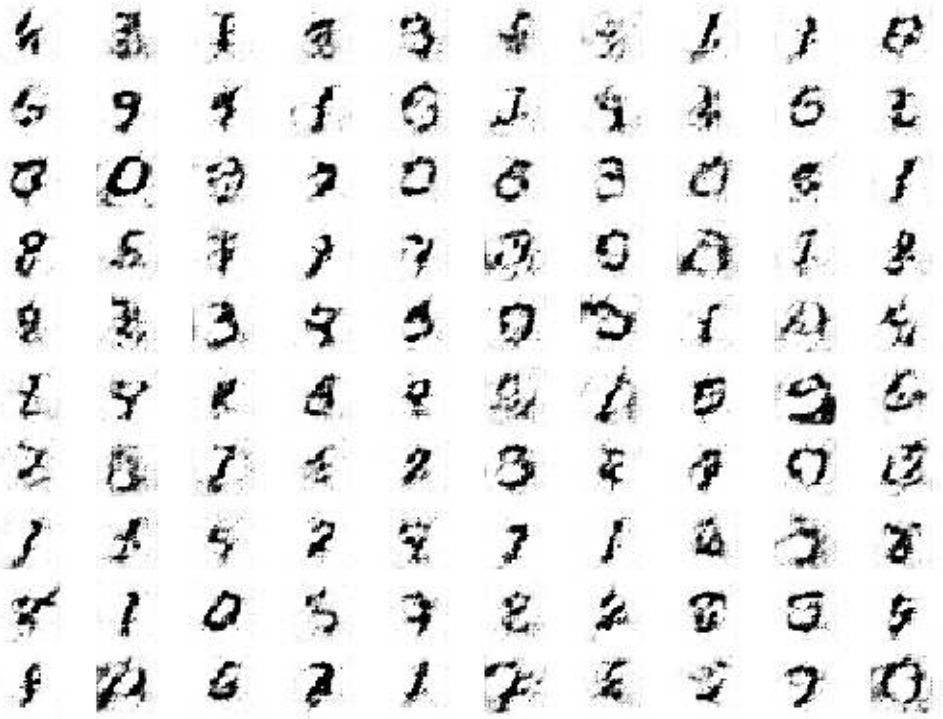

Figure 18.5: Plot of 100 Random Images Generated by the InfoGAN after 10 Training Epochs.

More epochs does not mean better quality, meaning that the best quality images may not be those from the final model saved at the end of training. Review the plots and choose a final model with the best image quality. In this case, we will use the model saved after 100 epochs or 93,700 training iterations.

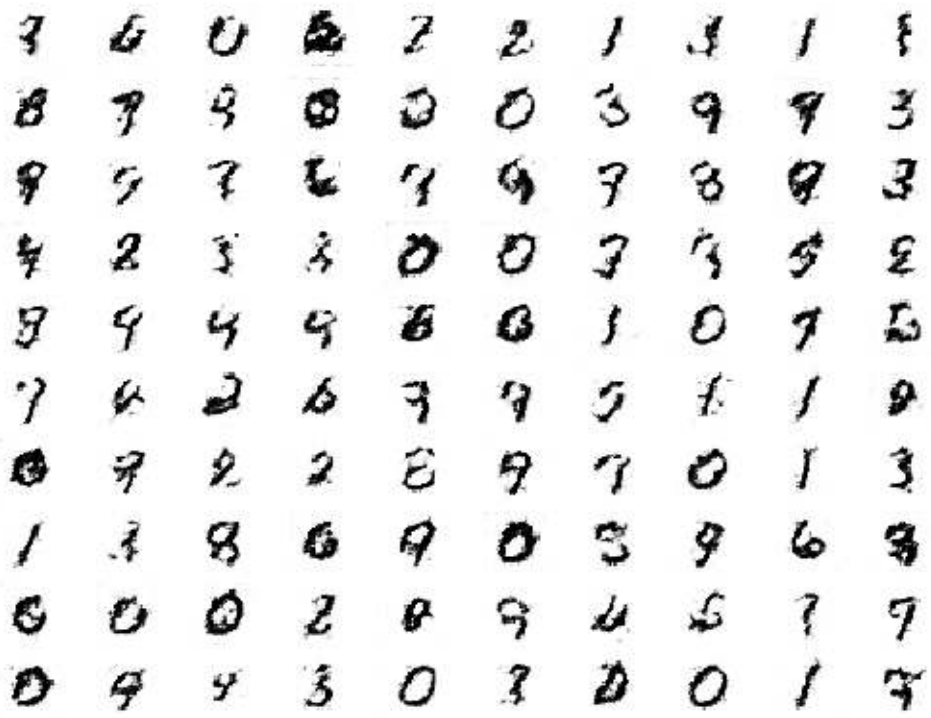

Figure 18.6: Plot of 100 Random Images Generated by the InfoGAN after 100 Training Epochs.

18.5 How to Use Control Codes With an InfoGAN

Now that we have trained the InfoGAN model, we can explore how to use it. First, we can load the model and use it to generate random images, as we did during training. The complete example is listed below. Change the model filename to match the model filename that generated the best images during your training run.

```
# example of loading the generator model and generating images
from math import sqrt
from numpy import hstack
from numpy.random import randn
from numpy.random import randint
from keras.models import load_model
from keras.utils import to_categorical
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_cat, n_samples):
  # generate points in the latent space
  z_latent = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_latent = z_latent.reshape(n_samples, latent_dim)
```

```
  # generate categorical codes
  cat_codes = randint(0, n_cat, n_samples)
  # one hot encode
  cat_codes = to_categorical(cat_codes, num_classes=n_cat)
  # concatenate latent points and control codes
  z_input = hstack((z_latent, cat_codes))
  return [z_input, cat_codes]

# create a plot of generated images
def create_plot(examples, n_examples):
  # plot images
  for i in range(n_examples):
    # define subplot
    pyplot.subplot(sqrt(n_examples), sqrt(n_examples), 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  pyplot.show()

# load model
model = load_model('model_93700.h5')
# number of values for the categorical control code
n_cat = 10
# size of the latent space
latent_dim = 62
# number of examples to generate
n_samples = 100
# generate points in latent space and control codes
z_input, _ = generate_latent_points(latent_dim, n_cat, n_samples)
# predict outputs
X = model.predict(z_input)
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
# plot the result
create_plot(X, n_samples)
```

Listing 18.12: Example of loading the saved generator model and using it to generate images.

Running the example will load the saved generator model and use it to generate 100 random images and plot the images on a 10×10 grid.

Figure 18.7: Plot of 100 Random Images Created by Loading the Saved InfoGAN Generator Model.

Next, we can update the example to test how much control our control variable gives us. We can update the `generate_latent_points()` function to take an argument of the value for the categorical value in [0,9], encode it, and use it as input along with noise vectors.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_cat, n_samples, digit):
  # generate points in the latent space
  z_latent = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_latent = z_latent.reshape(n_samples, latent_dim)
  # define categorical codes
  cat_codes = asarray([digit for _ in range(n_samples)])
  # one hot encode
  cat_codes = to_categorical(cat_codes, num_classes=n_cat)
  # concatenate latent points and control codes
  z_input = hstack((z_latent, cat_codes))
  return [z_input, cat_codes]
```

Listing 18.13: Example of a function for sampling points in latent space for a specific control code.

We can test this by generating a grid of 25 images with the categorical value 1. The complete example is listed below.

```
# example of testing different values of the categorical control variable
from math import sqrt
from numpy import asarray
from numpy import hstack
from numpy.random import randn
from keras.models import load_model
from keras.utils import to_categorical
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_cat, n_samples, digit):
  # generate points in the latent space
  z_latent = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_latent = z_latent.reshape(n_samples, latent_dim)
  # define categorical codes
  cat_codes = asarray([digit for _ in range(n_samples)])
  # one hot encode
  cat_codes = to_categorical(cat_codes, num_classes=n_cat)
  # concatenate latent points and control codes
  z_input = hstack((z_latent, cat_codes))
  return [z_input, cat_codes]

# create and save a plot of generated images
def save_plot(examples, n_examples):
  # plot images
  for i in range(n_examples):
    # define subplot
    pyplot.subplot(sqrt(n_examples), sqrt(n_examples), 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  pyplot.show()

# load model
model = load_model('model_93700.h5')
# number of categorical control codes
n_cat = 10
# size of the latent space
latent_dim = 62
# number of examples to generate
n_samples = 25
# define digit
digit = 1
# generate points in latent space and control codes
z_input, _ = generate_latent_points(latent_dim, n_cat, n_samples, digit)
# predict outputs
X = model.predict(z_input)
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
# plot the result
save_plot(X, n_samples)
```

Listing 18.14: Example of loading the saved generator model and using it to generate images

for a specific control code.

The result is a grid of 25 generated images generated with the categorical code set to the value 1.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The values of the control code are expected to influence the generated images; specifically, they are expected to influence the digit type. They are not expected to be ordered though, e.g. control codes of 1, 2, and 3 to create those digits. Nevertheless, in this case, the control code with a value of 1 has resulted in images generated that look like a 1.

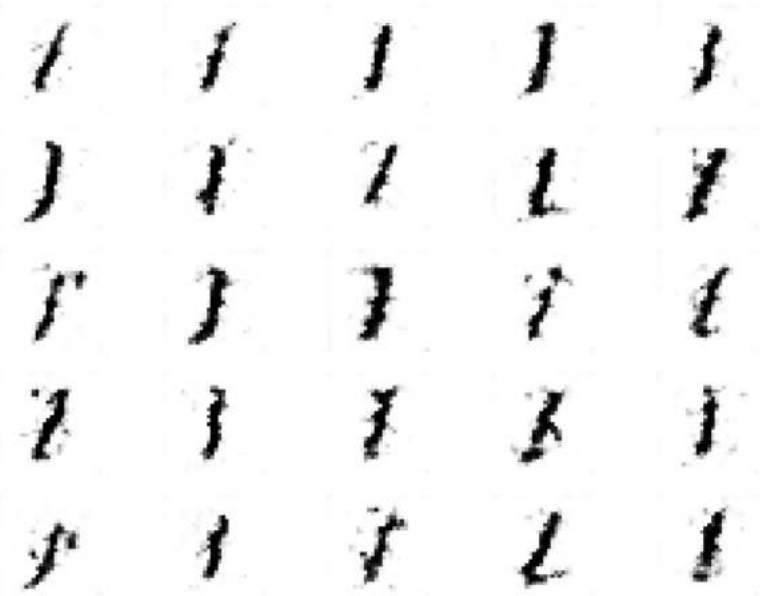

Figure 18.8: Plot of 25 Images Generated by the InfoGAN Model With the Categorical Control Code Set to 1.

Experiment with different digits and review what the value is controlling exactly about the image. For example, setting the value to 5 in this case (digit = 5) results in generated images that look like the number 8.

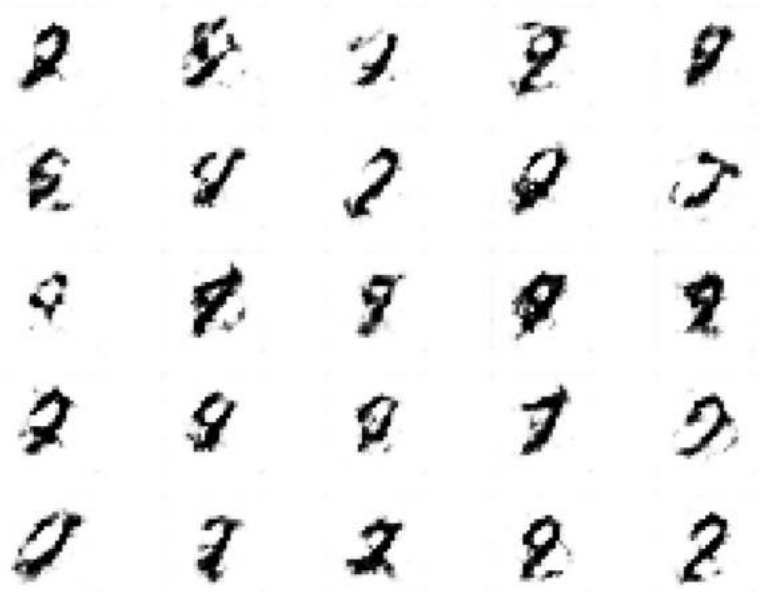

Figure 18.9: Plot of 25 Images Generated by the InfoGAN Model With the Categorical Control Code Set to 5.

18.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Change Cardinality**. Update the example to use different cardinality of the categorical control variable (e.g. more or fewer values) and review the effect on the training process and the control over generated images.
- **Uniform Control Variables**. Update the example and add two uniform continuous control variables to the auxiliary model and review the effect on the training process and the control over generated images.
- **Gaussian Control Variables**. Update the example and add two Gaussian continuous control variables to the auxiliary model and review the effect on the training process and the control over generated images.

If you explore any of these extensions, I'd love to know.

18.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

18.7.1 Papers

- InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets, 2016.
https://arxiv.org/abs/1606.03657

- Understanding Mutual Information and its use in InfoGAN, 2016.
https://mrdrozdov.github.io/static/papers/infogan.pdf

- The IM Algorithm: A Variational Approach To Information Maximization, 2004.
http://web4.cs.ucl.ac.uk/staff/D.Barber/publications/barber-agakov-IM-nips03.pdf

18.7.2 API

- Keras Datasets API..
https://keras.io/datasets/

- Keras Sequential Model API.
https://keras.io/models/sequential/

- Keras Convolutional Layers API.
https://keras.io/layers/convolutional/

- How can I "freeze" Keras layers?.
https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers

- MatplotLib API.
https://matplotlib.org/api/

- NumPy Random sampling (numpy.random) API.
https://docs.scipy.org/doc/numpy/reference/routines.random.html

- NumPy Array manipulation routines.
https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html

18.7.3 Articles

- InfoGAN (Official), OpenAI, GitHub.
https://github.com/openai/InfoGAN

- Mutual information, Wikipedia.
https://en.wikipedia.org/wiki/Mutual_information

- Conditional mutual information, Wikipedia.
https://en.wikipedia.org/wiki/Conditional_mutual_information

18.8 Summary

In this tutorial, you discovered how to implement an Information Maximizing Generative Adversarial Network model from scratch. Specifically, you learned:

- The InfoGAN is motivated by the desire to disentangle and control the properties in generated images.
- The InfoGAN involves the addition of control variables to generate an auxiliary model that predicts the control variables, trained via mutual information loss function.
- How to develop and train an InfoGAN model from scratch and use the control variables to control which digit is generated by the model.

18.8.1 Next

In the next tutorial, you will discover the auxiliary classifier GAN, that introduces additional models to improve the stability of the training algorithm.

Chapter 19

How to Develop an Auxiliary Classifier GAN (AC-GAN)

Generative Adversarial Networks, or GANs, are an architecture for training generative models, such as deep convolutional neural networks for generating images. The conditional generative adversarial network, or cGAN for short, is a type of GAN that involves the conditional generation of images by a generator model. Image generation can be conditional on a class label, if available, allowing the targeted generated of images of a given type. The Auxiliary Classifier GAN, or AC-GAN for short, is an extension of the conditional GAN that changes the discriminator to predict the class label of a given image rather than receive it as input. It has the effect of stabilizing the training process and allowing the generation of large high-quality images whilst learning a representation in the latent space that is independent of the class label. In this tutorial, you will discover how to develop an auxiliary classifier generative adversarial network for generating photographs of clothing. After completing this tutorial, you will know:

- The auxiliary classifier GAN is a type of conditional GAN that requires that the discriminator predict the class label of a given image.
- How to develop generator, discriminator, and composite models for the AC-GAN.
- How to train, evaluate, and use an AC-GAN to generate photographs of clothing from the Fashion-MNIST dataset.

Let's get started.

19.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Auxiliary Classifier Generative Adversarial Networks
2. Fashion-MNIST Clothing Photograph Dataset
3. How to Define AC-GAN Models
4. How to Develop an AC-GAN for Fashion-MNIST
5. How to Generate Items of Clothing With a Fit AC-GAN

19.2 Auxiliary Classifier Generative Adversarial Networks

The generative adversarial network is an architecture for training a generative model, typically deep convolutional neural networks for generating image. The architecture is comprised of both a generator model that takes random points from a latent space as input and generates images, and a discriminator for classifying images as either real (from the dataset) or fake (generated). Both models are then trained simultaneously in a zero-sum game. A conditional GAN, cGAN or CGAN for short, is an extension of the GAN architecture that adds structure to the latent space (see Chapter 17). The training of the GAN model is changed so that the generator is provided both with a point in the latent space and a class label as input, and attempts to generate an image for that class. The discriminator is provided with both an image and the class label and must classify whether the image is real or fake as before.

The addition of the class as input makes the image generation process, and image classification process, conditional on the class label, hence the name. The effect is both a more stable training process and a resulting generator model that can be used to generate images of a given specific type, e.g. for a class label. The Auxiliary Classifier GAN, or AC-GAN for short, is a further extension of the GAN architecture building upon the CGAN extension. It was introduced by Augustus Odena, et al. from Google Brain in the 2016 paper titled *Conditional Image Synthesis with Auxiliary Classifier GANs*. As with the conditional GAN, the generator model in the AC-GAN is provided both with a point in the latent space and the class label as input, e.g. the image generation process is conditional.

The main difference is in the discriminator model, which is only provided with the image as input, unlike the conditional GAN that is provided with the image and class label as input. The discriminator model must then predict whether the given image is real or fake as before, and must also predict the class label of the image.

> ... the model [...] is class conditional, but with an auxiliary decoder that is tasked with reconstructing class labels.
>
> — *Conditional Image Synthesis With Auxiliary Classifier GANs*, 2016.

The architecture is described in such a way that the discriminator and auxiliary classifier may be considered separate models that share model weights. In practice, the discriminator and auxiliary classifier can be implemented as a single neural network model with two outputs. The first output is a single probability via the sigmoid activation function that indicates the *realness* of the input image and is optimized using binary cross-entropy like a normal GAN discriminator model. The second output is a probability of the image belonging to each class via the softmax activation function, like any given multiclass classification neural network model, and is optimized using categorical cross-entropy. To summarize:

- **Generator Model**:
  - **Input**: Random point from the latent space, and the class label.
  - **Output**: Generated image.
- **Discriminator Model**:

- **Input**: Image.
- **Output**: Probability that the provided image is real, probability of the image belonging to each known class.

The plot below summarizes the inputs and outputs of a range of conditional GANs, including the AC-GAN, providing some context for the differences.

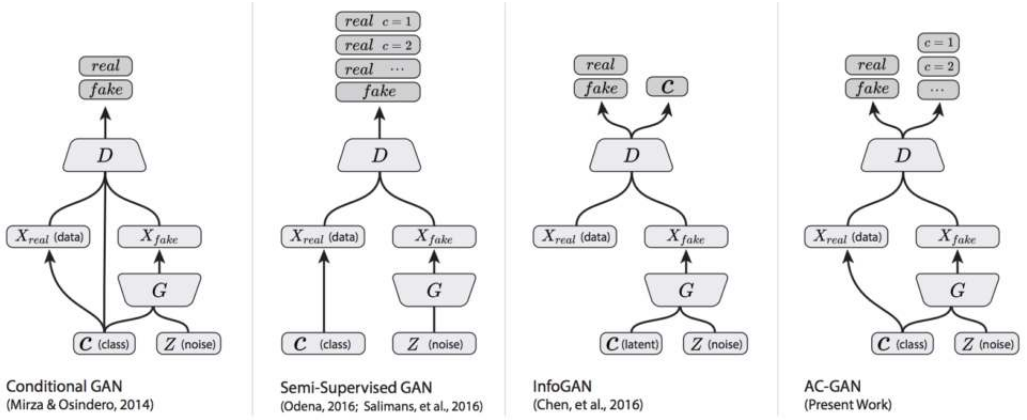

Figure 19.1: Summary of the Differences Between the Conditional GAN, Semi-Supervised GAN, InfoGAN, and AC-GAN. Taken from an early version of: Conditional Image Synthesis With Auxiliary Classifier GANs.

The discriminator seeks to maximize the probability of correctly classifying real and fake images (LS) and correctly predicting the class label (LC) of a real or fake image (e.g. $LS + LC$). The generator seeks to minimize the ability of the discriminator to discriminate real and fake images whilst also maximizing the ability of the discriminator predicting the class label of real and fake images (e.g. $LC - LS$).

> The objective function has two parts: the log-likelihood of the correct source, LS, and the log-likelihood of the correct class, LC. [...] D is trained to maximize $LS + LC$ while G is trained to maximize $LC - LS$.
>
> — *Conditional Image Synthesis With Auxiliary Classifier GANs*, 2016.

The resulting generator learns a latent space representation that is independent of the class label, unlike the conditional GAN. The effect of changing the conditional GAN in this way is both a more stable training process and the ability of the model to generate higher quality images with a larger size than had been previously possible, e.g. 128×128 pixels.

> ... we demonstrate that adding more structure to the GAN latent space along with a specialized cost function results in higher quality samples. [...] Importantly, we demonstrate quantitatively that our high resolution samples are not just naive resizings of low resolution samples.
>
> — *Conditional Image Synthesis With Auxiliary Classifier GANs*, 2016.

19.3 Fashion-MNIST Clothing Photograph Dataset

The Fashion-MNIST dataset is proposed as a more challenging replacement dataset for the MNIST handwritten digit dataset. It is a dataset comprised of 60,000 small square 28×28 pixel grayscale images of items of 10 types of clothing, such as shoes, t-shirts, dresses, and more. Keras provides access to the Fashion-MNIST dataset via the `fashion_mnist.load_dataset()` function. It returns two tuples, one with the input and output elements for the standard training dataset, and another with the input and output elements for the standard test dataset. The loading and preparation of this dataset was described in Section 17.3.

We will use the images in the training dataset as the basis for training a Generative Adversarial Network. Specifically, the generator model will learn how to generate new plausible items of clothing, using a discriminator that will try to distinguish between real images from the Fashion-MNIST training dataset and new images output by the generator model, and predict the class label for each. This is a relatively simple problem that does not require a sophisticated generator or discriminator model, although it does require the generation of a grayscale output image.

19.4 How to Define AC-GAN Models

In this section, we will develop the generator, discriminator, and composite models for the AC-GAN. The appendix of the AC-GAN paper provides suggestions for generator and discriminator configurations that we will use as inspiration. The table below summarizes these suggestions for the CIFAR-10 dataset, taken from the paper.

Operation	Kernel	Strides	Feature maps	BN?	Dropout	Nonlinearity
$G_x(z) - 110 \times 1 \times 1$ input						
Linear	N/A	N/A	384	×	0.0	ReLU
Transposed Convolution	5×5	2×2	192	√	0.0	ReLU
Transposed Convolution	5×5	2×2	96	√	0.0	ReLU
Transposed Convolution	5×5	2×2	3	×	0.0	Tanh
$D(x) - 32 \times 32 \times 3$ input						
Convolution	3×3	2×2	16	×	0.5	Leaky ReLU
Convolution	3×3	1×1	32	√	0.5	Leaky ReLU
Convolution	3×3	2×2	64	√	0.5	Leaky ReLU
Convolution	3×3	1×1	128	√	0.5	Leaky ReLU
Convolution	3×3	2×2	256	√	0.5	Leaky ReLU
Convolution	3×3	1×1	512	√	0.5	Leaky ReLU
Linear	N/A	N/A	11	×	0.0	Soft-Sigmoid
Generator Optimizer	Adam ($\alpha = [0.0001, 0.0002, 0.0003]$, $\beta_1 = 0.5$, $\beta_2 = 0.999$)					
Discriminator Optimizer	Adam ($\alpha = [0.0001, 0.0002, 0.0003]$, $\beta_1 = 0.5$, $\beta_2 = 0.999$)					
Batch size	100					
Iterations	50000					
Leaky ReLU slope	0.2					
Activation noise standard deviation	$[0, 0.1, 0.2]$					
Weight, bias initialization	Isotropic gaussian ($\mu = 0$, $\sigma = 0.02$), Constant(0)					

Figure 19.2: AC-GAN Generator and Discriminator Model Configuration Suggestions. Taken from: Conditional Image Synthesis With Auxiliary Classifier GANs.

19.4.1 AC-GAN Discriminator Model

Let's start with the discriminator model. The discriminator model must take as input an image and predict both the probability of the *realness* of the image and the probability of the image belonging to each of the given classes. The input images will have the shape $28 \times 28 \times 1$ and there are 10 classes for the items of clothing in the Fashion-MNIST dataset. The model can be defined as per the DCGAN architecture. That is, using Gaussian weight initialization, `BatchNormalization`, `LeakyReLU`, `Dropout`, and a 2×2 stride for downsampling instead of pooling layers. For example, below is the bulk of the discriminator model defined using the Keras functional API.

```
...
# weight initialization
init = RandomNormal(stddev=0.02)
# image input
in_image = Input(shape=in_shape)
# downsample to 14x14
fe = Conv2D(32, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
fe = LeakyReLU(alpha=0.2)(fe)
fe = Dropout(0.5)(fe)
# normal
fe = Conv2D(64, (3,3), padding='same', kernel_initializer=init)(fe)
fe = BatchNormalization()(fe)
fe = LeakyReLU(alpha=0.2)(fe)
fe = Dropout(0.5)(fe)
# downsample to 7x7
fe = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(fe)
fe = BatchNormalization()(fe)
fe = LeakyReLU(alpha=0.2)(fe)
fe = Dropout(0.5)(fe)
# normal
fe = Conv2D(256, (3,3), padding='same', kernel_initializer=init)(fe)
fe = BatchNormalization()(fe)
fe = LeakyReLU(alpha=0.2)(fe)
fe = Dropout(0.5)(fe)
# flatten feature maps
fe = Flatten()(fe)
...
```

Listing 19.1: Example of a defining a normal DCGAN discriminator model.

The main difference is that the model has two output layers. The first is a single node with the sigmoid activation for predicting the realness of the image.

```
...
# real/fake output
out1 = Dense(1, activation='sigmoid')(fe)
```

Listing 19.2: Example of a defining the first output layer for the discriminator model.

The second is multiple nodes, one for each class, using the softmax activation function to predict the class label of the given image.

```
...
# class label output
out2 = Dense(n_classes, activation='softmax')(fe)
```

Listing 19.3: Example of a defining the second output layer for the discriminator model.

We can then construct the image with a single input and two outputs.

```
...
# define model
model = Model(in_image, [out1, out2])
```

Listing 19.4: Example of a defining the discriminator model with two output layers.

The model must be trained with two loss functions, binary cross-entropy for the first output layer, and categorical cross-entropy loss for the second output layer. Rather than comparing a one hot encoding of the class labels to the second output layer, as we might do normally, we can compare the integer class labels directly. We can achieve this automatically using the sparse categorical cross-entropy loss function. This will have the identical effect of the categorical cross-entropy but avoids the step of having to manually one hot encode the target labels. When compiling the model, we can inform Keras to use the two different loss functions for the two output layers by specifying a list of function names as strings; for example:

```
loss=['binary_crossentropy', 'sparse_categorical_crossentropy']
```

Listing 19.5: Example of a defining two loss functions for the discriminator model.

The model is fit using the Adam version of stochastic gradient descent with a small learning rate and modest momentum, as is recommended for DCGANs.

```
...
# compile model
opt = Adam(lr=0.0002, beta_1=0.5)
model.compile(loss=['binary_crossentropy', 'sparse_categorical_crossentropy'],
    optimizer=opt)
```

Listing 19.6: Example of a compiling the discriminator model.

Tying this together, the `define_discriminator()` function will define and compile the discriminator model for the AC-GAN. The shape of the input images and the number of classes are parameterized and set with defaults, allowing them to be easily changed for your own project in the future.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1), n_classes=10):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=in_shape)
  # downsample to 14x14
  fe = Conv2D(32, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # normal
  fe = Conv2D(64, (3,3), padding='same', kernel_initializer=init)(fe)
  fe = BatchNormalization()(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # downsample to 7x7
```

```
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(fe)
  fe = BatchNormalization()(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # normal
  fe = Conv2D(256, (3,3), padding='same', kernel_initializer=init)(fe)
  fe = BatchNormalization()(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # real/fake output
  out1 = Dense(1, activation='sigmoid')(fe)
  # class label output
  out2 = Dense(n_classes, activation='softmax')(fe)
  # define model
  model = Model(in_image, [out1, out2])
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'sparse_categorical_crossentropy'],
      optimizer=opt)
  return model
```

Listing 19.7: Example of a function for defining the discriminator model.

We can define and summarize this model. The complete example is listed below.

```
# example of defining the discriminator model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers import BatchNormalization
from keras.initializers import RandomNormal
from keras.optimizers import Adam
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1), n_classes=10):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=in_shape)
  # downsample to 14x14
  fe = Conv2D(32, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # normal
  fe = Conv2D(64, (3,3), padding='same', kernel_initializer=init)(fe)
  fe = BatchNormalization()(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # downsample to 7x7
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(fe)
```

```
  fe = BatchNormalization()(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # normal
  fe = Conv2D(256, (3,3), padding='same', kernel_initializer=init)(fe)
  fe = BatchNormalization()(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # real/fake output
  out1 = Dense(1, activation='sigmoid')(fe)
  # class label output
  out2 = Dense(n_classes, activation='softmax')(fe)
  # define model
  model = Model(in_image, [out1, out2])
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'sparse_categorical_crossentropy'],
      optimizer=opt)
  return model

# define the discriminator model
model = define_discriminator()
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='discriminator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 19.8: Example of defining and summarizing the discriminator model.

The model summary was left out for brevity. A plot of the model is created, showing the linear processing of the input image and the two clear output layers.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

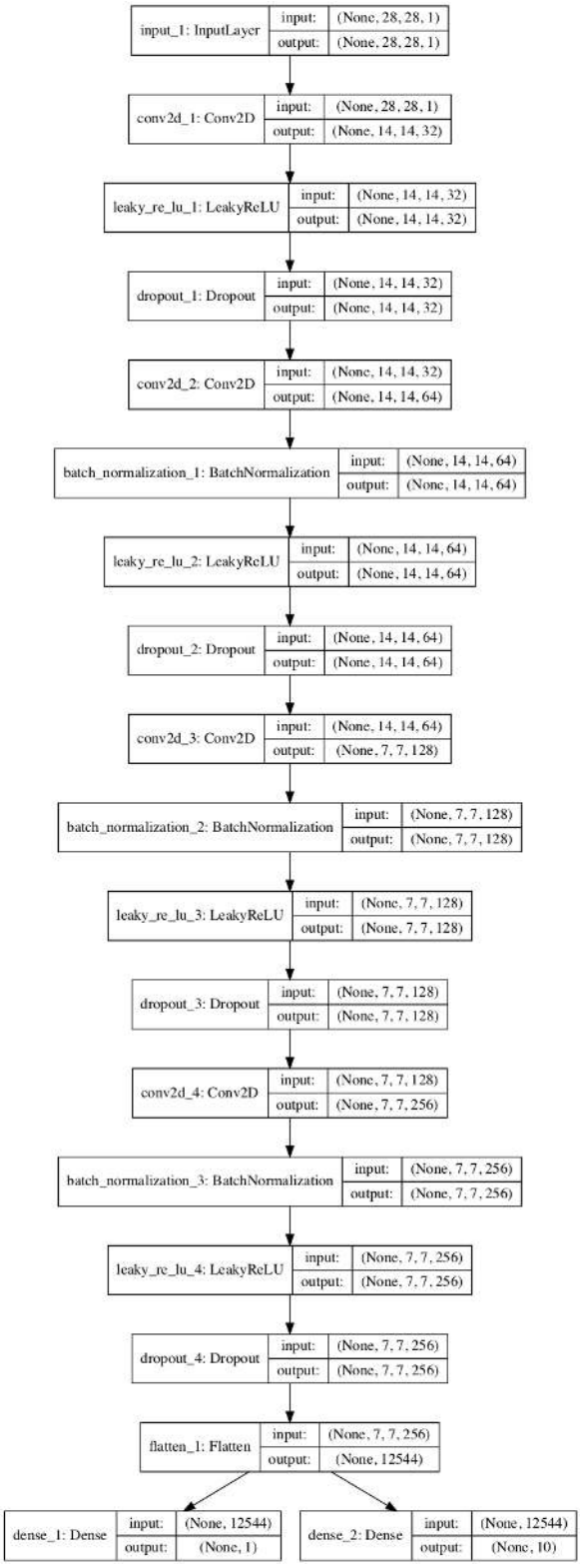

Figure 19.3: Plot of the Discriminator Model for the Auxiliary Classifier GAN.

Now that we have defined our AC-GAN discriminator model, we can develop the generator model.

19.4.2 AC-GAN Generator Model

The generator model must take a random point from the latent space as input, and the class label, then output a generated grayscale image with the shape $28 \times 28 \times 1$. The AC-GAN paper describes the AC-GAN generator model taking a vector input that is a concatenation of the point in latent space (100 dimensions) and the one hot encoded class label (10 dimensions) that is 110 dimensions. An alternative approach that has proven effective and is now generally recommended is to interpret the class label as an additional channel or feature map early in the generator model. This can be achieved by using a learned embedding with an arbitrary number of dimensions (e.g. 50), the output of which can be interpreted by a fully connected layer with a linear activation resulting in one additional 7×7 feature map.

```
...
# label input
in_label = Input(shape=(1,))
# embedding for categorical input
li = Embedding(n_classes, 50)(in_label)
# linear multiplication
n_nodes = 7 * 7
li = Dense(n_nodes, kernel_initializer=init)(li)
# reshape to additional channel
li = Reshape((7, 7, 1))(li)
```

Listing 19.9: Example of defining the first input for the generator model.

The point in latent space can be interpreted by a fully connected layer with sufficient activations to create multiple 7×7 feature maps, in this case, 384, and provide the basis for a low-resolution version of our output image. The 7×7 single feature map interpretation of the class label can then be channel-wise concatenated, resulting in 385 feature maps.

```
...
# image generator input
in_lat = Input(shape=(latent_dim,))
# foundation for 7x7 image
n_nodes = 384 * 7 * 7
gen = Dense(n_nodes, kernel_initializer=init)(in_lat)
gen = Activation('relu')(gen)
gen = Reshape((7, 7, 384))(gen)
# merge image gen and label input
merge = Concatenate()([gen, li])
```

Listing 19.10: Example of defining the second input for the generator model.

These feature maps can then go through the process of two transpose convolutional layers to upsample the 7×7 feature maps first to 14×14 pixels, and then finally to 28×28 features, quadrupling the area of the feature maps with each upsampling step. The output of the generator is a single feature map or grayscale image with the shape 28×28 and pixel values in the range [-1, 1] given the choice of a Tanh activation function. We use ReLU activation for the upsampling layers instead of `LeakyReLU` given the suggestion in the AC-GAN paper.

```
# upsample to 14x14
gen = Conv2DTranspose(192, (5,5), strides=(2,2), padding='same',
    kernel_initializer=init)(merge)
gen = BatchNormalization()(gen)
gen = Activation('relu')(gen)
```

```
# upsample to 28x28
gen = Conv2DTranspose(1, (5,5), strides=(2,2), padding='same', kernel_initializer=init)(gen)
out_layer = Activation('tanh')(gen)
```

Listing 19.11: Example of defining the body of the generator model.

We can tie all of this together and into the `define_generator()` function defined below that will create and return the generator model for the AC-GAN. The model is intentionally not compiled as it is not trained directly; instead, it is trained via the discriminator model.

```
# define the standalone generator model
def define_generator(latent_dim, n_classes=10):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # label input
  in_label = Input(shape=(1,))
  # embedding for categorical input
  li = Embedding(n_classes, 50)(in_label)
  # linear multiplication
  n_nodes = 7 * 7
  li = Dense(n_nodes, kernel_initializer=init)(li)
  # reshape to additional channel
  li = Reshape((7, 7, 1))(li)
  # image generator input
  in_lat = Input(shape=(latent_dim,))
  # foundation for 7x7 image
  n_nodes = 384 * 7 * 7
  gen = Dense(n_nodes, kernel_initializer=init)(in_lat)
  gen = Activation('relu')(gen)
  gen = Reshape((7, 7, 384))(gen)
  # merge image gen and label input
  merge = Concatenate()([gen, li])
  # upsample to 14x14
  gen = Conv2DTranspose(192, (5,5), strides=(2,2), padding='same',
      kernel_initializer=init)(merge)
  gen = BatchNormalization()(gen)
  gen = Activation('relu')(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(1, (5,5), strides=(2,2), padding='same',
      kernel_initializer=init)(gen)
  out_layer = Activation('tanh')(gen)
  # define model
  model = Model([in_lat, in_label], out_layer)
  return model
```

Listing 19.12: Example of a function for defining the generator model.

We can create this model and summarize and plot its structure. The complete example is listed below.

```
# example of defining the generator model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2DTranspose
from keras.layers import Embedding
```

```
from keras.layers import Concatenate
from keras.layers import Activation
from keras.layers import BatchNormalization
from keras.initializers import RandomNormal
from keras.utils.vis_utils import plot_model

# define the standalone generator model
def define_generator(latent_dim, n_classes=10):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # label input
  in_label = Input(shape=(1,))
  # embedding for categorical input
  li = Embedding(n_classes, 50)(in_label)
  # linear multiplication
  n_nodes = 7 * 7
  li = Dense(n_nodes, kernel_initializer=init)(li)
  # reshape to additional channel
  li = Reshape((7, 7, 1))(li)
  # image generator input
  in_lat = Input(shape=(latent_dim,))
  # foundation for 7x7 image
  n_nodes = 384 * 7 * 7
  gen = Dense(n_nodes, kernel_initializer=init)(in_lat)
  gen = Activation('relu')(gen)
  gen = Reshape((7, 7, 384))(gen)
  # merge image gen and label input
  merge = Concatenate()([gen, li])
  # upsample to 14x14
  gen = Conv2DTranspose(192, (5,5), strides=(2,2), padding='same',
      kernel_initializer=init)(merge)
  gen = BatchNormalization()(gen)
  gen = Activation('relu')(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(1, (5,5), strides=(2,2), padding='same',
      kernel_initializer=init)(gen)
  out_layer = Activation('tanh')(gen)
  # define model
  model = Model([in_lat, in_label], out_layer)
  return model

# define the size of the latent space
latent_dim = 100
# define the generator model
model = define_generator(latent_dim)
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='generator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 19.13: Example of defining and summarizing the generator model.

The model summary was left out for brevity. A plot of the network is created summarizing the input and output shapes for each layer. The plot confirms the two inputs to the network and the correct concatenation of the inputs.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

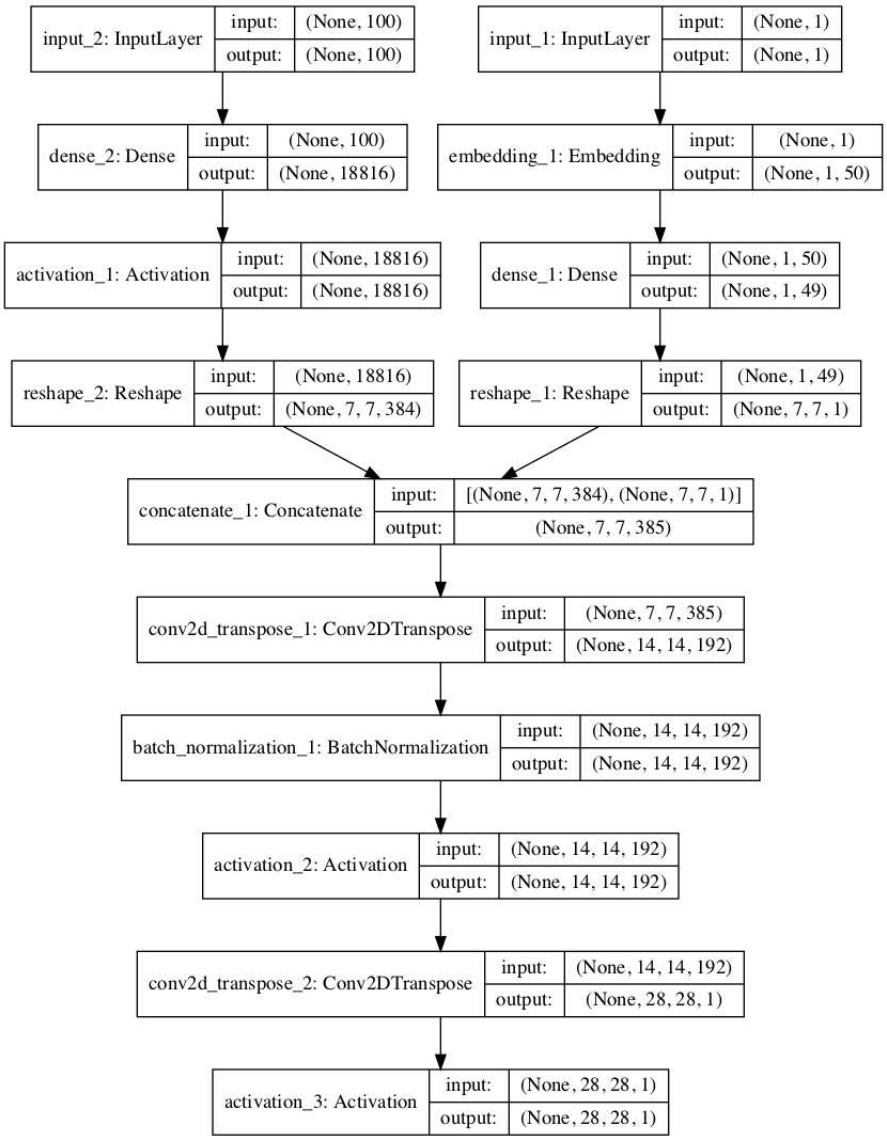

Figure 19.4: Plot of the Generator Model for the Auxiliary Classifier GAN.

Now that we have defined the generator model, we can show how it might be fit.

19.4.3 AC-GAN Composite Model

The generator model is not updated directly; instead, it is updated via the discriminator model. This can be achieved by creating a composite model that stacks the generator model on top of the discriminator model. The input to this composite model is the input to the generator model, namely a random point from the latent space and a class label. The generator model is connected directly to the discriminator model, which takes the generated image directly as

input. Finally, the discriminator model predicts both the realness of the generated image and the class label. As such, the composite model is optimized using two loss functions, one for each output of the discriminator model.

The discriminator model is updated in a standalone manner using real and fake examples, and we will review how to do this in the next section. Therefore, we do not want to update the discriminator model when updating (training) the composite model; we only want to use this composite model to update the weights of the generator model. This can be achieved by setting the layers of the discriminator as not trainable prior to compiling the composite model. This only has an effect on the layer weights when viewed or used by the composite model and prevents them from being updated when the composite model is updated. The `define_gan()` function below implements this, taking the already defined generator and discriminator models as input and defining a new composite model that can be used to update the generator model only.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  for layer in d_model.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # connect the outputs of the generator to the inputs of the discriminator
  gan_output = d_model(g_model.output)
  # define gan model as taking noise and label and outputting real/fake and label outputs
  model = Model(g_model.input, gan_output)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'sparse_categorical_crossentropy'],
      optimizer=opt)
  return model
```

Listing 19.14: Example of a function for defining the composite model for training the generator.

Now that we have defined the models used in the AC-GAN, we can fit them on the Fashion-MNIST dataset.

19.5 How to Develop an AC-GAN for Fashion-MNIST

The first step is to load and prepare the Fashion-MNIST dataset. We only require the images in the training dataset. The images are black and white, therefore we must add an additional channel dimension to transform them to be three dimensional, as expected by the convolutional layers of our models. Finally, the pixel values must be scaled to the range [-1,1] to match the output of the generator model. The `load_real_samples()` function below implements this, returning the loaded and scaled Fashion-MNIST training dataset ready for modeling.

```
# load images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
```

```
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  print(X.shape, trainy.shape)
  return [X, trainy]
```

Listing 19.15: Example of a function for loading and preparing the Fashion-MNIST dataset.

We will require one batch (or a half batch) of real images from the dataset each update to the GAN model. A simple way to achieve this is to select a random sample of images from the dataset each time. The `generate_real_samples()` function below implements this, taking the prepared dataset as an argument, selecting and returning a random sample of Fashion-MNIST images and clothing class labels. The `dataset` argument provided to the function is a list comprised of the images and class labels as returned from the `load_real_samples()` function. The function also returns their corresponding class label for the discriminator, specifically $class = 1$ indicating that they are real images.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # split into images and labels
  images, labels = dataset
  # choose random instances
  ix = randint(0, images.shape[0], n_samples)
  # select images and labels
  X, labels = images[ix], labels[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return [X, labels], y
```

Listing 19.16: Example of a function for creating a sample of real images.

Next, we need inputs for the generator model. These are random points from the latent space, specifically Gaussian distributed random variables. The `generate_latent_points()` function implements this, taking the size of the latent space as an argument and the number of points required, and returning them as a batch of input samples for the generator model. The function also returns randomly selected integers in [0,9] inclusively for the 10 class labels in the Fashion-MNIST dataset.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples, n_classes=10):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  # generate labels
  labels = randint(0, n_classes, n_samples)
  return [z_input, labels]
```

Listing 19.17: Example of a function for sampling points in the latent space and classes.

Next, we need to use the points in the latent space and clothing class labels as input to the generator in order to generate new images. The `generate_fake_samples()` function below implements this, taking the generator model and size of the latent space as arguments, then generating points in the latent space and using them as input to the generator model. The function returns the generated images, their corresponding clothing class label, and their discriminator class label, specifically $class = 0$ to indicate they are fake or generated.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  z_input, labels_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  images = generator.predict([z_input, labels_input])
  # create class labels
  y = zeros((n_samples, 1))
  return [images, labels_input], y
```

Listing 19.18: Example of a function for generating synthetic images using the generator model.

There are no agreed-upon ways to determine when to stop training a GAN; instead, images can be subjectively inspected in order to choose a final model. Therefore, we can periodically generate a sample of images using the generator model and save the generator model to file for later use. The `summarize_performance()` function below implements this, generating 100 images, plotting them, and saving the plot and the generator to file with a filename that includes the training step number.

```
# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, latent_dim, n_samples=100):
  # prepare fake examples
  [X, _], _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(100):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename1 = 'generated_plot_%04d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # save the generator model
  filename2 = 'model_%04d.h5' % (step+1)
  g_model.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))
```

Listing 19.19: Example of a function for summarizing model performance and saving the generator.

We are now ready to fit the GAN models. The model is fit for 100 training epochs, which is arbitrary, as the model begins generating plausible items of clothing after perhaps 20 epochs. A batch size of 64 samples is used, and each training epoch involves $\frac{60000}{64}$, or about 937, batches of real and fake samples and updates to the model. The `summarize_performance()` function is called every 10 epochs, or every (937×10) 9,370 training steps. For a given training step, first the discriminator model is updated for a half batch of real samples, then a half batch of fake samples, together forming one batch of weight updates. The generator is then updated via the combined GAN model. Importantly, the class label is set to 1, or real, for the fake samples.

This has the effect of updating the generator toward getting better at generating real samples on the next batch.

The discriminator and composite model return three loss values from the call to the `train_on_batch()` function. The first value is the sum of the loss values and can be ignored, whereas the second value is the loss for the real/fake output layer and the third value is the loss for the clothing label classification. The `train()` function below implements this, taking the defined models, dataset, and size of the latent dimension as arguments and parameterizing the number of epochs and batch size with default arguments. The generator model is saved at the end of training.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=64):
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset[0].shape[0] / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_steps):
    # get randomly selected 'real' samples
    [X_real, labels_real], y_real = generate_real_samples(dataset, half_batch)
    # update discriminator model weights
    _,d_r1,d_r2 = d_model.train_on_batch(X_real, [y_real, labels_real])
    # generate 'fake' examples
    [X_fake, labels_fake], y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator model weights
    _,d_f,d_f2 = d_model.train_on_batch(X_fake, [y_fake, labels_fake])
    # prepare points in latent space as input for the generator
    [z_input, z_labels] = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    _,g_1,g_2 = gan_model.train_on_batch([z_input, z_labels], [y_gan, z_labels])
    # summarize loss on this batch
    print('>%d, dr[%.3f,%.3f], df[%.3f,%.3f], g[%.3f,%.3f]' % (i+1, d_r1,d_r2, d_f,d_f2,
        g_1,g_2))
    # evaluate the model performance every 'epoch'
    if (i+1) % (bat_per_epo * 10) == 0:
      summarize_performance(i, g_model, latent_dim)
```

Listing 19.20: Example of a function for training the AC-GAN models.

We can then define the size of the latent space, define all three models, and train them on the loaded Fashion-MNIST dataset.

```
...
# size of the latent space
latent_dim = 100
# create the discriminator
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
```

```
# load image data
dataset = load_real_samples()
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Listing 19.21: Example of configuring and starting the training process.

Tying all of this together, the complete example is listed below.

```
# example of fitting an auxiliary classifier gan (ac-gan) on fashion mnsit
from numpy import zeros
from numpy import ones
from numpy import expand_dims
from numpy.random import randn
from numpy.random import randint
from keras.datasets.fashion_mnist import load_data
from keras.optimizers import Adam
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import BatchNormalization
from keras.layers import Dropout
from keras.layers import Embedding
from keras.layers import Activation
from keras.layers import Concatenate
from keras.initializers import RandomNormal
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1), n_classes=10):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=in_shape)
  # downsample to 14x14
  fe = Conv2D(32, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # normal
  fe = Conv2D(64, (3,3), padding='same', kernel_initializer=init)(fe)
  fe = BatchNormalization()(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # downsample to 7x7
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(fe)
  fe = BatchNormalization()(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  fe = Dropout(0.5)(fe)
  # normal
  fe = Conv2D(256, (3,3), padding='same', kernel_initializer=init)(fe)
  fe = BatchNormalization()(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
```

```
  fe = Dropout(0.5)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # real/fake output
  out1 = Dense(1, activation='sigmoid')(fe)
  # class label output
  out2 = Dense(n_classes, activation='softmax')(fe)
  # define model
  model = Model(in_image, [out1, out2])
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'sparse_categorical_crossentropy'],
      optimizer=opt)
  return model

# define the standalone generator model
def define_generator(latent_dim, n_classes=10):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # label input
  in_label = Input(shape=(1,))
  # embedding for categorical input
  li = Embedding(n_classes, 50)(in_label)
  # linear multiplication
  n_nodes = 7 * 7
  li = Dense(n_nodes, kernel_initializer=init)(li)
  # reshape to additional channel
  li = Reshape((7, 7, 1))(li)
  # image generator input
  in_lat = Input(shape=(latent_dim,))
  # foundation for 7x7 image
  n_nodes = 384 * 7 * 7
  gen = Dense(n_nodes, kernel_initializer=init)(in_lat)
  gen = Activation('relu')(gen)
  gen = Reshape((7, 7, 384))(gen)
  # merge image gen and label input
  merge = Concatenate()([gen, li])
  # upsample to 14x14
  gen = Conv2DTranspose(192, (5,5), strides=(2,2), padding='same',
      kernel_initializer=init)(merge)
  gen = BatchNormalization()(gen)
  gen = Activation('relu')(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(1, (5,5), strides=(2,2), padding='same',
      kernel_initializer=init)(gen)
  out_layer = Activation('tanh')(gen)
  # define model
  model = Model([in_lat, in_label], out_layer)
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  for layer in d_model.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
```

```
  # connect the outputs of the generator to the inputs of the discriminator
  gan_output = d_model(g_model.output)
  # define gan model as taking noise and label and outputting real/fake and label outputs
  model = Model(g_model.input, gan_output)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'sparse_categorical_crossentropy'],
      optimizer=opt)
  return model

# load images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  print(X.shape, trainy.shape)
  return [X, trainy]

# select real samples
def generate_real_samples(dataset, n_samples):
  # split into images and labels
  images, labels = dataset
  # choose random instances
  ix = randint(0, images.shape[0], n_samples)
  # select images and labels
  X, labels = images[ix], labels[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return [X, labels], y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples, n_classes=10):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  # generate labels
  labels = randint(0, n_classes, n_samples)
  return [z_input, labels]

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  z_input, labels_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  images = generator.predict([z_input, labels_input])
  # create class labels
  y = zeros((n_samples, 1))
  return [images, labels_input], y

# generate samples and save as a plot and save the model
```

```
def summarize_performance(step, g_model, latent_dim, n_samples=100):
  # prepare fake examples
  [X, _], _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(100):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename1 = 'generated_plot_%04d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # save the generator model
  filename2 = 'model_%04d.h5' % (step+1)
  g_model.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=64):
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset[0].shape[0] / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_steps):
    # get randomly selected 'real' samples
    [X_real, labels_real], y_real = generate_real_samples(dataset, half_batch)
    # update discriminator model weights
    _,d_r1,d_r2 = d_model.train_on_batch(X_real, [y_real, labels_real])
    # generate 'fake' examples
    [X_fake, labels_fake], y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    # update discriminator model weights
    _,d_f,d_f2 = d_model.train_on_batch(X_fake, [y_fake, labels_fake])
    # prepare points in latent space as input for the generator
    [z_input, z_labels] = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    _,g_1,g_2 = gan_model.train_on_batch([z_input, z_labels], [y_gan, z_labels])
    # summarize loss on this batch
    print('>%d, dr[%.3f,%.3f], df[%.3f,%.3f], g[%.3f,%.3f]' % (i+1, d_r1,d_r2, d_f,d_f2,
        g_1,g_2))
    # evaluate the model performance every 'epoch'
    if (i+1) % (bat_per_epo * 10) == 0:
      summarize_performance(i, g_model, latent_dim)

# size of the latent space
latent_dim = 100
# create the discriminator
```

```
discriminator = define_discriminator()
# create the generator
generator = define_generator(latent_dim)
# create the gan
gan_model = define_gan(generator, discriminator)
# load image data
dataset = load_real_samples()
# train model
train(generator, discriminator, gan_model, dataset, latent_dim)
```

Listing 19.22: Example of training the AC-GAN model on the Fashion-MNIST dataset.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

The loss is reported each training iteration, including the real/fake and class loss for the discriminator on real examples (dr), the discriminator on fake examples (df), and the generator updated via the composite model when generating images (g).

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
>1, dr[0.934,2.967], df[1.310,3.006], g[0.878,3.368]
>2, dr[0.711,2.836], df[0.939,3.262], g[0.947,2.751]
>3, dr[0.649,2.980], df[1.001,3.147], g[0.844,3.226]
>4, dr[0.732,3.435], df[0.823,3.715], g[1.048,3.292]
>5, dr[0.860,3.076], df[0.591,2.799], g[1.123,3.313]
...
```

Listing 19.23: Example output from training the AC-GAN model on the Fashion-MNIST dataset.

A total of 10 sample images are generated and 10 models saved over the run. Plots of generated clothing after 10 iterations already look plausible.

Figure 19.5: Example of AC-GAN Generated Items of Clothing after 10 Epochs.

The images remain reliable throughout the training process.

Figure 19.6: Example of AC-GAN Generated Items of Clothing After 100 Epochs.

19.6 How to Generate Items of Clothing With the AC-GAN

In this section, we can load a saved model and use it to generate new items of clothing that plausibly could have come from the Fashion-MNIST dataset. The AC-GAN technically does not conditionally generate images based on the class label, at least not in the same way as the conditional GAN.

> AC-GANs learn a representation for z that is independent of class label.
>
> — *Conditional Image Synthesis With Auxiliary Classifier GANs*, 2016.

Nevertheless, if used in this way, the generated images mostly match the class label. The example below loads the model from the end of the run (any saved model would do), and generates 100 examples of class 7 (sneaker).

```
# example of loading the generator model and generating images
from math import sqrt
from numpy import asarray
from numpy.random import randn
```

```
from keras.models import load_model
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples, n_class):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  # generate labels
  labels = asarray([n_class for _ in range(n_samples)])
  return [z_input, labels]

# create and save a plot of generated images
def save_plot(examples, n_examples):
  # plot images
  for i in range(n_examples):
    # define subplot
    pyplot.subplot(sqrt(n_examples), sqrt(n_examples), 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  pyplot.show()

# load model
model = load_model('model_93700.h5')
latent_dim = 100
n_examples = 100 # must be a square
n_class = 7 # sneaker
# generate images
latent_points, labels = generate_latent_points(latent_dim, n_examples, n_class)
# generate images
X  = model.predict([latent_points, labels])
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
# plot the result
save_plot(X, n_examples)
```

Listing 19.24: Example of using the saved AC-GAN generator to create images.

Running the example, in this case, generates 100 very plausible photos of sneakers.

Figure 19.7: Example of 100 Photos of Sneakers Generated by an AC-GAN.

It may be fun to experiment with other class values. For example, below are 100 generated coats (`n_class`=4). Most of the images are coats, although there are a few pants in there, showing that the latent space is partially, but not completely, class-conditional.

Figure 19.8: Example of 100 Photos of Coats Generated by an AC-GAN.

19.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Generate Images**. Generate images for each clothing class and compare results across different saved models (e.g. epoch 10, 20, etc.).
- **Alternate Configuration**. Update the configuration of the generator, discriminator, or both models to have more or less capacity and compare results.
- **CIFAR-10 Dataset**. Update the example to train on the CIFAR-10 dataset and use model configuration described in the appendix of the paper.

If you explore any of these extensions, I'd love to know.

19.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

19.8.1 Papers

- Conditional Image Synthesis With Auxiliary Classifier GANs, 2016.
  https://arxiv.org/abs/1610.09585
- Conditional Image Synthesis With Auxiliary Classifier GANs, Reviewer Comments.
  https://openreview.net/forum?id=rJXTf9Bxg
- Conditional Image Synthesis with Auxiliary Classifier GANs, NIPS 2016, YouTube.
  https://www.youtube.com/watch?v=myP2TN0_MaE

19.8.2 API

- Keras Datasets API..
  https://keras.io/datasets/
- Keras Sequential Model API.
  https://keras.io/models/sequential/
- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/
- How can I "freeze" Keras layers?.
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers
- MatplotLib API.
  https://matplotlib.org/api/
- NumPy Random sampling (numpy.random) API.
  https://docs.scipy.org/doc/numpy/reference/routines.random.html
- NumPy Array manipulation routines.
  https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html

19.8.3 Articles

- How to Train a GAN? Tips and tricks to make GANs work.
  https://github.com/soumith/ganhacks
- Fashion-MNIST Project, GitHub.
  https://github.com/zalandoresearch/fashion-mnist

19.9 Summary

In this tutorial, you discovered how to develop an auxiliary classifier generative adversarial network for generating photographs of clothing. Specifically, you learned:

- The auxiliary classifier GAN is a type of conditional GAN that requires that the discriminator predict the class label of a given image.

- How to develop generator, discriminator, and composite models for the AC-GAN.
- How to train, evaluate, and use an AC-GAN to generate photographs of clothing from the Fashion-MNIST dataset.

19.9.1 Next

In the next tutorial, you will discover the semi-supervised GAN that trains a classifier in concert with the GAN model that can achieve good results on very limited training data.

Chapter 20

How to Develop a Semi-Supervised GAN (SGAN)

Semi-supervised learning is the challenging problem of training a classifier in a dataset that contains a small number of labeled examples and a much larger number of unlabeled examples. The Generative Adversarial Network, or GAN, is an architecture that makes effective use of large, unlabeled datasets to train an image generator model via an image discriminator model. The discriminator model can be used as a starting point for developing a classifier model in some cases.

The semi-supervised GAN, or SGAN, model is an extension of the GAN architecture that involves the simultaneous training of a supervised discriminator, unsupervised discriminator, and a generator model. The result is both a supervised classification model that generalizes well to unseen examples and a generator model that outputs plausible examples of images from the domain. In this tutorial, you will discover how to develop a Semi-Supervised Generative Adversarial Network from scratch. After completing this tutorial, you will know:

- The semi-supervised GAN is an extension of the GAN architecture for training a classifier model while making use of labeled and unlabeled data.
- There are at least three approaches to implementing the supervised and unsupervised discriminator models in Keras used in the semi-supervised GAN.
- How to train a semi-supervised GAN from scratch on MNIST and load and use the trained classifier for making predictions.

Let's get started.

20.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. What Is the Semi-Supervised GAN?
2. How to Implement the Semi-Supervised Discriminator
3. How to Develop a Semi-Supervised GAN for MNIST
4. How to Use the Final SGAN Classifier Model

20.2 What Is the Semi-Supervised GAN?

Semi-supervised learning refers to a problem where a predictive model is required and there are few labeled examples and many unlabeled examples. The most common example is a classification predictive modeling problem in which there may be a very large dataset of examples, but only a small fraction have target labels. The model must learn from the small set of labeled examples and somehow harness the larger dataset of unlabeled examples in order to generalize to classifying new examples in the future. The Semi-Supervised GAN, or sometimes SGAN for short, is an extension of the Generative Adversarial Network architecture for addressing semi-supervised learning problems.

> One of the primary goals of this work is to improve the effectiveness of generative adversarial networks for semi-supervised learning (improving the performance of a supervised task, in this case, classification, by learning on additional unlabeled examples).
>
> — *Improved Techniques for Training GANs*, 2016.

The discriminator in a traditional GAN is trained to predict whether a given image is real (from the dataset) or fake (generated), allowing it to learn features from unlabeled images. The discriminator can then be used via transfer learning as a starting point when developing a classifier for the same dataset, allowing the supervised prediction task to benefit from the unsupervised training of the GAN. In the Semi-Supervised GAN, the discriminator model is updated to predict $K + 1$ classes, where K is the number of classes in the prediction problem and the additional class label is added for a new *fake* class. It involves directly training the discriminator model for both the unsupervised GAN task and the supervised classification task simultaneously.

> We train a generative model G and a discriminator D on a dataset with inputs belonging to one of N classes. At training time, D is made to predict which of $N + 1$ classes the input belongs to, where an extra class is added to correspond to the outputs of G.
>
> — *Semi-Supervised Learning with Generative Adversarial Networks*, 2016.

As such, the discriminator is trained in two modes: a supervised and unsupervised mode.

- **Unsupervised Training**: In the unsupervised mode, the discriminator is trained in the same way as the traditional GAN, to predict whether the example is either real or fake.
- **Supervised Training**: In the supervised mode, the discriminator is trained to predict the class label of real examples.

Training in unsupervised mode allows the model to learn useful feature extraction capabilities from a large unlabeled dataset, whereas training in supervised mode allows the model to use the extracted features and apply class labels. The result is a classifier model that can achieve state-of-the-art results on standard problems such as MNIST when trained on very few labeled examples, such as tens, hundreds, or one thousand. Additionally, the training process can also

result in better quality images output by the generator model. For example, Augustus Odena in his 2016 paper titled *Semi-Supervised Learning with Generative Adversarial Networks* shows how a GAN-trained classifier is able to perform as well as or better than a standalone CNN model on the MNIST handwritten digit recognition task when trained with 25, 50, 100, and 1,000 labeled examples.

EXAMPLES	CNN	SGAN
1000	0.965	0.964
100	0.895	0.928
50	0.859	0.883
25	0.750	0.802

Figure 20.1: Example of the Table of Results Comparing Classification Accuracy of a CNN and SGAN on MNIST. Taken from: Semi-Supervised Learning with Generative Adversarial Networks.

Tim Salimans, et al. from OpenAI in their 2016 paper titled *Improved Techniques for Training GANs* achieved at the time state-of-the-art results on a number of image classification tasks using a semi-supervised GAN, including MNIST.

Model	Number of incorrectly predicted test examples for a given number of labeled samples			
	20	50	100	200
DGN [21]			333 ± 14	
Virtual Adversarial [22]			212	
CatGAN [14]			191 ± 10	
Skip Deep Generative Model [23]			132 ± 7	
Ladder network [24]			106 ± 37	
Auxiliary Deep Generative Model [23]			96 ± 2	
Our model	1677 ± 452	221 ± 136	93 ± 6.5	90 ± 4.2
Ensemble of 10 of our models	1134 ± 445	142 ± 96	86 ± 5.6	81 ± 4.3

Figure 20.2: Example of the Table of Results Comparing Classification Accuracy of other GAN models to a SGAN on MNIST. Taken From: Improved Techniques for Training GANs.

20.3 How to Implement the Semi-Supervised Discriminator

There are a number of ways that we can implement the discriminator model for the semi-supervised GAN. In this section, we will review three candidate approaches.

20.3.1 Traditional Discriminator Model

Consider a discriminator model for the standard GAN model. It must take an image as input and predict whether it is real or fake. More specifically, it predicts the likelihood of the input image being real. The output layer uses a sigmoid activation function to predict a probability value in [0,1] and the model is typically optimized using a binary cross-entropy loss function. For example, we can define a simple discriminator model that takes grayscale images as input with the size of 28×28 pixels and predicts a probability of the image being real. We can use best practices and downsample the image using convolutional layers with a 2×2 stride and a leaky ReLU activation function. The `define_discriminator()` function below implements this and defines our standard discriminator model.

```
# example of defining the discriminator model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import Flatten
from keras.optimizers import Adam
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  # image input
  in_image = Input(shape=in_shape)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(in_image)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # dropout
  fe = Dropout(0.4)(fe)
  # output layer
  d_out_layer = Dense(1, activation='sigmoid')(fe)
  # define and compile discriminator model
  d_model = Model(in_image, d_out_layer)
  d_model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002, beta_1=0.5))
  return d_model

# create model
model = define_discriminator()
# plot the model
plot_model(model, to_file='discriminator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 20.1: Example of defining and summarizing the discriminator model.

Running the example creates a plot of the discriminator model, clearly showing the $28 \times 28 \times 1$

shape of the input image and the prediction of a single probability value.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

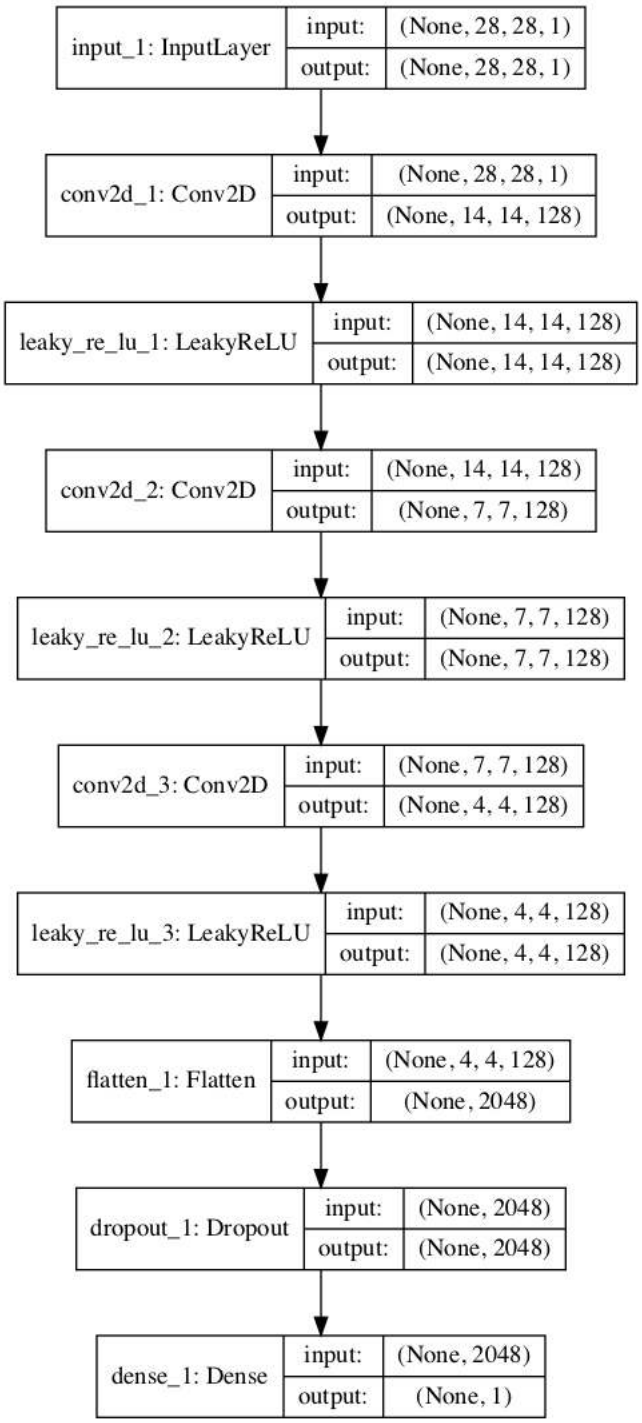

Figure 20.3: Plot of a Standard GAN Discriminator Model.

20.3.2 Separate Discriminator Models With Shared Weights

Starting with the standard GAN discriminator model, we can update it to create two models that share feature extraction weights. Specifically, we can define one classifier model that predicts whether an input image is real or fake, and a second classifier model that predicts the class of a given model.

- **Binary Classifier Model**. Predicts whether the image is real or fake, sigmoid activation function in the output layer, and optimized using the binary cross-entropy loss function.
- **Multiclass Classifier Model**. Predicts the class of the image, softmax activation function in the output layer, and optimized using the categorical cross-entropy loss function.

Both models have different output layers but share all feature extraction layers. This means that updates to one of the classifier models will impact both models. The example below creates the traditional discriminator model with binary output first, then re-uses the feature extraction layers and creates a new multiclass prediction model, in this case with 10 classes.

```
# example of defining semi-supervised discriminator model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import Flatten
from keras.optimizers import Adam
from keras.utils.vis_utils import plot_model

# define the standalone supervised and unsupervised discriminator models
def define_discriminator(in_shape=(28,28,1), n_classes=10):
  # image input
  in_image = Input(shape=in_shape)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(in_image)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # dropout
  fe = Dropout(0.4)(fe)
  # unsupervised output
  d_out_layer = Dense(1, activation='sigmoid')(fe)
  # define and compile unsupervised discriminator model
  d_model = Model(in_image, d_out_layer)
  d_model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002, beta_1=0.5))
  # supervised output
  c_out_layer = Dense(n_classes, activation='softmax')(fe)
  # define and compile supervised discriminator model
```

```
  c_model = Model(in_image, c_out_layer)
  c_model.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(lr=0.0002,
      beta_1=0.5), metrics=['accuracy'])
  return d_model, c_model

# create model
d_model, c_model = define_discriminator()
# plot the model
plot_model(d_model, to_file='discriminator1_plot.png', show_shapes=True,
    show_layer_names=True)
plot_model(c_model, to_file='discriminator2_plot.png', show_shapes=True,
    show_layer_names=True)
```

Listing 20.2: Example of defining and summarizing two separate discriminator models.

Running the example creates and plots both models. The plot for the first model is the same as before. The plot of the second model shows the same expected input shape and same feature extraction layers, with a new 10 class classification output layer.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

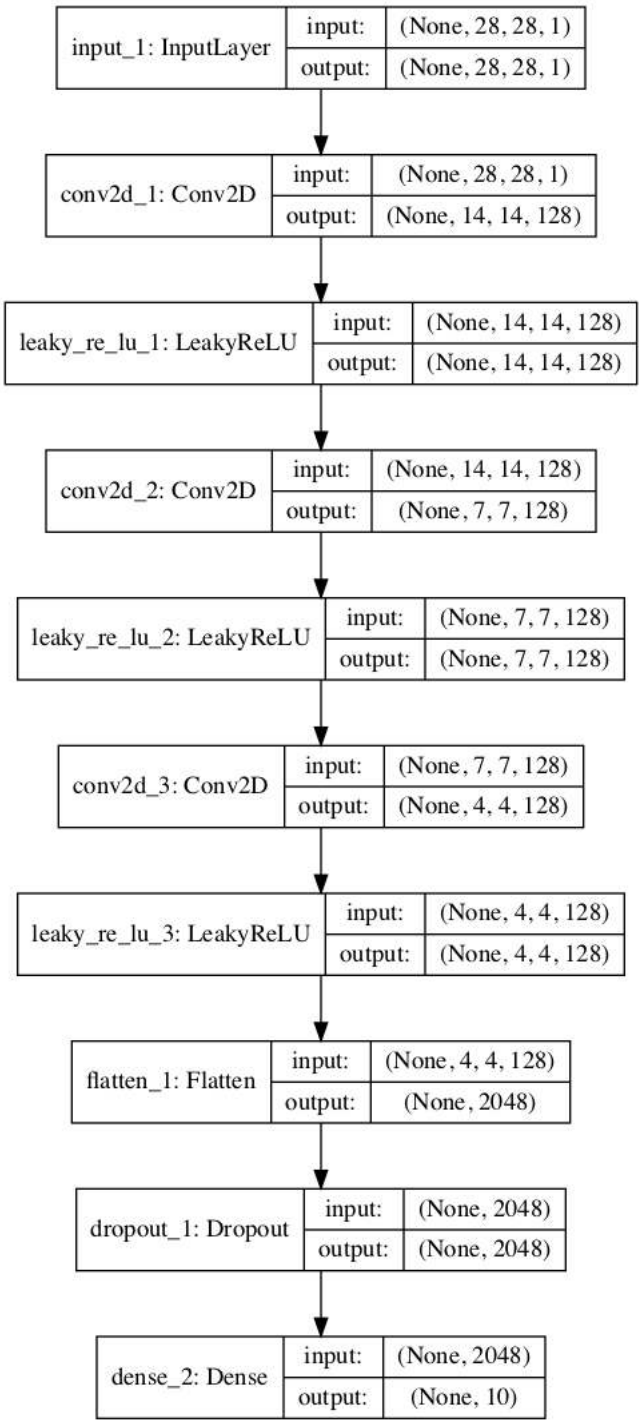

Figure 20.4: Plot of a Supervised Multiclass Classification GAN Discriminator Model.

20.3.3 Single Discriminator Model With Multiple Outputs

Another approach to implementing the semi-supervised discriminator model is to have a single model with multiple output layers. Specifically, this is a single model with one output layer for the unsupervised task and one output layer for the supervised task. This is like having separate models for the supervised and unsupervised tasks in that they both share the same feature

extraction layers, except that in this case, each input image always has two output predictions, specifically a real/fake prediction and a supervised class prediction.

A problem with this approach is that when the model is updated with unlabeled and generated images, there is no supervised class label. In that case, these images must have an output label of *unknown* or *fake* from the supervised output. This means that an additional class label is required for the supervised output layer. The example below implements the multi-output single model approach for the discriminator model in the semi-supervised GAN architecture. We can see that the model is defined with two output layers and that the output layer for the supervised task is defined with `n_classes` + 1, in this case 11, making room for the additional *unknown* class label. We can also see that the model is compiled to two loss functions, one for each output layer of the model.

```
# example of defining semi-supervised discriminator model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import Flatten
from keras.optimizers import Adam
from keras.utils.vis_utils import plot_model

# define the standalone supervised and unsupervised discriminator models
def define_discriminator(in_shape=(28,28,1), n_classes=10):
  # image input
  in_image = Input(shape=in_shape)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(in_image)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # dropout
  fe = Dropout(0.4)(fe)
  # unsupervised output
  d_out_layer = Dense(1, activation='sigmoid')(fe)
  # supervised output
  c_out_layer = Dense(n_classes + 1, activation='softmax')(fe)
  # define and compile supervised discriminator model
  model = Model(in_image, [d_out_layer, c_out_layer])
  model.compile(loss=['binary_crossentropy', 'sparse_categorical_crossentropy'],
      optimizer=Adam(lr=0.0002, beta_1=0.5), metrics=['accuracy'])
  return model

# create model
model = define_discriminator()
# plot the model
plot_model(model, to_file='multioutput_discriminator_plot.png', show_shapes=True,
```

```
    show_layer_names=True)
```

Listing 20.3: Example of defining and summarizing the multiple-output discriminator.

Running the example creates and plots the single multi-output model. The plot clearly shows the shared layers and the separate unsupervised and supervised output layers.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

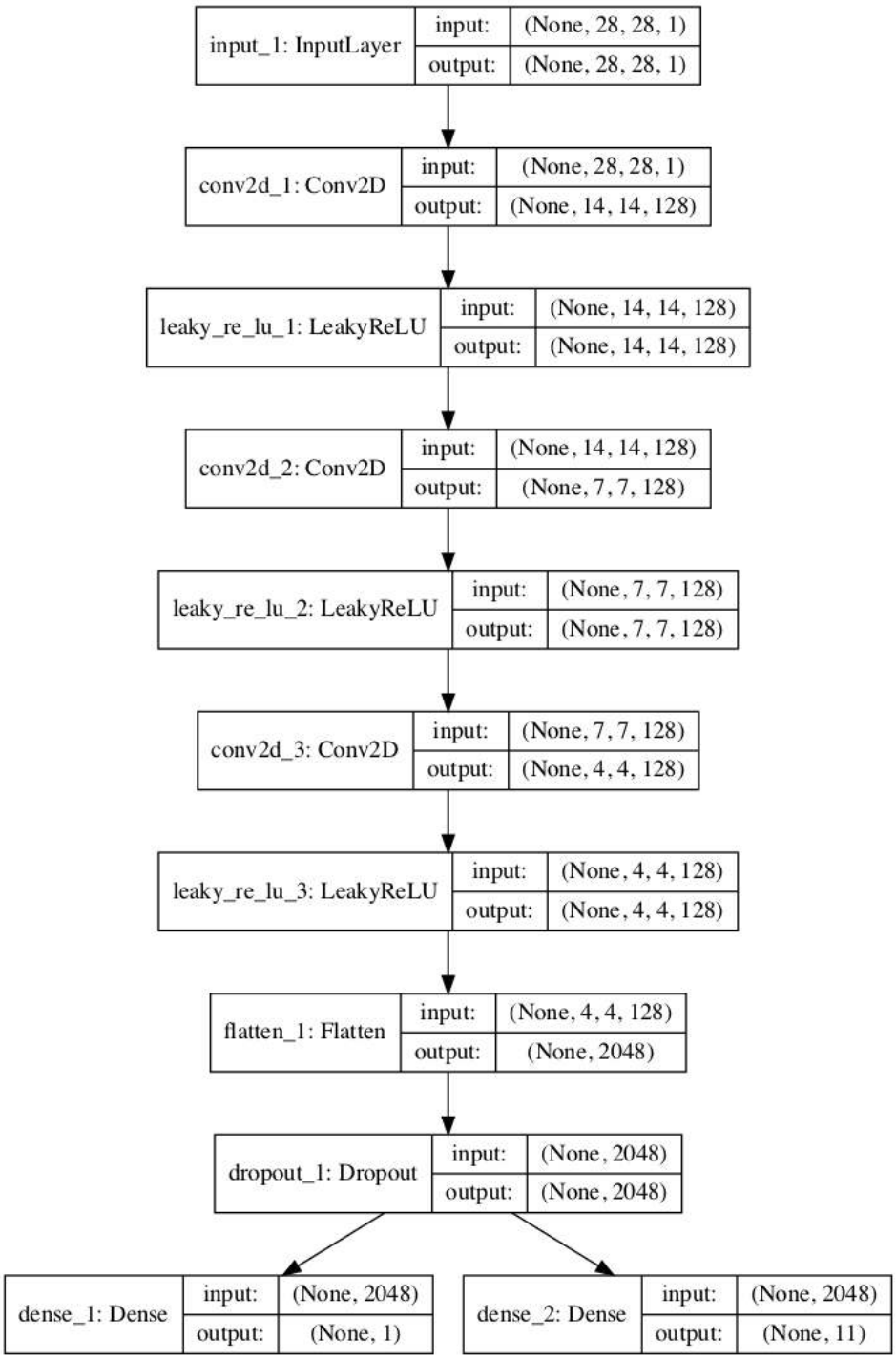

Figure 20.5: Plot of a Semi-Supervised GAN Discriminator Model With Unsupervised and Supervised Output Layers.

20.3.4 Stacked Discriminator Models With Shared Weights

A final approach is very similar to the prior two semi-supervised approaches and involves creating separate logical unsupervised and supervised models but attempts to reuse the output layers of one model to feed as input into another model. The approach is based on the definition of the

semi-supervised model in the 2016 paper by Tim Salimans, et al. from OpenAI titled *Improved Techniques for Training GANs*. In the paper, they describe an efficient implementation, where first the supervised model is created with K output classes and a softmax activation function. The unsupervised model is then defined that takes the output of the supervised model prior to the softmax activation, then calculates a normalized sum of the exponential outputs.

$$D(x) = \frac{Z(x)}{Z(x)+1}, \text{where } Z(x) = \sum_{k=1}^{K} \exp[l_k(x)] \tag{20.1}$$

To make this clearer, we can implement this activation function in NumPy and run some sample activations through it to see what happens. The complete example is listed below.

```
# example of custom activation function
import numpy as np

# custom activation function
def custom_activation(output):
  logexpsum = np.sum(np.exp(output))
  result = logexpsum / (logexpsum + 1.0)
  return result

# all -10s
output = np.asarray([-10.0, -10.0, -10.0])
print(custom_activation(output))
# all -1s
output = np.asarray([-1.0, -1.0, -1.0])
print(custom_activation(output))
# all 0s
output = np.asarray([0.0, 0.0, 0.0])
print(custom_activation(output))
# all 1s
output = np.asarray([1.0, 1.0, 1.0])
print(custom_activation(output))
# all 10s
output = np.asarray([10.0, 10.0, 10.0])
print(custom_activation(output))
```

Listing 20.4: Example of demonstrating the custom activation function.

Remember, the output of the unsupervised model prior to the softmax activation function will be the activations of the nodes directly. They will be small positive or negative values, but not normalized, as this would be performed by the softmax activation. The custom activation function will output a value between 0.0 and 1.0. A value close to 0.0 is output for a small or negative activation and a value close to 1.0 for a positive or large activation. We can see this when we run the example.

```
0.0001361812414310667 4
0.5246331135813284
0.75
0.890768227426964
0.9999848669190928
```

Listing 20.5: Example output from demonstrating the custom activation function.

This means that the model is encouraged to output a strong class prediction for real examples, and a small class prediction or low activation for fake examples. It's a clever trick and allows the re-use of the same output nodes from the supervised model in both models. The activation function can be implemented almost directly via the Keras backend and called from a `Lambda` layer, e.g. a layer that will apply a custom function to the input to the layer.

The complete example is listed below. First, the supervised model is defined with a softmax activation and categorical cross-entropy loss function. The unsupervised model is stacked on top of the output layer of the supervised model before the softmax activation, and the activations of the nodes pass through our custom activation function via the `Lambda` layer. No need for a sigmoid activation function as we have already normalized the activation. As before, the unsupervised model is fit using binary cross-entropy loss.

```
# example of defining semi-supervised discriminator model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers import Activation
from keras.layers import Lambda
from keras.optimizers import Adam
from keras.utils.vis_utils import plot_model
from keras import backend

# custom activation function
def custom_activation(output):
  logexpsum = backend.sum(backend.exp(output), axis=-1, keepdims=True)
  result = logexpsum / (logexpsum + 1.0)
  return result

# define the standalone supervised and unsupervised discriminator models
def define_discriminator(in_shape=(28,28,1), n_classes=10):
  # image input
  in_image = Input(shape=in_shape)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(in_image)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # dropout
  fe = Dropout(0.4)(fe)
  # output layer nodes
  fe = Dense(n_classes)(fe)
  # supervised output
  c_out_layer = Activation('softmax')(fe)
  # define and compile supervised discriminator model
```

```
  c_model = Model(in_image, c_out_layer)
  c_model.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(lr=0.0002,
      beta_1=0.5), metrics=['accuracy'])
  # unsupervised output
  d_out_layer = Lambda(custom_activation)(fe)
  # define and compile unsupervised discriminator model
  d_model = Model(in_image, d_out_layer)
  d_model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002, beta_1=0.5))
  return d_model, c_model

# create model
d_model, c_model = define_discriminator()
# plot the model
plot_model(d_model, to_file='stacked_discriminator1_plot.png', show_shapes=True,
    show_layer_names=True)
plot_model(c_model, to_file='stacked_discriminator2_plot.png', show_shapes=True,
    show_layer_names=True)
```

Listing 20.6: Example of defining and summarizing the discriminator models with the custom activation function.

Running the example creates and plots the two models, which look much the same as the two models in the first example. Stacked version of the unsupervised discriminator model:

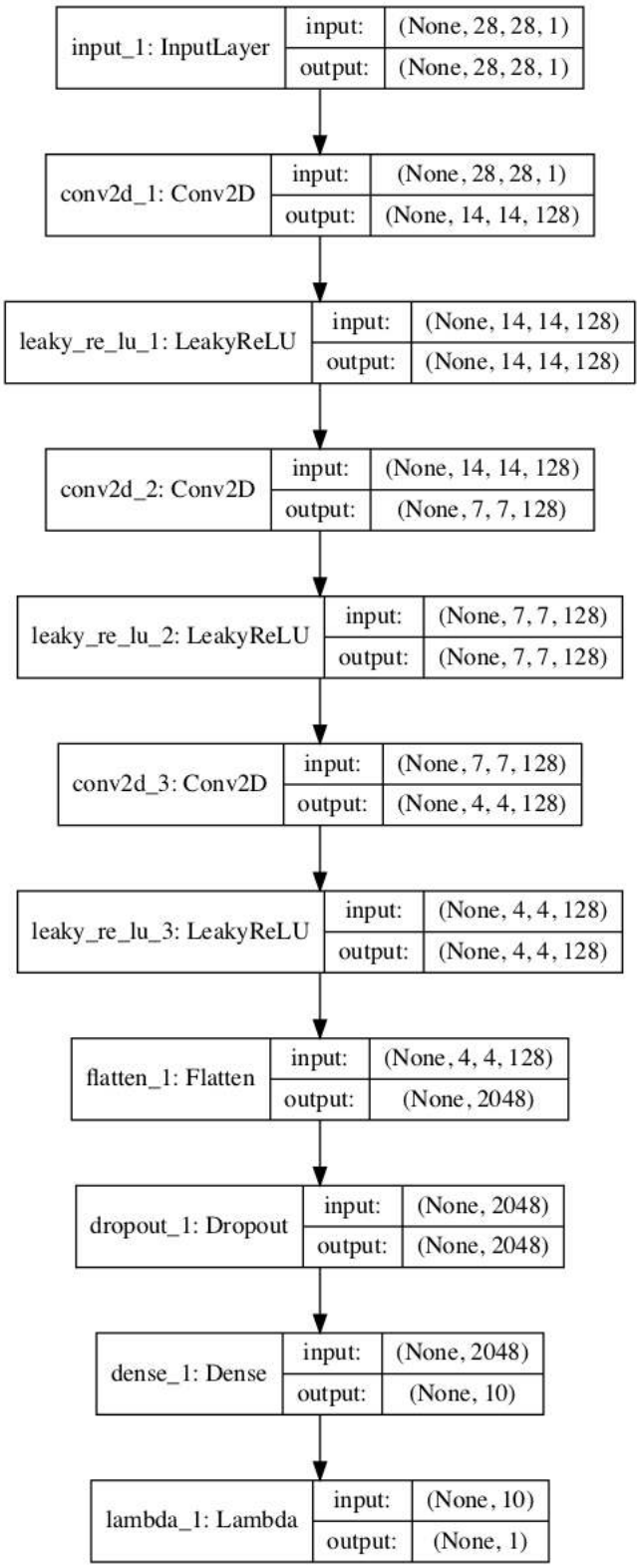

Figure 20.6: Plot of the Stacked Version of the Unsupervised Discriminator Model of the Semi-Supervised GAN.

Stacked version of the supervised discriminator model:

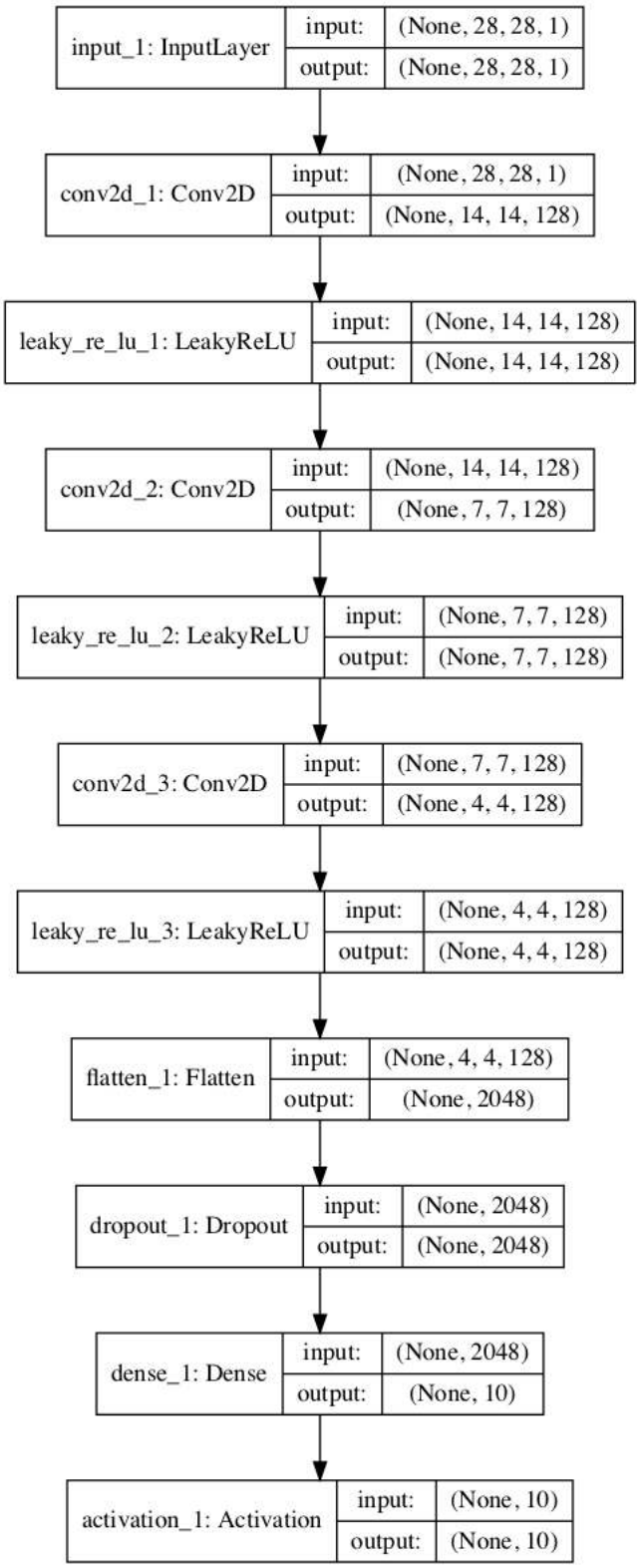

Figure 20.7: Plot of the Stacked Version of the Supervised Discriminator Model of the Semi-Supervised GAN.

Now that we have seen how to implement the discriminator model in the semi-supervised GAN, we can develop a complete example for image generation and semi-supervised classification.

20.4 How to Develop a Semi-Supervised GAN for MNIST

In this section, we will develop a semi-supervised GAN model for the MNIST handwritten digit dataset (described in Section 7.2). The dataset has 10 classes for the digits 0-9, therefore the classifier model will have 10 output nodes. The model will be fit on the training dataset that contains 60,000 examples. Only 100 of the images in the training dataset will be used with labels, 10 from each of the 10 classes. We will start off by defining the models. We will use the stacked discriminator model, exactly as defined in the previous section. Next, we can define the generator model. In this case, the generator model will take as input a point in the latent space and will use transpose convolutional layers to output a 28×28 grayscale image. The `define_generator()` function below implements this and returns the defined generator model.

```
# define the standalone generator model
def define_generator(latent_dim):
  # image generator input
  in_lat = Input(shape=(latent_dim,))
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  gen = Dense(n_nodes)(in_lat)
  gen = LeakyReLU(alpha=0.2)(gen)
  gen = Reshape((7, 7, 128))(gen)
  # upsample to 14x14
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
  gen = LeakyReLU(alpha=0.2)(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
  gen = LeakyReLU(alpha=0.2)(gen)
  # output
  out_layer = Conv2D(1, (7,7), activation='tanh', padding='same')(gen)
  # define model
  model = Model(in_lat, out_layer)
  return model
```

Listing 20.7: Example of a function for defining the generator model.

The generator model will be fit via the unsupervised discriminator model. We will use the composite model architecture, common to training the generator model when implemented in Keras. Specifically, weight sharing is used where the output of the generator model is passed directly to the unsupervised discriminator model, and the weights of the discriminator are marked as not trainable. The `define_gan()` function below implements this, taking the already-defined generator and discriminator models as input and returning the composite model used to train the weights of the generator model.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect image output from generator as input to discriminator
  gan_output = d_model(g_model.output)
  # define gan model as taking noise and outputting a classification
  model = Model(g_model.input, gan_output)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
```

```
  return model
```

Listing 20.8: Example of a function for defining the composite model for updating the generator.

We can load the training dataset and scale the pixels to the range [-1, 1] to match the output values of the generator model.

```
# load the images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  print(X.shape, trainy.shape)
  return [X, trainy]
```

Listing 20.9: Example of a function for loading and preparing the MNIST dataset.

We can also define a function to select a subset of the training dataset in which we keep the labels and train the supervised version of the discriminator model. The `select_supervised_samples()` function below implements this and is careful to ensure that the selection of examples is random and that the classes are balanced. The number of labeled examples is parameterized and set at 100, meaning that each of the 10 classes will have 10 randomly selected examples.

```
# select a supervised subset of the dataset, ensures classes are balanced
def select_supervised_samples(dataset, n_samples=100, n_classes=10):
  X, y = dataset
  X_list, y_list = list(), list()
  n_per_class = int(n_samples / n_classes)
  for i in range(n_classes):
    # get all images for this class
    X_with_class = X[y == i]
    # choose random instances
    ix = randint(0, len(X_with_class), n_per_class)
    # add to list
    [X_list.append(X_with_class[j]) for j in ix]
    [y_list.append(i) for j in ix]
  return asarray(X_list), asarray(y_list)
```

Listing 20.10: Example of a function for selecting supervised samples of real images.

Next, we can define a function for retrieving a batch of real training examples. A sample of images and labels is selected, with replacement. This same function can be used to retrieve examples from the labeled and unlabeled dataset, later when we train the models. In the case of the *unlabeled dataset*, we will ignore the labels.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # split into images and labels
  images, labels = dataset
  # choose random instances
  ix = randint(0, images.shape[0], n_samples)
  # select images and labels
```

```
  X, labels = images[ix], labels[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return [X, labels], y
```

Listing 20.11: Example of a function for selecting unsupervised samples of real images.

Next, we can define functions to help in generating images using the generator model. First, the `generate_latent_points()` function will create a batch worth of random points in the latent space that can be used as input for generating images. The `generate_fake_samples()` function will call this function to generate a batch worth of images that can be fed to the unsupervised discriminator model or the composite GAN model during training.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  z_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = z_input.reshape(n_samples, latent_dim)
  return z_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  z_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  images = generator.predict(z_input)
  # create class labels
  y = zeros((n_samples, 1))
  return images, y
```

Listing 20.12: Example of functions for sampling the latent space and generating synthetic images.

Next, we can define a function to be called when we want to evaluate the performance of the model. This function will generate and plot 100 images using the current state of the generator model. This plot of images can be used to subjectively evaluate the performance of the generator model. The supervised discriminator model is then evaluated on the entire training dataset, and the classification accuracy is reported. Finally, the generator model and the supervised discriminator model are saved to file, to be used later. The `summarize_performance()` function below implements this and can be called periodically, such as the end of every training epoch. The results can be reviewed at the end of the run to select a classifier and even generator models.

```
# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, c_model, latent_dim, dataset, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(100):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
```

```
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename1 = 'generated_plot_%04d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # evaluate the classifier model
  X, y = dataset
  _, acc = c_model.evaluate(X, y, verbose=0)
  print('Classifier Accuracy: %.3f%%' % (acc * 100))
  # save the generator model
  filename2 = 'g_model_%04d.h5' % (step+1)
  g_model.save(filename2)
  # save the classifier model
  filename3 = 'c_model_%04d.h5' % (step+1)
  c_model.save(filename3)
  print('>Saved: %s, %s, and %s' % (filename1, filename2, filename3))
```

Listing 20.13: Example of a function for summarizing model performance and saving models to file.

Next, we can define a function to train the models. The defined models and loaded training dataset are provided as arguments, and the number of training epochs and batch size are parameterized with default values, in this case 20 epochs and a batch size of 100. The chosen model configuration was found to overfit the training dataset quickly, hence the relatively smaller number of training epochs. Increasing the epochs to 100 or more results in much higher-quality generated images, but a lower-quality classifier model. Balancing these two concerns might make a fun extension.

First, the labeled subset of the training dataset is selected, and the number of training steps is calculated. The training process is almost identical to the training of a vanilla GAN model, with the addition of updating the supervised model with labeled examples. A single cycle through updating the models involves first updating the supervised discriminator model with labeled examples, then updating the unsupervised discriminator model with unlabeled real and generated examples. Finally, the generator model is updated via the composite model.

The shared weights of the discriminator model get updated with 1.5 batches worth of samples, whereas the weights of the generator model are updated with one batch worth of samples each iteration. Changing this so that each model is updated by the same amount might improve the model training process.

```
# train the generator and discriminator
def train(g_model, d_model, c_model, gan_model, dataset, latent_dim, n_epochs=20,
    n_batch=100):
  # select supervised dataset
  X_sup, y_sup = select_supervised_samples(dataset)
  print(X_sup.shape, y_sup.shape)
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset[0].shape[0] / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  print('n_epochs=%d, n_batch=%d, 1/2=%d, b/e=%d, steps=%d' % (n_epochs, n_batch,
      half_batch, bat_per_epo, n_steps))
```

```
  # manually enumerate epochs
  for i in range(n_steps):
    # update supervised discriminator (c)
    [Xsup_real, ysup_real], _ = generate_real_samples([X_sup, y_sup], half_batch)
    c_loss, c_acc = c_model.train_on_batch(Xsup_real, ysup_real)
    # update unsupervised discriminator (d)
    [X_real, _], y_real = generate_real_samples(dataset, half_batch)
    d_loss1 = d_model.train_on_batch(X_real, y_real)
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    d_loss2 = d_model.train_on_batch(X_fake, y_fake)
    # update generator (g)
    X_gan, y_gan = generate_latent_points(latent_dim, n_batch), ones((n_batch, 1))
    g_loss = gan_model.train_on_batch(X_gan, y_gan)
    # summarize loss on this batch
    print('>%d, c[%.3f,%.0f], d[%.3f,%.3f], g[%.3f]' % (i+1, c_loss, c_acc*100, d_loss1,
        d_loss2, g_loss))
    # evaluate the model performance every so often
    if (i+1) % (bat_per_epo * 1) == 0:
      summarize_performance(i, g_model, c_model, latent_dim, dataset)
```

Listing 20.14: Example of a function for training the GAN models.

Finally, we can define the models and call the function to train and save the models.

```
...
# size of the latent space
latent_dim = 100
# create the discriminator models
d_model, c_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, c_model, gan_model, dataset, latent_dim)
```

Listing 20.15: Example of configuring and starting the training process.

Tying all of this together, the complete example of training a semi-supervised GAN on the MNIST handwritten digit image classification task is listed below.

```
# example of semi-supervised gan for mnist
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy import asarray
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
```

```
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import Lambda
from keras.layers import Activation
from matplotlib import pyplot
from keras import backend

# custom activation function
def custom_activation(output):
  logexpsum = backend.sum(backend.exp(output), axis=-1, keepdims=True)
  result = logexpsum / (logexpsum + 1.0)
  return result

# define the standalone supervised and unsupervised discriminator models
def define_discriminator(in_shape=(28,28,1), n_classes=10):
  # image input
  in_image = Input(shape=in_shape)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(in_image)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # downsample
  fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
  fe = LeakyReLU(alpha=0.2)(fe)
  # flatten feature maps
  fe = Flatten()(fe)
  # dropout
  fe = Dropout(0.4)(fe)
  # output layer nodes
  fe = Dense(n_classes)(fe)
  # supervised output
  c_out_layer = Activation('softmax')(fe)
  # define and compile supervised discriminator model
  c_model = Model(in_image, c_out_layer)
  c_model.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(lr=0.0002,
      beta_1=0.5), metrics=['accuracy'])
  # unsupervised output
  d_out_layer = Lambda(custom_activation)(fe)
  # define and compile unsupervised discriminator model
  d_model = Model(in_image, d_out_layer)
  d_model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002, beta_1=0.5))
  return d_model, c_model

# define the standalone generator model
def define_generator(latent_dim):
  # image generator input
  in_lat = Input(shape=(latent_dim,))
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  gen = Dense(n_nodes)(in_lat)
  gen = LeakyReLU(alpha=0.2)(gen)
  gen = Reshape((7, 7, 128))(gen)
```

```
  # upsample to 14x14
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
  gen = LeakyReLU(alpha=0.2)(gen)
  # upsample to 28x28
  gen = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
  gen = LeakyReLU(alpha=0.2)(gen)
  # output
  out_layer = Conv2D(1, (7,7), activation='tanh', padding='same')(gen)
  # define model
  model = Model(in_lat, out_layer)
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect image output from generator as input to discriminator
  gan_output = d_model(g_model.output)
  # define gan model as taking noise and outputting a classification
  model = Model(g_model.input, gan_output)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# load the images
def load_real_samples():
  # load dataset
  (trainX, trainy), (_, _) = load_data()
  # expand to 3d, e.g. add channels
  X = expand_dims(trainX, axis=-1)
  # convert from ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [-1,1]
  X = (X - 127.5) / 127.5
  print(X.shape, trainy.shape)
  return [X, trainy]

# select a supervised subset of the dataset, ensures classes are balanced
def select_supervised_samples(dataset, n_samples=100, n_classes=10):
  X, y = dataset
  X_list, y_list = list(), list()
  n_per_class = int(n_samples / n_classes)
  for i in range(n_classes):
    # get all images for this class
    X_with_class = X[y == i]
    # choose random instances
    ix = randint(0, len(X_with_class), n_per_class)
    # add to list
    [X_list.append(X_with_class[j]) for j in ix]
    [y_list.append(i) for j in ix]
  return asarray(X_list), asarray(y_list)

# select real samples
def generate_real_samples(dataset, n_samples):
  # split into images and labels
```

```
  images, labels = dataset
  # choose random instances
  ix = randint(0, images.shape[0], n_samples)
  # select images and labels
  X, labels = images[ix], labels[ix]
  # generate class labels
  y = ones((n_samples, 1))
  return [X, labels], y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  z_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = z_input.reshape(n_samples, latent_dim)
  return z_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  z_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  images = generator.predict(z_input)
  # create class labels
  y = zeros((n_samples, 1))
  return images, y

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, c_model, latent_dim, dataset, n_samples=100):
  # prepare fake examples
  X, _ = generate_fake_samples(g_model, latent_dim, n_samples)
  # scale from [-1,1] to [0,1]
  X = (X + 1) / 2.0
  # plot images
  for i in range(100):
    # define subplot
    pyplot.subplot(10, 10, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename1 = 'generated_plot_%04d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # evaluate the classifier model
  X, y = dataset
  _, acc = c_model.evaluate(X, y, verbose=0)
  print('Classifier Accuracy: %.3f%%' % (acc * 100))
  # save the generator model
  filename2 = 'g_model_%04d.h5' % (step+1)
  g_model.save(filename2)
  # save the classifier model
  filename3 = 'c_model_%04d.h5' % (step+1)
  c_model.save(filename3)
  print('>Saved: %s, %s, and %s' % (filename1, filename2, filename3))
```

```
# train the generator and discriminator
def train(g_model, d_model, c_model, gan_model, dataset, latent_dim, n_epochs=20,
    n_batch=100):
  # select supervised dataset
  X_sup, y_sup = select_supervised_samples(dataset)
  print(X_sup.shape, y_sup.shape)
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset[0].shape[0] / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  print('n_epochs=%d, n_batch=%d, 1/2=%d, b/e=%d, steps=%d' % (n_epochs, n_batch,
      half_batch, bat_per_epo, n_steps))
  # manually enumerate epochs
  for i in range(n_steps):
    # update supervised discriminator (c)
    [Xsup_real, ysup_real], _ = generate_real_samples([X_sup, y_sup], half_batch)
    c_loss, c_acc = c_model.train_on_batch(Xsup_real, ysup_real)
    # update unsupervised discriminator (d)
    [X_real, _], y_real = generate_real_samples(dataset, half_batch)
    d_loss1 = d_model.train_on_batch(X_real, y_real)
    X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
    d_loss2 = d_model.train_on_batch(X_fake, y_fake)
    # update generator (g)
    X_gan, y_gan = generate_latent_points(latent_dim, n_batch), ones((n_batch, 1))
    g_loss = gan_model.train_on_batch(X_gan, y_gan)
    # summarize loss on this batch
    print('>%d, c[%.3f,%.0f], d[%.3f,%.3f], g[%.3f]' % (i+1, c_loss, c_acc*100, d_loss1,
        d_loss2, g_loss))
    # evaluate the model performance every so often
    if (i+1) % (bat_per_epo * 1) == 0:
      summarize_performance(i, g_model, c_model, latent_dim, dataset)

# size of the latent space
latent_dim = 100
# create the discriminator models
d_model, c_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, c_model, gan_model, dataset, latent_dim)
```

Listing 20.16: Example of training the SGAN model on the MNIST dataset.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

At the start of the run, the size of the training dataset is summarized, as is the supervised subset, confirming our configuration. The performance of each model is summarized at the end

of each update, including the loss and accuracy of the supervised discriminator model (c), the loss of the unsupervised discriminator model on real and generated examples (d), and the loss of the generator model updated via the composite model (g). The loss for the supervised model will shrink to a small value close to zero and accuracy will hit 100%, which will be maintained for the entire run. The loss of the unsupervised discriminator and generator should remain at modest values throughout the run if they are kept in equilibrium.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
(60000, 28, 28, 1) (60000,)
(100, 28, 28, 1) (100,)
n_epochs=20, n_batch=100, 1/2=50, b/e=600, steps=12000
>1, c[2.305,6], d[0.096,2.399], g[0.095]
>2, c[2.298,18], d[0.089,2.399], g[0.095]
>3, c[2.308,10], d[0.084,2.401], g[0.095]
>4, c[2.304,8], d[0.080,2.404], g[0.095]
>5, c[2.254,18], d[0.077,2.407], g[0.095]
...
```

Listing 20.17: Example output of loss from training the SGAN model on the MNIST dataset.

The supervised classification model is evaluated on the entire training dataset at the end of every training epoch, in this case after every 600 training updates. At this time, the performance of the model is summarized, showing that it rapidly achieves good skill. This is surprising given that the model is only trained on 10 labeled examples of each class.

```
...
Classifier Accuracy: 94.640%
Classifier Accuracy: 93.622%
Classifier Accuracy: 91.870%
Classifier Accuracy: 92.525%
Classifier Accuracy: 92.180%
```

Listing 20.18: Example output of accuracy from training the SGAN model on the MNIST dataset.

The models are also saved at the end of each training epoch and plots of generated images are also created. The quality of the generated images is good given the relatively small number of training epochs.

Figure 20.8: Plot of Handwritten Digits Generated by the Semi-Supervised GAN After 8400 Updates.

20.5 How to Use the Final SGAN Classifier Model

Now that we have trained the generator and discriminator models, we can make use of them. In the case of the semi-supervised GAN, we are less interested in the generator model and more interested in the supervised model. Reviewing the results for the specific run, we can select a specific saved model that is known to have good performance on the test dataset. In this case, the model saved after 12 training epochs, or 7,200 updates, that had a classification accuracy of about 95.432% on the training dataset. We can load the model directly via the `load_model()` Keras function.

```
...
# load the model
model = load_model('c_model_7200.h5')
```

Listing 20.19: Example of loading a saved supervised model.

Once loaded, we can evaluate it on the entire training dataset again to confirm the finding, then evaluate it on the holdout test dataset. Recall, the feature extraction layers expect the input images to have the pixel values scaled to the range [-1,1], therefore, this must be performed

before any images are provided to the model. The complete example of loading the saved semi-supervised classifier model and evaluating it in the complete MNIST dataset is listed below.

```
# example of loading the classifier model and generating images
from numpy import expand_dims
from keras.models import load_model
from keras.datasets.mnist import load_data
# load the model
model = load_model('c_model_7200.h5')
# load the dataset
(trainX, trainy), (testX, testy) = load_data()
# expand to 3d, e.g. add channels
trainX = expand_dims(trainX, axis=-1)
testX = expand_dims(testX, axis=-1)
# convert from ints to floats
trainX = trainX.astype('float32')
testX = testX.astype('float32')
# scale from [0,255] to [-1,1]
trainX = (trainX - 127.5) / 127.5
testX = (testX - 127.5) / 127.5
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
print('Train Accuracy: %.3f%%' % (train_acc * 100))
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Test Accuracy: %.3f%%' % (test_acc * 100))
```

Listing 20.20: Example of loading the saved supervised model for classification.

Running the example loads the model and evaluates it on the MNIST dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieves the expected performance of 95.432% on the training dataset, confirming we have loaded the correct model. We can also see that the accuracy on the holdout test dataset is as good, or slightly better, at about 95.920%. This shows that the learned classifier has good generalization.

```
Train Accuracy: 95.432%
Test Accuracy: 95.920%
```

Listing 20.21: Example output from loading the saved supervised model for classification.

We have successfully demonstrated the training and evaluation of a semi-supervised classifier model fit via the GAN architecture.

20.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Standalone Classifier**. Fit a standalone classifier model on the labeled dataset directly and compare performance to the SGAN model.
- **Number of Labeled Examples**. Repeat the example of more or fewer labeled examples and compare the performance of the model

- **Model Tuning**. Tune the performance of the discriminator and generator model to further lift the performance of the supervised model closer toward state-of-the-art results.

If you explore any of these extensions, I'd love to know.

20.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

20.7.1 Papers

- Semi-Supervised Learning with Generative Adversarial Networks, 2016.
https://arxiv.org/abs/1606.01583
- Improved Techniques for Training GANs, 2016.
https://arxiv.org/abs/1606.03498
- Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks, 2015.
https://arxiv.org/abs/1511.06390
- Semi-supervised Learning with GANs: Manifold Invariance with Improved Inference, 2017.
https://arxiv.org/abs/1705.08850
- Semi-Supervised Learning with GANs: Revisiting Manifold Regularization, 2018.
https://arxiv.org/abs/1805.08957

20.7.2 API

- Keras Datasets API..
https://keras.io/datasets/
- Keras Sequential Model API.
https://keras.io/models/sequential/
- Keras Convolutional Layers API.
https://keras.io/layers/convolutional/
- How can I "freeze" Keras layers?.
https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers
- MatplotLib API.
https://matplotlib.org/api/
- NumPy Random sampling (numpy.random) API.
https://docs.scipy.org/doc/numpy/reference/routines.random.html
- NumPy Array manipulation routines.
https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html

20.7.3 Projects

- Improved GAN Project (Official), GitHub.
  https://github.com/openai/improved-gan

20.8 Summary

In this tutorial, you discovered how to develop a Semi-Supervised Generative Adversarial Network from scratch. Specifically, you learned:

- The semi-supervised GAN is an extension of the GAN architecture for training a classifier model while making use of labeled and unlabeled data.
- There are at least three approaches to implementing the supervised and unsupervised discriminator models in Keras used in the semi-supervised GAN.
- How to train a semi-supervised GAN from scratch on MNIST and load and use the trained classifier for making predictions.

20.8.1 Next

This was the final tutorial in this part. In the next part, you will discover image-to-image translation with GAN models.

Part VI

Image Translation

Overview

In this part you will discover how to develop GAN models for image-to-image translation with paired and unpaired image datasets using the Pix2Pix and CycleGAN approaches. The models in this part are somewhat complex, therefore we will carefully step through their description, development, and finally application in separate chapters. After reading the chapters in this part, you will know:

- The Pix2Pix approach to modeling paired image-to-image translation (Chapter 21).
- How to implement the PatchGAN discriminator and U-Net generator of the Pix2Pix architecture (Chapter 22).
- How to develop a Pix2Pix application for transforming satellite photos to Google maps, and the reverse (Chapter 23).
- The CycleGAN approach to modeling unpaired image-to-image translation (Chapter 24).
- How to implement the PatchGAN discriminator and Encoder-Decoder generator of the CycleGAN architecture (Chapter 25).
- How to develop a CycleGAN application for transforming photos of horses to zebra, and the reverse (Chapter 26).

Chapter 21

Introduction to Pix2Pix

Image-to-image translation is the controlled conversion of a given source image to a target image. An example might be the conversion of black and white photographs to color photographs. Image-to-image translation is a challenging problem and often requires specialized models and loss functions for a given translation task or dataset. The Pix2Pix GAN is a general approach for image-to-image translation. It is based on the conditional generative adversarial network, where a target image is generated, conditional on a given input image. In this case, the Pix2Pix GAN changes the loss function so that the generated image is both plausible in the content of the target domain, and is a plausible translation of the input image. In this tutorial, you will discover the Pix2Pix conditional generative adversarial network for image-to-image translation. After reading this tutorial, you will know:

- Image-to-image translation often requires specialized models and hand-crafted loss functions.
- Pix2Pix GAN provides a general purpose model and loss function for image-to-image translation.
- The Pix2Pix GAN was demonstrated on a wide variety of image generation tasks, including translating photographs from day to night and product sketches to photographs.

Let's get started.

21.1 Overview

This tutorial is divided into five parts; they are:

1. The Problem of Image-to-Image Translation
2. Pix2Pix GAN for Image-to-Image Translation
3. Pix2Pix Architectural Details
4. Applications of the Pix2Pix GAN
5. Insight into Pix2Pix Architectural Choices

21.2 The Problem of Image-to-Image Translation

Image-to-image translation is the problem of changing a given image in a specific or controlled way. Examples include translating a photograph of a landscape from day to night or translating a segmented image to a photograph.

> In analogy to automatic language translation, we define automatic image-to-image translation as the task of translating one possible representation of a scene into another, given sufficient training data.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

It is a challenging problem that typically requires the development of a specialized model and hand-crafted loss function for the type of translation task being performed. Classical approaches use per-pixel classification or regression models, the problem with which is that each predicted pixel is independent of the pixels predicted before it and the broader structure of the image might be missed.

> Image-to-image translation problems are often formulated as per-pixel classification or regression. These formulations treat the output space as "unstructured" in the sense that each output pixel is considered conditionally independent from all others given the input image.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

Ideally, a technique is required that is general, meaning that the same general model and loss function can be used for multiple different image-to-image translation tasks.

21.3 Pix2Pix GAN for Image-to-Image Translation

Pix2Pix is a Generative Adversarial Network, or GAN, model designed for general purpose image-to-image translation. The approach was presented by Phillip Isola, et al. in their 2016 paper titled *Image-to-Image Translation with Conditional Adversarial Networks* and presented at CVPR in 2017. The GAN architecture is an approach to training a generator model, typically used for generating images. A discriminator model is trained to classify images as real (from the dataset) or fake (generated), and the generator is trained to fool the discriminator model. The Conditional GAN, or cGAN, is an extension of the GAN architecture that provides control over the image that is generated, e.g. allowing an image of a given class to be generated (see Chapter 17). Pix2Pix GAN is an implementation of the cGAN where the generation of an image is conditional on a given image.

> Just as GANs learn a generative model of data, conditional GANs (cGANs) learn a conditional generative model. This makes cGANs suitable for image-to-image translation tasks, where we condition on an input image and generate a corresponding output image.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

The generator model is provided with a given image as input and generates a translated version of the image. The discriminator model is given an input image and a real or generated paired image and must determine whether the paired image is real or fake. Finally, the generator model is trained to both fool the discriminator model and to minimize the loss between the generated image and the expected target image. As such, the Pix2Pix GAN must be trained on image datasets that are comprised of input images (before translation) and output or target images (after translation). This general architecture allows the Pix2Pix model to be trained for a range of image-to-image translation tasks.

21.4 Pix2Pix Architectural Details

The Pix2Pix GAN architecture involves the careful specification of a generator model, discriminator model, and model optimization procedure. Both the generator and discriminator models use standard Convolution-BatchNormalization-ReLU blocks of layers as is common for deep convolutional neural networks. Specific layer configurations are provided in the appendix of the paper.

> Both generator and discriminator use modules of the form convolution-BatchNorm-ReLu.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

Let's take a closer look at each of the two model architectures and the loss function used to optimize the model weights.

21.4.1 U-Net Generator Model

The generator model takes an image as input, and unlike a standard GAN model, it does not take a point from the latent space as input. Instead, the source of randomness comes from the use of dropout layers that are used both during training and when a prediction is made.

> Instead, for our final models, we provide noise only in the form of dropout, applied on several layers of our generator at both training and test time.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

- **Input**: Image from source domain
- **Output**: Image in target domain

A U-Net model architecture is used for the generator, instead of the common encoder-decoder model. The encoder-decoder generator architecture involves taking an image as input and downsampling it over a few layers until a bottleneck layer, where the representation is then upsampled again over a few layers before outputting the final image with the desired size. The U-Net model architecture is very similar in that it involves downsampling to a bottleneck and upsampling again to an output image, but links or skip-connections are made between layers of the same size in the encoder and the decoder, allowing the bottleneck to be circumvented.

> For many image translation problems, there is a great deal of low-level information shared between the input and output, and it would be desirable to shuttle this information directly across the net. [...] To give the generator a means to circumvent the bottleneck for information like this, we add skip connections, following the general shape of a "U-Net".
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

For example, the first layer of the encoder has the same-sized feature maps as the last layer of the decoder and is merged with the decoder. This is repeated with each layer in the encoder and corresponding layer of the decoder, forming a U-shaped model.

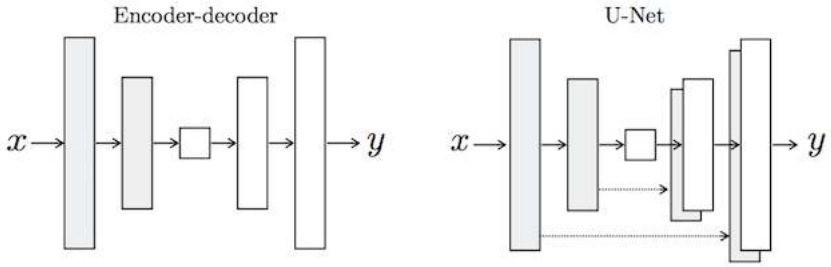

Figure 21.1: Depiction of the Encoder-Decoder Generator and U-Net Generator Models. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

21.4.2 PatchGAN Discriminator Model

The discriminator model takes an image from the source domain and an image from the target domain and predicts the likelihood of whether the image from the target domain is a real or generated version of the source image.

- **Input**: Image from source domain, and Image from the target domain.
- **Output**: Probability that the image from the target domain is a real translation of the source image.

The input to the discriminator model highlights the need to have an image dataset comprised of paired source and target images when training the model. Unlike the standard GAN model that uses a deep convolutional neural network to classify images, the Pix2Pix model uses a PatchGAN. This is a deep convolutional neural network designed to classify patches of an input image as real or fake, rather than the entire image.

> ... we design a discriminator architecture - which we term a PatchGAN - that only penalizes structure at the scale of patches. This discriminator tries to classify if each $N \times N$ patch in an image is real or fake. We run this discriminator convolutionally across the image, averaging all responses to provide the ultimate output of D.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

The PatchGAN discriminator model is implemented as a deep convolutional neural network, but the number of layers is configured such that the effective receptive field of each output of the network maps to a specific size in the input image. The output of the network is a single feature map of real/fake predictions that can be averaged to give a single score. A patch size of 70×70 was found to be effective across a range of image-to-image translation tasks.

21.4.3 Composite Adversarial and L1 Loss

The discriminator model is trained in a standalone manner in the same way as a standard GAN model, minimizing the negative log likelihood of identifying real and fake images, although conditioned on a source image. The training of the discriminator is too fast compared to the generator, therefore the discriminator loss is halved in order to slow down the training process.

$$\text{Discriminator Loss} = 0.5 \times \text{Discriminator Loss} \tag{21.1}$$

The generator model is trained using both the adversarial loss for the discriminator model and the L1 or mean absolute pixel difference between the generated translation of the source image and the expected target image. The adversarial loss and the L1 loss are combined into a composite loss function, which is used to update the generator model. L2 loss was also evaluated and found to result in blurry images.

> The discriminator's job remains unchanged, but the generator is tasked to not only fool the discriminator but also to be near the ground truth output in an L2 sense. We also explore this option, using L1 distance rather than L2 as L1 encourages less blurring
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

The adversarial loss influences whether the generator model can output images that are plausible in the target domain, whereas the L1 loss regularizes the generator model to output images that are a plausible translation of the source image. As such, the combination of the L1 loss to the adversarial loss is controlled by a new hyperparameter lambda (λ), which is set to 10 or 100, e.g. giving 10 or 100 times the importance of the L1 loss than the adversarial loss to the generator during training.

$$\text{Generator Loss} = \text{Adversarial Loss} + \lambda \times \text{L1 Loss} \tag{21.2}$$

21.5 Applications of the Pix2Pix GAN

The Pix2Pix GAN was demonstrated on a range of interesting image-to-image translation tasks. For example, the paper lists nine applications; they are:

- Semantic labels $\Leftrightarrow$ photo, trained on the Cityscapes dataset.
- Architectural labels $\Rightarrow$ photo, trained on Facades.
- Map $\Leftrightarrow$ aerial photo, trained on data scraped from Google Maps.

- Black and White ⇒ color photos.
- Edges ⇒ photo.
- Sketch ⇒ photo.
- Day ⇒ night photographs.
- Thermal ⇒ color photos.
- Photo with missing pixels ⇒ inpainted photo, trained on Paris StreetView.

In this section, we'll review a few of the examples taken from the paper.

21.5.1 Semantic Labels to Photographs

The example below demonstrates the translation of semantic labeled images to photographs of street scenes.

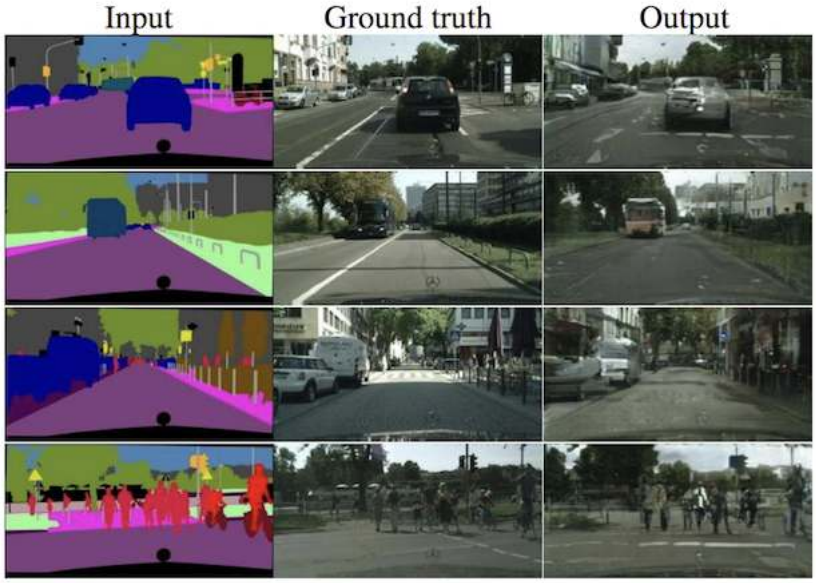

Figure 21.2: Pix2Pix GAN Translation of Semantic Images to Photographs of a Cityscape. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

Another example is provided demonstrating semantic labeled images of building facades to photographs.

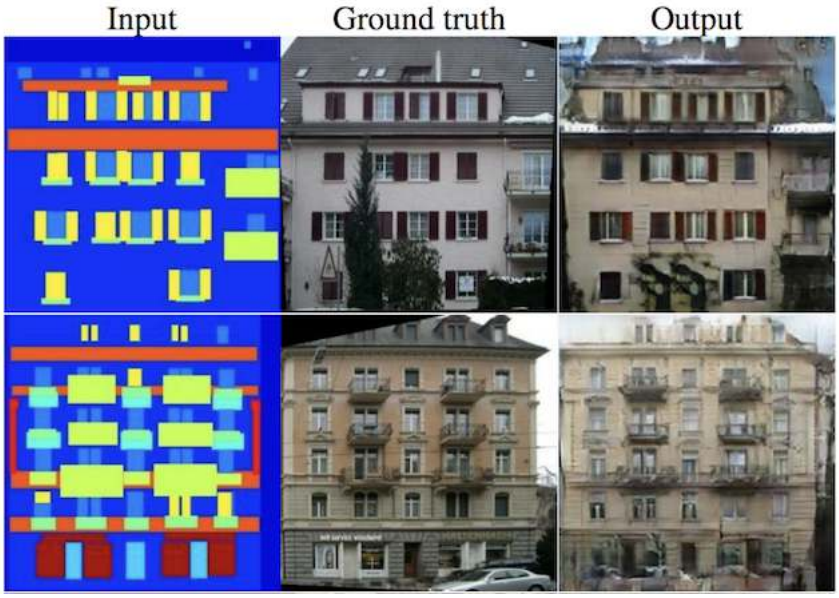

Figure 21.3: Pix2Pix GAN Translation of Semantic Images to Photographs of Building Facades. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

21.5.2 Daytime to Nighttime Photographs

The example below demonstrates the translation of daytime to nighttime photographs.

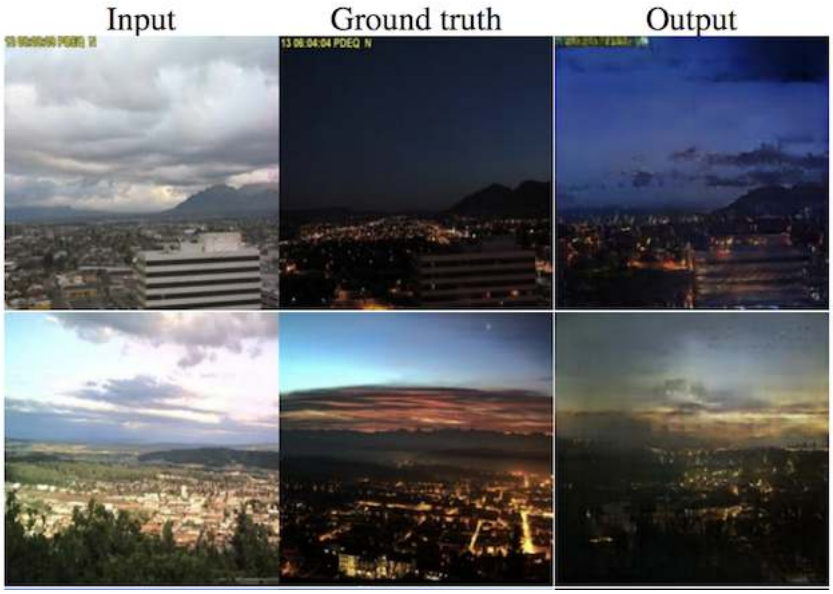

Figure 21.4: Pix2Pix GAN Translation of Daytime Photographs to Nighttime. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

21.5.3 Product Sketch to Photograph

The example below demonstrates the translation of product sketches of bags to photographs.

Figure 21.5: Pix2Pix GAN Translation of Product Sketches of Bags to Photographs. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

Similar examples are given for translating sketches of shoes to photographs.

Figure 21.6: Pix2Pix GAN Translation of Product Sketches of Shoes to Photographs. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

21.5.4 Photograph Inpainting

The example below demonstrates inpainting photographs of streetscapes taken in Paris.

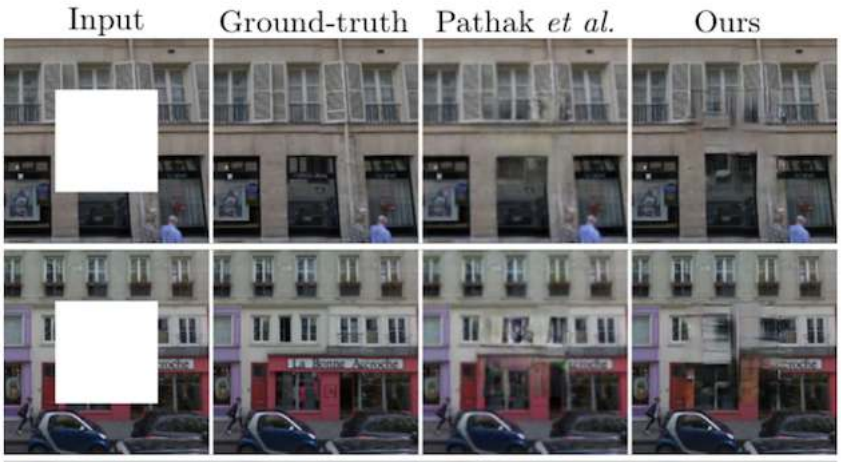

Figure 21.7: Pix2Pix GAN Inpainting Photographs of Paris. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

21.5.5 Thermal Image to Color Photograph

The example below demonstrates the translation of thermal images to color photographs of streetscapes.

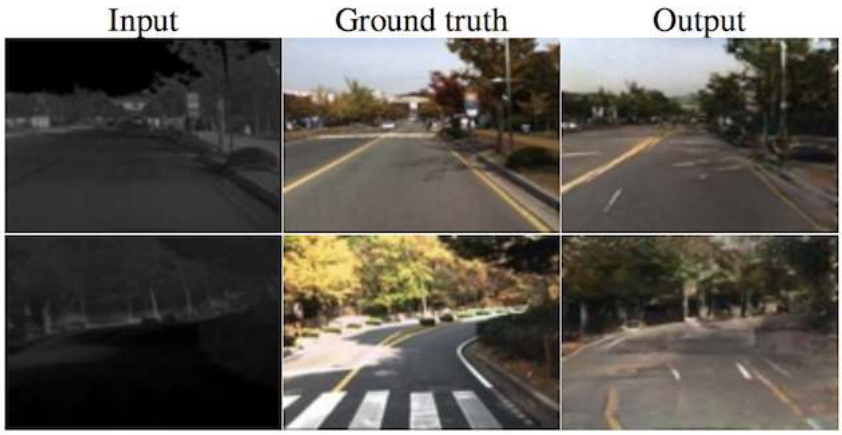

Figure 21.8: Pix2Pix GAN Translation of Thermal Images to Color Photographs. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

21.6 Insight into Pix2Pix Architectural Choices

The authors explore and analyze the effect of different model configurations and loss functions on image quality, supporting the architectural choices. The findings from these experiments shed light on perhaps why the Pix2Pix approach is effective across a wide range of image translation tasks.

21.6.1 Analysis of Loss Function

Experiments are performed to compare different loss functions used to train the generator model. These included only using the L1 loss, only the conditional adversarial loss, only using unconditional adversarial loss, and combinations of L1 with each adversarial loss. The results were interesting, showing that L1 and conditional adversarial loss alone can generate reasonable images, although the L1 images were blurry and the cGAN images introduced artifacts. The combination of the two gave the clearest results.

> L1 alone leads to reasonable but blurry results. The cGAN alone [...] gives much sharper results but introduces visual artifacts on certain applications. Adding both terms together (with lambda = 100) reduces these artifacts.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

Figure 21.9: Generated Images Using L1, Conditional Adversarial (cGAN), and Composite Loss Functions. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

21.6.2 Analysis of Generator Model

The U-Net generator model architecture was compared to a more common encoder-decoder generator model architecture. Both approaches were compared with just L1 loss and L1 + conditional adversarial loss, showing that the encoder-decoder was able to generate images in both cases, but the images were much sharper when using the U-Net architecture.

> The encoder-decoder is unable to learn to generate realistic images in our experiments. The advantages of the U-Net appear not to be specific to conditional GANs: when both U-Net and encoder-decoder are trained with an L1 loss, the U-Net again achieves the superior results.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

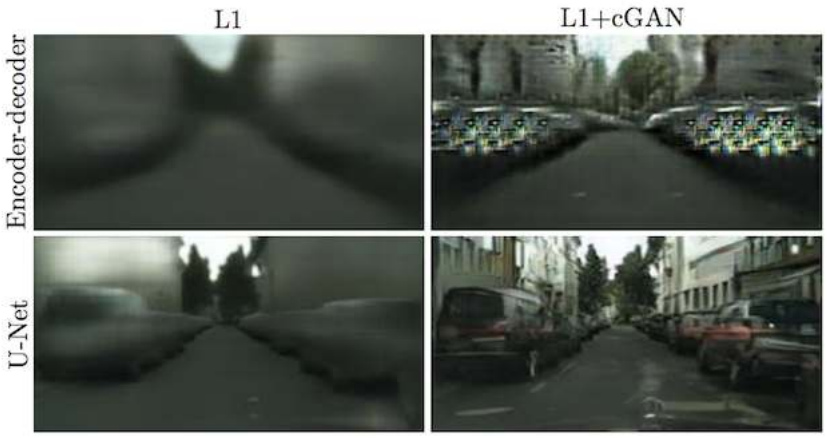

Figure 21.10: Generated Images Using the Encoder-Decoder and U-Net Generator Models Under Different Loss. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

Analysis of Discriminator Model Experiments were performed comparing the PatchGAN discriminator with different sized effective receptive fields. Versions of the model were tested from a 1×1 receptive field or PixelGAN, through to a full-sized 286×286 or ImageGAN, as well as smaller 16×16 and 70×70 PatchGANs. The larger the receptive field, the deeper the network. This means that the 1×1 PixelGAN is the shallowest model and the 286×286 ImageGAN is the deepest model. The results showed that very small receptive fields can generate effective images, although the full sized ImageGAN provides crisper results, but is harder to train. Using a smaller 70×70 receptive field provided a good trade-off of performance (model depth) and image quality.

> The 70×70 PatchGAN [...] achieves slightly better scores. Scaling beyond this, to the full 286×286 ImageGAN, does not appear to improve the visual quality [...] This may be because the ImageGAN has many more parameters and greater depth than the 70×70 PatchGAN, and may be harder to train.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

Figure 21.11: Generated Images Using PatchGANs With Different Sized Receptive Fields. Taken from: Image-to-Image Translation With Conditional Adversarial Networks.

21.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Image-to-Image Translation with Conditional Adversarial Networks, 2016.
https://arxiv.org/abs/1611.07004
- Image-to-Image Translation with Conditional Adversarial Nets, Homepage.
https://phillipi.github.io/pix2pix/
- Image-to-image translation with conditional adversarial nets, GitHub.
https://github.com/phillipi/pix2pix
- pytorch-CycleGAN-and-pix2pix, GitHub.
https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix
- Interactive Image-to-Image Demo, 2017.
https://affinelayer.com/pixsrv/
- Pix2Pix Datasets.
http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/

21.8 Summary

In this tutorial, you discovered the Pix2Pix conditional generative adversarial networks for image-to-image translation. Specifically, you learned:

- Image-to-image translation often requires specialized models and hand-crafted loss functions.
- Pix2Pix GAN provides a general purpose model and loss function for image-to-image translation.
- The Pix2Pix GAN was demonstrated on a wide variety of image generation tasks, including translating photographs from day to night and product sketches to photographs.

21.8.1 Next

In the next tutorial, you will discover how to implement Pix2Pix models from scratch with Keras.

Chapter 22

How to Implement Pix2Pix Models

The Pix2Pix GAN is a generator model for performing image-to-image translation trained on paired examples. For example, the model can be used to translate images of daytime to nighttime, or from sketches of products like shoes to photographs of products. The benefit of the Pix2Pix model is that compared to other GANs for conditional image generation, it is relatively simple and capable of generating large high-quality images across a variety of image translation tasks. The model is very impressive but has an architecture that appears somewhat complicated to implement for beginners. In this tutorial, you will discover how to implement the Pix2Pix GAN architecture from scratch using the Keras deep learning framework. After completing this tutorial, you will know:

- How to develop the PatchGAN discriminator model for the Pix2Pix GAN.
- How to develop the U-Net encoder-decoder generator model for the Pix2Pix GAN.
- How to implement the composite model for updating the generator and how to train both models.

Let's get started.

22.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What Is the Pix2Pix GAN?
2. How to Implement the PatchGAN Discriminator Model
3. How to Implement the U-Net Generator Model
4. How to Implement Adversarial and L1 Loss
5. How to Update Model Weights

22.2 What Is the Pix2Pix GAN?

Pix2Pix is a Generative Adversarial Network, or GAN, model designed for general purpose image-to-image translation. The approach was presented by Phillip Isola, et al. in their 2016 paper titled *Image-to-Image Translation with Conditional Adversarial Networks* and presented at CVPR in 2017 (introduced in Chapter 21). The GAN architecture is comprised of a generator model for outputting new plausible synthetic images and a discriminator model that classifies images as real (from the dataset) or fake (generated). The discriminator model is updated directly, whereas the generator model is updated via the discriminator model. As such, the two models are trained simultaneously in an adversarial process where the generator seeks to better fool the discriminator and the discriminator seeks to better identify the counterfeit images.

The Pix2Pix model is a type of conditional GAN, or cGAN, where the generation of the output image is conditional on an input, in this case, a source image. The discriminator is provided both with a source image and the target image and must determine whether the target is a plausible transformation of the source image. Again, the discriminator model is updated directly, and the generator model is updated via the discriminator model, although the loss function is updated. The generator is trained via adversarial loss, which encourages the generator to generate plausible images in the target domain. The generator is also updated via L1 loss measured between the generated image and the expected output image. This additional loss encourages the generator model to create plausible translations of the source image.

The Pix2Pix GAN has been demonstrated on a range of image-to-image translation tasks such as converting maps to satellite photographs, black and white photographs to color, and sketches of products to product photographs. Now that we are familiar with the Pix2Pix GAN, let's explore how we can implement it using the Keras deep learning library.

22.3 How to Implement the PatchGAN Discriminator Model

The discriminator model in the Pix2Pix GAN is implemented as a PatchGAN. The PatchGAN is designed based on the size of the receptive field, sometimes called the effective receptive field. The receptive field is the relationship between one output activation of the model to an area on the input image (actually volume as it proceeded down the input channels). A PatchGAN with the size 70×70 is used, which means that the output (or each output) of the model maps to a 70×70 square of the input image. In effect, a 70×70 PatchGAN will classify 70×70 patches of the input image as real or fake.

> ... we design a discriminator architecture - which we term a PatchGAN - that only penalizes structure at the scale of patches. This discriminator tries to classify if each $N \times N$ patch in an image is real or fake. We run this discriminator convolutionally across the image, averaging all responses to provide the ultimate output of D.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

Before we dive into the configuration details of the PatchGAN, it is important to get a handle on the calculation of the receptive field. The receptive field is not the size of the output of the discriminator model, e.g. it does not refer to the shape of the activation map output by the

model. It is a definition of the model in terms of one pixel in the output activation map to the input image. The output of the model may be a single value or a square activation map of values that predict whether each patch of the input image is real or fake. Traditionally, the receptive field refers to the size of the activation map of a single convolutional layer with regards to the input of the layer, the size of the filter, and the size of the stride. The effective receptive field generalizes this idea and calculates the receptive field for the output of a stack of convolutional layers with regard to the raw image input. The terms are often used interchangeably.

The authors of the Pix2Pix GAN provide a Matlab script to calculate the effective receptive field size for different model configurations in a script called `receptive_field_sizes.m`[1]. It can be helpful to work through an example for the 70×70 PatchGAN receptive field calculation. The 70×70 PatchGAN has a fixed number of three layers (excluding the output and second last layers), regardless of the size of the input image. The calculation of the receptive field in one dimension is calculated as:

$$\text{receptive field} = (\text{output size} - 1) \times \text{stride} + \text{kernel size} \tag{22.1}$$

Where output size is the size of the prior layers activation map, stride is the number of pixels the filter is moved when applied to the activation, and kernel size is the size of the filter to be applied. The PatchGAN uses a fixed stride of 2×2 (except in the output and second last layers) and a fixed kernel size of 4×4. We can, therefore, calculate the receptive field size starting with one pixel in the output of the model and working backward to the input image. We can develop a Python function called `receptive_field()` to calculate the receptive field, then calculate and print the receptive field for each layer in the Pix2Pix PatchGAN model. The complete example is listed below.

```
# example of calculating the receptive field for the PatchGAN

# calculate the effective receptive field size
def receptive_field(output_size, kernel_size, stride_size):
    return (output_size - 1) * stride_size + kernel_size

# output layer 1x1 pixel with 4x4 kernel and 1x1 stride
rf = receptive_field(1, 4, 1)
print(rf)
# second last layer with 4x4 kernel and 1x1 stride
rf = receptive_field(rf, 4, 1)
print(rf)
# 3 PatchGAN layers with 4x4 kernel and 2x2 stride
rf = receptive_field(rf, 4, 2)
print(rf)
rf = receptive_field(rf, 4, 2)
print(rf)
rf = receptive_field(rf, 4, 2)
print(rf)
```

Listing 22.1: Example of calculating the receptive field for a PatchGAN.

Running the example prints the size of the receptive field for each layer in the model from the output layer to the input layer. We can see that each 1×1 pixel in the output layer maps to a 70×70 receptive field in the input layer.

[1] https://github.com/phillipi/pix2pix/blob/master/scripts/receptive_field_sizes.m

```
4
7
16
34
70
```

Listing 22.2: Example output from calculating the receptive field for a PatchGAN.

The authors of the Pix2Pix paper explore different PatchGAN configurations, including a 1×1 receptive field called a PixelGAN and a receptive field that matches the 256×256 pixel images input to the model (resampled to 286×286) called an ImageGAN. They found that the 70×70 PatchGAN resulted in the best trade-off of performance and image quality.

> The 70×70 PatchGAN [...] achieves slightly better scores. Scaling beyond this, to the full 286×286 ImageGAN, does not appear to improve the visual quality of the results.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

The configuration for the PatchGAN is provided in the appendix of the paper and can be confirmed by reviewing the `defineD_n_layers()`[2] function in the official Torch implementation. The model takes two images as input, specifically a source and a target image. These images are concatenated together at the channel level, e.g. 3 color channels of each image become 6 channels of the input.

> Let Ck denote a Convolution-BatchNorm-ReLU layer with k filters. [...] All convolutions are 4×4 spatial filters applied with stride 2. [...] The 70×70 discriminator architecture is: C64-C128-C256-C512. After the last layer, a convolution is applied to map to a 1-dimensional output, followed by a Sigmoid function. As an exception to the above notation, BatchNorm is not applied to the first C64 layer. All ReLUs are leaky, with slope 0.2.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

The PatchGAN configuration is defined using a shorthand notation as: C64-C128-C256-C512, where C refers to a block of Convolution-BatchNorm-LeakyReLU layers and the number indicates the number of filters. Batch normalization is not used in the first layer. As mentioned, the kernel size is fixed at 4×4 and a stride of 2×2 is used on all but the last 2 layers of the model. The slope of the `LeakyReLU` is set to 0.2, and a sigmoid activation function is used in the output layer.

> Random jitter was applied by resizing the 256×256 input images to 286×286 and then randomly cropping back to size 256×256. Weights were initialized from a Gaussian distribution with mean 0 and standard deviation 0.02.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

[2] https://github.com/phillipi/pix2pix/blob/master/models.lua#L180

Model weights were initialized via random Gaussian with a mean of 0.0 and standard deviation of 0.02. Images input to the model are 256×256.

> ... we divide the objective by 2 while optimizing D, which slows down the rate at which D learns relative to G. We use minibatch SGD and apply the Adam solver, with a learning rate of 0.0002, and momentum parameters $\beta 1 = 0.5$, $\beta 2 = 0.999$.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

The model is trained with a batch size of one image and the Adam version of stochastic gradient descent is used with a small learning range and modest momentum. The loss for the discriminator is weighted by 50% for each model update. Tying this all together, we can define a function named `define_discriminator()` that creates the 70×70 PatchGAN discriminator model. The complete example of defining the model is listed below.

```
# example of defining a 70x70 patchgan discriminator model
from keras.optimizers import Adam
from keras.initializers import RandomNormal
from keras.models import Model
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import LeakyReLU
from keras.layers import Activation
from keras.layers import Concatenate
from keras.layers import BatchNormalization
from keras.utils.vis_utils import plot_model

# define the discriminator model
def define_discriminator(image_shape):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # source image input
  in_src_image = Input(shape=image_shape)
  # target image input
  in_target_image = Input(shape=image_shape)
  # concatenate images channel-wise
  merged = Concatenate()([in_src_image, in_target_image])
  # C64
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(merged)
  d = LeakyReLU(alpha=0.2)(d)
  # C128
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C256
  d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C512
  d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # second last output layer
  d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
```

```
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # patch output
  d = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
  patch_out = Activation('sigmoid')(d)
  # define model
  model = Model([in_src_image, in_target_image], patch_out)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, loss_weights=[0.5])
  return model

# define image shape
image_shape = (256,256,3)
# create the model
model = define_discriminator(image_shape)
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='discriminator_model_plot.png', show_shapes=True,
    show_layer_names=True)
```

Listing 22.3: Example of defining and summarizing the PatchGAN discriminator model.

Running the example first summarizes the model, providing insight into how the input shape is transformed across the layers and the number of parameters in the model. The output is omitted here for brevity. A plot of the model is created showing much the same information in a graphical form. The model is not complex, with a linear path with two input images and a single output prediction.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

We can see that the two input images are concatenated together to create one $256 \times 256 \times 6$ input to the first hidden convolutional layer. This concatenation of input images could occur before the input layer of the model, but allowing the model to perform the concatenation makes the behavior of the model clearer. We can see that the model output will be an activation map with the size 16×16 pixels or activations and a single channel, with each value in the map corresponding to a 70×70 pixel patch of the input 256×256 image. If the input image was half the size at 128×128, then the output feature map would also be halved to 8×8 The model is a binary classification model, meaning it predicts an output as a probability in the range [0,1], in this case, the likelihood of whether the input image is real or from the target dataset. The patch of values can be averaged to give a real/fake prediction by the model. When trained, the target is compared to a matrix of target values, 0 for fake and 1 for real.

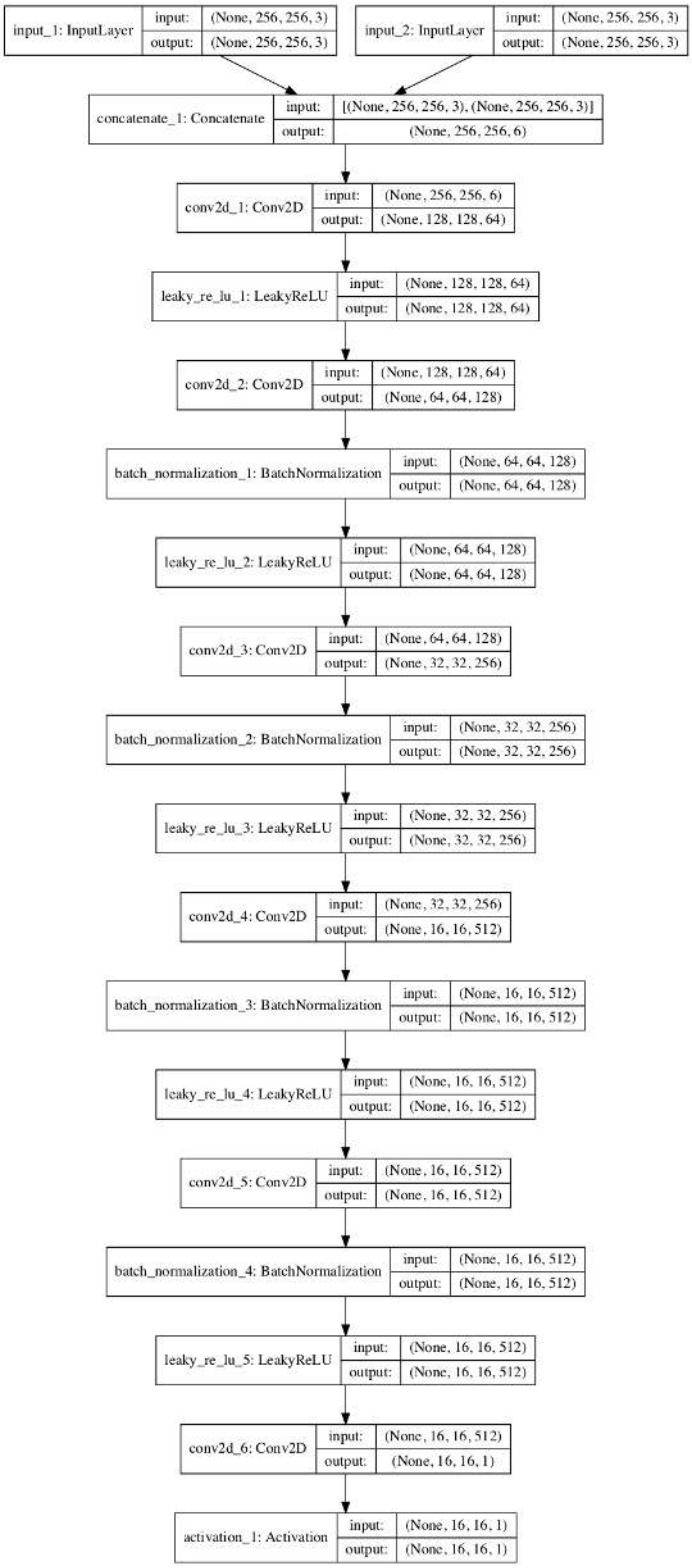

Figure 22.1: Plot of the PatchGAN Model Used in the Pix2Pix GAN Architecture.

Now that we know how to implement the PatchGAN discriminator model, we can now look at implementing the U-Net generator model.

22.4 How to Implement the U-Net Generator Model

The generator model for the Pix2Pix GAN is implemented as a U-Net. The U-Net model is an encoder-decoder model for image translation where skip connections are used to connect layers in the encoder with corresponding layers in the decoder that have the same sized feature maps. The encoder part of the model is comprised of convolutional layers that use a 2×2 stride to downsample the input source image down to a bottleneck layer. The decoder part of the model reads the bottleneck output and uses transpose convolutional layers to upsample to the required output image size.

> ... the input is passed through a series of layers that progressively downsample, until a bottleneck layer, at which point the process is reversed.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

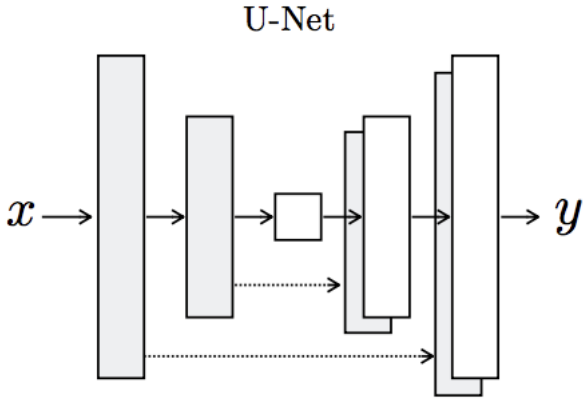

Figure 22.2: Architecture of the U-Net Generator Model. Taken from Image-to-Image Translation With Conditional Adversarial Networks.

Skip connections are added between the layers with the same sized feature maps so that the first downsampling layer is connected with the last upsampling layer, the second downsampling layer is connected with the second last upsampling layer, and so on. The connections concatenate the channels of the feature map in the downsampling layer with the feature map in the upsampling layer.

> Specifically, we add skip connections between each layer i and layer $n - i$, where n is the total number of layers. Each skip connection simply concatenates all channels at layer i with those at layer $n - i$.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

Unlike traditional generator models in the GAN architecture, the U-Net generator does not take a point from the latent space as input. Instead, dropout layers are used as a source of randomness both during training and when the model is used to make a prediction, e.g. generate an image at inference time. Similarly, batch normalization is used in the same way during

training and inference, meaning that statistics are calculated for each batch and not fixed at the end of the training process. This is referred to as instance normalization, specifically when the batch size is set to 1 as it is with the Pix2Pix model.

> At inference time, we run the generator net in exactly the same manner as during the training phase. This differs from the usual protocol in that we apply dropout at test time, and we apply batch normalization using the statistics of the test batch, rather than aggregated statistics of the training batch.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

In Keras, layers like `Dropout` and `BatchNormalization` operate differently during training and in inference model. We can set the `training` argument when calling these layers to `True` to ensure that they always operate in training-model, even when used during inference. For example, a `Dropout` layer that will drop out during inference as well as training can be added to the model as follows:

```
...
g = Dropout(0.5)(g, training=True)
```

Listing 22.4: Example of configuring a `Dropout` layer to operate during inference.

As with the discriminator model, the configuration details of the generator model are defined in the appendix of the paper and can be confirmed when comparing against the `defineG_unet()` function in the official Torch implementation[3]. The encoder uses blocks of Convolution-BatchNorm-LeakyReLU like the discriminator model, whereas the decoder model uses blocks of Convolution-BatchNorm-Dropout-ReLU with a dropout rate of 50%. All convolutional layers use a filter size of 4×4 and a stride of 2×2.

> Let Ck denote a Convolution-BatchNorm-ReLU layer with k filters. CDk denotes a Convolution-BatchNormDropout-ReLU layer with a dropout rate of 50%. All convolutions are 4×4 spatial filters applied with stride 2.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

The architecture of the U-Net model is defined using the shorthand notation as:

- **Encoder**: C64-C128-C256-C512-C512-C512-C512-C512
- **Decoder**: CD512-CD1024-CD1024-C1024-C1024-C512-C256-C128

The last layer of the encoder is the bottleneck layer, which does not use batch normalization, according to an amendment to the paper and confirmation in the code, and uses a ReLU activation instead of leaky ReLU.

> ... the activations of the bottleneck layer are zeroed by the batchnorm operation, effectively making the innermost layer skipped. This issue can be fixed by removing batchnorm from this layer, as has been done in the public code

[3] https://github.com/phillipi/pix2pix/blob/master/models.lua#L47

— *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

The number of filters in the U-Net decoder is a little misleading as it is the number of filters for the layer after concatenation with the equivalent layer in the encoder. This may become more clear when we create a plot of the model. The output of the model uses a single convolutional layer with three channels, and Tanh activation function is used in the output layer, common to GAN generator models. Batch normalization is not used in the first layer of the encoder.

> After the last layer in the decoder, a convolution is applied to map to the number of output channels (3 in general [...]), followed by a Tanh function [...] BatchNorm is not applied to the first C64 layer in the encoder. All ReLUs in the encoder are leaky, with slope 0.2, while ReLUs in the decoder are not leaky.

— *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

Tying this all together, we can define a function named `define_generator()` that defines the U-Net encoder-decoder generator model. Two helper functions are also provided for defining encoder blocks of layers and decoder blocks of layers. The complete example of defining the model is listed below.

```
# example of defining a u-net encoder-decoder generator model
from keras.initializers import RandomNormal
from keras.models import Model
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Activation
from keras.layers import Concatenate
from keras.layers import Dropout
from keras.layers import BatchNormalization
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define an encoder block
def define_encoder_block(layer_in, n_filters, batchnorm=True):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # add downsampling layer
  g = Conv2D(n_filters, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(layer_in)
  # conditionally add batch normalization
  if batchnorm:
    g = BatchNormalization()(g, training=True)
  # leaky relu activation
  g = LeakyReLU(alpha=0.2)(g)
  return g

# define a decoder block
def decoder_block(layer_in, skip_in, n_filters, dropout=True):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # add upsampling layer
```

```
  g = Conv2DTranspose(n_filters, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(layer_in)
  # add batch normalization
  g = BatchNormalization()(g, training=True)
  # conditionally add dropout
  if dropout:
    g = Dropout(0.5)(g, training=True)
  # merge with skip connection
  g = Concatenate()([g, skip_in])
  # relu activation
  g = Activation('relu')(g)
  return g

# define the standalone generator model
def define_generator(image_shape=(256,256,3)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=image_shape)
  # encoder model: C64-C128-C256-C512-C512-C512-C512-C512
  e1 = define_encoder_block(in_image, 64, batchnorm=False)
  e2 = define_encoder_block(e1, 128)
  e3 = define_encoder_block(e2, 256)
  e4 = define_encoder_block(e3, 512)
  e5 = define_encoder_block(e4, 512)
  e6 = define_encoder_block(e5, 512)
  e7 = define_encoder_block(e6, 512)
  # bottleneck, no batch norm and relu
  b = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(e7)
  b = Activation('relu')(b)
  # decoder model: CD512-CD1024-CD1024-C1024-C1024-C512-C256-C128
  d1 = decoder_block(b, e7, 512)
  d2 = decoder_block(d1, e6, 512)
  d3 = decoder_block(d2, e5, 512)
  d4 = decoder_block(d3, e4, 512, dropout=False)
  d5 = decoder_block(d4, e3, 256, dropout=False)
  d6 = decoder_block(d5, e2, 128, dropout=False)
  d7 = decoder_block(d6, e1, 64, dropout=False)
  # output
  g = Conv2DTranspose(3, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d7)
  out_image = Activation('tanh')(g)
  # define model
  model = Model(in_image, out_image)
  return model

# define image shape
image_shape = (256,256,3)
# create the model
model = define_generator(image_shape)
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='generator_model_plot.png', show_shapes=True,
    show_layer_names=True)
```

Listing 22.5: Example of defining and summarizing the U-Net generator model.

Running the example first summarizes the model. The output of the model summary was omitted here for brevity. The model has a single input and output, but the skip connections make the summary difficult to read. A plot of the model is created showing much the same information in a graphical form. The model is complex, and the plot helps to understand the skip connections and their impact on the number of filters in the decoder.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

Working backward from the output layer, if we look at the `Concatenate` layers and the first `Conv2DTranspose` layer of the decoder, we can see the number of channels as:

- 128, 256, 512, 1024, 1024, 1024, 1024, 512

Reversing this list gives the stated configuration of the number of filters for each layer in the decoder from the paper of:

- CD512-CD1024-CD1024-C1024-C1024-C512-C256-C128

The model summary and plot where left out here for brevity. Now that we have defined both models, we can look at how the generator model is updated via the discriminator model.

22.5 How to Implement Adversarial and L1 Loss

The discriminator model can be updated directly, whereas the generator model must be updated via the discriminator model. This can be achieved by defining a new composite model in Keras that connects the output of the generator model as input to the discriminator model. The discriminator model can then predict whether a generated image is real or fake. We can update the weights of the composite model in such a way that the generated image has the label of *real* instead of *fake*, which will cause the generator weights to be updated towards generating a better fake image. We can also mark the discriminator weights as not trainable in this context, to avoid the misleading update. Additionally, the generator needs to be updated to better match the targeted translation of the input image. This means that the composite model must also output the generated image directly, allowing it to be compared to the target image. Therefore, we can summarize the inputs and outputs of this composite model as follows:

- **Inputs**: Source image
- **Outputs**: Classification of real/fake, generated target image.

The weights of the generator will be updated via both adversarial loss via the discriminator output and L1 loss via the direct image output. The loss scores are added together, where the L1 loss is treated as a regularizing term and weighted via a hyperparameter called lambda (λ), set to 100.

$$\text{loss} = \text{adversarial loss} + \lambda \times \text{L1 loss} \tag{22.2}$$

The `define_gan()` function below implements this, taking the defined generator and discriminator models as input and creating the composite GAN model that can be used to update the generator model weights. The source image input is provided both to the generator and the discriminator as input and the output of the generator is also connected to the discriminator as input. Two loss functions are specified when the model is compiled for the discriminator and generator outputs respectively. The `loss_weights` argument is used to define the weighting of each loss when added together to update the generator model weights.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model, image_shape):
  # make weights in the discriminator not trainable
  for layer in d_model.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # define the source image
  in_src = Input(shape=image_shape)
  # connect the source image to the generator input
  gen_out = g_model(in_src)
  # connect the source input and generator output to the discriminator input
  dis_out = d_model([in_src, gen_out])
  # src image as input, generated image and classification output
  model = Model(in_src, [dis_out, gen_out])
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'mae'], optimizer=opt, loss_weights=[1,100])
  return model
```

Listing 22.6: Example of a function for defining the composite model for training the generator.

Tying this together with the model definitions from the previous sections, the complete example is listed below.

```
# example of defining a composite model for training the generator model
from keras.optimizers import Adam
from keras.initializers import RandomNormal
from keras.models import Model
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Activation
from keras.layers import Concatenate
from keras.layers import Dropout
from keras.layers import BatchNormalization
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define the discriminator model
def define_discriminator(image_shape):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # source image input
  in_src_image = Input(shape=image_shape)
  # target image input
  in_target_image = Input(shape=image_shape)
  # concatenate images channel-wise
```

```
  merged = Concatenate()([in_src_image, in_target_image])
  # C64
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(merged)
  d = LeakyReLU(alpha=0.2)(d)
  # C128
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C256
  d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C512
  d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # second last output layer
  d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # patch output
  d = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
  patch_out = Activation('sigmoid')(d)
  # define model
  model = Model([in_src_image, in_target_image], patch_out)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, loss_weights=[0.5])
  return model

# define an encoder block
def define_encoder_block(layer_in, n_filters, batchnorm=True):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # add downsampling layer
  g = Conv2D(n_filters, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(layer_in)
  # conditionally add batch normalization
  if batchnorm:
    g = BatchNormalization()(g, training=True)
  # leaky relu activation
  g = LeakyReLU(alpha=0.2)(g)
  return g

# define a decoder block
def decoder_block(layer_in, skip_in, n_filters, dropout=True):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # add upsampling layer
  g = Conv2DTranspose(n_filters, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(layer_in)
  # add batch normalization
  g = BatchNormalization()(g, training=True)
  # conditionally add dropout
  if dropout:
    g = Dropout(0.5)(g, training=True)
```

```
  # merge with skip connection
  g = Concatenate()([g, skip_in])
  # relu activation
  g = Activation('relu')(g)
  return g

# define the standalone generator model
def define_generator(image_shape=(256,256,3)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=image_shape)
  # encoder model: C64-C128-C256-C512-C512-C512-C512-C512
  e1 = define_encoder_block(in_image, 64, batchnorm=False)
  e2 = define_encoder_block(e1, 128)
  e3 = define_encoder_block(e2, 256)
  e4 = define_encoder_block(e3, 512)
  e5 = define_encoder_block(e4, 512)
  e6 = define_encoder_block(e5, 512)
  e7 = define_encoder_block(e6, 512)
  # bottleneck, no batch norm and relu
  b = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(e7)
  b = Activation('relu')(b)
  # decoder model: CD512-CD1024-CD1024-C1024-C1024-C512-C256-C128
  d1 = decoder_block(b, e7, 512)
  d2 = decoder_block(d1, e6, 512)
  d3 = decoder_block(d2, e5, 512)
  d4 = decoder_block(d3, e4, 512, dropout=False)
  d5 = decoder_block(d4, e3, 256, dropout=False)
  d6 = decoder_block(d5, e2, 128, dropout=False)
  d7 = decoder_block(d6, e1, 64, dropout=False)
  # output
  g = Conv2DTranspose(3, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d7)
  out_image = Activation('tanh')(g)
  # define model
  model = Model(in_image, out_image)
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model, image_shape):
  # make weights in the discriminator not trainable
  for layer in d_model.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # define the source image
  in_src = Input(shape=image_shape)
  # connect the source image to the generator input
  gen_out = g_model(in_src)
  # connect the source input and generator output to the discriminator input
  dis_out = d_model([in_src, gen_out])
  # src image as input, generated image and classification output
  model = Model(in_src, [dis_out, gen_out])
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'mae'], optimizer=opt, loss_weights=[1,100])
  return model
```

```
# define image shape
image_shape = (256,256,3)
# define the models
d_model = define_discriminator(image_shape)
g_model = define_generator(image_shape)
# define the composite model
gan_model = define_gan(g_model, d_model, image_shape)
# summarize the model
gan_model.summary()
# plot the model
plot_model(gan_model, to_file='gan_model_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 22.7: Example of defining and summarizing the composite model for training the generator.

Running the example first summarizes the composite model, showing the 256×256 image input, the same shaped output from `model_2` (the generator) and the PatchGAN classification prediction from `model_1` (the discriminator).

```
__________________________________________________________________________________
Layer (type)                 Output Shape       Param #    Connected to
==================================================================================
input_4 (InputLayer)         (None, 256, 256, 3) 0
__________________________________________________________________________________
model_2 (Model)              (None, 256, 256, 3) 54429315  input_4[0][0]
__________________________________________________________________________________
model_1 (Model)              (None, 16, 16, 1)  6968257    input_4[0][0]
                                                          model_2[1][0]
==================================================================================
Total params: 61,397,572
Trainable params: 54,419,459
Non-trainable params: 6,978,113
__________________________________________________________________________________
```

Listing 22.8: Example output from defining and summarizing the composite model for training the generator.

A plot of the composite model is also created, showing how the input image flows into the generator and discriminator, and that the model has two outputs or end-points from each of the two models.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

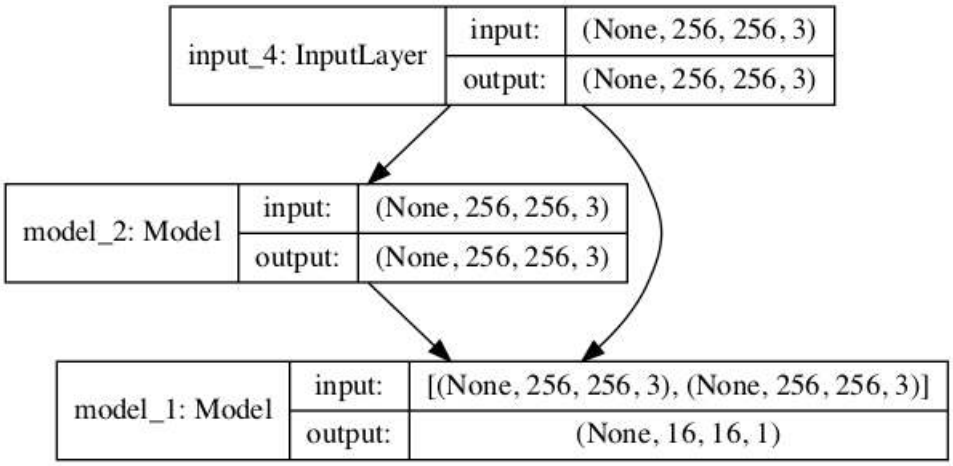

Figure 22.3: Plot of the Composite GAN Model Used to Train the Generator in the Pix2Pix GAN Architecture.

22.6 How to Update Model Weights

Training the defined models is relatively straightforward. First, we must define a helper function that will select a batch of real source and target images and the associated output (1.0). Here, the dataset is a list of two arrays of images.

```
# select a batch of random samples, returns images and target
def generate_real_samples(dataset, n_samples, patch_shape):
  # unpack dataset
  trainA, trainB = dataset
  # choose random instances
  ix = randint(0, trainA.shape[0], n_samples)
  # retrieve selected images
  X1, X2 = trainA[ix], trainB[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, patch_shape, patch_shape, 1))
  return [X1, X2], y
```

Listing 22.9: Example of a function for selecting samples of real images.

Similarly, we need a function to generate a batch of fake images and the associated output (0.0). Here, the samples are an array of source images for which target images will be generated.

```
# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, samples, patch_shape):
  # generate fake instance
  X = g_model.predict(samples)
  # create 'fake' class labels (0)
  y = zeros((len(X), patch_shape, patch_shape, 1))
  return X, y
```

Listing 22.10: Example of a function for generating synthetic images with the generator.

Now, we can define the steps of a single training iteration. First, we must select a batch of source and target images by calling `generate_real_samples()`. Typically, the batch size

(n_batch) is set to 1. In this case, we will assume 256×256 input images, which means the n_patch for the PatchGAN discriminator will be 16 to indicate a 16×16 output feature map.

```
...
# select a batch of real samples
[X_realA, X_realB], y_real = generate_real_samples(dataset, n_batch, n_patch)
```

Listing 22.11: Example of selecting a sample of real images.

Next, we can use the batches of selected real source images to generate corresponding batches of generated or fake target images.

```
...
# generate a batch of fake samples
X_fakeB, y_fake = generate_fake_samples(g_model, X_realA, n_patch)
```

Listing 22.12: Example of generating a sample of synthetic images.

We can then use the real and fake images, as well as their targets, to update the standalone discriminator model.

```
...
# update discriminator for real samples
d_loss1 = d_model.train_on_batch([X_realA, X_realB], y_real)
# update discriminator for generated samples
d_loss2 = d_model.train_on_batch([X_realA, X_fakeB], y_fake)
```

Listing 22.13: Example of updating the PatchGAN discriminator model.

So far, this is normal for updating a GAN in Keras. Next, we can update the generator model via adversarial loss and L1 loss. Recall that the composite GAN model takes a batch of source images as input and predicts first the classification of real/fake and second the generated target. Here, we provide a target to indicate the generated images are *real* ($class = 1$) to the discriminator output of the composite model. The real target images are provided for calculating the L1 loss between them and the generated target images. We have two loss functions, but three loss values calculated for a batch update, where only the first loss value is of interest as it is the weighted sum of the adversarial and L1 loss values for the batch.

```
...
# update the generator
g_loss, _, _ = gan_model.train_on_batch(X_realA, [y_real, X_realB])
```

Listing 22.14: Example of updating the U-Net generator model.

That's all there is to it. We can define all of this in a function called `train()` that takes the defined models and a loaded dataset (as a list of two NumPy arrays) and trains the models.

```
# train pix2pix models
def train(d_model, g_model, gan_model, dataset, n_epochs=100, n_batch=1, n_patch=16):
  # unpack dataset
  trainA, trainB = dataset
  # calculate the number of batches per training epoch
  bat_per_epo = int(len(trainA) / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # manually enumerate epochs
  for i in range(n_steps):
```

```
    # select a batch of real samples
    [X_realA, X_realB], y_real = generate_real_samples(dataset, n_batch, n_patch)
    # generate a batch of fake samples
    X_fakeB, y_fake = generate_fake_samples(g_model, X_realA, n_patch)
    # update discriminator for real samples
    d_loss1 = d_model.train_on_batch([X_realA, X_realB], y_real)
    # update discriminator for generated samples
    d_loss2 = d_model.train_on_batch([X_realA, X_fakeB], y_fake)
    # update the generator
    g_loss, _, _ = gan_model.train_on_batch(X_realA, [y_real, X_realB])
    # summarize performance
    print('>%d, d1[%.3f] d2[%.3f] g[%.3f]' % (i+1, d_loss1, d_loss2, g_loss))
```

Listing 22.15: Example of a function that implements the Pix2Pix training algorithm.

The train function can then be called directly with our defined models and loaded dataset.

```
...
# load image data
dataset = ...
# train model
train(d_model, g_model, gan_model, dataset)
```

Listing 22.16: Example of calling the Pix2Pix training algorithm.

22.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

22.7.1 Official

- Image-to-Image Translation with Conditional Adversarial Networks, 2016.
  https://arxiv.org/abs/1611.07004
- Image-to-Image Translation with Conditional Adversarial Nets, Homepage.
  https://phillipi.github.io/pix2pix/
- Image-to-image translation with conditional adversarial nets, GitHub.
  https://github.com/phillipi/pix2pix
- pytorch-CycleGAN-and-pix2pix, GitHub.
  https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix
- Interactive Image-to-Image Demo, 2017.
  https://affinelayer.com/pixsrv/
- Pix2Pix Datasets
  http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/

22.7.2 API

- Keras Datasets API.
  https://keras.io/datasets/
- Keras Sequential Model API
  https://keras.io/models/sequential/
- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/
- How can I "freeze" Keras layers?
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers

22.7.3 Articles

- Question: PatchGAN Discriminator, 2017.
  https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/issues/39
- `receptive_field_sizes.m`
  https://github.com/phillipi/pix2pix/blob/master/scripts/receptive_field_sizes.m

22.8 Summary

In this tutorial, you discovered how to implement the Pix2Pix GAN architecture from scratch using the Keras deep learning framework. Specifically, you learned:

- How to develop the PatchGAN discriminator model for the Pix2Pix GAN.
- How to develop the U-Net encoder-decoder generator model for the Pix2Pix GAN.
- How to implement the composite model for updating the generator and how to train both models.

22.8.1 Next

In the next tutorial, you will discover how to train a Pix2Pix model to translate satellite images to Google Maps images.

Chapter 23

How to Develop a Pix2Pix End-to-End

The Pix2Pix Generative Adversarial Network, or GAN, is an approach to training a deep convolutional neural network for image-to-image translation tasks. The careful configuration of architecture as a type of image-conditional GAN allows for both the generation of large images compared to prior GAN models (e.g. such as 256×256 pixels) and the capability of performing well on a variety of different image-to-image translation tasks. In this tutorial, you will discover how to develop a Pix2Pix generative adversarial network for image-to-image translation.

After completing this tutorial, you will know:

- How to load and prepare the satellite image to Google maps image-to-image translation dataset.
- How to develop a Pix2Pix model for translating satellite photographs to Google Maps images.
- How to use the final Pix2Pix generator model to translate ad hoc satellite images.

Let's get started.

23.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What Is the Pix2Pix GAN?
2. Satellite to Map Image Translation Dataset
3. How to Develop and Train a Pix2Pix Model
4. How to Translate Images With a Pix2Pix Model
5. How to Translate Google Maps to Satellite Images

23.2 What Is the Pix2Pix GAN?

Pix2Pix is a Generative Adversarial Network, or GAN, model designed for general purpose image-to-image translation. The approach was presented by Phillip Isola, et al. in their 2016 paper titled *Image-to-Image Translation with Conditional Adversarial Networks* and presented at CVPR in 2017 (introduced in Chapter 21). The GAN architecture is comprised of a generator model for outputting new plausible synthetic images, and a discriminator model that classifies images as real (from the dataset) or fake (generated). The discriminator model is updated directly, whereas the generator model is updated via the discriminator model. As such, the two models are trained simultaneously in an adversarial process where the generator seeks to better fool the discriminator and the discriminator seeks to better identify the counterfeit images.

The Pix2Pix model is a type of conditional GAN, or cGAN, where the generation of the output image is conditional on an input, in this case, a source image. The discriminator is provided both with a source image and the target image and must determine whether the target is a plausible transformation of the source image. The generator is trained via adversarial loss, which encourages the generator to generate plausible images in the target domain. The generator is also updated via L1 loss measured between the generated image and the expected output image. This additional loss encourages the generator model to create plausible translations of the source image.

The Pix2Pix GAN has been demonstrated on a range of image-to-image translation tasks such as converting maps to satellite photographs, black and white photographs to color, and sketches of products to product photographs. Now that we are familiar with the Pix2Pix GAN, let's prepare a dataset that we can use with image-to-image translation.

23.3 Satellite to Map Image Translation Dataset

In this tutorial, we will use the so-called *maps* dataset used in the Pix2Pix paper. This is a dataset comprised of satellite images of New York and their corresponding Google maps pages. The image translation problem involves converting satellite photos to Google maps format, or the reverse, Google maps images to Satellite photos. The dataset is provided on the Pix2Pix website and can be downloaded as a 255-megabyte zip file.

- Download Maps Dataset (maps.tar.gz). [1]

Download the dataset and unzip it into your current working directory. This will create a directory called `maps/` with the following structure:

```
maps
├──train
└──val
```

Listing 23.1: Example directory structure for the maps dataset.

The train folder contains 1,097 images, whereas the validation dataset contains 1,099 images. Images have a digit filename and are in JPEG format. Each image is 1,200 pixels wide and 600 pixels tall and contains both the satellite image on the left and the Google maps image on the right.

[1] http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/maps.tar.gz

Figure 23.1: Sample Image From the Maps Dataset Including Both Satellite and Google Maps Image.

We can prepare this dataset for training a Pix2Pix GAN model in Keras. We will just work with the images in the training dataset. Each image will be loaded, rescaled, and split into the satellite and Google Maps elements. The result will be 1,097 color image pairs with the width and height of 256×256 pixels. The `load_images()` function below implements this. It enumerates the list of images in a given directory, loads each with the target size of 256×512 pixels, splits each image into satellite and map elements and returns an array of each.

```
# load all images in a directory into memory
def load_images(path, size=(256,512)):
  src_list, tar_list = list(), list()
  # enumerate filenames in directory, assume all are images
  for filename in listdir(path):
    # load and resize the image
    pixels = load_img(path + filename, target_size=size)
    # convert to an array
    pixels = img_to_array(pixels)
    # split into satellite and map
    sat_img, map_img = pixels[:, :256], pixels[:, 256:]
    src_list.append(sat_img)
    tar_list.append(map_img)
  return [asarray(src_list), asarray(tar_list)]
```

Listing 23.2: Example of a function for loading, scaling and splitting the map images.

We can call this function with the path to the training dataset. Once loaded, we can save the prepared arrays to a new file in compressed format for later use. The complete example is listed below.

```
# load, split and scale the maps dataset ready for training
from os import listdir
from numpy import asarray
from keras.preprocessing.image import img_to_array
```

```
from keras.preprocessing.image import load_img
from numpy import savez_compressed

# load all images in a directory into memory
def load_images(path, size=(256,512)):
  src_list, tar_list = list(), list()
  # enumerate filenames in directory, assume all are images
  for filename in listdir(path):
    # load and resize the image
    pixels = load_img(path + filename, target_size=size)
    # convert to numpy array
    pixels = img_to_array(pixels)
    # split into satellite and map
    sat_img, map_img = pixels[:, :256], pixels[:, 256:]
    src_list.append(sat_img)
    tar_list.append(map_img)
  return [asarray(src_list), asarray(tar_list)]

# dataset path
path = 'maps/train/'
# load dataset
[src_images, tar_images] = load_images(path)
print('Loaded: ', src_images.shape, tar_images.shape)
# save as compressed numpy array
filename = 'maps_256.npz'
savez_compressed(filename, src_images, tar_images)
print('Saved dataset: ', filename)
```

Listing 23.3: Example of preparing and saving the maps dataset.

Running the example loads all images in the training dataset, summarizes their shape to ensure the images were loaded correctly, then saves the arrays to a new file called `maps_256.npz` in compressed NumPy array format.

```
Loaded: (1096, 256, 256, 3) (1096, 256, 256, 3)
Saved dataset: maps_256.npz
```

Listing 23.4: Example output from preparing and saving the maps dataset.

This file can be loaded later via the `load()` NumPy function and retrieving each array in turn. We can then plot some images pairs to confirm the data has been handled correctly.

```
# load the prepared dataset
from numpy import load
from matplotlib import pyplot
# load the dataset
data = load('maps_256.npz')
src_images, tar_images = data['arr_0'], data['arr_1']
print('Loaded: ', src_images.shape, tar_images.shape)
# plot source images
n_samples = 3
for i in range(n_samples):
  pyplot.subplot(2, n_samples, 1 + i)
  pyplot.axis('off')
  pyplot.imshow(src_images[i].astype('uint8'))
# plot target image
for i in range(n_samples):
```

```
  pyplot.subplot(2, n_samples, 1 + n_samples + i)
  pyplot.axis('off')
  pyplot.imshow(tar_images[i].astype('uint8'))
pyplot.show()
```

Listing 23.5: Example of loading and plotting the prepared dataset.

Running this example loads the prepared dataset and summarizes the shape of each array, confirming our expectations of a little over one thousand 256×256 image pairs.

```
Loaded: (1096, 256, 256, 3) (1096, 256, 256, 3)
```

Listing 23.6: Example output from loading and plotting the prepared dataset.

A plot of three image pairs is also created showing the satellite images on the top and Google Maps images on the bottom. We can see that satellite images are quite complex and that although the Google Maps images are much simpler, they have color codings for things like major roads, water, and parks.

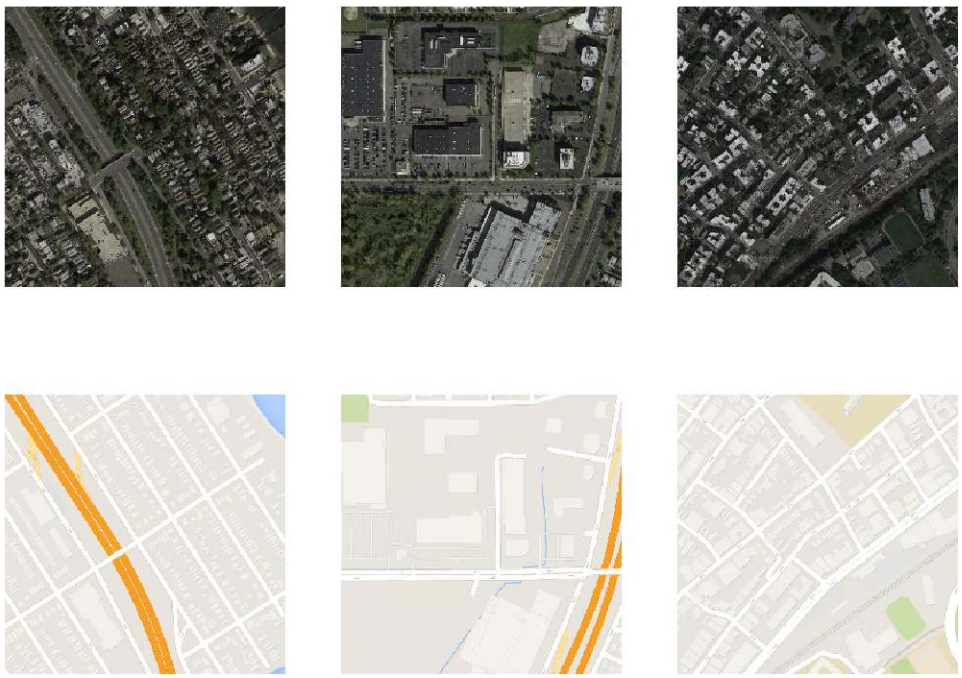

Figure 23.2: Plot of Three Image Pairs Showing Satellite Images (top) and Google Maps Images (bottom).

Now that we have prepared the dataset for image translation, we can develop our Pix2Pix GAN model.

23.4 How to Develop and Train a Pix2Pix Model

In this section, we will develop the Pix2Pix model for translating satellite photos to Google maps images. The same model architecture and configuration described in the paper was used across a range of image translation tasks. This architecture is both described in the body of the paper, with additional detail in the appendix of the paper, and a fully working implementation provided as open source with the Torch deep learning framework. The implementation in this section will use the Keras deep learning framework based directly on the model described in the paper and implemented in the author's code base, designed to take and generate color images with the size 256×256 pixels (model implementation was covered in Chapter 22). The architecture is comprised of two models: the discriminator and the generator.

The discriminator is a deep convolutional neural network that performs image classification. Specifically, conditional-image classification. It takes both the source image (e.g. satellite photo) and the target image (e.g. Google maps image) as input and predicts the likelihood of whether target image is a real or fake translation of the source image. The discriminator design is based on the effective receptive field of the model, which defines the relationship between one output of the model to the number of pixels in the input image. This is called a PatchGAN model and is carefully designed so that each output prediction of the model maps to a 70×70 square or patch of the input image. The benefit of this approach is that the same model can be applied to input images of different sizes, e.g. larger or smaller than 256×256 pixels. The output of the model depends on the size of the input image but may be one value or a square activation map of values. Each value is a probability for the likelihood that a patch in the input image is real. These values can be averaged to give an overall likelihood or classification score if needed.

The `define_discriminator()` function below implements the 70×70 PatchGAN discriminator model as per the design of the model in the paper. The model takes two input images that are concatenated together and predicts a patch output of predictions. The model is optimized using binary cross-entropy, and a weighting is used so that updates to the model have half (0.5) the usual effect. The authors of Pix2Pix recommend this weighting of model updates to slow down changes to the discriminator, relative to the generator model during training.

```
# define the discriminator model
def define_discriminator(image_shape):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # source image input
  in_src_image = Input(shape=image_shape)
  # target image input
  in_target_image = Input(shape=image_shape)
  # concatenate images channel-wise
  merged = Concatenate()([in_src_image, in_target_image])
  # C64
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(merged)
  d = LeakyReLU(alpha=0.2)(d)
  # C128
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C256
  d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
```

```
  # C512
  d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # second last output layer
  d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # patch output
  d = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
  patch_out = Activation('sigmoid')(d)
  # define model
  model = Model([in_src_image, in_target_image], patch_out)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, loss_weights=[0.5])
  return model
```

Listing 23.7: Example of a function for defining the PatchGAN discriminator.

The generator model is more complex than the discriminator model. The generator is an encoder-decoder model using a U-Net architecture. The model takes a source image (e.g. satellite photo) and generates a target image (e.g. Google maps image). It does this by first downsampling or encoding the input image down to a bottleneck layer, then upsampling or decoding the bottleneck representation to the size of the output image. The U-Net architecture means that skip-connections are added between the encoding layers and the corresponding decoding layers, forming a U-shape. The encoder and decoder of the generator are comprised of standardized blocks of convolutional, batch normalization, dropout, and activation layers. This standardization means that we can develop helper functions to create each block of layers and call it repeatedly to build-up the encoder and decoder parts of the model.

The `define_generator()` function below implements the U-Net encoder-decoder generator model. It uses the `define_encoder_block()` helper function to create blocks of layers for the encoder and the `decoder_block()` function to create blocks of layers for the decoder. The Tanh activation function is used in the output layer, meaning that pixel values in the generated image will be in the range [-1,1].

```
# define an encoder block
def define_encoder_block(layer_in, n_filters, batchnorm=True):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # add downsampling layer
  g = Conv2D(n_filters, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(layer_in)
  # conditionally add batch normalization
  if batchnorm:
    g = BatchNormalization()(g, training=True)
  # leaky relu activation
  g = LeakyReLU(alpha=0.2)(g)
  return g

# define a decoder block
def decoder_block(layer_in, skip_in, n_filters, dropout=True):
  # weight initialization
  init = RandomNormal(stddev=0.02)
```

```
  # add upsampling layer
  g = Conv2DTranspose(n_filters, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(layer_in)
  # add batch normalization
  g = BatchNormalization()(g, training=True)
  # conditionally add dropout
  if dropout:
    g = Dropout(0.5)(g, training=True)
  # merge with skip connection
  g = Concatenate()([g, skip_in])
  # relu activation
  g = Activation('relu')(g)
  return g

# define the standalone generator model
def define_generator(image_shape=(256,256,3)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=image_shape)
  # encoder model
  e1 = define_encoder_block(in_image, 64, batchnorm=False)
  e2 = define_encoder_block(e1, 128)
  e3 = define_encoder_block(e2, 256)
  e4 = define_encoder_block(e3, 512)
  e5 = define_encoder_block(e4, 512)
  e6 = define_encoder_block(e5, 512)
  e7 = define_encoder_block(e6, 512)
  # bottleneck, no batch norm and relu
  b = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(e7)
  b = Activation('relu')(b)
  # decoder model
  d1 = decoder_block(b, e7, 512)
  d2 = decoder_block(d1, e6, 512)
  d3 = decoder_block(d2, e5, 512)
  d4 = decoder_block(d3, e4, 512, dropout=False)
  d5 = decoder_block(d4, e3, 256, dropout=False)
  d6 = decoder_block(d5, e2, 128, dropout=False)
  d7 = decoder_block(d6, e1, 64, dropout=False)
  # output
  g = Conv2DTranspose(3, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d7)
  out_image = Activation('tanh')(g)
  # define model
  model = Model(in_image, out_image)
  return model
```

Listing 23.8: Example of functions for defining the U-Net generator.

The discriminator model is trained directly on real and generated images, whereas the generator model is not. Instead, the generator model is trained via the discriminator model. It is updated to minimize the loss predicted by the discriminator for generated images marked as *real*. As such, it is encouraged to generate more realistic images. The generator is also updated to minimize the L1 loss or mean absolute error between the generated image and the target image. The generator is updated via a weighted sum of both the adversarial loss and the L1 loss, where the authors of the model recommend a weighting of 100 to 1 in favor of the L1 loss.

This is to encourage the generator strongly toward generating plausible translations of the input image, and not just plausible images in the target domain.

This can be achieved by defining a new logical model comprised of the weights in the existing standalone generator and discriminator model. This logical or composite model involves stacking the generator on top of the discriminator. A source image is provided as input to the generator and to the discriminator, although the output of the generator is connected to the discriminator as the corresponding *target* image. The discriminator then predicts the likelihood that the generated image was a real translation of the source image. The discriminator is updated in a standalone manner, so the weights are reused in this composite model but are marked as not trainable. The composite model is updated with two targets, one indicating that the generated images were real (cross-entropy loss), forcing large weight updates in the generator toward generating more realistic images, and the executed real translation of the image, which is compared against the output of the generator model (L1 loss). The `define_gan()` function below implements this, taking the already-defined generator and discriminator models as arguments and using the Keras functional API to connect them together into a composite model. Both loss functions are specified for the two outputs of the model and the weights used for each are specified in the `loss_weights` argument to the `compile()` function.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model, image_shape):
  # make weights in the discriminator not trainable
  for layer in d_model.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # define the source image
  in_src = Input(shape=image_shape)
  # connect the source image to the generator input
  gen_out = g_model(in_src)
  # connect the source input and generator output to the discriminator input
  dis_out = d_model([in_src, gen_out])
  # src image as input, generated image and classification output
  model = Model(in_src, [dis_out, gen_out])
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'mae'], optimizer=opt, loss_weights=[1,100])
  return model
```

Listing 23.9: Example of a function for defining the composite model for training the generator.

Next, we can load our paired images dataset in compressed NumPy array format. This will return a list of two NumPy arrays: the first for source images and the second for corresponding target images.

```
# load and prepare training images
def load_real_samples(filename):
  # load the compressed arrays
  data = load(filename)
  # unpack the arrays
  X1, X2 = data['arr_0'], data['arr_1']
  # scale from [0,255] to [-1,1]
  X1 = (X1 - 127.5) / 127.5
  X2 = (X2 - 127.5) / 127.5
  return [X1, X2]
```

Listing 23.10: Example of a function for loading the prepared dataset.

Training the discriminator will require batches of real and fake images. The `generate_real_samples()` function below will prepare a batch of random pairs of images from the training dataset, and the corresponding discriminator label of $class = 1$ to indicate they are real.

```
# select a batch of random samples, returns images and target
def generate_real_samples(dataset, n_samples, patch_shape):
  # unpack dataset
  trainA, trainB = dataset
  # choose random instances
  ix = randint(0, trainA.shape[0], n_samples)
  # retrieve selected images
  X1, X2 = trainA[ix], trainB[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, patch_shape, patch_shape, 1))
  return [X1, X2], y
```

Listing 23.11: Example of a function for selecting a sample of real images.

The `generate_fake_samples()` function below uses the generator model and a batch of real source images to generate an equivalent batch of target images for the discriminator. These are returned with the label $class = 0$ to indicate to the discriminator that they are fake.

```
# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, samples, patch_shape):
  # generate fake instance
  X = g_model.predict(samples)
  # create 'fake' class labels (0)
  y = zeros((len(X), patch_shape, patch_shape, 1))
  return X, y
```

Listing 23.12: Example of a function for generating synthetic images with the generator.

Typically, GAN models do not converge; instead, an equilibrium is found between the generator and discriminator models. As such, we cannot easily judge when training should stop. Therefore, we can save the model and use it to generate sample image-to-image translations periodically during training, such as every 10 training epochs. We can then review the generated images at the end of training and use the image quality to choose a final model. The `summarize_performance()` function implements this, taking the generator model at a point during training and using it to generate a number, in this case three, of translations of randomly selected images in the dataset. The source, generated image, and expected target are then plotted as three rows of images and the plot saved to file. Additionally, the model is saved to an H5 formatted file that makes it easier to load later. Both the image and model filenames include the training iteration number, allowing us to easily tell them apart at the end of training.

```
# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, dataset, n_samples=3):
  # select a sample of input images
  [X_realA, X_realB], _ = generate_real_samples(dataset, n_samples, 1)
  # generate a batch of fake samples
  X_fakeB, _ = generate_fake_samples(g_model, X_realA, 1)
  # scale all pixels from [-1,1] to [0,1]
  X_realA = (X_realA + 1) / 2.0
```

```
  X_realB = (X_realB + 1) / 2.0
  X_fakeB = (X_fakeB + 1) / 2.0
  # plot real source images
  for i in range(n_samples):
    pyplot.subplot(3, n_samples, 1 + i)
    pyplot.axis('off')
    pyplot.imshow(X_realA[i])
  # plot generated target image
  for i in range(n_samples):
    pyplot.subplot(3, n_samples, 1 + n_samples + i)
    pyplot.axis('off')
    pyplot.imshow(X_fakeB[i])
  # plot real target image
  for i in range(n_samples):
    pyplot.subplot(3, n_samples, 1 + n_samples*2 + i)
    pyplot.axis('off')
    pyplot.imshow(X_realB[i])
  # save plot to file
  filename1 = 'plot_%06d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # save the generator model
  filename2 = 'model_%06d.h5' % (step+1)
  g_model.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))
```

Listing 23.13: Example of a function for summarizing model performance and saving the generator.

Finally, we can train the generator and discriminator models. The `train()` function below implements this, taking the defined generator, discriminator, composite model, and loaded dataset as input. The number of epochs is set at 100 to keep training times down, although 200 was used in the paper. A batch size of 1 is used as is recommended in the paper. Training involves a fixed number of training iterations. There are 1,097 images in the training dataset. One epoch is one iteration through this number of examples, with a batch size of one means 1,097 training steps. The generator is saved and evaluated every 10 epochs or every 10,097 training steps, and the model will run for 100 epochs, or a total of 100,097 training steps. Each training step involves first selecting a batch of real examples, then using the generator to generate a batch of matching fake samples using the real source images. The discriminator is then updated with the batch of real images and then fake images.

Next, the generator model is updated providing the real source images as input and providing class labels of 1 (real) and the real target images as the expected outputs of the model required for calculating loss. The generator has two loss scores as well as the weighted sum score returned from the call to `train_on_batch()`. We are only interested in the weighted sum score (the first value returned) as it is used to update the model weights. Finally, the loss for each update is reported to the console each training iteration and model performance is evaluated every 10 training epochs.

```
# train pix2pix model
def train(d_model, g_model, gan_model, dataset, n_epochs=100, n_batch=1):
  # determine the output square shape of the discriminator
  n_patch = d_model.output_shape[1]
  # unpack dataset
```

```
  trainA, trainB = dataset
  # calculate the number of batches per training epoch
  bat_per_epo = int(len(trainA) / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # manually enumerate epochs
  for i in range(n_steps):
    # select a batch of real samples
    [X_realA, X_realB], y_real = generate_real_samples(dataset, n_batch, n_patch)
    # generate a batch of fake samples
    X_fakeB, y_fake = generate_fake_samples(g_model, X_realA, n_patch)
    # update discriminator for real samples
    d_loss1 = d_model.train_on_batch([X_realA, X_realB], y_real)
    # update discriminator for generated samples
    d_loss2 = d_model.train_on_batch([X_realA, X_fakeB], y_fake)
    # update the generator
    g_loss, _, _ = gan_model.train_on_batch(X_realA, [y_real, X_realB])
    # summarize performance
    print('>%d, d1[%.3f] d2[%.3f] g[%.3f]' % (i+1, d_loss1, d_loss2, g_loss))
    # summarize model performance
    if (i+1) % (bat_per_epo * 10) == 0:
      summarize_performance(i, g_model, dataset)
```

Listing 23.14: Example of a function for training the Pix2Pix GAN models.

Tying all of this together, the complete code example of training a Pix2Pix GAN to translate satellite photos to Google maps images is listed below.

```
# example of pix2pix gan for satellite to map image-to-image translation
from numpy import load
from numpy import zeros
from numpy import ones
from numpy.random import randint
from keras.optimizers import Adam
from keras.initializers import RandomNormal
from keras.models import Model
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Activation
from keras.layers import Concatenate
from keras.layers import Dropout
from keras.layers import BatchNormalization
from keras.layers import LeakyReLU
from matplotlib import pyplot

# define the discriminator model
def define_discriminator(image_shape):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # source image input
  in_src_image = Input(shape=image_shape)
  # target image input
  in_target_image = Input(shape=image_shape)
  # concatenate images channel-wise
  merged = Concatenate()([in_src_image, in_target_image])
```

```
  # C64
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(merged)
  d = LeakyReLU(alpha=0.2)(d)
  # C128
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C256
  d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C512
  d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # second last output layer
  d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)
  # patch output
  d = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
  patch_out = Activation('sigmoid')(d)
  # define model
  model = Model([in_src_image, in_target_image], patch_out)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, loss_weights=[0.5])
  return model

# define an encoder block
def define_encoder_block(layer_in, n_filters, batchnorm=True):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # add downsampling layer
  g = Conv2D(n_filters, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(layer_in)
  # conditionally add batch normalization
  if batchnorm:
    g = BatchNormalization()(g, training=True)
  # leaky relu activation
  g = LeakyReLU(alpha=0.2)(g)
  return g

# define a decoder block
def decoder_block(layer_in, skip_in, n_filters, dropout=True):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # add upsampling layer
  g = Conv2DTranspose(n_filters, (4,4), strides=(2,2), padding='same',
      kernel_initializer=init)(layer_in)
  # add batch normalization
  g = BatchNormalization()(g, training=True)
  # conditionally add dropout
  if dropout:
    g = Dropout(0.5)(g, training=True)
  # merge with skip connection
```

```
  g = Concatenate()([g, skip_in])
  # relu activation
  g = Activation('relu')(g)
  return g

# define the standalone generator model
def define_generator(image_shape=(256,256,3)):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=image_shape)
  # encoder model
  e1 = define_encoder_block(in_image, 64, batchnorm=False)
  e2 = define_encoder_block(e1, 128)
  e3 = define_encoder_block(e2, 256)
  e4 = define_encoder_block(e3, 512)
  e5 = define_encoder_block(e4, 512)
  e6 = define_encoder_block(e5, 512)
  e7 = define_encoder_block(e6, 512)
  # bottleneck, no batch norm and relu
  b = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(e7)
  b = Activation('relu')(b)
  # decoder model
  d1 = decoder_block(b, e7, 512)
  d2 = decoder_block(d1, e6, 512)
  d3 = decoder_block(d2, e5, 512)
  d4 = decoder_block(d3, e4, 512, dropout=False)
  d5 = decoder_block(d4, e3, 256, dropout=False)
  d6 = decoder_block(d5, e2, 128, dropout=False)
  d7 = decoder_block(d6, e1, 64, dropout=False)
  # output
  g = Conv2DTranspose(3, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d7)
  out_image = Activation('tanh')(g)
  # define model
  model = Model(in_image, out_image)
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model, image_shape):
  # make weights in the discriminator not trainable
  for layer in d_model.layers:
    if not isinstance(layer, BatchNormalization):
      layer.trainable = False
  # define the source image
  in_src = Input(shape=image_shape)
  # connect the source image to the generator input
  gen_out = g_model(in_src)
  # connect the source input and generator output to the discriminator input
  dis_out = d_model([in_src, gen_out])
  # src image as input, generated image and classification output
  model = Model(in_src, [dis_out, gen_out])
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss=['binary_crossentropy', 'mae'], optimizer=opt, loss_weights=[1,100])
  return model
```

```
# load and prepare training images
def load_real_samples(filename):
  # load the compressed arrays
  data = load(filename)
  # unpack the arrays
  X1, X2 = data['arr_0'], data['arr_1']
  # scale from [0,255] to [-1,1]
  X1 = (X1 - 127.5) / 127.5
  X2 = (X2 - 127.5) / 127.5
  return [X1, X2]

# select a batch of random samples, returns images and target
def generate_real_samples(dataset, n_samples, patch_shape):
  # unpack dataset
  trainA, trainB = dataset
  # choose random instances
  ix = randint(0, trainA.shape[0], n_samples)
  # retrieve selected images
  X1, X2 = trainA[ix], trainB[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, patch_shape, patch_shape, 1))
  return [X1, X2], y

# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, samples, patch_shape):
  # generate fake instance
  X = g_model.predict(samples)
  # create 'fake' class labels (0)
  y = zeros((len(X), patch_shape, patch_shape, 1))
  return X, y

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, dataset, n_samples=3):
  # select a sample of input images
  [X_realA, X_realB], _ = generate_real_samples(dataset, n_samples, 1)
  # generate a batch of fake samples
  X_fakeB, _ = generate_fake_samples(g_model, X_realA, 1)
  # scale all pixels from [-1,1] to [0,1]
  X_realA = (X_realA + 1) / 2.0
  X_realB = (X_realB + 1) / 2.0
  X_fakeB = (X_fakeB + 1) / 2.0
  # plot real source images
  for i in range(n_samples):
    pyplot.subplot(3, n_samples, 1 + i)
    pyplot.axis('off')
    pyplot.imshow(X_realA[i])
  # plot generated target image
  for i in range(n_samples):
    pyplot.subplot(3, n_samples, 1 + n_samples + i)
    pyplot.axis('off')
    pyplot.imshow(X_fakeB[i])
  # plot real target image
  for i in range(n_samples):
    pyplot.subplot(3, n_samples, 1 + n_samples*2 + i)
    pyplot.axis('off')
    pyplot.imshow(X_realB[i])
```

```
  # save plot to file
  filename1 = 'plot_%06d.png' % (step+1)
  pyplot.savefig(filename1)
  pyplot.close()
  # save the generator model
  filename2 = 'model_%06d.h5' % (step+1)
  g_model.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))

# train pix2pix models
def train(d_model, g_model, gan_model, dataset, n_epochs=100, n_batch=1):
  # determine the output square shape of the discriminator
  n_patch = d_model.output_shape[1]
  # unpack dataset
  trainA, trainB = dataset
  # calculate the number of batches per training epoch
  bat_per_epo = int(len(trainA) / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # manually enumerate epochs
  for i in range(n_steps):
    # select a batch of real samples
    [X_realA, X_realB], y_real = generate_real_samples(dataset, n_batch, n_patch)
    # generate a batch of fake samples
    X_fakeB, y_fake = generate_fake_samples(g_model, X_realA, n_patch)
    # update discriminator for real samples
    d_loss1 = d_model.train_on_batch([X_realA, X_realB], y_real)
    # update discriminator for generated samples
    d_loss2 = d_model.train_on_batch([X_realA, X_fakeB], y_fake)
    # update the generator
    g_loss, _, _ = gan_model.train_on_batch(X_realA, [y_real, X_realB])
    # summarize performance
    print('>%d, d1[%.3f] d2[%.3f] g[%.3f]' % (i+1, d_loss1, d_loss2, g_loss))
    # summarize model performance
    if (i+1) % (bat_per_epo * 10) == 0:
      summarize_performance(i, g_model, dataset)

# load image data
dataset = load_real_samples('maps_256.npz')
print('Loaded', dataset[0].shape, dataset[1].shape)
# define input shape based on the loaded dataset
image_shape = dataset[0].shape[1:]
# define the models
d_model = define_discriminator(image_shape)
g_model = define_generator(image_shape)
# define the composite model
gan_model = define_gan(g_model, d_model, image_shape)
# train model
train(d_model, g_model, gan_model, dataset)
```

Listing 23.15: Example of training the Pix2Pix GAN on the prepared maps dataset.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

The loss is reported each training iteration, including the discriminator loss on real examples ($d1$), discriminator loss on generated or fake examples ($d2$), and generator loss, which is a weighted average of adversarial and L1 loss (g). If loss for the discriminator goes to zero and stays there for a long time, consider re-starting the training run as it is an example of a training failure.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
>1, d1[0.566] d2[0.520] g[82.266]
>2, d1[0.469] d2[0.484] g[66.813]
>3, d1[0.428] d2[0.477] g[79.520]
>4, d1[0.362] d2[0.405] g[78.143]
>5, d1[0.416] d2[0.406] g[72.452]
...
>109596, d1[0.303] d2[0.006] g[5.792]
>109597, d1[0.001] d2[1.127] g[14.343]
>109598, d1[0.000] d2[0.381] g[11.851]
>109599, d1[1.289] d2[0.547] g[6.901]
>109600, d1[0.437] d2[0.005] g[10.460]
>Saved: plot_109600.png and model_109600.h5
```

Listing 23.16: Example output from training the Pix2Pix GAN on the prepared maps dataset.

Models are saved every 10 epochs and saved to a file with the training iteration number. Additionally, images are generated every 10 epochs and compared to the expected target images. These plots can be assessed at the end of the run and used to select a final generator model based on generated image quality. At the end of the run, will you will have 10 saved model files and 10 plots of generated images. After the first 10 epochs, map images are generated that look plausible, although the lines for streets are not entirely straight and images contain some blurring. Nevertheless, large structures are in the right places with mostly the right colors.

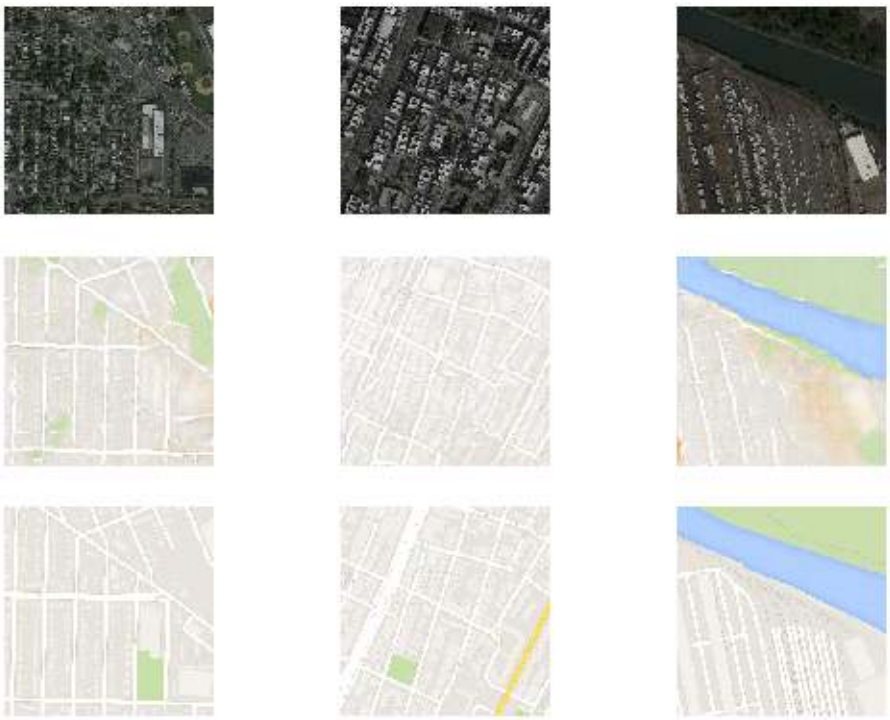

Figure 23.3: Plot of Satellite to Google Maps Translated Images Using Pix2Pix After 10 Training Epochs.

Generated images after about 50 training epochs begin to look very realistic, and the quality appears to remain good for the remainder of the training process. The first generated image example below (left column, middle row) includes more useful detail than the real Google Maps image.

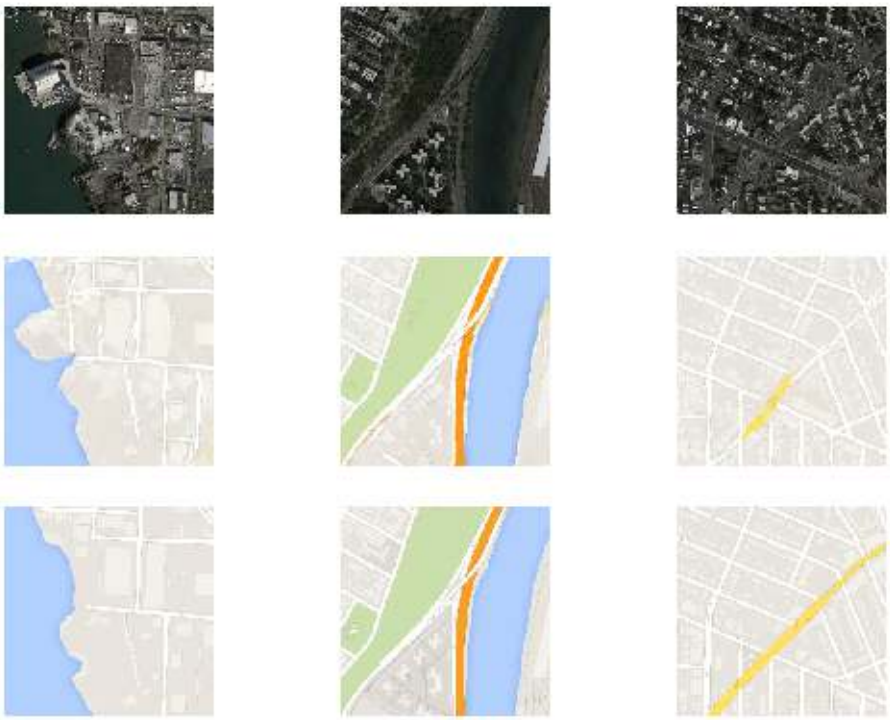

Figure 23.4: Plot of Satellite to Google Maps Translated Images Using Pix2Pix After 100 Training Epochs.

Now that we have developed and trained the Pix2Pix model, we can explore how they can be used in a standalone manner.

23.5 How to Translate Images With a Pix2Pix Model

Training the Pix2Pix model results in many saved models and samples of generated images for each. More training epochs does not necessarily mean a better quality model. Therefore, we can choose a model based on the quality of the generated images and use it to perform ad hoc image-to-image translation. In this case, we will use the model saved at the end of the run, e.g. after 10 0epochs or 109,600 training iterations. A good starting point is to load the model and use it to make ad hoc translations of source images in the training dataset. First, we can load the training dataset. We can use the same function named `load_real_samples()` for loading the dataset as was used when training the model.

```
# load and prepare training images
def load_real_samples(filename):
  # load the compressed arrays
  data = load(filename)
  # unpack the arrays
  X1, X2 = data['arr_0'], data['arr_1']
```

```
  # scale from [0,255] to [-1,1]
  X1 = (X1 - 127.5) / 127.5
  X2 = (X2 - 127.5) / 127.5
  return [X1, X2]
```

Listing 23.17: Example of a function for loading the prepared dataset.

This function can be called as follows:

```
...
# load dataset
[X1, X2] = load_real_samples('maps_256.npz')
print('Loaded', X1.shape, X2.shape)
```

Listing 23.18: Example of loading the prepared dataset.

Next, we can load the saved Keras model.

```
...
# load model
model = load_model('model_109600.h5')
```

Listing 23.19: Example of loading the saved generator model.

Next, we can choose a random image pair from the training dataset to use as an example.

```
...
# select random example
ix = randint(0, len(X1), 1)
src_image, tar_image = X1[ix], X2[ix]
```

Listing 23.20: Example of choosing a random image pair from the dataset.

We can provide the source satellite image as input to the model and use it to predict a Google Maps image.

```
...
# generate image from source
gen_image = model.predict(src_image)
```

Listing 23.21: Example of generating translated version of the image.

Finally, we can plot the source, generated image, and the expected target image. The `plot_images()` function below implements this, providing a nice title above each image.

```
# plot source, generated and target images
def plot_images(src_img, gen_img, tar_img):
  images = vstack((src_img, gen_img, tar_img))
  # scale from [-1,1] to [0,1]
  images = (images + 1) / 2.0
  titles = ['Source', 'Generated', 'Expected']
  # plot images row by row
  for i in range(len(images)):
    # define subplot
    pyplot.subplot(1, 3, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(images[i])
```

```
    # show title
    pyplot.title(titles[i])
  pyplot.show()
```

Listing 23.22: Example of a function for plotting generated images.

This function can be called with each of our source, generated, and target images.

```
...
# plot all three images
plot_images(src_image, gen_image, tar_image)
```

Listing 23.23: Example of plotting images.

Tying all of this together, the complete example of performing an ad hoc image-to-image translation with an example from the training dataset is listed below.

```
# example of loading a pix2pix model and using it for image to image translation
from keras.models import load_model
from numpy import load
from numpy import vstack
from matplotlib import pyplot
from numpy.random import randint

# load and prepare training images
def load_real_samples(filename):
  # load the compressed arrays
  data = load(filename)
  # unpack the arrays
  X1, X2 = data['arr_0'], data['arr_1']
  # scale from [0,255] to [-1,1]
  X1 = (X1 - 127.5) / 127.5
  X2 = (X2 - 127.5) / 127.5
  return [X1, X2]

# plot source, generated and target images
def plot_images(src_img, gen_img, tar_img):
  images = vstack((src_img, gen_img, tar_img))
  # scale from [-1,1] to [0,1]
  images = (images + 1) / 2.0
  titles = ['Source', 'Generated', 'Expected']
  # plot images row by row
  for i in range(len(images)):
    # define subplot
    pyplot.subplot(1, 3, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(images[i])
    # show title
    pyplot.title(titles[i])
  pyplot.show()

# load dataset
[X1, X2] = load_real_samples('maps_256.npz')
print('Loaded', X1.shape, X2.shape)
# load model
```

```
model = load_model('model_109600.h5')
# select random example
ix = randint(0, len(X1), 1)
src_image, tar_image = X1[ix], X2[ix]
# generate image from source
gen_image = model.predict(src_image)
# plot all three images
plot_images(src_image, gen_image, tar_image)
```

Listing 23.24: Example of using the saved generator to translate images.

Running the example will select a random image from the training dataset, translate it to a Google Maps, and plot the result compared to the expected image.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the generated image captures large roads with orange and yellow as well as green park areas. The generated image is not perfect but is very close to the expected image.

Figure 23.5: Plot of Satellite to Google Maps Image Translation With Final Pix2Pix GAN Model.

We may also want to use the model to translate a given standalone image. We can select an image from the validation dataset under `maps/val/` and crop the satellite element of the image. This can then be saved and used as input to the model. In this case, we will use `maps/val/1.jpg`.

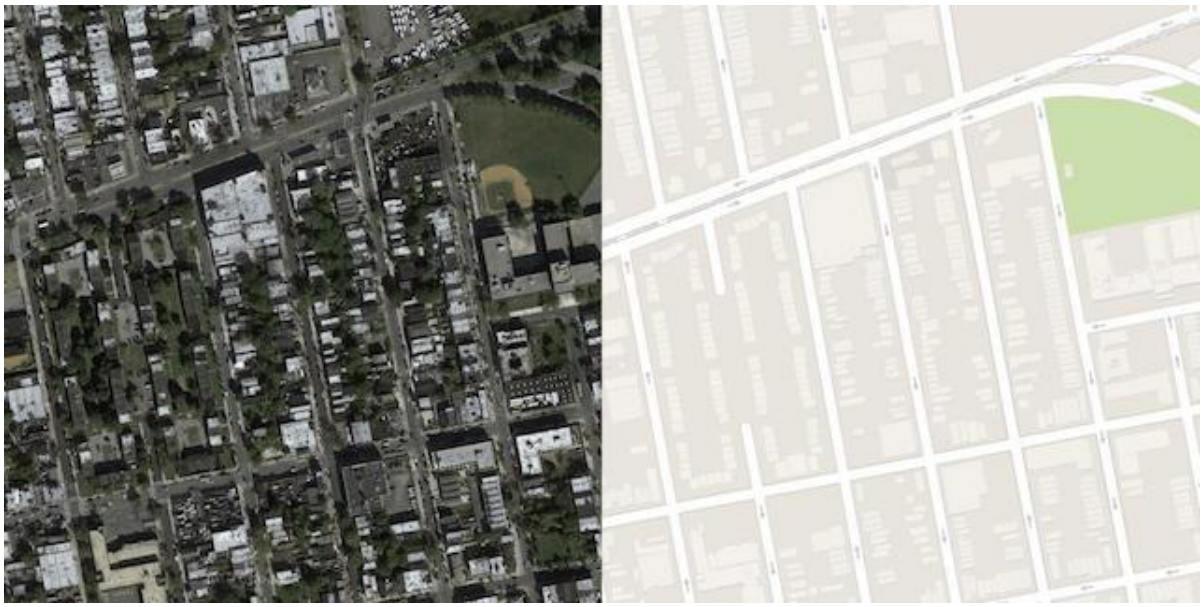

Figure 23.6: Example Image From the Validation Part of the Maps Dataset.

We can use an image program to create a rough crop of the satellite element of this image to use as input and save the file as `satellite.jpg` in the current working directory.

Figure 23.7: Example of a Cropped Satellite Image to Use as Input to the Pix2Pix Model.

We must load the image as a NumPy array of pixels with the size of 256×256, rescale the

pixel values to the range [-1,1], and then expand the single image dimensions to represent one input sample. The `load_image()` function below implements this, returning image pixels that can be provided directly to a loaded Pix2Pix model.

```
# load an image
def load_image(filename, size=(256,256)):
  # load image with the preferred size
  pixels = load_img(filename, target_size=size)
  # convert to an array
  pixels = img_to_array(pixels)
  # scale from [0,255] to [-1,1]
  pixels = (pixels - 127.5) / 127.5
  # reshape to 1 sample
  pixels = expand_dims(pixels, 0)
  return pixels
```

Listing 23.25: Example of a function for loading and preparing an image for translation.

We can then load our cropped satellite image.

```
...
# load source image
src_image = load_image('satellite.jpg')
print('Loaded', src_image.shape)
```

Listing 23.26: Example of loading and preparing an image.

As before, we can load our saved Pix2Pix generator model and generate a translation of the loaded image.

```
...
# load model
model = load_model('model_109600.h5')
# generate image from source
gen_image = model.predict(src_image)
```

Listing 23.27: Example of loading the generator and translating an image.

Finally, we can scale the pixel values back to the range [0,1] and plot the result.

```
...
# scale from [-1,1] to [0,1]
gen_image = (gen_image + 1) / 2.0
# plot the image
pyplot.imshow(gen_image[0])
pyplot.axis('off')
pyplot.show()
```

Listing 23.28: Example of plotting a single translated image.

Tying this all together, the complete example of performing an ad hoc image translation with a single image file is listed below.

```
# example of loading a pix2pix model and using it for one-off image translation
from keras.models import load_model
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img
from numpy import expand_dims
from matplotlib import pyplot
```

```
# load an image
def load_image(filename, size=(256,256)):
  # load image with the preferred size
  pixels = load_img(filename, target_size=size)
  # convert to numpy array
  pixels = img_to_array(pixels)
  # scale from [0,255] to [-1,1]
  pixels = (pixels - 127.5) / 127.5
  # reshape to 1 sample
  pixels = expand_dims(pixels, 0)
  return pixels

# load source image
src_image = load_image('satellite.jpg')
print('Loaded', src_image.shape)
# load model
model = load_model('model_109600.h5')
# generate image from source
gen_image = model.predict(src_image)
# scale from [-1,1] to [0,1]
gen_image = (gen_image + 1) / 2.0
# plot the image
pyplot.imshow(gen_image[0])
pyplot.axis('off')
pyplot.show()
```

Listing 23.29: Example of translating a single image.

Running the example loads the image from file, creates a translation of it, and plots the result. The generated image appears to be a reasonable translation of the source image. The streets do not appear to be straight lines and the detail of the buildings is a bit lacking. Perhaps with further training or choice of a different model, higher-quality images could be generated.

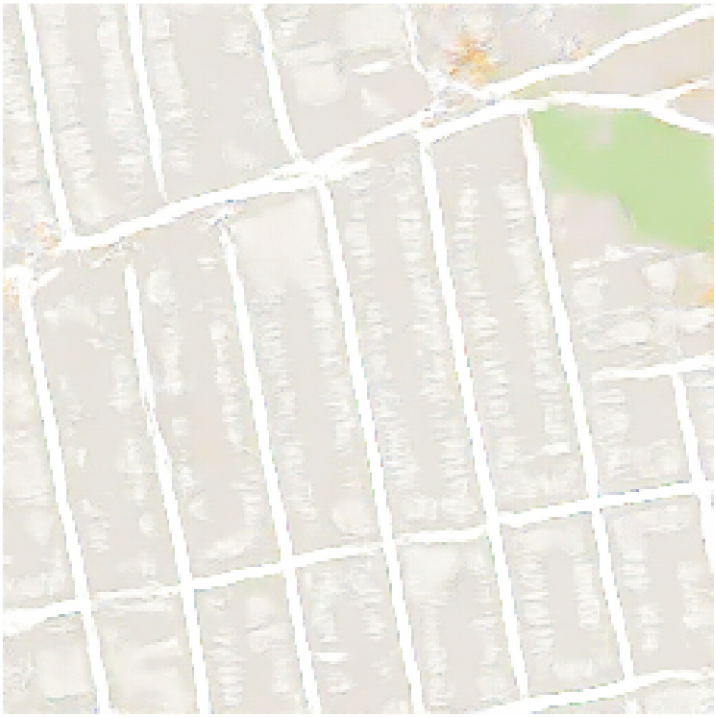

Figure 23.8: Plot of Satellite Image Translated to Google Maps With Final Pix2Pix GAN Model.

23.6 How to Translate Google Maps to Satellite Images

Now that we are familiar with how to develop and use a Pix2Pix model for translating satellite images to Google maps, we can also explore the reverse. That is, we can develop a Pix2Pix model to translate Google Maps images to plausible satellite images. This requires that the model invent (or hallucinate) plausible buildings, roads, parks, and more. We can use the same code to train the model with one small difference. We can change the order of the datasets returned from the `load_real_samples()` function; for example:

```
# load and prepare training images
def load_real_samples(filename):
  # load the compressed arrays
  data = load(filename)
  # unpack the arrays
  X1, X2 = data['arr_0'], data['arr_1']
  # scale from [0,255] to [-1,1]
  X1 = (X1 - 127.5) / 127.5
  X2 = (X2 - 127.5) / 127.5
  # return in reverse order
  return [X2, X1]
```

Listing 23.30: Example of a function for loading the paired images in reverse order.

The order of `X1` and `X2` is reversed in this version of the function. This means that the model will take Google Maps images as input and learn to generate satellite images. The complete example is omitted here for brevity. Run the example as before.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

As before, the loss of the model is reported each training iteration. If loss for the discriminator goes to zero and stays there for a long time, consider re-starting the training run as it is an example of a training failure.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
>1, d1[0.442] d2[0.650] g[49.790]
>2, d1[0.317] d2[0.478] g[56.476]
>3, d1[0.376] d2[0.450] g[48.114]
>4, d1[0.396] d2[0.406] g[62.903]
>5, d1[0.496] d2[0.460] g[40.650]
...
>109596, d1[0.311] d2[0.057] g[25.376]
>109597, d1[0.028] d2[0.070] g[16.618]
>109598, d1[0.007] d2[0.208] g[18.139]
>109599, d1[0.358] d2[0.076] g[22.494]
>109600, d1[0.279] d2[0.049] g[9.941]
>Saved: plot_109600.png and model_109600.h5
```

Listing 23.31: Example output from training the Pix2Pix GAN on the prepared maps dataset with reverse order.

It is harder to judge the quality of generated satellite images, nevertheless, plausible images are generated after just 10 epochs.

Figure 23.9: Plot of Google Maps to Satellite Translated Images Using Pix2Pix After 10 Training Epochs.

As before, image quality will improve and will continue to vary over the training process. A final model can be chosen based on generated image quality, not total training epochs. The model appears to have little difficulty in generating reasonable water, parks, roads, and more.

Figure 23.10: Plot of Google Maps to Satellite Translated Images Using Pix2Pix After 90 Training Epochs.

23.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Standalone Satellite**. Develop an example of translating standalone Google Maps images to satellite images, as we did for satellite to Google Maps images.
- **New Image**. Locate a satellite image for an entirely new location and translate it to a Google Maps and consider the result compared to the actual image in Google maps.
- **More Training**. Continue training the model for another 100 epochs and evaluate whether the additional training results in further improvements in image quality.
- **Image Augmentation**. Use some minor image augmentation during training as described in the Pix2Pix paper and evaluate whether it results in better quality generated images.

If you explore any of these extensions, I'd love to know.

23.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

23.8.1 Official

- Image-to-Image Translation with Conditional Adversarial Networks, 2016.
  https://arxiv.org/abs/1611.07004
- Image-to-Image Translation with Conditional Adversarial Nets, Homepage.
  https://phillipi.github.io/pix2pix/
- Image-to-image translation with conditional adversarial nets, GitHub.
  https://github.com/phillipi/pix2pix
- pytorch-CycleGAN-and-pix2pix, GitHub.
  https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix
- Interactive Image-to-Image Demo, 2017.
  https://affinelayer.com/pixsrv/
- Pix2Pix Datasets
  http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/

23.8.2 API

- Keras Datasets API.
  https://keras.io/datasets/
- Keras Sequential Model API
  https://keras.io/models/sequential/
- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/
- How can I "freeze" Keras layers?
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers

23.9 Summary

In this tutorial, you discovered how to develop a Pix2Pix generative adversarial network for image-to-image translation. Specifically, you learned:

- How to load and prepare the satellite image to Google maps image-to-image translation dataset.
- How to develop a Pix2Pix model for translating satellite photographs to Google Maps images.
- How to use the final Pix2Pix generator model to translate ad hoc satellite images.

23.9.1 Next

In the next tutorial, you will discover the CycleGAN model architecture for unpaired image-to-image translation.

Chapter 24

Introduction to the CycleGAN

Image-to-image translation involves generating a new synthetic version of a given image with a specific modification, such as translating a summer landscape to winter. Training a model for image-to-image translation typically requires a large dataset of paired examples. These datasets can be difficult and expensive to prepare, and in some cases impossible, such as photographs of paintings by long dead artists. The CycleGAN is a technique that involves the automatic training of image-to-image translation models without paired examples. The models are trained in an unsupervised manner using a collection of images from the source and target domain that do not need to be related in any way.

This simple technique is powerful, achieving visually impressive results on a range of application domains, most notably translating photographs of horses to zebra, and the reverse. In this tutorial, you will discover the CycleGAN technique for unpaired image-to-image translation. After reading this tutorial, you will know:

- Image-to-Image translation involves the controlled modification of an image and requires large datasets of paired images that are complex to prepare or sometimes don't exist.
- CycleGAN is a technique for training unsupervised image translation models via the GAN architecture using unpaired collections of images from two different domains.
- CycleGAN has been demonstrated on a range of applications including season translation, object transfiguration, style transfer, and generating photos from paintings.

Let's get started.

24.1 Overview

This tutorial is divided into five parts; they are:

1. Problem With Image-to-Image Translation
2. Unpaired Image-to-Image Translation with CycleGAN
3. What Is the CycleGAN Model Architecture
4. Applications of CycleGAN
5. Implementation Tips for CycleGAN

24.2 Problem With Image-to-Image Translation

Image-to-image translation is an image synthesis task that requires the generation of a new image that is a controlled modification of a given image.

> Image-to-image translation is a class of vision and graphics problems where the goal is to learn the mapping between an input image and an output image using a training set of aligned image pairs.

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

Examples of image-to-image translation include:

- Translating summer landscapes to winter landscapes (or the reverse).
- Translating paintings to photographs (or the reverse).
- Translating horses to zebras (or the reverse).

Traditionally, training an image-to-image translation model requires a dataset comprised of paired examples. That is, a large dataset of many examples of input images X (e.g. summer landscapes) and the same image with the desired modification that can be used as an expected output image Y (e.g. winter landscapes). The requirement for a paired training dataset is a limitation. These datasets are challenging and expensive to prepare, e.g. photos of different scenes under different conditions. In many cases, the datasets simply do not exist, such as famous paintings and their respective photographs.

> However, obtaining paired training data can be difficult and expensive. [...] Obtaining input-output pairs for graphics tasks like artistic stylization can be even more difficult since the desired output is highly complex, typically requiring artistic authoring. For many tasks, like object transfiguration (e.g., zebra $\leftrightarrow$ horse), the desired output is not even well-defined.

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

As such, there is a desire for techniques for training an image-to-image translation system that does not require paired examples. Specifically, where any two collections of unrelated images can be used and the general characteristics extracted from each collection and used in the image translation process. For example, to be able to take a large collection of photos of summer landscapes and a large collection of photos of winter landscapes with unrelated scenes and locations as the first location and be able to translate specific photos from one group to the other. This is called the problem of unpaired image-to-image translation.

24.3 Unpaired Image-to-Image Translation With CycleGAN

A successful approach for unpaired image-to-image translation is CycleGAN. CycleGAN is an approach to training image-to-image translation models using the generative adversarial network, or GAN, model architecture.

> [...] we present a method that can learn to [capture] special characteristics of one image collection and figuring out how these characteristics could be translated into the other image collection, all in the absence of any paired training examples.

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

The GAN architecture is an approach to training a model for image synthesis that is comprised of two models: a generator model and a discriminator model. The generator takes a point from a latent space as input and generates new plausible images from the domain, and the discriminator takes an image as input and predicts whether it is real (from a dataset) or fake (generated). Both models are trained in a game, such that the generator is updated to better fool the discriminator and the discriminator is updated to better detect generated images. The CycleGAN is an extension of the GAN architecture that involves the simultaneous training of two generator models and two discriminator models.

One generator takes images from the first domain as input and outputs images for the second domain, and the other generator takes images from the second domain as input and generates images from the first domain. Discriminator models are then used to determine how plausible the generated images are and update the generator models accordingly. This extension alone might be enough to generate plausible images in each domain, but not sufficient to generate translations of the input images.

> ... adversarial losses alone cannot guarantee that the learned function can map an individual input x_i to a desired output y_i

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

The CycleGAN uses an additional extension to the architecture called cycle consistency. This is the idea that an image output by the first generator could be used as input to the second generator and the output of the second generator should match the original image. The reverse is also true: that an output from the second generator can be fed as input to the first generator and the result should match the input to the second generator. Cycle consistency is a concept from machine translation where a phrase translated from English to French should translate from French back to English and be identical to the original phrase. The reverse process should also be true.

> ... we exploit the property that translation should be "cycle consistent", in the sense that if we translate, e.g., a sentence from English to French, and then translate it back from French to English, we should arrive back at the original sentence

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

The CycleGAN encourages cycle consistency by adding an additional loss to measure the difference between the generated output of the second generator and the original image, and the reverse. This acts as a regularization of the generator models, guiding the image generation process in the new domain toward image translation.

24.4 What Is the CycleGAN Model Architecture

At first glance, the architecture of the CycleGAN appears complex. Let's take a moment to step through all of the models involved and their inputs and outputs. Consider the problem where we are interested in translating images from summer to winter and winter to summer. We have two collections of photographs and they are unpaired, meaning they are photos of different locations at different times; we don't have the exact same scenes in winter and summer.

- **Collection 1**: Photos of summer landscapes.
- **Collection 2**: Photos of winter landscapes.

We will develop an architecture of two GANs, and each GAN has a discriminator and a generator model, meaning there are four models in total in the architecture. The first GAN will generate photos of winter given photos of summer, and the second GAN will generate photos of summer given photos of winter.

- **GAN 1**: Translates photos of summer (collection 1) to winter (collection 2).
- **GAN 2**: Translates photos of winter (collection 2) to summer (collection 1).

Each GAN has a conditional generator model that will synthesize an image given an input image. And each GAN has a discriminator model to predict how likely the generated image is to have come from the target image collection. The discriminator and generator models for a GAN are trained under normal adversarial loss like a standard GAN model. We can summarize the generator and discriminator models from GAN 1 as follows:

- **Generator Model 1**:
  - **Input**: Takes photos of summer (collection 1).
  - **Output**: Generates photos of winter (collection 2).
- **Discriminator Model 1**:
  - **Input**: Takes photos of winter from collection 2 and output from Generator Model 1.
  - **Output**: Likelihood of image is from collection 2.

Similarly, we can summarize the generator and discriminator models from GAN 2 as follows:

- **Generator Model 2**:
  - **Input**: Takes photos of winter (collection 2).
  - **Output**: Generates photos of summer (collection 1).
- **Discriminator Model 2**:
  - **Input**: Takes photos of summer from collection 1 and output from Generator Model 2.
  - **Output**: Likelihood of image is from collection 1.

So far, the models are sufficient for generating plausible images in the target domain but are not translations of the input image. Each of the GANs are also updated using cycle consistency loss. This is designed to encourage the synthesized images in the target domain that are to be translations of the input image. Cycle consistency loss compares an input photo to the Cycle GAN to the generated photo and calculates the difference between the two, e.g. using the L1 norm or summed absolute difference in pixel values. There are two ways in which cycle consistency loss is calculated and used to update the generator models each training iteration.

The first GAN (GAN 1) will take an image of a summer landscape, generate image of a winter landscape, which is provided as input to the second GAN (GAN 2), which in turn will generate an image of a summer landscape. The cycle consistency loss calculates the difference between the image input to GAN 1 and the image output by GAN 2 and the generator models are updated accordingly to reduce the difference in the images. This is a forward-cycle for cycle consistency loss. The same process is related in reverse for a backward cycle consistency loss from generator 2 to generator 1 and comparing the original photo of winter to the generated photo of winter.

- **Forward Cycle Consistency Loss**:
  - Input photo of summer (collection 1) to GAN 1
  - Output photo of winter from GAN 1
  - Input photo of winter from GAN 1 to GAN 2
  - Output photo of summer from GAN 2
  - Compare photo of summer (collection 1) to photo of summer from GAN 2
- **Backward Cycle Consistency Loss**:
  - Input photo of winter (collection 2) to GAN 2
  - Output photo of summer from GAN 2
  - Input photo of summer from GAN 2 to GAN 1
  - Output photo of winter from GAN 1
  - Compare photo of winter (collection 2) to photo of winter from GAN 1

24.5 Applications of CycleGAN

The CycleGAN approach is presented with many impressive applications. In this section, we will review five of these applications to get an idea of the capability of the technique.

24.5.1 Style Transfer

Style transfer refers to the learning of artistic style from one domain, often paintings, and applying the artistic style to another domain, such as photographs. The CycleGAN is demonstrated by applying the artistic style from Monet, Van Gogh, Cezanne, and Ukiyo-e to photographs of landscapes.

Figure 24.1: Example of Style Transfer from Famous Painters to Photographs of Landscapes. Taken from: Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks.

24.5.2 Object Transfiguration

Object transfiguration refers to the transformation of objects from one class, such as dogs into another class of objects, such as cats. The CycleGAN is demonstrated transforming photographs of horses into zebras and the reverse: photographs of zebras into horses. This type of transfiguration makes sense given that both horse and zebras look similar in size and structure, except for their coloring.

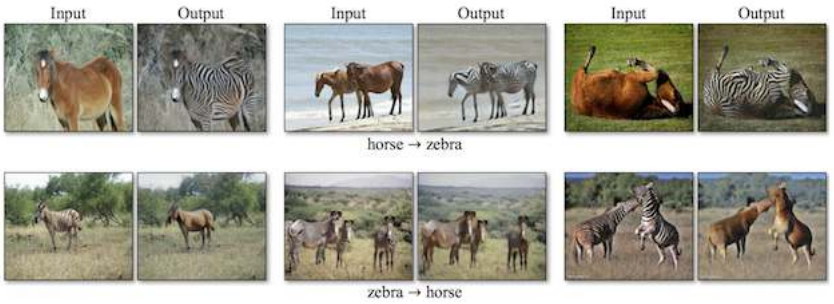

Figure 24.2: Example of Object Transfiguration from Horses to Zebra and Zebra to Horses. Taken from: Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks.

The CycleGAN is also demonstrated on translating photographs of apples to oranges, as well as the reverse: photographs of oranges to apples. Again, this transfiguration makes sense as both oranges and apples have the same structure and size.

Figure 24.3: Example of Object Transfiguration from Apples to Oranges and Oranges to Apples. Taken from: Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks.

24.5.3 Season Transfer

Season transfer refers to the translation of photographs taken in one season, such as summer, to another season, such as winter. The CycleGAN is demonstrated on translating photographs of winter landscapes to summer landscapes, and the reverse of summer landscapes to winter landscapes.

Figure 24.4: Example of Season Transfer from Winter to Summer and Summer to Winter. Taken from: Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks.

24.5.4 Photograph Generation From Paintings

Photograph generation from paintings, as its name suggests, is the synthesis of photorealistic images given a painting, typically by a famous artist or famous scene. The CycleGAN is demonstrated on translating many paintings by Monet to plausible photographs.

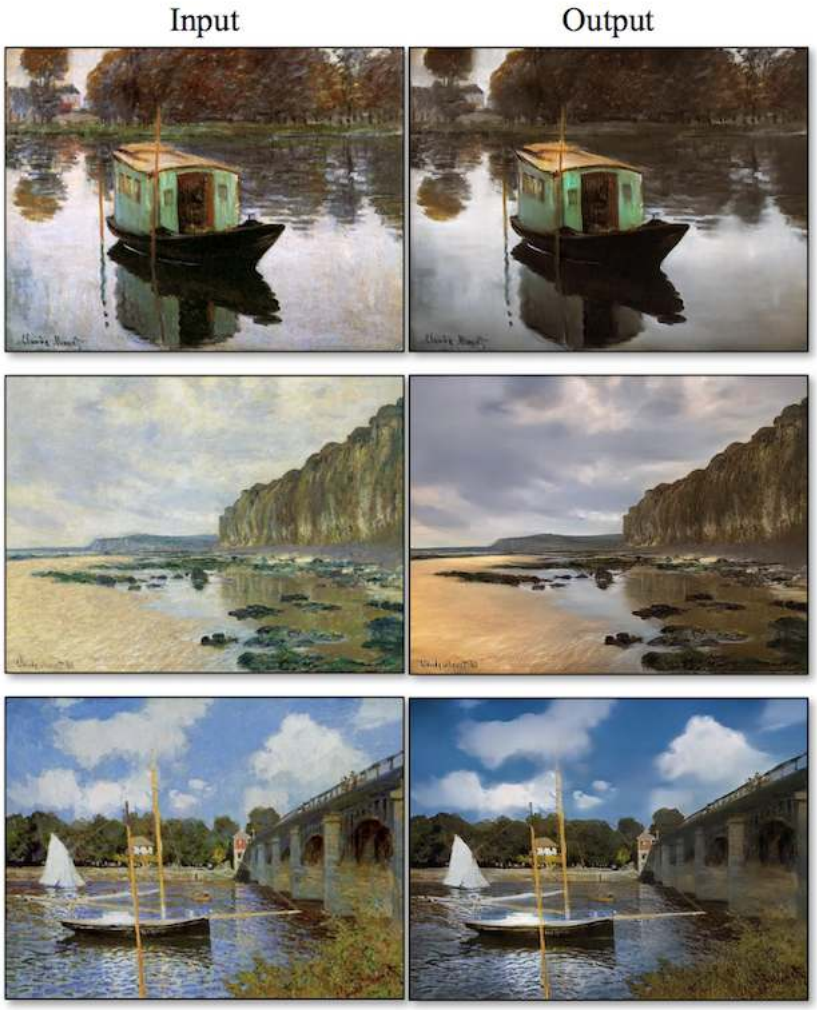

Figure 24.5: Example of Translation Paintings by Monet to Photorealistic Scenes. Taken from: Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks.

24.5.5 Photograph Enhancement

Photograph enhancement refers to transforms that improve the original image in some way. The CycleGAN is demonstrated on photo enhancement by improving the depth of field (e.g. giving a macro effect) on close-up photographs of flowers.

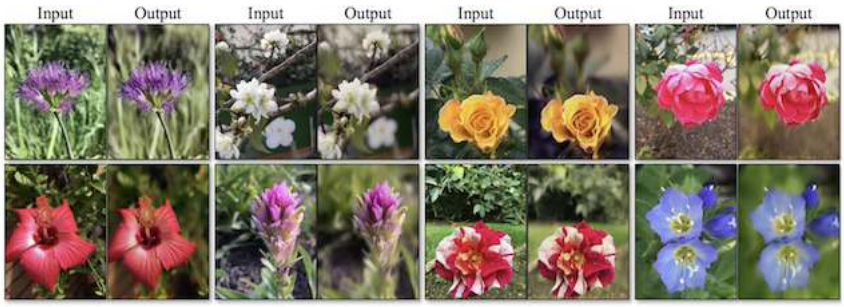

Figure 24.6: Example of Photograph Enhancement Improving the Depth of Field on Photos of Flowers. Taken from: Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks.

24.6 Implementation Tips for CycleGAN

The CycleGAN paper provides a number of technical details regarding how to implement the technique in practice. The generator network implementation is based on the approach described for style transfer by Justin Johnson in the 2016 paper titled *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*. The generator model starts with best practices for generators using the deep convolutional GAN, which is implemented using multiple residual blocks (e.g. from the ResNet). The discriminator models use PatchGAN, as described by Phillip Isola, et al. in their 2016 paper titled *Image-to-Image Translation with Conditional Adversarial Networks*.

> This discriminator tries to classify if each $N \times N$ patch in an image is real or fake. We run this discriminator convolutionally across the image, averaging all responses to provide the ultimate output of D.
>
> — *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

PatchGANs are used in the discriminator models to classify 70×70 overlapping patches of input images as belonging to the domain or having been generated. The discriminator output is then taken as the average of the prediction for each patch. The adversarial loss is implemented using a least-squared loss function, as described in Xudong Mao, et al's 2016 paper titled *Least Squares Generative Adversarial Networks* (described in Chapter 15).

> [...] we propose the Least Squares Generative Adversarial Networks (LSGANs) which adopt the least squares loss function for the discriminator. The idea is simple yet powerful: the least squares loss function is able to move the fake samples toward the decision boundary, because the least squares loss function penalizes samples that lie in a long way on the correct side of the decision boundary.
>
> — *Least squares generative adversarial networks*, 2016.

Additionally, a buffer of 50 generated images is used to update the discriminator models instead of freshly generated images, as described in Ashish Shrivastava's 2016 paper titled *Learning from Simulated and Unsupervised Images through Adversarial Training*.

> [...] we introduce a method to improve the stability of adversarial training by updating the discriminator using a history of refined images, rather than only the ones in the current minibatch.
>
> — *Learning from Simulated and Unsupervised Images through Adversarial Training*, 2016.

The models are trained with the Adam version of stochastic gradient descent and a small learning rate for 100 epochs, then a further 100 epochs with a learning rate decay. The models are updated after each image, e.g. a batch size of 1. Additional model-specific details are provided in the appendix of the paper for each of the datasets on which the technique as demonstrated.

24.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

24.7.1 Papers

- Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, 2017.
  https://arxiv.org/abs/1703.10593
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution, 2016.
  https://arxiv.org/abs/1603.08155
- Image-to-Image Translation with Conditional Adversarial Networks, 2016.
  https://arxiv.org/abs/1611.07004
- Least squares generative adversarial networks, 2016.
  https://arxiv.org/abs/1611.04076
- Learning from Simulated and Unsupervised Images through Adversarial Training, 2016.
  https://arxiv.org/abs/1612.07828

24.7.2 Articles

- CycleGAN Project (official), GitHub.
  https://github.com/junyanz/CycleGAN/
- CycleGAN Project Page (official).
  https://junyanz.github.io/CycleGAN/

24.8 Summary

In this tutorial, you discovered the CycleGAN technique for unpaired image-to-image translation. Specifically, you learned:

- Image-to-Image translation involves the controlled modification of an image and requires large datasets of paired images that are complex to prepare or sometimes don't exist.

- CycleGAN is a technique for training unsupervised image translation models via the GAN architecture using unpaired collections of images from two different domains.
- CycleGAN has been demonstrated on a range of applications including season translation, object transfiguration, style transfer, and generating photos from paintings.

24.8.1 Next

In the next tutorial, you will discover how to implement CycleGAN models from scratch with Keras.

Chapter 25

How to Implement CycleGAN Models

The Cycle Generative adversarial Network, or CycleGAN for short, is a generator model for converting images from one domain to another domain. For example, the model can be used to translate images of horses to images of zebras, or photographs of city landscapes at night to city landscapes during the day. The benefit of the CycleGAN model is that it can be trained without paired examples. That is, it does not require examples of photographs before and after the translation in order to train the model, e.g. photos of the same city landscape during the day and at night. Instead, it is able to use a collection of photographs from each domain and extract and harness the underlying style of images in the collection in order to perform the translation. The model is very impressive but has an architecture that appears quite complicated to implement for beginners. In this tutorial, you will discover how to implement the CycleGAN architecture from scratch using the Keras deep learning framework. After completing this tutorial, you will know:

- How to implement the discriminator and generator models.
- How to define composite models to train the generator models via adversarial and cycle loss.
- How to implement the training process to update model weights each training iteration.

Let's get started.

25.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What Is the CycleGAN Architecture?
2. How to Implement the CycleGAN Discriminator Model
3. How to Implement the CycleGAN Generator Model
4. How to Implement Composite Models and Loss
5. How to Update Model Weights

25.2 What Is the CycleGAN Architecture?

The CycleGAN model was described by Jun-Yan Zhu, et al. in their 2017 paper titled *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* (introduced in Chapter 24). The model architecture is comprised of two generator models: one generator (Generator-A) for generating images for the first domain (Domain-A) and the second generator (Generator-B) for generating images for the second domain (Domain-B).

- Generator-A → Domain-A
- Generator-B → Domain-B

The generator models perform image translation, meaning that the image generation process is conditional on an input image, specifically an image from the other domain. Generator-A takes an image from Domain-B as input and Generator-B takes an image from Domain-A as input.

- Domain-B → Generator-A → Domain-A
- Domain-A → Generator-B → Domain-B

Each generator has a corresponding discriminator model. The first discriminator model (Discriminator-A) takes real images from Domain-A and generated images from Generator-A and predicts whether they are real or fake. The second discriminator model (Discriminator-B) takes real images from Domain-B and generated images from Generator-B and predicts whether they are real or fake.

- Domain-A → Discriminator-A → [Real/Fake]
- Domain-B → Generator-A → Discriminator-A → [Real/Fake]
- Domain-B → Discriminator-B → [Real/Fake]
- Domain-A → Generator-B → Discriminator-B → [Real/Fake]

The discriminator and generator models are trained in an adversarial zero-sum process, like normal GAN models. The generators learn to better fool the discriminators and the discriminators learn to better detect fake images. Together, the models find an equilibrium during the training process. Additionally, the generator models are regularized not just to create new images in the target domain, but instead create translated versions of the input images from the source domain. This is achieved by using generated images as input to the corresponding generator model and comparing the output image to the original images. Passing an image through both generators is called a cycle. Together, each pair of generator models are trained to better reproduce the original source image, referred to as cycle consistency.

- Domain-B → Generator-A → Domain-A → Generator-B → Domain-B
- Domain-A → Generator-B → Domain-B → Generator-A → Domain-A

There is one further element to the architecture referred to as the identity mapping. This is where a generator is provided with images as input from the target domain and is expected to generate the same image without change. This addition to the architecture is optional, although it results in a better matching of the color profile of the input image.

- Domain-A → Generator-A → Domain-A
- Domain-B → Generator-B → Domain-B

Now that we are familiar with the model architecture, we can take a closer look at each model in turn and how they can be implemented. The paper provides a good description of the models and training process, although the official Torch implementation was used as the definitive description for each model and training process and provides the basis for the model implementations described below.

25.3 How to Implement the CycleGAN Discriminator Model

The discriminator model is responsible for taking a real or generated image as input and predicting whether it is real or fake. The discriminator model is implemented as a PatchGAN model (described in detail in Chapter 22).

> For the discriminator networks we use 70×70 PatchGANs, which aim to classify whether 70×70 overlapping image patches are real or fake.

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

The PatchGAN was described in the 2016 paper titled *Precomputed Real-time Texture Synthesis With Markovian Generative Adversarial Networks* and was used in the Pix2Pix model for image translation described in the 2016 paper titled *Image-to-Image Translation with Conditional Adversarial Networks*. The architecture is described as discriminating an input image as real or fake by averaging the prediction for $n \times n$ squares or patches of the source image.

> ... we design a discriminator architecture - which we term a PatchGAN - that only penalizes structure at the scale of patches. This discriminator tries to classify if each $N \times N$ patch in an image is real or fake. We run this discriminator convolutionally across the image, averaging all responses to provide the ultimate output of D.

— *Image-to-Image Translation with Conditional Adversarial Networks*, 2016.

This can be implemented directly by using a somewhat standard deep convolutional discriminator model. Instead of outputting a single value like a traditional discriminator model, the PatchGAN discriminator model can output a square or one-channel feature map of predictions. The 70×70 refers to the effective receptive field of the model on the input, not the actual shape of the output feature map. The receptive field of a convolutional layer refers to the number of pixels that one output of the layer maps to in the input to the layer. The effective receptive field

refers to the mapping of one pixel in the output of a deep convolutional model (multiple layers) to the input image. Here, the PatchGAN is an approach to designing a deep convolutional network based on the effective receptive field, where one output activation of the model maps to a 70×70 patch of the input image, regardless of the size of the input image.

The PatchGAN has the effect of predicting whether each 70×70 patch in the input image is real or fake. These predictions can then be averaged to give the output of the model (if needed) or compared directly to a matrix (or a vector if flattened) of expected values (e.g. 0 or 1 values). The discriminator model described in the paper takes 256×256 color images as input and defines an explicit architecture that is used on all of the test problems. The architecture uses blocks of Conv2D-InstanceNorm-LeakyReLU layers, with 4×4 filters and a 2×2 stride.

> Let Ck denote a 4×4 Convolution-InstanceNorm-LeakyReLU layer with k filters and stride 2. After the last layer, we apply a convolution to produce a 1-dimensional output. We do not use InstanceNorm for the first C64 layer. We use leaky ReLUs with a slope of 0.2.

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

The architecture for the discriminator is as: C64-C128-C256-C512. This is referred to as a 3-layer PatchGAN in the CycleGAN and Pix2Pix nomenclature, as excluding the first hidden layer, the model has three hidden layers that could be scaled up or down to give different sized PatchGAN models. Not listed in the paper, the model also has a final hidden layer $C512$ with a 1×1 stride, and an output layer $C1$, also with a 1×1 stride with a linear activation function. Given the model is mostly used with 256×256 sized images as input, the size of the output feature map of activations is 16×16. If 128×128 images were used as input, then the size of the output feature map of activations would be 8×8. The model does not use batch normalization; instead, instance normalization is used.

Instance normalization was described in the 2016 paper titled *Instance Normalization: The Missing Ingredient for Fast Stylization*. It is a very simple type of normalization and involves standardizing (e.g. scaling to a standard Gaussian) the values on each feature map. The intent is to remove image-specific contrast information from the image during image generation, resulting in better generated images.

> The key idea is to replace batch normalization layers in the generator architecture with instance normalization layers, and to keep them at test time (as opposed to freeze and simplify them out as done for batch normalization). Intuitively, the normalization process allows to remove instance-specific contrast information from the content image, which simplifies generation. In practice, this results in vastly improved images.

— *Instance Normalization: The Missing Ingredient for Fast Stylization*, 2016.

Although designed for generator models, it can also prove effective in discriminator models. An implementation of instance normalization is provided in the `keras-contrib` project that provides early access to community-supplied Keras features. The `keras-contrib` library can be installed via `pip` as follows:

```
sudo pip install git+https://www.github.com/keras-team/keras-contrib.git
```

Listing 25.1: Example of installing `keras-contrib` with `pip`.

Or, if you are using an Anaconda virtual environment, such as on EC2:

```
git clone https://www.github.com/keras-team/keras-contrib.git
cd keras-contrib
sudo ~/anaconda3/envs/tensorflow_p36/bin/python setup.py install
```

Listing 25.2: Example of installing `keras-contrib` with Anaconda.

The new `InstanceNormalization` layer can then be used as follows:

```
...
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization
# define layer
layer = InstanceNormalization(axis=-1)
...
```

Listing 25.3: Example of using the `InstanceNormalization` layer.

The `axis` argument is set to -1 to ensure that features are normalized per feature map. The network weights are initialized to Gaussian random numbers with a standard deviation of 0.02, as is described for DCGANs more generally.

> Weights are initialized from a Gaussian distribution N (0, 0.02).

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

The discriminator model is updated using a least squares loss (L2), a so-called Least-Squared Generative Adversarial Network, or LSGAN.

> ... we replace the negative log likelihood objective by a least-squares loss. This loss is more stable during training and generates higher quality results.

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

This can be implemented using mean squared error between the target values of $class = 1$ for real images and $class = 0$ for fake images. Additionally, the paper suggests dividing the loss for the discriminator by half during training, in an effort to slow down updates to the discriminator relative to the generator.

> In practice, we divide the objective by 2 while optimizing D, which slows down the rate at which D learns, relative to the rate of G.

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

This can be achieved by setting the `loss_weights` argument to 0.5 when compiling the model. This weighting does not appear to be implemented in the official Torch implementation when updating discriminator models are defined in the `fDx_basic()` function[1]. We can tie all

[1] https://github.com/junyanz/CycleGAN/blob/master/models/cycle_gan_model.lua#L136

of this together in the example below with a `define_discriminator()` function that defines the PatchGAN discriminator. The model configuration matches the description in the appendix of the paper with additional details from the official Torch implementation defined in the `defineD_n_layers()` function[2].

```
# example of defining a 70x70 patchgan discriminator model
from keras.optimizers import Adam
from keras.initializers import RandomNormal
from keras.models import Model
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import LeakyReLU
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization
from keras.utils.vis_utils import plot_model

# define the discriminator model
def define_discriminator(image_shape):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # source image input
  in_image = Input(shape=image_shape)
  # C64
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
  d = LeakyReLU(alpha=0.2)(d)
  # C128
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C256
  d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C512
  d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # second last output layer
  d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # patch output
  patch_out = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
  # define model
  model = Model(in_image, patch_out)
  # compile model
  model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5), loss_weights=[0.5])
  return model

# define image shape
image_shape = (256,256,3)
# create the model
model = define_discriminator(image_shape)
# summarize the model
model.summary()
```

[2] https://github.com/junyanz/CycleGAN/blob/master/models/architectures.lua#L338

```
# plot the model
plot_model(model, to_file='discriminator_model_plot.png', show_shapes=True,
    show_layer_names=True)
```

Listing 25.4: Example of defining and summarizing the PatchGAN discriminator.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

Running the example summarizes the model showing the size inputs and outputs for each layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 256, 256, 3)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 128, 128, 64)      3136
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 128, 128, 64)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 64, 64, 128)       131200
_________________________________________________________________
instance_normalization_1 (In (None, 64, 64, 128)       256
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 64, 64, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 256)       524544
_________________________________________________________________
instance_normalization_2 (In (None, 32, 32, 256)       512
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 512)       2097664
_________________________________________________________________
instance_normalization_3 (In (None, 16, 16, 512)       1024
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 16, 16, 512)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 512)       4194816
_________________________________________________________________
instance_normalization_4 (In (None, 16, 16, 512)       1024
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 16, 16, 512)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 16, 16, 1)         8193
=================================================================
Total params: 6,962,369
Trainable params: 6,962,369
Non-trainable params: 0
_________________________________________________________________
```

Listing 25.5: Example output from defining and summarizing the PatchGAN discriminator.

A plot of the model architecture is also created to help get an idea of the inputs, outputs, and transitions of the image data through the model.

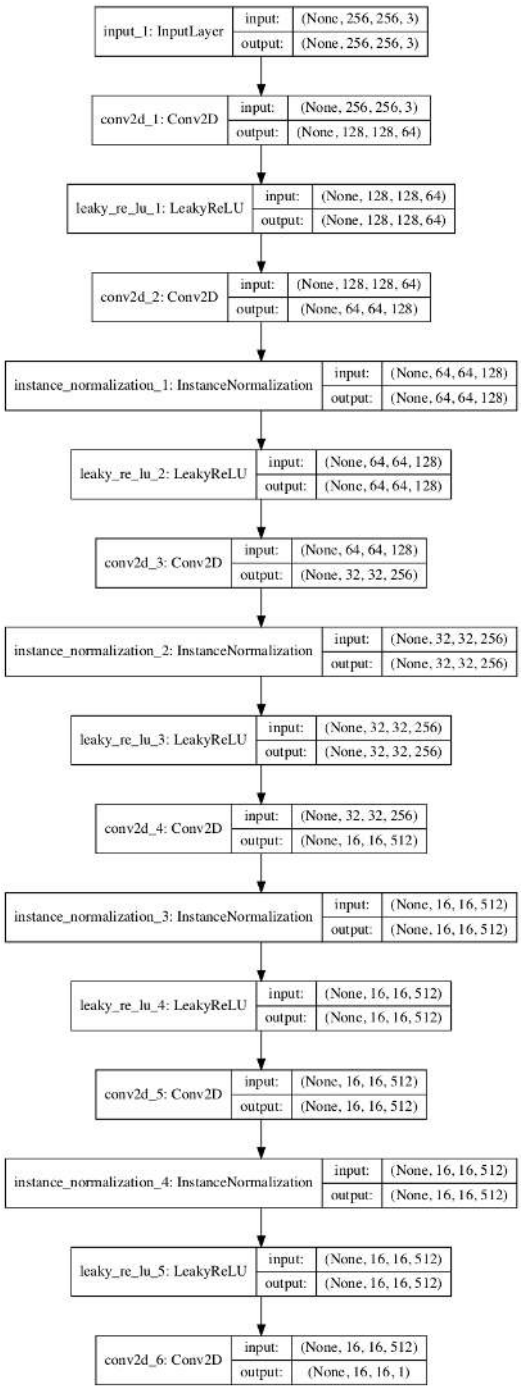

Figure 25.1: Plot of the PatchGAN Discriminator Model for the CycleGAN.

25.4 How to Implement the CycleGAN Generator Model

The CycleGAN Generator model takes an image as input and generates a translated image as output. The model uses a sequence of downsampling convolutional blocks to encode the input

image, a number of residual network (ResNet) convolutional blocks to transform the image, and a number of upsampling convolutional blocks to generate the output image.

> Let c7s1-k denote a 7×7 Convolution-InstanceNormReLU layer with k filters and stride 1. dk denotes a 3×3 Convolution-InstanceNorm-ReLU layer with k filters and stride 2. Reflection padding was used to reduce artifacts. Rk denotes a residual block that contains two 3×3 convolutional layers with the same number of filters on both layer. uk denotes a 3×3 fractional-strided-ConvolutionInstanceNorm-ReLU layer with k filters and stride $\frac{1}{2}$.

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

The architecture for the 6-resnet block generator for 128×128 images is as follows:

- c7s1-64,d128,d256,R256,R256,R256,R256,R256,R256,u128,u64,c7s1-3

First, we need a function to define the ResNet blocks. These are blocks comprised of two 3×3 CNN layers where the input to the block is concatenated to the output of the block, channel-wise. This is implemented in the `resnet_block()` function that creates two Conv-InstanceNorm blocks with 3×3 filters and 1×1 stride and without a ReLU activation after the second block, matching the official Torch implementation in the `build_conv_block()` function. Same padding is used instead of reflection padded recommended in the paper for simplicity.

```
# generator a resnet block
def resnet_block(n_filters, input_layer):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # first layer convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(input_layer)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # second convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  # concatenate merge channel-wise with input layer
  g = Concatenate()([g, input_layer])
  return g
```

Listing 25.6: Example of a function for defining a ResNet block.

Next, we can define a function that will create the 9-resnet block version for 256×256 input images. This can easily be changed to the 6-resnet block version by setting `image_shape` argument to $(128 \times 128 \times 3)$ and `n_resnet` function argument to 6. Importantly, the model outputs pixel values with the shape as the input and pixel values are in the range [-1, 1], typical for GAN generator models.

```
# define the standalone generator model
def define_generator(image_shape=(256,256,3), n_resnet=9):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=image_shape)
  # c7s1-64
```

```
  g = Conv2D(64, (7,7), padding='same', kernel_initializer=init)(in_image)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # d128
  g = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # d256
  g = Conv2D(256, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # R256
  for _ in range(n_resnet):
    g = resnet_block(256, g)
  # u128
  g = Conv2DTranspose(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # u64
  g = Conv2DTranspose(64, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # c7s1-3
  g = Conv2D(3, (7,7), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  out_image = Activation('tanh')(g)
  # define model
  model = Model(in_image, out_image)
  return model
```

Listing 25.7: Example of a function for defining the encoder-decoder generator.

The generator model is not compiled as it is trained via a composite model, seen in the next section. Tying this together, the complete example is listed below.

```
# example of an encoder-decoder generator for the cyclegan
from keras.models import Model
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import Activation
from keras.initializers import RandomNormal
from keras.layers import Concatenate
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization
from keras.utils.vis_utils import plot_model

# generator a resnet block
def resnet_block(n_filters, input_layer):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # first layer convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(input_layer)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # second convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
```

```
  # concatenate merge channel-wise with input layer
  g = Concatenate()([g, input_layer])
  return g

# define the standalone generator model
def define_generator(image_shape=(256,256,3), n_resnet=9):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=image_shape)
  # c7s1-64
  g = Conv2D(64, (7,7), padding='same', kernel_initializer=init)(in_image)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # d128
  g = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # d256
  g = Conv2D(256, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # R256
  for _ in range(n_resnet):
    g = resnet_block(256, g)
  # u128
  g = Conv2DTranspose(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # u64
  g = Conv2DTranspose(64, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # c7s1-3
  g = Conv2D(3, (7,7), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  out_image = Activation('tanh')(g)
  # define model
  model = Model(in_image, out_image)
  return model

# create the model
model = define_generator()
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='generator_model_plot.png', show_shapes=True,
    show_layer_names=True)
```

Listing 25.8: Example of defining and summarizing the Encoder-Decoder generator.

Running the example first summarizes the model. A plot of the generator model is also created, showing the skip connections in the ResNet blocks. The output of the model summary and the plot are omitted here for brevity.

25.5 How to Implement Composite Models and Loss

The generator models are not updated directly. Instead, the generator models are updated via composite models. An update to each generator model involves changes to the model weights based on four concerns:

- Adversarial loss (L2 or mean squared error).
- Identity loss (L1 or mean absolute error).
- Forward cycle loss (L1 or mean absolute error).
- Backward cycle loss (L1 or mean absolute error).

The adversarial loss is the standard approach for updating the generator via the discriminator, although in this case, the least squares loss function is used instead of the negative log likelihood (e.g. binary cross-entropy). First, we can use our function to define the two generators and two discriminators used in the CycleGAN.

```
...
# input shape
image_shape = (256,256,3)
# generator: A -> B
g_model_AtoB = define_generator(image_shape)
# generator: B -> A
g_model_BtoA = define_generator(image_shape)
# discriminator: A -> [real/fake]
d_model_A = define_discriminator(image_shape)
# discriminator: B -> [real/fake]
d_model_B = define_discriminator(image_shape)
```

Listing 25.9: Example of defining the four required models.

A composite model is required for each generator model that is responsible for only updating the weights of that generator model, although it is required to share the weights with the related discriminator model and the other generator model. This can be achieved by marking the weights of the other models as not trainable in the context of the composite model to ensure we are only updating the intended generator.

```
...
# ensure the model we're updating is trainable
g_model_1.trainable = True
# mark discriminator as not trainable
d_model.trainable = False
# mark other generator model as not trainable
g_model_2.trainable = False
```

Listing 25.10: Example of marking discriminator models as not trainable.

The model can be constructed piecewise using the Keras functional API. The first step is to define the input of the real image from the source domain, pass it through our generator model, then connect the output of the generator to the discriminator and classify it as real or fake.

```
...
# discriminator element
input_gen = Input(shape=image_shape)
gen1_out = g_model_1(input_gen)
output_d = d_model(gen1_out)
```

Listing 25.11: Example of defining the input from the source domain.

Next, we can connect the identity mapping element with a new input for the real image from the target domain, pass it through our generator model, and output the (hopefully) untranslated image directly.

```
...
# identity element
input_id = Input(shape=image_shape)
output_id = g_model_1(input_id)
```

Listing 25.12: Example of defining the input for the identity mapping.

So far, we have a composite model with two real image inputs and a discriminator classification and identity image output. Next, we need to add the forward and backward cycles. The forward cycle can be achieved by connecting the output of our generator to the other generator, the output of which can be compared to the input to our generator and should be identical.

```
...
# forward cycle
output_f = g_model_2(gen1_out)
```

Listing 25.13: Example of defining the output for the forward cycle.

The backward cycle is more complex and involves the input for the real image from the target domain passing through the other generator, then passing through our generator, which should match the real image from the target domain.

```
...
# backward cycle
gen2_out = g_model_2(input_id)
output_b = g_model_1(gen2_out)
```

Listing 25.14: Example of defining the output for the backward cycle.

That's it. We can then define this composite model with two inputs: one real image for the source and the target domain, and four outputs, one for the discriminator, one for the generator for the identity mapping, one for the other generator for the forward cycle, and one from our generator for the backward cycle.

```
...
# define model graph
model = Model([input_gen, input_id], [output_d, output_id, output_f, output_b])
```

Listing 25.15: Example of defining the input and output layers for the composite model.

The adversarial loss for the discriminator output uses least squares loss which is implemented as L2 or mean squared error. The outputs from the generators are compared to images and are optimized using L1 loss implemented as mean absolute error. The generator is updated as a weighted average of the four loss values. The adversarial loss is weighted normally, whereas

the forward and backward cycle loss is weighted using a parameter called lambda and is set to 10, e.g. 10 times more important than adversarial loss. The identity loss is also weighted as a fraction of the lambda parameter and is set to 0.5×10 or 5 in the official Torch implementation.

```
...
# compile model with weighting of least squares loss and L1 loss
model.compile(loss=['mse', 'mae', 'mae', 'mae'], loss_weights=[1, 5, 10, 10], optimizer=opt)
```

Listing 25.16: Example of compiling the composite model.

We can tie all of this together and define the function `define_composite_model()` for creating a composite model for training a given generator model.

```
# define a composite model for updating generators by adversarial and cycle loss
def define_composite_model(g_model_1, d_model, g_model_2, image_shape):
  # ensure the model we're updating is trainable
  g_model_1.trainable = True
  # mark discriminator as not trainable
  d_model.trainable = False
  # mark other generator model as not trainable
  g_model_2.trainable = False
  # discriminator element
  input_gen = Input(shape=image_shape)
  gen1_out = g_model_1(input_gen)
  output_d = d_model(gen1_out)
  # identity element
  input_id = Input(shape=image_shape)
  output_id = g_model_1(input_id)
  # forward cycle
  output_f = g_model_2(gen1_out)
  # backward cycle
  gen2_out = g_model_2(input_id)
  output_b = g_model_1(gen2_out)
  # define model graph
  model = Model([input_gen, input_id], [output_d, output_id, output_f, output_b])
  # define optimization algorithm configuration
  opt = Adam(lr=0.0002, beta_1=0.5)
  # compile model with weighting of least squares loss and L1 loss
  model.compile(loss=['mse', 'mae', 'mae', 'mae'], loss_weights=[1, 5, 10, 10],
      optimizer=opt)
  return model
```

Listing 25.17: Example of a function for defining the composite model for training the generator.

This function can then be called to prepare a composite model for training both the `g_model_AtoB` generator model and the `g_model_BtoA` model; for example:

```
...
# composite: A -> B -> [real/fake, A]
c_model_AtoBtoA = define_composite_model(g_model_AtoB, d_model_B, g_model_BtoA, image_shape)
# composite: B -> A -> [real/fake, B]
c_model_BtoAtoB = define_composite_model(g_model_BtoA, d_model_A, g_model_AtoB, image_shape)
```

Listing 25.18: Example of defining both composite models.

Summarizing and plotting the composite model is a bit of a mess as it does not help to see the inputs and outputs of the model clearly. We can summarize the inputs and outputs for each

of the composite models below. Recall that we are sharing or reusing the same set of weights if a given model is used more than once in the composite model.

- **Generator-A Composite Model**: Only Generator-A weights are trainable and weights for other models and not trainable.
  - **Adversarial**: Domain-B → Generator-A → Domain-A → Discriminator-A → [real/fake]
  - **Identity**: Domain-A → Generator-A → Domain-A
  - **Forward Cycle**: Domain-B → Generator-A → Domain-A → Generator-B → Domain-B
  - **Backward Cycle**: Domain-A → Generator-B → Domain-B → Generator-A → Domain-A
- **Generator-B Composite Model**: Only Generator-B weights are trainable and weights for other models are not trainable.
  - **Adversarial**: Domain-A → Generator-B → Domain-B → Discriminator-B → [real/fake]
  - **Identity**: Domain-B → Generator-B → Domain-B
  - **Forward Cycle**: Domain-A → Generator-B → Domain-B → Generator-A → Domain-A
  - **Backward Cycle**: Domain-B → Generator-A → Domain-A → Generator-B → Domain-B

A complete example of creating all of the models is listed below for completeness.

```
# example of defining composite models for training cyclegan generators
from keras.optimizers import Adam
from keras.models import Model
from keras.models import Input
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import Activation
from keras.layers import LeakyReLU
from keras.initializers import RandomNormal
from keras.layers import Concatenate
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization

# define the discriminator model
def define_discriminator(image_shape):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # source image input
  in_image = Input(shape=image_shape)
  # C64
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
  d = LeakyReLU(alpha=0.2)(d)
  # C128
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
```

```
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C256
  d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C512
  d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # second last output layer
  d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # patch output
  patch_out = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
  # define model
  model = Model(in_image, patch_out)
  # compile model
  model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5), loss_weights=[0.5])
  return model

# generator a resnet block
def resnet_block(n_filters, input_layer):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # first layer convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(input_layer)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # second convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  # concatenate merge channel-wise with input layer
  g = Concatenate()([g, input_layer])
  return g

# define the standalone generator model
def define_generator(image_shape, n_resnet=9):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=image_shape)
  # c7s1-64
  g = Conv2D(64, (7,7), padding='same', kernel_initializer=init)(in_image)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # d128
  g = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # d256
  g = Conv2D(256, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # R256
```

```
  for _ in range(n_resnet):
    g = resnet_block(256, g)
  # u128
  g = Conv2DTranspose(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # u64
  g = Conv2DTranspose(64, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # c7s1-3
  g = Conv2D(3, (7,7), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  out_image = Activation('tanh')(g)
  # define model
  model = Model(in_image, out_image)
  return model

# define a composite model for updating generators by adversarial and cycle loss
def define_composite_model(g_model_1, d_model, g_model_2, image_shape):
  # ensure the model we're updating is trainable
  g_model_1.trainable = True
  # mark discriminator as not trainable
  d_model.trainable = False
  # mark other generator model as not trainable
  g_model_2.trainable = False
  # discriminator element
  input_gen = Input(shape=image_shape)
  gen1_out = g_model_1(input_gen)
  output_d = d_model(gen1_out)
  # identity element
  input_id = Input(shape=image_shape)
  output_id = g_model_1(input_id)
  # forward cycle
  output_f = g_model_2(gen1_out)
  # backward cycle
  gen2_out = g_model_2(input_id)
  output_b = g_model_1(gen2_out)
  # define model graph
  model = Model([input_gen, input_id], [output_d, output_id, output_f, output_b])
  # define optimization algorithm configuration
  opt = Adam(lr=0.0002, beta_1=0.5)
  # compile model with weighting of least squares loss and L1 loss
  model.compile(loss=['mse', 'mae', 'mae', 'mae'], loss_weights=[1, 5, 10, 10],
      optimizer=opt)
  return model

# input shape
image_shape = (256,256,3)
# generator: A -> B
g_model_AtoB = define_generator(image_shape)
# generator: B -> A
g_model_BtoA = define_generator(image_shape)
# discriminator: A -> [real/fake]
d_model_A = define_discriminator(image_shape)
# discriminator: B -> [real/fake]
```

```
d_model_B = define_discriminator(image_shape)
# composite: A -> B -> [real/fake, A]
c_model_AtoB = define_composite_model(g_model_AtoB, d_model_B, g_model_BtoA, image_shape)
# composite: B -> A -> [real/fake, B]
c_model_BtoA = define_composite_model(g_model_BtoA, d_model_A, g_model_AtoB, image_shape)
```

Listing 25.19: Example of defining and summarizing the composite models for training the generators.

25.6 How to Update Model Weights

Training the defined models is relatively straightforward. First, we must define a helper function that will select a batch of real images and the associated target (1.0).

```
# select a batch of random samples, returns images and target
def generate_real_samples(dataset, n_samples, patch_shape):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, patch_shape, patch_shape, 1))
  return X, y
```

Listing 25.20: Example of a function for selecting a sample of real images.

Similarly, we need a function to generate a batch of fake images and the associated target (0.0).

```
# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, dataset, patch_shape):
  # generate fake instance
  X = g_model.predict(dataset)
  # create 'fake' class labels (0)
  y = zeros((len(X), patch_shape, patch_shape, 1))
  return X, y
```

Listing 25.21: Example of a function for creating a sample of synthetic images with the generator.

Now, we can define the steps of a single training iteration. We will model the order of updates based on the implementation in the official Torch implementation in the `OptimizeParameters()` function[3] (the official code uses a more confusing inverted naming convention).

1. Update Generator-B (A $\rightarrow$ B)
2. Update Discriminator-B
3. Update Generator-A (B $\rightarrow$ A)
4. Update Discriminator-A

[3] https://github.com/junyanz/CycleGAN/blob/master/models/cycle_gan_model.lua#L230

First, we must select a batch of real images by calling `generate_real_samples()` for both Domain-A and Domain-B. Typically, the batch size (`n_batch`) is set to 1. In this case, we will assume 256 × 256 input images, which means the `n_patch` for the PatchGAN discriminator will be 16.

```
...
# select a batch of real samples
X_realA, y_realA = generate_real_samples(trainA, n_batch, n_patch)
X_realB, y_realB = generate_real_samples(trainB, n_batch, n_patch)
```

Listing 25.22: Example of selecting samples of real images.

Next, we can use the batches of selected real images to generate corresponding batches of generated or fake images.

```
...
# generate a batch of fake samples
X_fakeA, y_fakeA = generate_fake_samples(g_model_BtoA, X_realB, n_patch)
X_fakeB, y_fakeB = generate_fake_samples(g_model_AtoB, X_realA, n_patch)
```

Listing 25.23: Example of generating samples of synthetic images.

The paper describes using a pool of previously generated images from which examples are randomly selected and used to update the discriminator model, where the pool size was set to 50 images.

> ... [we] update the discriminators using a history of generated images rather than the ones produced by the latest generators. We keep an image buffer that stores the 50 previously created images.

— *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017.

This can be implemented using a list for each domain and a using a function to populate the pool, then randomly replace elements from the pool once it is at capacity. The `update_image_pool()` function below implements this based on the official Torch implementation in `image_pool.lua`[4].

```
# update image pool for fake images
def update_image_pool(pool, images, max_size=50):
  selected = list()
  for image in images:
    if len(pool) < max_size:
      # stock the pool
      pool.append(image)
      selected.append(image)
    elif random() < 0.5:
      # use image, but don't add it to the pool
      selected.append(image)
    else:
      # replace an existing image and use replaced image
      ix = randint(0, len(pool))
      selected.append(pool[ix])
      pool[ix] = image
```

[4] https://github.com/junyanz/CycleGAN/blob/master/util/image_pool.lua

```
  return asarray(selected)
```

Listing 25.24: Example of a function for maintaining a pool of generated images.

We can then update our image pool with generated fake images, the results of which can be used to train the discriminator models.

```
...
# update fakes from pool
X_fakeA = update_image_pool(poolA, X_fakeA)
X_fakeB = update_image_pool(poolB, X_fakeB)
```

Listing 25.25: Example of using the pool of generated images.

Next, we can update Generator-A. The `train_on_batch()` function will return a value for each of the four loss functions, one for each output, as well as the weighted sum (first value) used to update the model weights which we are interested in.

```
...
# update generator B->A via adversarial and cycle loss
g_loss2, _, _, _, _ = c_model_BtoA.train_on_batch([X_realB, X_realA], [y_realA, X_realA,
    X_realB, X_realA])
```

Listing 25.26: Example of updating the first generator model.

We can then update the discriminator model using the fake images that may or may not have come from the image pool.

```
...
# update discriminator for A -> [real/fake]
dA_loss1 = d_model_A.train_on_batch(X_realA, y_realA)
dA_loss2 = d_model_A.train_on_batch(X_fakeA, y_fakeA)
```

Listing 25.27: Example of updating the first discriminator model.

We can then do the same for the other generator and discriminator models.

```
...
# update generator A->B via adversarial and cycle loss
g_loss1, _, _, _, _ = c_model_AtoB.train_on_batch([X_realA, X_realB], [y_realB, X_realB,
    X_realA, X_realB])
# update discriminator for B -> [real/fake]
dB_loss1 = d_model_B.train_on_batch(X_realB, y_realB)
dB_loss2 = d_model_B.train_on_batch(X_fakeB, y_fakeB)
```

Listing 25.28: Example of updating the second generator and discriminator models.

At the end of the training run, we can then report the current loss for the discriminator models on real and fake images and of each generator model.

```
...
# summarize performance
print('>%d, dA[%.3f,%.3f] dB[%.3f,%.3f] g[%.3f,%.3f]' % (i+1, dA_loss1,dA_loss2,
    dB_loss1,dB_loss2, g_loss1,g_loss2))
```

Listing 25.29: Example of reporting the loss for each model update.

Tying this all together, we can define a function named `train()` that takes an instance of each of the defined models and a loaded dataset (list of two NumPy arrays, one for each

domain) and trains the model. A batch size of 1 is used as is described in the paper and the models are fit for 100 training epochs.

```
# train cyclegan models
def train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA,
    dataset):
  # define properties of the training run
  n_epochs, n_batch, = 100, 1
  # determine the output square shape of the discriminator
  n_patch = d_model_A.output_shape[1]
  # unpack dataset
  trainA, trainB = dataset
  # prepare image pool for fakes
  poolA, poolB = list(), list()
  # calculate the number of batches per training epoch
  bat_per_epo = int(len(trainA) / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # manually enumerate epochs
  for i in range(n_steps):
    # select a batch of real samples
    X_realA, y_realA = generate_real_samples(trainA, n_batch, n_patch)
    X_realB, y_realB = generate_real_samples(trainB, n_batch, n_patch)
    # generate a batch of fake samples
    X_fakeA, y_fakeA = generate_fake_samples(g_model_BtoA, X_realB, n_patch)
    X_fakeB, y_fakeB = generate_fake_samples(g_model_AtoB, X_realA, n_patch)
    # update fakes from pool
    X_fakeA = update_image_pool(poolA, X_fakeA)
    X_fakeB = update_image_pool(poolB, X_fakeB)
    # update generator B->A via adversarial and cycle loss
    g_loss2, _, _, _, _ = c_model_BtoA.train_on_batch([X_realB, X_realA], [y_realA,
        X_realA, X_realB, X_realA])
    # update discriminator for A -> [real/fake]
    dA_loss1 = d_model_A.train_on_batch(X_realA, y_realA)
    dA_loss2 = d_model_A.train_on_batch(X_fakeA, y_fakeA)
    # update generator A->B via adversarial and cycle loss
    g_loss1, _, _, _, _ = c_model_AtoB.train_on_batch([X_realA, X_realB], [y_realB,
        X_realB, X_realA, X_realB])
    # update discriminator for B -> [real/fake]
    dB_loss1 = d_model_B.train_on_batch(X_realB, y_realB)
    dB_loss2 = d_model_B.train_on_batch(X_fakeB, y_fakeB)
    # summarize performance
    print('>%d, dA[%.3f,%.3f] dB[%.3f,%.3f] g[%.3f,%.3f]' % (i+1, dA_loss1,dA_loss2,
        dB_loss1,dB_loss2, g_loss1,g_loss2))
```

Listing 25.30: Example of a function for training the CycleGAN models.

The train function can then be called directly with our defined models and loaded dataset.

```
...
# load a dataset as a list of two arrays
dataset = ...
# train models
train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA, dataset)
```

Listing 25.31: Example of calling the train function for the CycleGAN.

As an improvement, it may be desirable to combine the update to each discriminator model into a single operation as is performed in the `fDx_basic()` function of the official implementation[5]. Additionally, the paper describes updating the models for another 100 epochs (200 in total), where the learning rate is decayed to 0.0. This too can be added as a minor extension to the training process.

25.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

25.7.1 Papers

- Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, 2017.
  https://arxiv.org/abs/1703.10593
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution, 2016.
  https://arxiv.org/abs/1603.08155
- Image-to-Image Translation with Conditional Adversarial Networks, 2016.
  https://arxiv.org/abs/1611.07004
- Least Squares Generative Adversarial Networks, 2016.
  https://arxiv.org/abs/1611.04076
- Precomputed Real-time Texture Synthesis With Markovian Generative Adversarial Networks, 2016.
  https://arxiv.org/abs/1604.04382
- Instance Normalization: The Missing Ingredient for Fast Stylization.
  https://arxiv.org/abs/1607.08022
- Layer Normalization.
  https://arxiv.org/abs/1607.06450

25.7.2 API

- Keras API.
  https://keras.io/
- `keras-contrib`: Keras community contributions, GitHub.
  https://github.com/keras-team/keras-contrib

[5] https://github.com/junyanz/CycleGAN/blob/master/models/cycle_gan_model.lua#L136

25.7.3 Projects

- CycleGAN Project (official), GitHub.
  https://github.com/junyanz/CycleGAN/
- pytorch-CycleGAN-and-pix2pix (official), GitHub.
  https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix
- CycleGAN Project Page (official).
  https://junyanz.github.io/CycleGAN/

25.8 Summary

In this tutorial, you discovered how to implement the CycleGAN architecture from scratch using the Keras deep learning framework. Specifically, you learned:

- How to implement the discriminator and generator models.
- How to define composite models to train the generator models via adversarial and cycle loss.
- How to implement the training process to update model weights each training iteration.

25.8.1 Next

In the next tutorial, you will discover how to develop a CycleGAN model to translate photographs of horses to zebra, and the reverse.

Chapter 26

How to Develop the CycleGAN End-to-End

The Cycle Generative Adversarial Network, or CycleGAN, is an approach to training a deep convolutional neural network for image-to-image translation tasks. Unlike other GAN models for image translation, the CycleGAN does not require a dataset of paired images. For example, if we are interested in translating photographs of oranges to apples, we do not require a training dataset of oranges that have been manually converted to apples. This allows the development of a translation model on problems where training datasets may not exist, such as translating paintings to photographs. In this tutorial, you will discover how to develop a CycleGAN model to translate photos of horses to zebras, and back again. After completing this tutorial, you will know:

- How to load and prepare the horses to zebras image translation dataset for modeling.
- How to train a pair of CycleGAN generator models for translating horses to zebras and zebras to horses.
- How to load saved CycleGAN models and use them to translate photographs.

Let's get started.

26.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. What Is the CycleGAN?
2. How to Prepare the Horses to Zebras Dataset
3. How to Develop a CycleGAN to Translate Horses to Zebras
4. How to Perform Image Translation with CycleGAN

26.2 What Is the CycleGAN?

The CycleGAN model was described by Jun-Yan Zhu, et al. in their 2017 paper titled *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* (introduced in Chapter 24). The benefit of the CycleGAN model is that it can be trained without paired examples. That is, it does not require examples of photographs before and after the translation in order to train the model, e.g. photos of the same city landscape during the day and at night. Instead, the model is able to use a collection of photographs from each domain and extract and harness the underlying style of images in the collection in order to perform the translation. The paper provides a good description of the models and training process, although the official Torch implementation was used as the definitive description for each model and training process and provides the basis for the model implementations described below.

26.3 How to Prepare the Horses to Zebras Dataset

One of the impressive examples of the CycleGAN in the paper was to transform photographs of horses to zebras, and the reverse, zebras to horses. The authors of the paper referred to this as the problem of *object transfiguration* and it was also demonstrated on photographs of apples and oranges. In this tutorial, we will develop a CycleGAN from scratch for image-to-image translation (or object transfiguration) from horses to zebras and the reverse. We will refer to this dataset as `horses2zebra`. The zip file for this dataset about 111 megabytes and can be downloaded from the CycleGAN webpage:

- Download Horses to Zebras Dataset (111 megabytes).[1]

Download the dataset into your current working directory. You will see the following directory structure:

```
horse2zebra
├──testA
├──testB
├──trainA
└──trainB
```

Listing 26.1: Example directory structure for the horse2zebra dataset.

The *A* category refers to horse and *B* category refers to zebra, and the dataset is comprised of train and test elements. We will load all photographs and use them as a training dataset. The photographs are square with the shape 256×256 and have filenames like `n02381460_2.jpg`. The example below will load all photographs from the train and test folders and create an array of images for category A and another for category B. Both arrays are then saved to a new file in compressed NumPy array format.

```
# example of preparing the horses and zebra dataset
from os import listdir
from numpy import asarray
from numpy import vstack
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.preprocessing.image import load_img
```

[1] https://people.eecs.berkeley.edu/~taesung_park/CycleGAN/datasets/horse2zebra.zip

```
from numpy import savez_compressed

# load all images in a directory into memory
def load_images(path, size=(256,256)):
  data_list = list()
  # enumerate filenames in directory, assume all are images
  for filename in listdir(path):
    # load and resize the image
    pixels = load_img(path + filename, target_size=size)
    # convert to numpy array
    pixels = img_to_array(pixels)
    # store
    data_list.append(pixels)
  return asarray(data_list)

# dataset path
path = 'horse2zebra/'
# load dataset A
dataA1 = load_images(path + 'trainA/')
dataAB = load_images(path + 'testA/')
dataA = vstack((dataA1, dataAB))
print('Loaded dataA: ', dataA.shape)
# load dataset B
dataB1 = load_images(path + 'trainB/')
dataB2 = load_images(path + 'testB/')
dataB = vstack((dataB1, dataB2))
print('Loaded dataB: ', dataB.shape)
# save as compressed numpy array
filename = 'horse2zebra_256.npz'
savez_compressed(filename, dataA, dataB)
print('Saved dataset: ', filename)
```

Listing 26.2: Example of preparing and saving the dataset ready for modeling.

Running the example first loads all images into memory, showing that there are 1,187 photos in category A (horses) and 1,474 in category B (zebras). The arrays are then saved in compressed NumPy format with the filename `horse2zebra_256.npz`. This data file is about 570 megabytes, larger than the raw images as we are storing pixel values as 32-bit floating point values.

```
Loaded dataA: (1187, 256, 256, 3)
Loaded dataB: (1474, 256, 256, 3)
Saved dataset: horse2zebra_256.npz
```

Listing 26.3: Example output from preparing and saving the horse2zebra dataset.

We can then load the dataset and plot some of the photos to confirm that we are handling the image data correctly. The complete example is listed below.

```
# load and plot the prepared dataset
from numpy import load
from matplotlib import pyplot
# load the face dataset
data = load('horse2zebra_256.npz')
dataA, dataB = data['arr_0'], data['arr_1']
print('Loaded: ', dataA.shape, dataB.shape)
# plot source images
n_samples = 3
```

```
for i in range(n_samples):
  pyplot.subplot(2, n_samples, 1 + i)
  pyplot.axis('off')
  pyplot.imshow(dataA[i].astype('uint8'))
# plot target image
for i in range(n_samples):
  pyplot.subplot(2, n_samples, 1 + n_samples + i)
  pyplot.axis('off')
  pyplot.imshow(dataB[i].astype('uint8'))
pyplot.show()
```

Listing 26.4: Example of loading and plotting the prepared dataset.

Running the example first loads the dataset, confirming the number of examples and shape of the color images match our expectations.

```
Loaded: (1187, 256, 256, 3) (1474, 256, 256, 3)
```

Listing 26.5: Example output from loading and plotting the prepared dataset.

A plot is created showing a row of three images from the horse photo dataset (dataA) and a row of three images from the zebra dataset (dataB).

Figure 26.1: Plot of Photographs from the Horses2Zeba Dataset.

Now that we have prepared the dataset for modeling, we can develop the CycleGAN generator models that can translate photos from one category to the other, and the reverse.

26.4 How to Develop a CycleGAN to Translate Horse to Zebra

In this section, we will develop the CycleGAN model for translating photos of horses to zebras and photos of zebras to horses. The same model architecture and configuration described in the paper was used across a range of image-to-image translation tasks. This architecture is both described in the body of the paper, with additional detail in the appendix of the paper, and a fully working implementation provided as open source implemented for the Torch deep learning framework. The implementation in this section will use the Keras deep learning framework based directly on the model described in the paper and implemented in the author's codebase, designed to take and generate color images with the size 256×256 pixels (model implementation was covered in Chapter 25).

The architecture is comprised of four models: two discriminator models, and two generator models. The discriminator is a deep convolutional neural network that performs image classification. It takes a source image as input and predicts the likelihood of whether the target image is a real or fake image. Two discriminator models are used, one for Domain-A (horses) and one for Domain-B (zebras). The discriminator design is based on the effective receptive field of the model, which defines the relationship between one output of the model to the number of pixels in the input image. This is called a PatchGAN model and is carefully designed so that each output prediction of the model maps to a 70×70 square or patch of the input image. The benefit of this approach is that the same model can be applied to input images of different sizes, e.g. larger or smaller than 256×256 pixels.

The output of the model depends on the size of the input image but may be one value or a square activation map of values. Each value is a probability that a patch in the input image is real. These values can be averaged to give an overall likelihood or classification score if needed. A pattern of Convolutional-BatchNorm-LeakyReLU layers is used in the model, which is common to deep convolutional discriminator models. Unlike other models, the CycleGAN discriminator uses `InstanceNormalization` instead of `BatchNormalization`. It is a very simple type of normalization and involves standardizing (e.g. scaling to a standard Gaussian) the values on each output feature map, rather than across features in a batch. An implementation of instance normalization is provided in the `keras-contrib` project that provides early access to community supplied Keras features (covered in Chapter 25).

The `define_discriminator()` function below implements the 70×70 PatchGAN discriminator model as per the design of the model in the paper. The model takes a 256×256 sized image as input and outputs a patch of predictions. The model is optimized using least squares loss (L2) implemented as mean squared error, and a weighting is used so that updates to the model have half (0.5) the usual effect. The authors of CycleGAN paper recommend this weighting of model updates to slow down changes to the discriminator, relative to the generator model during training.

```
# define the discriminator model
def define_discriminator(image_shape):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # source image input
  in_image = Input(shape=image_shape)
  # C64
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
```

```
  d = LeakyReLU(alpha=0.2)(d)
  # C128
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C256
  d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C512
  d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # second last output layer
  d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # patch output
  patch_out = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
  # define model
  model = Model(in_image, patch_out)
  # compile model
  model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5), loss_weights=[0.5])
  return model
```

Listing 26.6: Example of a function for defining the PatchGAN discriminator.

The generator model is more complex than the discriminator model. The generator is an encoder-decoder model architecture. The model takes a source image (e.g. horse photo) and generates a target image (e.g. zebra photo). It does this by first downsampling or encoding the input image down to a bottleneck layer, then interpreting the encoding with a number of ResNet layers that use skip connections, followed by a series of layers that upsample or decode the representation to the size of the output image. First, we need a function to define the ResNet blocks. These are blocks comprised of two 3×3 CNN layers where the input to the block is concatenated to the output of the block, channel-wise.

This is implemented in the `resnet_block()` function that creates two Convolution-InstanceNorm blocks with 3×3 filters and 1×1 stride and without a ReLU activation after the second block, matching the official Torch implementation in the `build_conv_block()` function. Same padding is used instead of reflection padded recommended in the paper for simplicity.

```
# generator a resnet block
def resnet_block(n_filters, input_layer):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # first layer convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(input_layer)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # second convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  # concatenate merge channel-wise with input layer
  g = Concatenate()([g, input_layer])
  return g
```

Listing 26.7: Example of a function for defining a ResNet block.

Next, we can define a function that will create the 9-resnet block version for 256×256 input images. This can easily be changed to the 6-resnet block version by setting the `image_shape` argument to $(128 \times 128 \times 3)$ and `n_resnet` function argument to 6. Importantly, the model outputs pixel values with the shape as the input and pixel values are in the range [-1, 1], typical for GAN generator models.

```
# define the standalone generator model
def define_generator(image_shape, n_resnet=9):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=image_shape)
  # c7s1-64
  g = Conv2D(64, (7,7), padding='same', kernel_initializer=init)(in_image)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # d128
  g = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # d256
  g = Conv2D(256, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # R256
  for _ in range(n_resnet):
    g = resnet_block(256, g)
  # u128
  g = Conv2DTranspose(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # u64
  g = Conv2DTranspose(64, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # c7s1-3
  g = Conv2D(3, (7,7), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  out_image = Activation('tanh')(g)
  # define model
  model = Model(in_image, out_image)
  return model
```

Listing 26.8: Example of a function for defining the encoder-decoder generator.

The discriminator models are trained directly on real and generated images, whereas the generator models are not. Instead, the generator models are trained via their related discriminator models. Specifically, they are updated to minimize the loss predicted by the discriminator for generated images marked as *real*, called adversarial loss. As such, they are encouraged to generate images that better fit into the target domain. The generator models are also updated based on how effective they are at the regeneration of a source image when used with the other

generator model, called cycle loss. Finally, a generator model is expected to output an image without translation when provided an example from the target domain, called identity loss. Altogether, each generator model is optimized via the combination of four outputs with four loss functions:

- Adversarial loss (L2 or mean squared error).
- Identity loss (L1 or mean absolute error).
- Forward cycle loss (L1 or mean absolute error).
- Backward cycle loss (L1 or mean absolute error).

This can be achieved by defining a composite model used to train each generator model that is responsible for only updating the weights of that generator model, although it is required to share the weights with the related discriminator model and the other generator model. This is implemented in the `define_composite_model()` function below that takes a defined generator model (`g_model_1`) as well as the defined discriminator model for the generator models output (`d_model`) and the other generator model (`g_model_2`). The weights of the other models are marked as not trainable as we are only interested in updating the first generator model, i.e. the focus of this composite model.

The discriminator is connected to the output of the generator in order to classify generated images as real or fake. A second input for the composite model is defined as an image from the target domain (instead of the source domain), which the generator is expected to output without translation for the identity mapping. Next, forward cycle loss involves connecting the output of the generator to the other generator, which will reconstruct the source image. Finally, the backward cycle loss involves the image from the target domain used for the identity mapping that is also passed through the other generator whose output is connected to our main generator as input and outputs a reconstructed version of that image from the target domain.

To summarize, a composite model has two inputs for the real photos from Domain-A and Domain-B, and four outputs for the discriminator output, identity generated image, forward cycle generated image, and backward cycle generated image. Only the weights of the first or main generator model are updated for the composite model and this is done via the weighted sum of all loss functions. The cycle loss is given more weight (10-times) than the adversarial loss as described in the paper, and the identity loss is always used with a weighting half that of the cycle loss (5-times), matching the official implementation source code.

```
# define a composite model for updating generators by adversarial and cycle loss
def define_composite_model(g_model_1, d_model, g_model_2, image_shape):
  # ensure the model we're updating is trainable
  g_model_1.trainable = True
  # mark discriminator as not trainable
  d_model.trainable = False
  # mark other generator model as not trainable
  g_model_2.trainable = False
  # discriminator element
  input_gen = Input(shape=image_shape)
  gen1_out = g_model_1(input_gen)
  output_d = d_model(gen1_out)
  # identity element
  input_id = Input(shape=image_shape)
```

```
  output_id = g_model_1(input_id)
  # forward cycle
  output_f = g_model_2(gen1_out)
  # backward cycle
  gen2_out = g_model_2(input_id)
  output_b = g_model_1(gen2_out)
  # define model graph
  model = Model([input_gen, input_id], [output_d, output_id, output_f, output_b])
  # define optimization algorithm configuration
  opt = Adam(lr=0.0002, beta_1=0.5)
  # compile model with weighting of least squares loss and L1 loss
  model.compile(loss=['mse', 'mae', 'mae', 'mae'], loss_weights=[1, 5, 10, 10],
      optimizer=opt)
  return model
```

Listing 26.9: Example of a function for defining the composite model for training the generator.

We need to create a composite model for each generator model, e.g. the Generator-A (BtoA) for zebra to horse translation, and the Generator-B (AtoB) for horse to zebra translation. All of this forward and backward across two domains gets confusing. Below is a complete listing of all of the inputs and outputs for each of the composite models. Identity and cycle loss are calculated as the L1 distance between the input and output image for each sequence of translations. Adversarial loss is calculated as the L2 distance between the model output and the target values of 1.0 for real and 0.0 for fake. Defining the models is the hard part of the CycleGAN; the rest is standard GAN training and is relatively straightforward. Next, we can load our paired images dataset in compressed NumPy array format. This will return a list of two NumPy arrays: the first for source images and the second for corresponding target images.

```
# load and prepare training images
def load_real_samples(filename):
  # load the dataset
  data = load(filename)
  # unpack arrays
  X1, X2 = data['arr_0'], data['arr_1']
  # scale from [0,255] to [-1,1]
  X1 = (X1 - 127.5) / 127.5
  X2 = (X2 - 127.5) / 127.5
  return [X1, X2]
```

Listing 26.10: Example of a function for loading the prepared dataset.

Each training iteration we will require a sample of real images from each domain as input to the discriminator and composite generator models. This can be achieved by selecting a random batch of samples. The `generate_real_samples()` function below implements this, taking a NumPy array for a domain as input and returning the requested number of randomly selected images, as well as the target for the PatchGAN discriminator model indicating the images are real ($target = 1.0$). As such, the shape of the PatchGAN output is also provided, which in the case of 256×256 images will be 16, or a $16 \times 16 \times 1$ activation map, defined by the `patch_shape` function argument.

```
# select a batch of random samples, returns images and target
def generate_real_samples(dataset, n_samples, patch_shape):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
```

```
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, patch_shape, patch_shape, 1))
  return X, y
```

Listing 26.11: Example of a function for selecting samples of real images.

Similarly, a sample of generated images is required to update each discriminator model in each training iteration. The `generate_fake_samples()` function below generates this sample given a generator model and the sample of real images from the source domain. Again, target values for each generated image are provided with the correct shape of the PatchGAN, indicating that they are fake or generated ($target = 0.0$).

```
# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, dataset, patch_shape):
  # generate fake instance
  X = g_model.predict(dataset)
  # create 'fake' class labels (0)
  y = zeros((len(X), patch_shape, patch_shape, 1))
  return X, y
```

Listing 26.12: Example of a function for creating samples of synthetic images with the generator.

Typically, GAN models do not converge; instead, an equilibrium is found between the generator and discriminator models. As such, we cannot easily judge whether training should stop. Therefore, we can save the model and use it to generate sample image-to-image translations periodically during training, such as every one or five training epochs. We can then review the generated images at the end of training and use the image quality to choose a final model. The `save_models()` function below will save each generator model to the current directory in H5 format, including the training iteration number in the filename.

```
# save the generator models to file
def save_models(step, g_model_AtoB, g_model_BtoA):
  # save the first generator model
  filename1 = 'g_model_AtoB_%06d.h5' % (step+1)
  g_model_AtoB.save(filename1)
  # save the second generator model
  filename2 = 'g_model_BtoA_%06d.h5' % (step+1)
  g_model_BtoA.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))
```

Listing 26.13: Example of a function for saving the generator models to file.

The `summarize_performance()` function below uses a given generator model to generate translated versions of a few randomly selected source photographs and saves the plot to file. The source images are plotted on the first row and the generated images are plotted on the second row. Again, the plot filename includes the training iteration number.

```
# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, trainX, name, n_samples=5):
  # select a sample of input images
  X_in, _ = generate_real_samples(trainX, n_samples, 0)
  # generate translated images
  X_out, _ = generate_fake_samples(g_model, X_in, 0)
  # scale all pixels from [-1,1] to [0,1]
```

```
  X_in = (X_in + 1) / 2.0
  X_out = (X_out + 1) / 2.0
  # plot real images
  for i in range(n_samples):
    pyplot.subplot(2, n_samples, 1 + i)
    pyplot.axis('off')
    pyplot.imshow(X_in[i])
  # plot translated image
  for i in range(n_samples):
    pyplot.subplot(2, n_samples, 1 + n_samples + i)
    pyplot.axis('off')
    pyplot.imshow(X_out[i])
  # save plot to file
  filename1 = '%s_generated_plot_%06d.png' % (name, (step+1))
  pyplot.savefig(filename1)
  pyplot.close()
```

Listing 26.14: Example of a function for summarizing and saving model performance.

We are nearly ready to define the training of the models. The discriminator models are updated directly on real and generated images, although in an effort to further manage how quickly the discriminator models learn, a pool of fake images is maintained. The paper defines an image pool of 50 generated images for each discriminator model that is first populated and probabilistically either adds new images to the pool by replacing an existing image or uses a generated image directly. We can implement this as a Python list of images for each discriminator and use the `update_image_pool()` function below to maintain each pool list.

```
# update image pool for fake images
def update_image_pool(pool, images, max_size=50):
  selected = list()
  for image in images:
    if len(pool) < max_size:
      # stock the pool
      pool.append(image)
      selected.append(image)
    elif random() < 0.5:
      # use image, but don't add it to the pool
      selected.append(image)
    else:
      # replace an existing image and use replaced image
      ix = randint(0, len(pool))
      selected.append(pool[ix])
      pool[ix] = image
  return asarray(selected)
```

Listing 26.15: Example of a function for managing the generated image pool.

We can now define the training of each of the generator models. The `train()` function below takes all six models (two discriminator, two generator, and two composite models) as arguments along with the dataset and trains the models. The batch size is fixed at one image to match the description in the paper and the models are fit for 100 epochs. Given that the horses dataset has 1,187 images, one epoch is defined as 1,187 batches and the same number of training iterations. Images are generated using both generators each epoch and models are saved every five epochs or (1187×5) 5,935 training iterations.

The order of model updates is implemented to match the official Torch implementation. First, a batch of real images from each domain is selected, then a batch of fake images for each domain is generated. The fake images are then used to update each discriminator's fake image pool. Next, the Generator-A model (zebras to horses) is updated via the composite model, followed by the Discriminator-A model (horses). Then the Generator-B (horses to zebra) composite model and Discriminator-B (zebras) models are updated. Loss for each of the updated models is then reported at the end of the training iteration. Importantly, only the weighted average loss used to update each generator is reported.

```
# train cyclegan models
def train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA,
    dataset):
  # define properties of the training run
  n_epochs, n_batch, = 100, 1
  # determine the output square shape of the discriminator
  n_patch = d_model_A.output_shape[1]
  # unpack dataset
  trainA, trainB = dataset
  # prepare image pool for fakes
  poolA, poolB = list(), list()
  # calculate the number of batches per training epoch
  bat_per_epo = int(len(trainA) / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # manually enumerate epochs
  for i in range(n_steps):
    # select a batch of real samples
    X_realA, y_realA = generate_real_samples(trainA, n_batch, n_patch)
    X_realB, y_realB = generate_real_samples(trainB, n_batch, n_patch)
    # generate a batch of fake samples
    X_fakeA, y_fakeA = generate_fake_samples(g_model_BtoA, X_realB, n_patch)
    X_fakeB, y_fakeB = generate_fake_samples(g_model_AtoB, X_realA, n_patch)
    # update fakes from pool
    X_fakeA = update_image_pool(poolA, X_fakeA)
    X_fakeB = update_image_pool(poolB, X_fakeB)
    # update generator B->A via adversarial and cycle loss
    g_loss2, _, _, _, _ = c_model_BtoA.train_on_batch([X_realB, X_realA], [y_realA,
        X_realA, X_realB, X_realA])
    # update discriminator for A -> [real/fake]
    dA_loss1 = d_model_A.train_on_batch(X_realA, y_realA)
    dA_loss2 = d_model_A.train_on_batch(X_fakeA, y_fakeA)
    # update generator A->B via adversarial and cycle loss
    g_loss1, _, _, _, _ = c_model_AtoB.train_on_batch([X_realA, X_realB], [y_realB,
        X_realB, X_realA, X_realB])
    # update discriminator for B -> [real/fake]
    dB_loss1 = d_model_B.train_on_batch(X_realB, y_realB)
    dB_loss2 = d_model_B.train_on_batch(X_fakeB, y_fakeB)
    # summarize performance
    print('>%d, dA[%.3f,%.3f] dB[%.3f,%.3f] g[%.3f,%.3f]' % (i+1, dA_loss1,dA_loss2,
        dB_loss1,dB_loss2, g_loss1,g_loss2))
    # evaluate the model performance every so often
    if (i+1) % (bat_per_epo * 1) == 0:
      # plot A->B translation
      summarize_performance(i, g_model_AtoB, trainA, 'AtoB')
      # plot B->A translation
```

```
      summarize_performance(i, g_model_BtoA, trainB, 'BtoA')
    if (i+1) % (bat_per_epo * 5) == 0:
      # save the models
      save_models(i, g_model_AtoB, g_model_BtoA)
```

Listing 26.16: Example of a function for training the CycleGAN models.

Tying all of this together, the complete example of training a CycleGAN model to translate photos of horses to zebras and zebras to horses is listed below.

```
# example of training a cyclegan on the horse2zebra dataset
from random import random
from numpy import load
from numpy import zeros
from numpy import ones
from numpy import asarray
from numpy.random import randint
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.initializers import RandomNormal
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import Conv2DTranspose
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import Concatenate
from tensorflow_addons.layers import InstanceNormalization

from matplotlib import pyplot

# define the discriminator model
def define_discriminator(image_shape):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # source image input
  in_image = Input(shape=image_shape)
  # C64
  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
  d = LeakyReLU(alpha=0.2)(d)
  # C128
  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C256
  d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # C512
  d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # second last output layer
  d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
  d = InstanceNormalization(axis=-1)(d)
  d = LeakyReLU(alpha=0.2)(d)
  # patch output
  patch_out = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
```

```
  # define model
  model = Model(in_image, patch_out)
  # compile model
  model.compile(loss='mse', optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
      loss_weights=[0.5])
  return model

# generator a resnet block
def resnet_block(n_filters, input_layer):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # first layer convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(input_layer)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # second convolutional layer
  g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  # concatenate merge channel-wise with input layer
  g = Concatenate()([g, input_layer])
  return g

# define the standalone generator model
def define_generator(image_shape, n_resnet=9):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=image_shape)
  # c7s1-64
  g = Conv2D(64, (7,7), padding='same', kernel_initializer=init)(in_image)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # d128
  g = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # d256
  g = Conv2D(256, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # R256
  for _ in range(n_resnet):
    g = resnet_block(256, g)
  # u128
  g = Conv2DTranspose(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # u64
  g = Conv2DTranspose(64, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)
  # c7s1-3
  g = Conv2D(3, (7,7), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  out_image = Activation('tanh')(g)
  # define model
```

```
  model = Model(in_image, out_image)
  return model

# define a composite model for updating generators by adversarial and cycle loss
def define_composite_model(g_model_1, d_model, g_model_2, image_shape):
    """Discriminators are compiled and will be trained separately, but generators are not
        compiled.
    """
  # ensure the model we're updating is trainable
  g_model_1.trainable = True
  # mark discriminator as not trainable
  d_model.trainable = False
  # mark other generator model as not trainable
  g_model_2.trainable = False
  # discriminator element
  input_gen = Input(shape=image_shape)
  gen1_out = g_model_1(input_gen)
  output_d = d_model(gen1_out)
  # identity element
  input_id = Input(shape=image_shape)
  output_id = g_model_1(input_id)
  # forward cycle
  output_f = g_model_2(gen1_out)
  # backward cycle
  gen2_out = g_model_2(input_id)
  output_b = g_model_1(gen2_out)
  # define model graph
  model = Model([input_gen, input_id], [output_d, output_id, output_f, output_b])
  # define optimization algorithm configuration
  opt = Adam(lr=0.0002, beta_1=0.5)
  # compile model with weighting of least squares loss and L1 loss
  model.compile(loss=['mse', 'mae', 'mae', 'mae'], loss_weights=[1, 5, 10, 10],
      optimizer=opt)
  return model

# load and prepare training images
def load_real_samples(filename):
  # load the dataset
  data = load(filename)
  # unpack arrays
  X1, X2 = data['arr_0'], data['arr_1']
  # scale from [0,255] to [-1,1]
  X1 = (X1 - 127.5) / 127.5
  X2 = (X2 - 127.5) / 127.5
  return [X1, X2]

# select a batch of random samples, returns images and target
def generate_real_samples(dataset, n_samples, patch_shape):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, patch_shape, patch_shape, 1))
  return X, y
```

```
# generate a batch of images, returns images and targets
def generate_fake_samples(g_model, dataset, patch_shape):
  # generate fake instance
  X = g_model.predict(dataset)
  # create 'fake' class labels (0)
  y = zeros((len(X), patch_shape, patch_shape, 1))
  return X, y

# save the generator models to file
def save_models(step, g_model_AtoB, g_model_BtoA):
  # save the first generator model
  filename1 = 'g_model_AtoB_%06d.h5' % (step+1)
  g_model_AtoB.save(filename1)
  # save the second generator model
  filename2 = 'g_model_BtoA_%06d.h5' % (step+1)
  g_model_BtoA.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))

# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, trainX, name, n_samples=5):
  # select a sample of input images
  X_in, _ = generate_real_samples(trainX, n_samples, 0)
  # generate translated images
  X_out, _ = generate_fake_samples(g_model, X_in, 0)
  # scale all pixels from [-1,1] to [0,1]
  X_in = (X_in + 1) / 2.0
  X_out = (X_out + 1) / 2.0
  # plot real images
  for i in range(n_samples):
    pyplot.subplot(2, n_samples, 1 + i)
    pyplot.axis('off')
    pyplot.imshow(X_in[i])
  # plot translated image
  for i in range(n_samples):
    pyplot.subplot(2, n_samples, 1 + n_samples + i)
    pyplot.axis('off')
    pyplot.imshow(X_out[i])
  # save plot to file
  filename1 = '%s_generated_plot_%06d.png' % (name, (step+1))
  pyplot.savefig(filename1)
  pyplot.close()

# update image pool for fake images
def update_image_pool(pool, images, max_size=50):
  selected = list()
  for image in images:
    if len(pool) < max_size:
      # stock the pool
      pool.append(image)
      selected.append(image)
    elif random() < 0.5:
      # use image, but don't add it to the pool
      selected.append(image)
    else:
      # replace an existing image and use replaced image
      ix = randint(0, len(pool))
```

```
      selected.append(pool[ix])
      pool[ix] = image
  return asarray(selected)

# train cyclegan models
def train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA,
    dataset):
  # define properties of the training run
  n_epochs, n_batch, = 100, 1
  # determine the output square shape of the discriminator
  n_patch = d_model_A.output_shape[1]
  # unpack dataset
  trainA, trainB = dataset
  # prepare image pool for fakes
  poolA, poolB = list(), list()
  # calculate the number of batches per training epoch
  bat_per_epo = int(len(trainA) / n_batch)
  # calculate the number of training iterations
  n_steps = bat_per_epo * n_epochs
  # manually enumerate epochs
  for i in range(n_steps):
    # select a batch of real samples
    X_realA, y_realA = generate_real_samples(trainA, n_batch, n_patch)
    X_realB, y_realB = generate_real_samples(trainB, n_batch, n_patch)
    # generate a batch of fake samples
    X_fakeA, y_fakeA = generate_fake_samples(g_model_BtoA, X_realB, n_patch)
    X_fakeB, y_fakeB = generate_fake_samples(g_model_AtoB, X_realA, n_patch)
    # update fakes from pool
    X_fakeA = update_image_pool(poolA, X_fakeA)
    X_fakeB = update_image_pool(poolB, X_fakeB)
    # update generator B->A via adversarial and cycle loss
    g_loss2, _, _, _, _ = c_model_BtoA.train_on_batch([X_realB, X_realA], [y_realA,
        X_realA, X_realB, X_realA])
    # update discriminator for A -> [real/fake]
    dA_loss1 = d_model_A.train_on_batch(X_realA, y_realA)
    dA_loss2 = d_model_A.train_on_batch(X_fakeA, y_fakeA)
    # update generator A->B via adversarial and cycle loss
    g_loss1, _, _, _, _ = c_model_AtoB.train_on_batch([X_realA, X_realB], [y_realB,
        X_realB, X_realA, X_realB])
    # update discriminator for B -> [real/fake]
    dB_loss1 = d_model_B.train_on_batch(X_realB, y_realB)
    dB_loss2 = d_model_B.train_on_batch(X_fakeB, y_fakeB)
    # summarize performance
    print('>%d, dA[%.3f,%.3f] dB[%.3f,%.3f] g[%.3f,%.3f]' % (i+1, dA_loss1,dA_loss2,
        dB_loss1,dB_loss2, g_loss1,g_loss2))
    # evaluate the model performance every so often
    if (i+1) % (bat_per_epo * 1) == 0:
      # plot A->B translation
      summarize_performance(i, g_model_AtoB, trainA, 'AtoB')
      # plot B->A translation
      summarize_performance(i, g_model_BtoA, trainB, 'BtoA')
    if (i+1) % (bat_per_epo * 5) == 0:
      # save the models
      save_models(i, g_model_AtoB, g_model_BtoA)

# load image data
```

```
dataset = load_real_samples('horse2zebra_256.npz')
print('Loaded', dataset[0].shape, dataset[1].shape)
# define input shape based on the loaded dataset
image_shape = dataset[0].shape[1:]
# generator: A -> B
g_model_AtoB = define_generator(image_shape)
# generator: B -> A
g_model_BtoA = define_generator(image_shape)
# discriminator: A -> [real/fake]
d_model_A = define_discriminator(image_shape)
# discriminator: B -> [real/fake]
d_model_B = define_discriminator(image_shape)
# composite: A -> B -> [real/fake, A]
c_model_AtoB = define_composite_model(g_model_AtoB, d_model_B, g_model_BtoA, image_shape)
# composite: B -> A -> [real/fake, B]
c_model_BtoA = define_composite_model(g_model_BtoA, d_model_A, g_model_AtoB, image_shape)
# train models
train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA, dataset)
```

Listing 26.17: Example of training the CycleGAN on the prepared horses2zebra dataset.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix C.

The loss is reported each training iteration, including the Discriminator-A loss on real and fake examples (dA), Discriminator-B loss on real and fake examples (dB), and Generator-AtoB and Generator-BtoA loss, each of which is a weighted average of adversarial, identity, forward, and backward cycle loss (g). If loss for the discriminator goes to zero and stays there for a long time, consider re-starting the training run as it is an example of a training failure.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
>1, dA[2.284,0.678] dB[1.422,0.918] g[18.747,18.452]
>2, dA[2.129,1.226] dB[1.039,1.331] g[19.469,22.831]
>3, dA[1.644,3.909] dB[1.097,1.680] g[19.192,23.757]
>4, dA[1.427,1.757] dB[1.236,3.493] g[20.240,18.390]
>5, dA[1.737,0.808] dB[1.662,2.312] g[16.941,14.915]
...
>118696, dA[0.004,0.016] dB[0.001,0.001] g[2.623,2.359]
>118697, dA[0.001,0.028] dB[0.003,0.002] g[3.045,3.194]
>118698, dA[0.002,0.008] dB[0.001,0.002] g[2.685,2.071]
>118699, dA[0.010,0.010] dB[0.001,0.001] g[2.430,2.345]
>118700, dA[0.002,0.008] dB[0.000,0.004] g[2.487,2.169]
>Saved: g_model_AtoB_118700.h5 and g_model_BtoA_118700.h5
```

Listing 26.18: Example output from training the CycleGAN on the prepared horses2zebra dataset.

Plots of generated images are saved at the end of every epoch or after every 1,187 training iterations and the iteration number is used in the filename.

```
AtoB_generated_plot_001187.png
AtoB_generated_plot_002374.png
...
BtoA_generated_plot_001187.png
BtoA_generated_plot_002374.png
```

Listing 26.19: Example output of saved plots of generated images.

Models are saved after every five epochs or (1187×5) 5,935 training iterations, and again the iteration number is used in the filenames.

```
g_model_AtoB_053415.h5
g_model_AtoB_059350.h5
...
g_model_BtoA_053415.h5
g_model_BtoA_059350.h5
```

Listing 26.20: Example output of saved generator models.

The plots of generated images can be used to choose a model and more training iterations may not necessarily mean better quality generated images. Horses to Zebras translation starts to become reliable after about 50 epochs.

Figure 26.2: Plot of Source Photographs of Horses (top row) and Translated Photographs of Zebras (bottom row) After 53,415 Training Iterations.

The translation from Zebras to Horses appears to be more challenging for the model to learn, although somewhat plausible translations also begin to be generated after 50 to 60 epochs. I suspect that better quality results could be achieved with an additional 100 training epochs with weight decay, as is used in the paper, and perhaps with a data generator that systematically works through each dataset rather than randomly sampling.

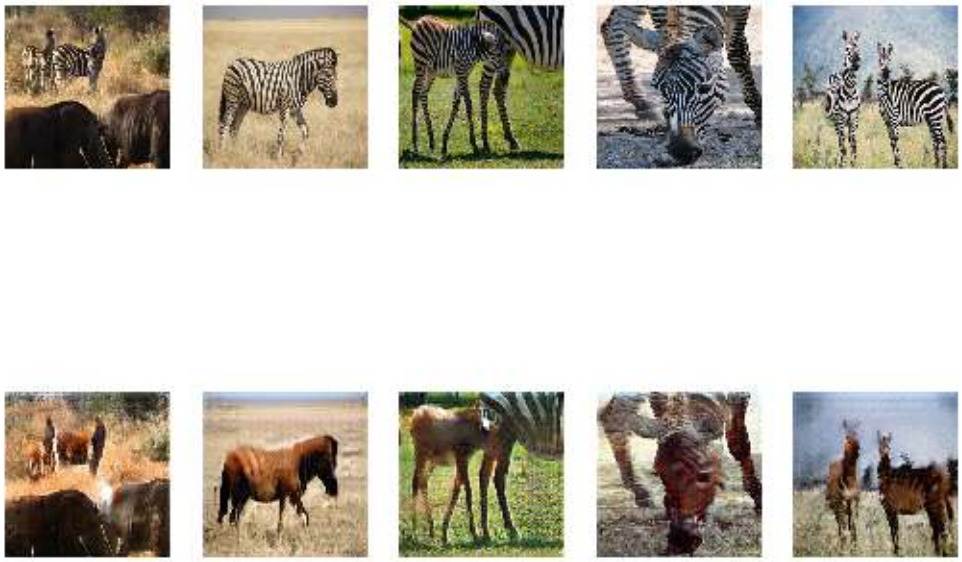

Figure 26.3: Plot of Source Photographs of Zebras (top row) and Translated Photographs of Horses (bottom row) After 90,212 Training Iterations.

Now that we have fit our CycleGAN generators, we can use them to translate photographs in an ad hoc manner.

26.5 How to Perform Image Translation with CycleGAN

The saved generator models can be loaded and used for ad hoc image translation. The first step is to load the dataset. We can use the same `load_real_samples()` function as we developed in the previous section.

```
...
# load dataset
A_data, B_data = load_real_samples('horse2zebra_256.npz')
print('Loaded', A_data.shape, B_data.shape)
```

Listing 26.21: Example of loading the prepared dataset.

Review the plots of generated images and select a pair of models that we can use for image generation. In this case, we will use the model saved around epoch 89 (training iteration 89,025). Our generator models used a custom layer from the `keras_contrib` library, specifically the `InstanceNormalization` layer. Therefore, we need to specify how to load this layer when loading each generator model. This can be achieved by specifying a dictionary mapping of the layer name to the object and passing this as an argument to the `load_model()` Keras function.

```
...
# load the models
cust = {'InstanceNormalization': InstanceNormalization}
model_AtoB = load_model('g_model_AtoB_089025.h5', cust)
model_BtoA = load_model('g_model_BtoA_089025.h5', cust)
```

Listing 26.22: Example of loading the saved generator models.

We can use the `select_sample()` function that we developed in the previous section to select a random photo from the dataset.

```
# select a random sample of images from the dataset
def select_sample(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  return X
```

Listing 26.23: Example of a function for selecting a random sample of images from the dataset.

Next, we can use the Generator-AtoB model, first by selecting a random image from Domain-A (horses) as input, using Generator-AtoB to translate it to Domain-B (zebras), then use the Generator-BtoA model to reconstruct the original image (horse).

```
...
# select input and generate translated images
A_real = select_sample(A_data, 1)
B_generated = model_AtoB.predict(A_real)
A_reconstructed = model_BtoA.predict(B_generated)
```

Listing 26.24: Example of using a generator model to translate images.

We can then plot the three photos side by side as the original or real photo, the translated photo, and the reconstruction of the original photo. The `show_plot()` function below implements this.

```
# plot the image, the translation, and the reconstruction
def show_plot(imagesX, imagesY1, imagesY2):
  images = vstack((imagesX, imagesY1, imagesY2))
  titles = ['Real', 'Generated', 'Reconstructed']
  # scale from [-1,1] to [0,1]
  images = (images + 1) / 2.0
  # plot images row by row
  for i in range(len(images)):
    # define subplot
```

```
    pyplot.subplot(1, len(images), 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(images[i])
    # title
    pyplot.title(titles[i])
  pyplot.show()
```

Listing 26.25: Example of a function for plotting generated images.

We can then call this function to plot our real and generated photos.

```
...
show_plot(A_real, B_generated, A_reconstructed)
```

Listing 26.26: Example of plotting generated images.

This is a good test of both models, however, we can also perform the same operation in reverse. Specifically, a real photo from Domain-B (zebra) translated to Domain-A (horse), then reconstructed as Domain-B (zebra).

```
...
# plot B->A->B
B_real = select_sample(B_data, 1)
A_generated = model_BtoA.predict(B_real)
B_reconstructed = model_AtoB.predict(A_generated)
show_plot(B_real, A_generated, B_reconstructed)
```

Listing 26.27: Example of generating and plotting images using the other generator model.

Tying all of this together, the complete example is listed below.

```
# example of using saved cyclegan models for image translation
from keras.models import load_model
from numpy import load
from numpy import vstack
from matplotlib import pyplot
from numpy.random import randint
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization

# load and prepare training images
def load_real_samples(filename):
  # load the dataset
  data = load(filename)
  # unpack arrays
  X1, X2 = data['arr_0'], data['arr_1']
  # scale from [0,255] to [-1,1]
  X1 = (X1 - 127.5) / 127.5
  X2 = (X2 - 127.5) / 127.5
  return [X1, X2]

# select a random sample of images from the dataset
def select_sample(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
```

```
  return X

# plot the image, the translation, and the reconstruction
def show_plot(imagesX, imagesY1, imagesY2):
  images = vstack((imagesX, imagesY1, imagesY2))
  titles = ['Real', 'Generated', 'Reconstructed']
  # scale from [-1,1] to [0,1]
  images = (images + 1) / 2.0
  # plot images row by row
  for i in range(len(images)):
    # define subplot
    pyplot.subplot(1, len(images), 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(images[i])
    # title
    pyplot.title(titles[i])
  pyplot.show()

# load dataset
A_data, B_data = load_real_samples('horse2zebra_256.npz')
print('Loaded', A_data.shape, B_data.shape)
# load the models
cust = {'InstanceNormalization': InstanceNormalization}
model_AtoB = load_model('g_model_AtoB_089025.h5', cust)
model_BtoA = load_model('g_model_BtoA_089025.h5', cust)
# plot A->B->A
A_real = select_sample(A_data, 1)
B_generated = model_AtoB.predict(A_real)
A_reconstructed = model_BtoA.predict(B_generated)
show_plot(A_real, B_generated, A_reconstructed)
# plot B->A->B
B_real = select_sample(B_data, 1)
A_generated = model_BtoA.predict(B_real)
B_reconstructed = model_AtoB.predict(A_generated)
show_plot(B_real, A_generated, B_reconstructed)
```

Listing 26.28: Example of loading the saved generator models and performing image translation.

Running the example first selects a random photo of a horse, translates it, and then tries to reconstruct the original photo.

Figure 26.4: Plot of a Real Photo of a Horse, Translation to Zebra, and Reconstructed Photo of a Horse Using CycleGAN.

Then a similar process is performed in reverse, selecting a random photo of a zebra, translating it to a horse, then reconstructing the original photo of the zebra.

Figure 26.5: Plot of a Real Photo of a Zebra, Translation to Horse, and Reconstructed Photo of a Zebra Using CycleGAN.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The models are not perfect, especially the zebra to horse model, so you may want to generate many translated examples to review. It also seems that both models are more effective when reconstructing an image, which is interesting as they are essentially performing the same translation task as when operating on real photographs. This may be a sign that the adversarial loss is not strong enough during training. We may also want to use a generator model in a standalone way on individual photograph files. First, we can select a photo from the training dataset. In this case, we will use `horse2zebra/trainA/n02381460_541.jpg`.

Figure 26.6: Photograph of a Horse.

We can develop a function to load this image and scale it to the preferred size of 256 × 256, scale pixel values to the range [-1,1], and convert the array of pixels to a single sample. The `load_image()` function below implements this.

```
def load_image(filename, size=(256,256)):
  # load and resize the image
  pixels = load_img(filename, target_size=size)
  # convert to an array
  pixels = img_to_array(pixels)
  # transform in a sample
  pixels = expand_dims(pixels, 0)
  # scale from [0,255] to [-1,1]
  pixels = (pixels - 127.5) / 127.5
  return pixels
```

Listing 26.29: Example of a function for loading and preparing an image for translation.

We can then load our selected image as well as the AtoB generator model, as we did before.

```
...
# load the image
image_src = load_image('horse2zebra/trainA/n02381460_541.jpg')
# load the model
cust = {'InstanceNormalization': InstanceNormalization}
model_AtoB = load_model('g_model_AtoB_089025.h5', cust)
```

Listing 26.30: Example of loading the image and the generator model.

We can then translate the loaded image, scale the pixel values back to the expected range, and plot the result.

```
...
# translate image
image_tar = model_AtoB.predict(image_src)
# scale from [-1,1] to [0,1]
image_tar = (image_tar + 1) / 2.0
# plot the translated image
```

```
pyplot.imshow(image_tar[0])
pyplot.show()
```

Listing 26.31: Example of translating the image and plotting the result.

Tying this all together, the complete example is listed below.

```
# example of using saved cyclegan models for image translation
from numpy import expand_dims
from keras.models import load_model
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img
from matplotlib import pyplot

# load an image to the preferred size
def load_image(filename, size=(256,256)):
  # load and resize the image
  pixels = load_img(filename, target_size=size)
  # convert to numpy array
  pixels = img_to_array(pixels)
  # transform in a sample
  pixels = expand_dims(pixels, 0)
  # scale from [0,255] to [-1,1]
  pixels = (pixels - 127.5) / 127.5
  return pixels

# load the image
image_src = load_image('horse2zebra/trainA/n02381460_541.jpg')
# load the model
cust = {'InstanceNormalization': InstanceNormalization}
model_AtoB = load_model('g_model_AtoB_100895.h5', cust)
# translate image
image_tar = model_AtoB.predict(image_src)
# scale from [-1,1] to [0,1]
image_tar = (image_tar + 1) / 2.0
# plot the translated image
pyplot.imshow(image_tar[0])
pyplot.show()
```

Listing 26.32: Example of loading and translating a single photograph.

Running the example loads the selected image, loads the generator model, translates the photograph of a horse to a zebra, and plots the results.

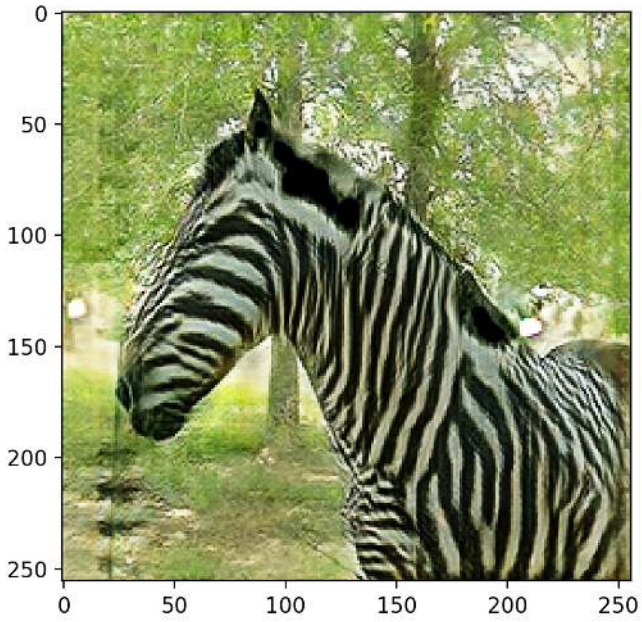

Figure 26.7: Photograph of a Horse Translated to a Photograph of a Zebra using CycleGAN.

26.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Smaller Image Size**. Update the example to use a smaller image size, such as 128×128, and adjust the size of the generator model to use 6 ResNet layers as is used in the CycleGAN paper.
- **Different Dataset**. Update the example to use the apples to oranges dataset.
- **Without Identity Mapping**. Update the example to train the generator models without the identity mapping and compare results.

If you explore any of these extensions, I'd love to know.

26.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

26.7.1 Papers

- Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, 2017. https://arxiv.org/abs/1703.10593

26.7.2 Projects

- CycleGAN Project (official), GitHub.
  https://github.com/junyanz/CycleGAN/
- pytorch-CycleGAN-and-pix2pix (official), GitHub.
  https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix
- CycleGAN Project Page (official).
  https://junyanz.github.io/CycleGAN/

26.7.3 API

- Keras Datasets API.
  https://keras.io/datasets/
- Keras Sequential Model API
  https://keras.io/models/sequential/
- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/
- How can I "freeze" Keras layers?
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers
- Keras Contrib Project.
  https://github.com/keras-team/keras-contrib

26.7.4 Articles

- CycleGAN Dataset.
  https://people.eecs.berkeley.edu/~taesung_park/CycleGAN/datasets

26.8 Summary

In this tutorial, you discovered how to develop a CycleGAN model to translate photos of horses to zebras, and back again. Specifically, you learned:

- How to load and prepare the horses to zebra image translation dataset for modeling.
- How to train a pair of CycleGAN generator models for translating horses to zebra and zebra to horses.
- How to load saved CycleGAN models and use them to translate photographs.

26.8.1 Next

This was the final tutorial in this part. In the next part, you will discover some state-of-the-art GAN model architectures.

Part VII

Advanced GANs

Overview

In this part you will discover some of the more interesting state-of-the-art GAN models that are capable of generating large photorealistic images. These are models that have only been described in the last few years. After reading the chapters in this part, you will know:

- How to leverage what works well in GAN and scale them up in the BigGAN (Chapter 27).
- How to support the generation large images with the Progressive Growing GAN (Chapter 28).
- How to give fine-grained control over the generation of large images with the StyleGAN (Chapter 29).

Chapter 27

Introduction to the BigGAN

Generative Adversarial Networks, or GANs, are perhaps the most effective generative model for image synthesis. Nevertheless, they are typically restricted to generating small images and the training process remains fragile, dependent upon specific augmentations and hyperparameters in order to achieve good results. The BigGAN is an approach to pull together a suite of recent best practices in training class-conditional images and scaling up the batch size and number of model parameters. The result is the routine generation of both high-resolution (large) and high-quality (high-fidelity) images. In this tutorial, you will discover the BigGAN model for scaling up class-conditional image synthesis. After reading this tutorial, you will know:

- Image size and training brittleness remain large problems for GANs.
- Scaling up model size and batch size can result in dramatically larger and higher-quality images.
- Specific model architectural and training configurations required to scale up GANs.

Let's get started.

27.1 Overview

This tutorial is divided into four parts; they are:

1. Brittleness of GAN Training
2. Develop Better GANs by Scaling Up
3. How to Scale-Up GANs With BigGAN
4. Example of Images Generated by BigGAN

27.2 Brittleness of GAN Training

Generative Adversarial Networks, or GANs for short, are capable of generating high-quality synthetic images. Nevertheless, the size of generated images remains relatively small, e.g. 64×64 or 128×128 pixels. Additionally, the model training process remains brittle regardless of the large number of studies that have investigated and proposed improvements.

> Without auxiliary stabilization techniques, this training procedure is notoriously brittle, requiring finely-tuned hyperparameters and architectural choices to work at all.
>
> — *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

Most of the improvements to the training process have focused on changes to the objective function or constraining the discriminator model during the training process.

> Much recent research has accordingly focused on modifications to the vanilla GAN procedure to impart stability, drawing on a growing body of empirical and theoretical insights. One line of work is focused on changing the objective function [...] to encourage convergence. Another line is focused on constraining D through gradient penalties [...] or normalization [...] both to counteract the use of unbounded loss functions and ensure D provides gradients everywhere to G.
>
> — *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

More recently, work has focused on the effective application of the GAN for generating both high-quality and larger images. One approach is to try scaling up GAN models that already work well.

27.3 Develop Better GANs by Scaling Up

The BigGAN is an implementation of the GAN architecture designed to leverage the best from what has been reported to work more generally. It was described by Andrew Brock, et al. in their 2018 paper titled *Large Scale GAN Training for High Fidelity Natural Image Synthesis* and presented at the ICLR 2019 conference. Specifically, the BigGAN is designed for class-conditional image generation. That is, the generation of images using both a point from latent space and image class information as input. Example datasets used to train class-conditional GANs include the CIFAR or ImageNet image classification datasets that have tens, hundreds, or thousands of image classes. As its name suggests, the BigGAN is focused on scaling up the GAN models. This includes GAN models with:

- More model parameters (e.g. more feature maps).
- Larger Batch Sizes
- Architectural changes

> We demonstrate that GANs benefit dramatically from scaling, and train models with two to four times as many parameters and eight times the batch size compared to prior art. We introduce two simple, general architectural changes that improve scalability, and modify a regularization scheme to improve conditioning, demonstrably boosting performance.
>
> — *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

The BigGAN architecture also introduces a *truncation trick* used during image generation that results in an improvement in image quality, and a corresponding regularizing technique to better support this trick. The result is an approach capable of generating larger and higher-quality images, such as 256×256 and 512×512 images.

> When trained on ImageNet at 128×128 resolution, our models (BigGANs) improve the state-of-the-art [...] We also successfully train BigGANs on ImageNet at 256×256 and 512×512 resolution ...
>
> — *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

27.4 How to Scale-Up GANs With BigGAN

The contribution of the BigGAN model is the design decisions for both the models and the training process. These design decisions are important for both re-implementing the BigGAN, but also in providing insight on configuration options that may prove beneficial with GANs more generally. The focus of the BigGAN model is to increase the number of model parameters and batch size, then configure the model and training process to achieve the best results. In this section, we will review the specific design decisions in the BigGAN.

27.4.1 Self-Attention Module and Hinge Loss

The base for the model is the Self-Attention GAN, or SAGAN for short, described by Han Zhang, et al. in the 2018 paper tilted *Self-Attention Generative Adversarial Networks*. This involves introducing an attention map that is applied to feature maps, allowing the generator and discriminator models to focus on different parts of the image. This involves adding an attention module to the deep convolutional model architecture.

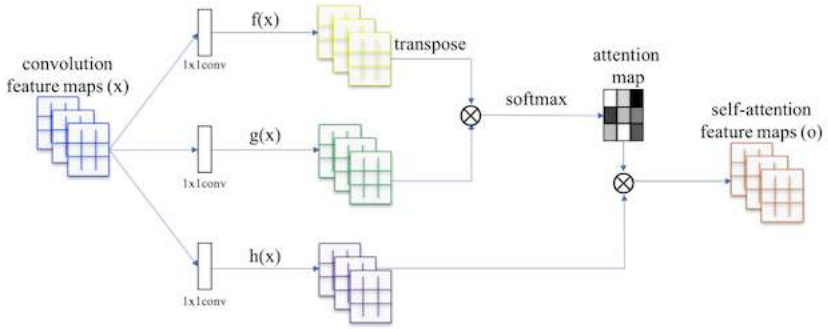

Figure 27.1: Summary of the Self-Attention Module Used in the Self-Attention GAN. Taken from: Self-Attention Generative Adversarial Networks.

Additionally, the model is trained via hinge loss, commonly used for training support vector machines.

> In SAGAN, the proposed attention module has been applied to both generator and discriminator, which are trained in an alternating fashion by minimizing the hinge version of the adversarial loss

— *Self-Attention Generative Adversarial Networks*, 2018.

The BigGAN uses the model architecture with attention modules from SAGAN and is trained via hinge loss. *Appendix B* of the paper titled *Architectural Details* provides a summary of the modules and their configurations used in the generator and discriminator models. There are two versions of the model described: BigGAN and BigGAN-deep, the latter involving deeper resnet modules and, in turn, achieving better results.

27.4.2 Class Conditional Information

The class information is provided to the generator model via class-conditional batch normalization. This was described by Vincent Dumoulin, et al. in their 2016 paper titled *A Learned Representation For Artistic Style*. In the paper, the technique is referred to as *conditional instance normalization* that involves normalizing activations based on the statistics from images of a given style, or in the case of BigGAN, images of a given class.

> We call this approach conditional instance normalization. The goal of the procedure is [to] transform a layer's activations x into a normalized activation z specific to painting style s.

— *A Learned Representation For Artistic Style*, 2016.

Class information is provided to the discriminator via projection. This is described by Takeru Miyato, et al. in their 2018 paper titled *Spectral Normalization for Generative Adversarial Networks*. This involves using an integer embedding of the class value that is concatenated into an intermediate layer of the network.

> Discriminator for conditional GANs. For computational ease, we embedded the integer label y in 0, ..., 1000 into 128 dimension before concatenating the vector to the output of the intermediate layer.

— *Spectral Normalization for Generative Adversarial Networks*, 2018.

Instead of using one class embedding per class label, a shared embedding was used in order to reduce the number of weights.

> Instead of having a separate layer for each embedding, we opt to use a shared embedding, which is linearly projected to each layer's gains and biases. This reduces computation and memory costs, and improves training speed (in number of iterations required to reach a given performance) by 37%.

— *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

27.4.3 Spectral Normalization

The weights of the generator are normalized using spectral normalization. Spectral normalization for use in GANs was described by Takeru Miyato, et al. in their 2018 paper titled *Spectral Normalization for Generative Adversarial Networks*. Specifically, it involves normalizing the spectral norm of the weight matrix.

> Our spectral normalization normalizes the spectral norm of the weight matrix W so that it satisfies the Lipschitz constraint $\sigma(W) = 1$:
>
> — *Spectral Normalization for Generative Adversarial Networks*, 2018.

The efficient implementation requires a change to the weight updates during minibatch stochastic gradient descent, described in *Appendix A* of the spectral normalization paper.

Algorithm 1 SGD with spectral normalization

- Initialize $\tilde{\boldsymbol{u}}_l \in \mathcal{R}^{d_l}$ for $l = 1, \ldots, L$ with a random vector (sampled from isotropic distribution).
- For each update and each layer l:
  1. Apply power iteration method to a unnormalized weight W^l:

$$\tilde{\boldsymbol{v}}_l \leftarrow (W^l)^{\mathrm{T}} \tilde{\boldsymbol{u}}_l / \|(W^l)^{\mathrm{T}} \tilde{\boldsymbol{u}}_l\|_2 \quad (20)$$

$$\tilde{\boldsymbol{u}}_l \leftarrow W^l \tilde{\boldsymbol{v}}_l / \|W^l \tilde{\boldsymbol{v}}_l\|_2 \quad (21)$$

  2. Calculate $\bar{W}_{\mathrm{SN}}$ with the spectral norm:

$$\bar{W}^l_{\mathrm{SN}}(W^l) = W^l / \sigma(W^l), \text{ where } \sigma(W^l) = \tilde{\boldsymbol{u}}_l^{\mathrm{T}} W^l \tilde{\boldsymbol{v}}_l \quad (22)$$

  3. Update W^l with SGD on mini-batch dataset $\mathcal{D}_M$ with a learning rate α:

$$W^l \leftarrow W^l - \alpha \nabla_{W^l} \ell(\bar{W}^l_{\mathrm{SN}}(W^l), \mathcal{D}_M) \quad (23)$$

Figure 27.2: Algorithm for SGD With Spectral Normalization. Taken from: Spectral Normalization for Generative Adversarial Networks.

27.4.4 Update Discriminator More Than Generator

In the GAN training algorithm, it is common to first update the discriminator model and then to update the generator model. The BigGAN slightly modifies this and updates the discriminator model twice before updating the generator model in each training iteration.

27.4.5 Moving Average of Model Weights

The generator model is evaluated based on the images that are generated. Before images are generated for evaluation, the model weights are averaged across prior training iterations using a moving average. This approach to model weight moving average for generator evaluation was described and used by Tero Karras, et al. in their 2017 paper titled *Progressive Growing of GANs for Improved Quality, Stability, and Variation*.

> ... for visualizing generator output at any given point during the training, we use an exponential running average for the weights of the generator with decay 0.999.
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

27.4.6 Orthogonal Weight Initialization

Model weights are initialized using Orthogonal Initialization. This was described by Andrew Saxe, et al. in their 2013 paper titled *Exact Solutions To The Nonlinear Dynamics Of Learning In Deep Linear Neural Networks*. This involves setting the weights to be a random orthogonal matrix.

> ... the initial weights in each layer to be a random orthogonal matrix (satisfying $W^T \cdot W = I$) ...

— *Exact Solutions To The Nonlinear Dynamics Of Learning In Deep Linear Neural Networks*, 2013.

Keras supports orthogonal weight initialization directly[1].

27.4.7 Larger Batch Size

Very large batch sizes were tested and evaluated. This includes batch sizes of 256, 512, 1024, and 2,048 images. Larger batch sizes generally resulted in better quality images, with the best image quality achieved with a batch size of 2,048 images.

> ... simply increasing the batch size by a factor of 8 improves the state-of-the-art IS by 46%.

— *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

The intuition is that the larger batch size provides more *modes*, and in turn, provides better gradient information for updating the models.

> We conjecture that this is a result of each batch covering more modes, providing better gradients for both networks.

— *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

27.4.8 More Model Parameters

The number of model parameters was also dramatically increased. This was achieved by doubling the number of channels or feature maps (filters) in each layer.

> We then increase the width (number of channels) in each layer by 50%, approximately doubling the number of parameters in both models. This leads to a further IS improvement of 21%, which we posit is due to the increased capacity of the model relative to the complexity of the dataset.

— *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

[1] https://keras.io/initializers/

27.4.9 Skip-z Connections

Skip connections were added to the generator model to directly connect the input latent point to specific layers deep in the network. These are referred to as skip-z connections, where z refers to the input latent vector.

> Next, we add direct skip connections (skip-z) from the noise vector z to multiple layers of G rather than just the initial layer. The intuition behind this design is to allow G to use the latent space to directly influence features at different resolutions and levels of hierarchy. [...] Skip-z provides a modest performance improvement of around 4%, and improves training speed by a further 18%.
>
> — *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

27.4.10 Truncation Trick

The truncation trick involves using a different truncated distribution for the generator's latent space during training than during inference or image synthesis. A Gaussian distribution is used during training, and a truncated Gaussian is used during inference. This is referred to as the *truncation trick*.

> We call this the Truncation Trick: truncating a z vector by resampling the values with magnitude above a chosen threshold leads to improvement in individual sample quality at the cost of reduction in overall sample variety.
>
> — *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

The truncation trick provides a trade-off between image quality or fidelity and image variety. A more narrow sampling range results in better quality, whereas a larger sampling range results in more variety in sampled images.

> This technique allows fine-grained, post-hoc selection of the trade-off between sample quality and variety for a given G.
>
> — *Large Scale GAN Training for High Fidelity Natural Image Synthesis*, 2018.

27.4.11 Orthogonal Regularization

Not all models respond well to the truncation trick. Some of the deeper models would provide saturated artifacts when the truncation trick was used. To better encourage a broader range of models to work well with the truncation trick, orthogonal regularization was used. This was introduced by Andrew Brock, et al. in their 2016 paper titled *Neural Photo Editing with Introspective Adversarial Networks*. This is related to the orthogonal weight initialization and introduces a weight regularization term to encourage the weights to maintain their orthogonal property.

> Orthogonality is a desirable quality in ConvNet filters, partially because multiplication by an orthogonal matrix leaves the norm of the original matrix unchanged. [...] we propose a simple weight regularization technique, Orthogonal Regularization, that encourages weights to be orthogonal by pushing them towards the nearest orthogonal manifold.
>
> — *Neural Photo Editing with Introspective Adversarial Networks*, 2016.

27.5 Example of Images Generated by BigGAN

The BigGAN is capable of generating large, high-quality images. In this section, we will review a few examples presented in the paper. Below are some examples of high-quality images generated by BigGAN.

Figure 27.3: Examples of High-Quality Class-Conditional Images Generated by BigGAN. Taken from: Large Scale GAN Training for High Fidelity Natural Image Synthesis.

Below are examples of large and high-quality images generated by BigGAN.

Figure 27.4: Examples of Large High-Quality Class-Conditional Images Generated by BigGAN. Taken from: Large Scale GAN Training for High Fidelity Natural Image Synthesis.

One of the issues described when training BigGAN generators is the idea of *class leakage*, a new type of failure mode. Below is an example of class leakage from a partially trained BigGAN, showing a cross between a tennis ball and perhaps a dog.

Figure 27.5: Examples of Class Leakage in an Image Generated by Partially Trained BigGAN. Taken from: Large Scale GAN Training for High Fidelity Natural Image Synthesis.

Below are some additional images generated by the BigGAN at 256×256 resolution.

Figure 27.6: Examples of Large High-Quality 256×256 Class-Conditional Images Generated by BigGAN. Taken from: Large Scale GAN Training for High Fidelity Natural Image Synthesis.

Below are some more images generated by the BigGAN at 512×512 resolution.

Figure 27.7: Examples of Large High-Quality 512 × 512 Class-Conditional Images Generated by BigGAN. Taken from: Large Scale GAN Training for High Fidelity Natural Image Synthesis.

27.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

27.6.1 Papers

- Large Scale GAN Training for High Fidelity Natural Image Synthesis, 2018.
  https://arxiv.org/abs/1809.11096
- Large Scale GAN Training for High Fidelity Natural Image Synthesis, ICLR 2019.
  https://openreview.net/forum?id=B1xsqj09Fm
- Self-Attention Generative Adversarial Networks, 2018.
  https://arxiv.org/abs/1805.08318
- A Learned Representation For Artistic Style, 2016.
  https://arxiv.org/abs/1610.07629
- Spectral Normalization for Generative Adversarial Networks, 2018.
  https://arxiv.org/abs/1802.05957
- Progressive Growing of GANs for Improved Quality, Stability, and Variation, 2017.
  https://arxiv.org/abs/1710.10196
- Exact Solutions To The Nonlinear Dynamics Of Learning In Deep Linear Neural Networks, 2013.
  https://arxiv.org/abs/1312.6120
- Neural Photo Editing with Introspective Adversarial Networks, 2016.
  https://arxiv.org/abs/1609.07093

27.6.2 Code

- BigGAN on TensorFlow Hub, Official.
  https://tfhub.dev/s?q=biggan
- BigGAN Demo.
  https://colab.research.google.com/github/tensorflow/hub/blob/master/examples/colab/biggan_generation_with_tf_hub.ipynb

27.6.3 Articles

- Reducing the Need for Labeled Data in Generative Adversarial Networks, Google AI Blog.
  https://ai.googleblog.com/2019/03/reducing-need-for-labeled-data-in.html

27.7 Summary

In this tutorial, you discovered the BigGAN model for scaling up class-conditional image synthesis. Specifically, you learned:

- Image size and training brittleness remain large problems for GANs.
- Scaling up model size and batch size can result in dramatically larger and higher-quality images.
- Specific model architectural and training configurations required to scale up GANs.

27.7.1 Next

In the next tutorial, you will discover the progressive growing GAN for incrementally increasing the capacity of the generator and discriminator models.

Chapter 28

Introduction to the Progressive Growing GAN

Progressive Growing GAN is an extension to the GAN training process that allows for the stable training of generator models that can output large high-quality images. It involves starting with a very small image and incrementally adding blocks of layers that increase the output size of the generator model and the input size of the discriminator model until the desired image size is achieved. This approach has proven effective at generating high-quality synthetic faces that are startlingly realistic. In this tutorial, you will discover the progressive growing generative adversarial network for generating large images. After reading this tutorial, you will know:

- GANs are effective at generating sharp images, although they are limited to small image sizes because of model stability.
- Progressive growing GAN is a stable approach to training GAN models to generate large high-quality images that involves incrementally increasing the size of the model during training.
- Progressive growing GAN models are capable of generating photorealistic synthetic faces and objects at high resolution that are remarkably realistic.

Let's get started.

28.1 Overview

This tutorial is divided into five parts; they are:

1. GANs Are Generally Limited to Small Images
2. Generate Large Images by Progressively Adding Layers
3. How to Progressively Grow a GAN
4. Images Generated by the Progressive Growing GAN
5. How to Configure Progressive Growing GAN Models

28.2 GANs Are Generally Limited to Small Images

Generative Adversarial Networks, or GANs for short, are an effective approach for training deep convolutional neural network models for generating synthetic images. Training a GAN model involves two models: a generator used to output synthetic images, and a discriminator model used to classify images as real or fake, which is used to train the generator model. The two models are trained together in an adversarial manner, seeking an equilibrium. Compared to other approaches, they are both fast and result in crisp images. A problem with GANs is that they are limited to small dataset sizes, often a few hundred pixels and often less than 100-pixel square images.

> GANs produce sharp images, albeit only in fairly small resolutions and with somewhat limited variation, and the training continues to be unstable despite recent progress.
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

Generating high-resolution images is believed to be challenging for GAN models as the generator must learn how to output both large structure and fine details at the same time. The high resolution makes any issues in the fine detail of generated images easy to spot for the discriminator and the training process fails.

> The generation of high-resolution images is difficult because higher resolution makes it easier to tell the generated images apart from training images ...
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

Large images, such as 1024-pixel square images, also require significantly more memory, which is in relatively limited supply on modern GPU hardware compared to main memory. As such, the batch size that defines the number of images used to update model weights each training iteration must be reduced to ensure that the large images fit into memory. This, in turn, introduces further instability into the training process.

> Large resolutions also necessitate using smaller minibatches due to memory constraints, further compromising training stability.
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

Additionally, the training of GAN models remains unstable, even in the presence of a suite of empirical techniques designed to improve the stability of the model training process.

28.3 Generate Large Images by Progressively Adding Layers

A solution to the problem of training stable GAN models for larger images is to progressively increase the number of layers during the training process. This approach is called Progressive Growing GAN, Progressive GAN, or PGGAN for short. The approach was proposed by Tero Karras, et al. from Nvidia in the 2017 paper titled *Progressive Growing of GANs for Improved Quality, Stability, and Variation* and presented at the 2018 ICLR conference.

> Our primary contribution is a training methodology for GANs where we start with low-resolution images, and then progressively increase the resolution by adding layers to the networks.
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

Progressive Growing GAN involves using a generator and discriminator model with the same general structure and starting with very small images, such as 4×4 pixels. During training, new blocks of convolutional layers are systematically added to both the generator model and the discriminator models.

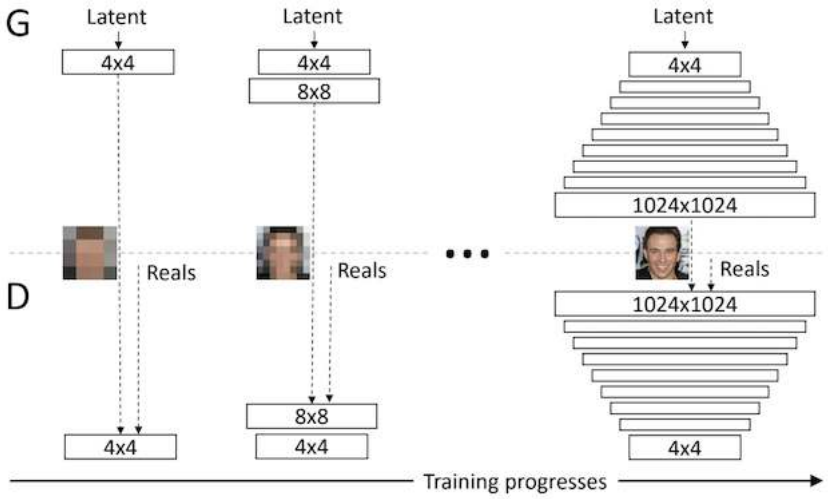

Figure 28.1: Example of Progressively Adding Layers to Generator and Discriminator Models. Taken from: Progressive Growing of GANs for Improved Quality, Stability, and Variation.

The incremental addition of the layers allows the models to effectively learn coarse-level detail and later learn ever finer detail, both on the generator and discriminator side.

> This incremental nature allows the training to first discover large-scale structure of the image distribution and then shift attention to increasingly finer scale detail, instead of having to learn all scales simultaneously.
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

This approach allows the generation of large high-quality images, such as 1024×1024 photorealistic faces of celebrities that do not exist.

28.4 How to Progressively Grow a GAN

Progressive Growing GAN requires that the capacity of both the generator and discriminator model be expanded by adding layers during the training process. This is much like the greedy layer-wise training process that was common for developing deep learning neural networks prior to the development of ReLU and Batch Normalization. Unlike greedy layer-wise pre-training, progressive growing GAN involves adding blocks of layers and phasing in the addition of the blocks of layers rather than adding them directly.

> When new layers are added to the networks, we fade them in smoothly [...] This avoids sudden shocks to the already well-trained, smaller-resolution layers.
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

Further, all layers remain trainable during the training process, including existing layers when new layers are added.

> All existing layers in both networks remain trainable throughout the training process.
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

The phasing in of a new block of layers involves using a skip connection to connect the new block to the input of the discriminator or output of the generator and adding it to the existing input or output layer with a weighting. The weighting controls the influence of the new block and is achieved using a parameter alpha (α) that starts at zero or a very small number and linearly increases to 1.0 over training iterations. This is demonstrated in the figure below, taken from the paper. It shows a generator that outputs a 16×16 image and a discriminator that takes a 16×16 pixel image. The models are grown to the size of 32×32.

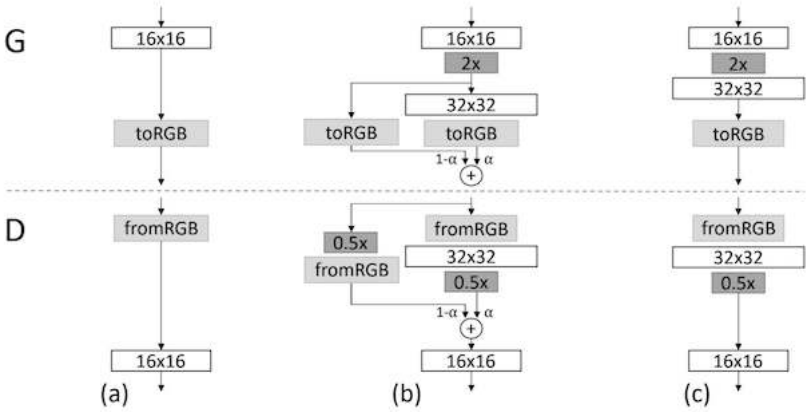

Figure 28.2: Example of Phasing in the Addition of New Layers to the Generator and Discriminator Models. Taken from: Progressive Growing of GANs for Improved Quality, Stability, and Variation.

Let's take a closer look at how to progressively add layers to the generator and discriminator when going from 16×16 to 32×32 pixels.

28.4.1 Growing the Generator

For the generator, this involves adding a new block of convolutional layers that outputs a 32×32 image. The output of this new layer is combined with the output of the 16×16 layer that is upsampled using nearest neighbor interpolation to 32×32. This is different from many GAN generators that use a transpose convolutional layer.

> ... doubling [...] the image resolution using nearest neighbor filtering

> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

The contribution of the upsampled 16×16 layer is weighted by $(1 - \alpha)$, whereas the contribution of the new 32×32 layer is weighted by α. Alpha is small initially, giving the most weight to the scaled-up version of the 16×16 image, although slowly transitions to giving more weight and then all weight to the new 32×32 output layers over training iterations.

> During the transition we treat the layers that operate on the higher resolution like a residual block, whose weight α increases linearly from 0 to 1.

> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

28.4.2 Growing the Discriminator

For the discriminator, this involves adding a new block of convolutional layers for the input of the model to support image sizes with 32×32 pixels. The input image is downsampled to 16×16 using average pooling so that it can pass through the existing 16×16 convolutional layers. The output of the new 32×32 block of layers is also downsampled using average pooling so that it can be provided as input to the existing 16×16 block. This is different from most GAN models that use a 2×2 stride in the convolutional layers to downsample.

> ... halving the image resolution using [...] average pooling

> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

The two downsampled versions of the input are combined in a weighted manner, starting with a full weighting to the downsampled raw input and linearly transitioning to a full weighting for the interpreted output of the new input layer block.

28.5 Images Generated by the Progressive Growing GAN

In this section, we can review some of the impressive results achieved with the Progressive Growing GAN described in the paper. Many example images are provided in the appendix of the paper and I recommend reviewing it. Additionally, a YouTube video was also created summarizing the impressive results of the model.

- Progressive Growing of GANs for Improved Quality, Stability, and Variation, YouTube. [1]

28.5.1 Synthetic Photographs of Celebrity Faces

Perhaps the most impressive accomplishment of the Progressive Growing GAN is the generation of large 1024×1024 pixel photorealistic generated faces. The model was trained on a high-quality version of the celebrity faces dataset, called CELEBA-HQ. As such, the faces look familiar as they contain elements of many real celebrity faces, although none of the people actually exist.

[1] https://www.youtube.com/watch?v=G06dEcZ-QTg

Figure 28.3: Example of Photorealistic Generated Faces Using Progressive Growing GAN. Taken from: Progressive Growing of GANs for Improved Quality, Stability, and Variation.

Interestingly, the model required to generate the faces was trained on 8 GPUs for 4 days, perhaps out of the range of most developers.

> We trained the network on 8 Tesla V100 GPUs for 4 days, after which we no longer observed qualitative differences between the results of consecutive training iterations. Our implementation used an adaptive minibatch size depending on the current output resolution so that the available memory budget was optimally utilized.
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

28.5.2 Synthetic Photographs of Objects

The model was also demonstrated on generating 256×256-pixel photorealistic synthetic objects from the LSUN dataset, such as bikes, buses, and churches.

Figure 28.4: Example of Photorealistic Generated Objects Using Progressive Growing GAN. Taken from: Progressive Growing of GANs for Improved Quality, Stability, and Variation.

28.6 How to Configure Progressive Growing GAN Models

The paper describes the configuration details of the model used to generate the 1024×1024 synthetic photographs of celebrity faces. Specifically, the details are provided in *Appendix A*. Although we may not be interested or have the resources to develop such a large model, the configuration details may be useful when implementing a Progressive Growing GAN. Both the discriminator and generator models were grown using blocks of convolutional layers, each using a specific number of filters with the size 3×3 and the `LeakyReLU` activation layer with the slope of 0.2. Upsampling was achieved via nearest neighbor sampling and downsampling was achieved using average pooling.

> Both networks consist mainly of replicated 3-layer blocks that we introduce one by one during the course of the training. [...] We use leaky ReLU with leakiness 0.2 in all layers of both networks, except for the last layer that uses linear activation.
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

The generator used a 512-element vector of Gaussian random variables. It also used an output layer with a 1×1-sized filters and a linear activation function, instead of the more common hyperbolic tangent activation function (Tanh). The discriminator also used an output layer with 1×1-sized filters and a linear activation function. The Wasserstein GAN loss was used with the gradient penalty, so-called WGAN-GP as described in the 2017 paper titled *Improved Training of Wasserstein GANs*. The least squares loss was tested and showed good results, but not as good as WGAN-GP. The models start with a 4×4 input image and grow until they reach the 1024×1024 target. Tables were provided that list the number of layers

and number of filters used in each layer for the generator and discriminator models, reproduced below.

Generator	Act.	Output shape	Params
Latent vector	–	512 × 1 × 1	–
Conv 4 × 4	LReLU	512 × 4 × 4	4.2M
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Upsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Upsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Upsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Upsample	–	512 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	1.2M
Conv 3 × 3	LReLU	256 × 64 × 64	590k
Upsample	–	256 × 128 × 128	–
Conv 3 × 3	LReLU	128 × 128 × 128	295k
Conv 3 × 3	LReLU	128 × 128 × 128	148k
Upsample	–	128 × 256 × 256	–
Conv 3 × 3	LReLU	64 × 256 × 256	74k
Conv 3 × 3	LReLU	64 × 256 × 256	37k
Upsample	–	64 × 512 × 512	–
Conv 3 × 3	LReLU	32 × 512 × 512	18k
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Upsample	–	32 × 1024 × 1024	–
Conv 3 × 3	LReLU	16 × 1024 × 1024	4.6k
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Conv 1 × 1	linear	3 × 1024 × 1024	51
Total trainable parameters			**23.1M**

Discriminator	Act.	Output shape	Params
Input image	–	3 × 1024 × 1024	–
Conv 1 × 1	LReLU	16 × 1024 × 1024	64
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Conv 3 × 3	LReLU	32 × 1024 × 1024	4.6k
Downsample	–	32 × 512 × 512	–
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Conv 3 × 3	LReLU	64 × 512 × 512	18k
Downsample	–	64 × 256 × 256	–
Conv 3 × 3	LReLU	64 × 256 × 256	37k
Conv 3 × 3	LReLU	128 × 256 × 256	74k
Downsample	–	128 × 128 × 128	–
Conv 3 × 3	LReLU	128 × 128 × 128	148k
Conv 3 × 3	LReLU	256 × 128 × 128	295k
Downsample	–	256 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	590k
Conv 3 × 3	LReLU	512 × 64 × 64	1.2M
Downsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Downsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Downsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Downsample	–	512 × 4 × 4	–
Minibatch stddev	–	513 × 4 × 4	–
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Conv 4 × 4	LReLU	512 × 1 × 1	4.2M
Fully-connected	linear	1 × 1 × 1	513
Total trainable parameters			**23.1M**

Figure 28.5: Tables Showing Generator and Discriminator Configuration for the Progressive Growing GAN. Taken from: Progressive Growing of GANs for Improved Quality, Stability, and Variation.

Batch normalization is not used; instead, two other techniques are added, including minibatch standard deviation pixel-wise normalization. The standard deviation of activations across images in the minibatch is added as a new channel prior to the last block of convolutional layers in the discriminator model. This is referred to as *Minibatch standard deviation*.

> We inject the across-minibatch standard deviation as an additional feature map at 4×4 resolution toward the end of the discriminator
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

A pixel-wise normalization is performed in the generator after each convolutional layer that normalizes each pixel value in the activation map across the channels to a unit length. This is a type of activation constraint that is more generally referred to as *local response normalization*. The bias for all layers is initialized as zero and model weights are initialized as a random Gaussian rescaled using the He weight initialization method.

> We initialize all bias parameters to zero and all weights according to the normal distribution with unit variance. However, we scale the weights with a layer-specific constant at runtime ...
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

The models are optimized using the Adam version of stochastic gradient descent with a small learning rate and low momentum.

> We train the networks using Adam with $a = 0.001$, $B1 = 0$, $B2 = 0.99$, and eta = 10^{-8}.
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

Image generation uses a weighted average of prior models rather a given model snapshot, much like a horizontal ensemble.

> ... visualizing generator output at any given point during the training, we use an exponential running average for the weights of the generator with decay 0.999
>
> — *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, 2017.

28.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Progressive Growing of GANs for Improved Quality, Stability, and Variation, 2017.
  https://arxiv.org/abs/1710.10196
- Progressive Growing of GANs for Improved Quality, Stability, and Variation, Official.
  https://research.nvidia.com/publication/2017-10_Progressive-Growing-of
- progressive_growing_of_gans Project (official), GitHub.
  https://github.com/tkarras/progressive_growing_of_gans
- Progressive Growing of GANs for Improved Quality, Stability, and Variation. Open Review.
  https://openreview.net/forum?id=Hk99zCeAb&noteId=Hk99zCeAb
- Progressive Growing of GANs for Improved Quality, Stability, and Variation, YouTube.
  https://www.youtube.com/watch?v=G06dEcZ-QTg

28.8 Summary

In this tutorial, you discovered the progressive growing generative adversarial network for generating large images. Specifically, you learned:

- GANs are effective at generating sharp images, although they are limited to small image sizes because of model stability.
- Progressive growing GAN is a stable approach to training GAN models to generate large high-quality images that involves incrementally increasing the size of the model during training.
- Progressive growing GAN models are capable of generating photorealistic synthetic faces and objects at high resolution that are remarkably realistic.

28.8.1 Next

In the next tutorial, you will discover the StyleGAN that gives fine-grained control over the style of generated images.

Chapter 29

Introduction to the StyleGAN

Generative Adversarial Networks, or GANs for short, are effective at generating large high-quality images. Most improvement has been made to discriminator models in an effort to train more effective generator models, although less effort has been put into improving the generator models. The Style Generative Adversarial Network, or StyleGAN for short, is an extension to the GAN architecture that proposes large changes to the generator model, including the use of a mapping network to map points in latent space to an intermediate latent space, the use of the intermediate latent space to control style at each point in the generator model, and the introduction to noise as a source of variation at each point in the generator model.

The resulting model is capable not only of generating impressively photorealistic high-quality photos of faces, but also offers control over the style of the generated image at different levels of detail through varying the style vectors and noise. In this tutorial, you will discover the Style Generative Adversarial Network that gives control over the style of generated synthetic images. After reading this tutorial, you will know:

- The lack of control over the style of synthetic images generated by traditional GAN models.
- The architecture of StyleGAN model that introduces control over the style of generated images at different levels of detail.
- Impressive results achieved with the StyleGAN architecture when used to generate synthetic human faces.

Let's get started.

29.1 Overview

This tutorial is divided into four parts; they are:

1. Lacking Control Over Synthesized Images
2. Control Style Using New Generator Model
3. What Is the StyleGAN Model Architecture
4. Examples of StyleGAN Generated Images

29.2 Lacking Control Over Synthesized Images

Generative adversarial networks are effective at generating high-quality and large-resolution synthetic images. The generator model takes as input a point from latent space and generates an image. This model is trained by a second model, called the discriminator, that learns to differentiate real images from the training dataset from fake images generated by the generator model. As such, the two models compete in an adversarial game and find a balance or equilibrium during the training process. Many improvements to the GAN architecture have been achieved through enhancements to the discriminator model. These changes are motivated by the idea that a better discriminator model will, in turn, lead to the generation of more realistic synthetic images.

As such, the generator has been somewhat neglected and remains a black box. For example, the source of randomness used in the generation of synthetic images is not well understood, including both the amount of randomness in the sampled points and the structure of the latent space.

> Yet the generators continue to operate as black boxes, and despite recent efforts, the understanding of various aspects of the image synthesis process, [...] is still lacking. The properties of the latent space are also poorly understood ...
>
> — *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018.

This limited understanding of the generator is perhaps most exemplified by the general lack of control over the generated images. There are few tools to control the properties of generated images, e.g. the style. This includes high-level features such as background and foreground, and fine-grained details such as the features of synthesized objects or subjects. This requires both disentangling features or properties in images and adding controls for these properties to the generator model.

29.3 Control Style Using New Generator Model

The Style Generative Adversarial Network, or StyleGAN for short, is an extension to the GAN architecture to give control over the disentangled style properties of generated images.

> Our generator starts from a learned constant input and adjusts the "style" of the image at each convolution layer based on the latent code, therefore directly controlling the strength of image features at different scales
>
> — *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018.

The StyleGAN is an extension of the progressive growing GAN that is an approach for training generator models capable of synthesizing very large high-quality images via the incremental expansion of both discriminator and generator models from small to large images during the training process. In addition to the incremental growing of the models during training, the style GAN changes the architecture of the generator significantly. The StyleGAN generator no longer takes a point from the latent space as input; instead, there are two new sources of randomness used to generate a synthetic image: a standalone mapping network and noise layers.

The output from the mapping network is a vector that defines the styles that is integrated at each point in the generator model via a new layer called adaptive instance normalization. The use of this style vector gives control over the style of the generated image. Stochastic variation is introduced through noise added at each point in the generator model. The noise is added to entire feature maps that allow the model to interpret the style in a fine-grained, per-pixel manner. This per-block incorporation of style vector and noise allows each block to localize both the interpretation of style and the stochastic variation to a given level of detail.

> The new architecture leads to an automatically learned, unsupervised separation of high-level attributes (e.g., pose and identity when trained on human faces) and stochastic variation in the generated images (e.g., freckles, hair), and it enables intuitive, scale-specific control of the synthesis
>
> — *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018.

29.4 What Is the StyleGAN Model Architecture

The StyleGAN is described as a progressive growing GAN architecture with five modifications, each of which was added and evaluated incrementally in an ablative study. The incremental list of changes to the generator are:

- Baseline Progressive GAN.
- Addition of tuning and bilinear upsampling.
- Addition of mapping network and AdaIN (styles).
- Removal of latent vector input to generator.
- Addition of noise to each block.
- Addition Mixing regularization.

The image below summarizes the StyleGAN generator architecture.

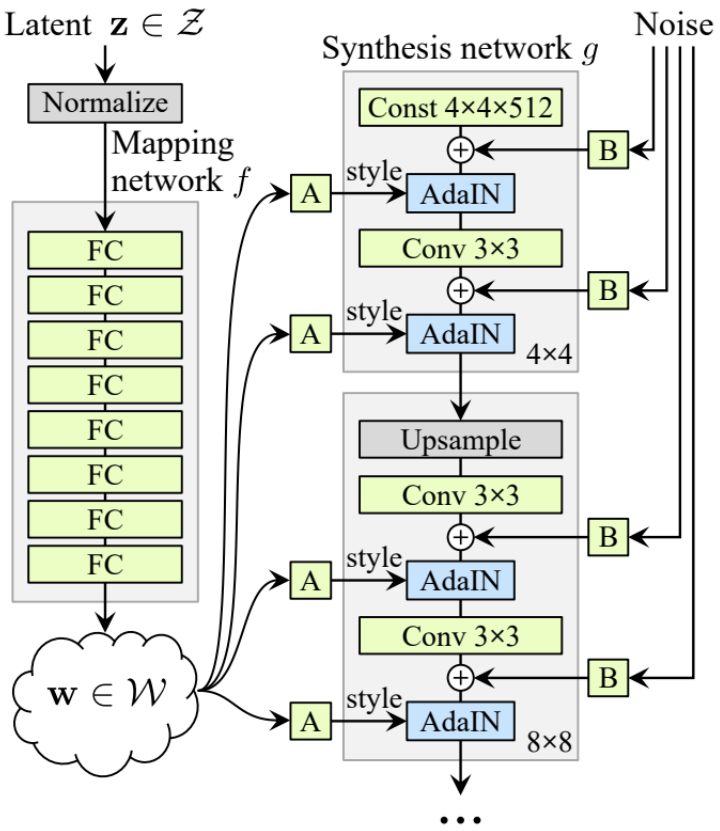

Figure 29.1: Summary of the StyleGAN Generator Model Architecture. Taken from: A Style-Based Generator Architecture for Generative Adversarial Networks.

We can review each of these changes in more detail.

29.4.1 Baseline Progressive GAN

The StyleGAN generator and discriminator models are trained using the progressive growing GAN training method (see Chapter 28). This means that both models start with small images, in this case, 4×4 images. The models are fit until stable, then both discriminator and generator are expanded to double the width and height (quadruple the area), e.g. 8×8. A new block is added to each model to support the larger image size, which is faded in slowly over training. Once faded-in, the models are again trained until reasonably stable and the process is repeated with ever-larger image sizes until the desired target image size is met, such as 1024×1024.

29.4.2 Bilinear Sampling

The progressive growing GAN uses nearest neighbor layers for upsampling instead of transpose convolutional layers that are common in other generator models. The first point of deviation in the StyleGAN is that bilinear upsampling layers are used instead of nearest neighbor.

> We replace the nearest-neighbor up/downsampling in both networks with bilinear sampling, which we implement by lowpass filtering the activations with a separable 2nd order binomial filter after each upsampling layer and before each downsampling layer.

> — *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018.

29.4.3 Mapping Network and AdaIN

Next, a standalone mapping network is used that takes a randomly sampled point from the latent space as input and generates a style vector. The mapping network is comprised of eight fully connected layers.

> For simplicity, we set the dimensionality of both [the latent and intermediate latent] spaces to 512, and the mapping f is implemented using an 8-layer MLP ...
>
> — *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018.

The style vector is then transformed and incorporated into each block of the generator model after the convolutional layers via an operation called adaptive instance normalization or AdaIN. The AdaIN layers involve first standardizing the output of feature map to a standard Gaussian, then adding the style vector as a bias term.

> Learned affine transformations then specialize [the intermediate latent vector] to styles $y = (ys, yb)$ that control adaptive instance normalization (AdaIN) operations after each convolution layer of the synthesis network g.
>
> — *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018.

$$\text{AdaIN}(x_i, y) = y_{s,i}\frac{x_i - \mu(x_i)}{\sigma(x_i)} + y_{b,i} \tag{29.1}$$

The addition of the new mapping network to the architecture also results in the renaming of the generator model to a *synthesis network*.

29.4.4 Removal of Latent Point Input

The next change involves modifying the generator model so that it no longer takes a point from the latent space as input. Instead, the model has a constant $4 \times 4 \times 512$ constant value input in order to start the image synthesis process.

29.4.5 Addition of Noise

The output of each convolutional layer in the synthesis network is a block of activation maps. Gaussian noise is added to each of these activation maps prior to the AdaIN operations. A different sample of noise is generated for each block and is interpreted using per-layer scaling factors.

> These are single-channel images consisting of uncorrelated Gaussian noise, and we feed a dedicated noise image to each layer of the synthesis network. The noise image is broadcasted to all feature maps using learned per-feature scaling factors and then added to the output of the corresponding convolution ...
>
> — *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018.

This noise is used to introduce style-level variation at a given level of detail.

29.4.6 Mixing regularization

Mixing regularization involves first generating two style vectors from the mapping network. A split point in the synthesis network is chosen and all AdaIN operations prior to the split point use the first style vector and all AdaIN operations after the split point get the second style vector.

> ... we employ mixing regularization, where a given percentage of images are generated using two random latent codes instead of one during training.
>
> — *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018.

This encourages the layers and blocks to localize the style to specific parts of the model and corresponding level of detail in the generated image.

29.5 Examples of StyleGAN Generated Images

The StyleGAN is both effective at generating large high-quality images and at controlling the style of the generated images. In this section, we will review some examples of generated images. A video demonstrating the capability of the model was released by the authors of the paper, providing a useful overview.

- StyleGAN Results Video, YouTube. [1]

29.5.1 High-Quality Faces

The image below taken from the paper shows synthetic faces generated with the StyleGAN with the sizes 4×4, 8×8, 16×16, and 32×32.

Figure 29.2: Example of High-Quality Generated Faces Using the StyleGAN. Taken from: A Style-Based Generator Architecture for Generative Adversarial Networks.

[1] https://www.youtube.com/watch?v=kSLJriaOumA

29.5.2 Varying Style by Level of Detail

The use of different style vectors at different points of the synthesis network gives control over the styles of the resulting image at different levels of detail. For example, blocks of layers in the synthesis network at lower resolutions (e.g. 4×4 and 8×8) control high-level styles such as pose and hairstyle. Blocks of layers in the middle of the network (e.g. as 16×16 and 32×32) control hairstyles and facial expression. Finally, blocks of layers closer to the output end of the network (e.g. 64×64 to 1024×1024) control color schemes and very fine details.

The image below taken from the paper shows generated images on the left and across the top. The two rows of intermediate images are examples of the style vectors used to generate the images on the left, where the style vectors used for the images on the top are used only in the lower levels. This allows the images on the left to adopt high-level styles such as pose and hairstyle from the images on the top in each column.

> Copying the styles corresponding to coarse spatial resolutions (4^2 - 8^2) brings high-level aspects such as pose, general hair style, face shape, and eyeglasses from source B, while all colors (eyes, hair, lighting) and finer facial features resemble A.
>
> — *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018.

Figure 29.3: Example of One Set of Generated Faces (Left) Adopting the Coarse Style of Another Set of Generated Faces (Top). Taken from: A Style-Based Generator Architecture for Generative Adversarial Networks.

29.5.3 Varying Style by Level of Detail

The authors varied the use of noise at different levels of detail in the model (e.g. fine, middle, coarse), much like the previous example of varying style. The result is that noise gives control over the generation of detail, from broader structure when noise is used in the coarse blocks of layers to the generation of fine detail when noise is added to the layers closer to the output of the network.

> We can see that the artificial omission of noise leads to featureless "painterly" look. Coarse noise causes large-scale curling of hair and appearance of larger background features, while the fine noise brings out the finer curls of hair, finer background detail, and skin pores.

— *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018.

Figure 29.4: Example of Varying Noise at Different Levels of the Generator Model. Taken from: A Style-Based Generator Architecture for Generative Adversarial Networks.

29.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- A Style-Based Generator Architecture for Generative Adversarial Networks, 2018. https://arxiv.org/abs/1812.04948
- Progressive Growing of GANs for Improved Quality, Stability, and Variation, 2017. https://arxiv.org/abs/1710.10196
- StyleGAN - Official TensorFlow Implementation, GitHub. https://github.com/NVlabs/stylegan
- StyleGAN Results Video, YouTube. https://www.youtube.com/watch?v=kSLJriaOumA

29.7 Summary

In this tutorial, you discovered the Style Generative Adversarial Network that gives control over the style of generated synthetic images. Specifically, you learned:

- The lack of control over the style of synthetic images generated by traditional GAN models.
- The architecture of StyleGAN model GAN model that introduces control over the style of generated images at different levels of detail
- Impressive results achieved with the StyleGAN architecture when used to generate synthetic human faces.

29.7.1 Next

This was the final tutorial in this part. In the next part, you will discover additional resources and tutorials for setting up your workstation and running large models in the cloud.

Part VIII

Appendix

Appendix A

Getting Help

This is just the beginning of your journey with Generative Adversarial Networks with Python. As you start to work on methods or expand your existing knowledge of algorithms you may need help. This chapter points out some of the best sources of help.

A.1 Applied Neural Networks

This section lists some of the best books on the practical considerations of working with neural network models.

- *Deep Learning With Python*, 2017.
  https://amzn.to/2NJq1pf
- *Deep Learning*, 2016.
  https://amzn.to/2NJW3gE
- *Neural Networks: Tricks of the Trade*, 2012.
  https://amzn.to/2S83KiB
- *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
  https://amzn.to/2poqOxc
- *Neural Networks for Pattern Recognition*, 1995.
  https://amzn.to/2I9gNMP

A.2 Programming Computer Vision Books

Programming books can be useful when getting started in computer vision. Specifically, for providing more insight into using top computer vision libraries such as PIL and OpenCV. The list below provides a selection of some of the top computer vision programming books:

- *Learning OpenCV 3*, 2017.
  https://amzn.to/2EqF0Pv
- *Programming Computer Vision with Python*, 2012.
  https://amzn.to/2QKTaAL

- *Practical Computer Vision with SimpleCV*, 2012.
https://amzn.to/2QnFqMY

A.3 Official Keras Destinations

This section lists the official Keras sites that you may find helpful.

- Keras Official Blog.
https://blog.keras.io/
- Keras API Documentation.
https://keras.io/
- Keras Source Code Project.
https://github.com/keras-team/keras

A.4 Where to Get Help with Keras

This section lists the 9 best places I know where you can get help with Keras.

- Keras Users Google Group.
https://groups.google.com/forum/#!forum/keras-users
- Keras Slack Channel (you must request to join).
https://keras-slack-autojoin.herokuapp.com/
- Keras on Gitter.
https://gitter.im/Keras-io/Lobby#
- Keras tag on StackOverflow.
https://stackoverflow.com/questions/tagged/keras
- Keras tag on CrossValidated.
https://stats.stackexchange.com/questions/tagged/keras
- Keras tag on DataScience.
https://datascience.stackexchange.com/questions/tagged/keras
- Keras Topic on Quora.
https://www.quora.com/topic/Keras
- Keras GitHub Issues.
https://github.com/keras-team/keras/issues
- Keras on Twitter.
https://twitter.com/hashtag/keras

A.5 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

- Boil your question down to the simplest form. E.g. not something broad like *my model does not work* or *how does x work*.
- Search for answers before asking questions.
- Provide complete code and error messages.
- Boil your code down to the smallest possible working example that demonstrates the issue.

These are excellent resources both for posting unique questions, but also for searching through the answers to questions on related topics.

A.6 Contact the Author

You are not alone. If you ever have any questions about GANs or this book, please contact me directly. I will do my best to help.

Jason Brownlee
`Jason@MachineLearningMastery.com`

Appendix B

How to Setup Python on Your Workstation

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning and deep learning software. These instructions are suitable for Windows, macOS, and Linux platforms. I will demonstrate them on macOS, so you may see some mac dialogs and file extensions.

B.1 Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda
4. Install Deep Learning Libraries

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2 Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

- 1. Visit the Anaconda homepage.
  https://www.continuum.io/
- 2. Click `Anaconda` from the menu and click `Download` to go to the download page.
  https://www.continuum.io/downloads

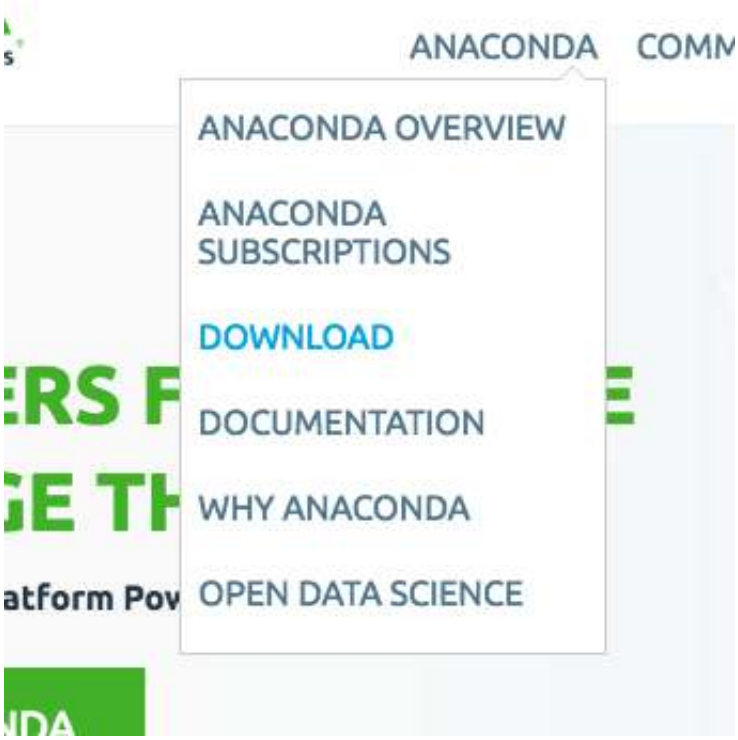

Figure B.1: Click Anaconda and Download.

- 3. Choose the download suitable for your platform (Windows, OSX, or Linux):
  - Choose Python 3.6
  - Choose the Graphical Installer

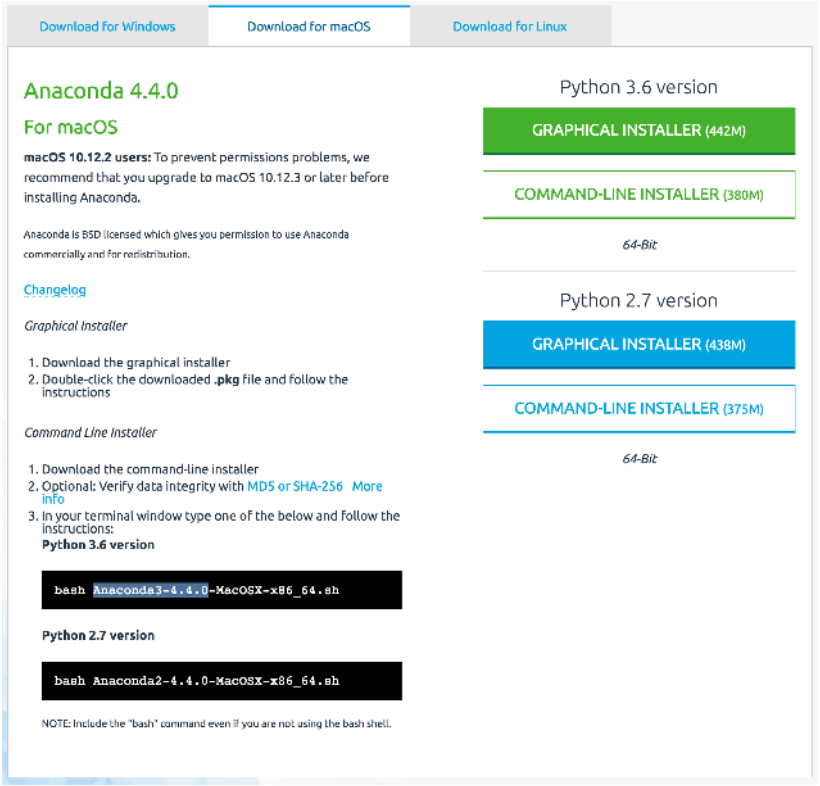

Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on macOS, so I chose the macOS version. The file is about 426 MB. You should have a file with a name like:

```
Anaconda3-4.4.0-MacOSX-x86_64.pkg
```

Listing B.1: Example filename on macOS.

B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

- 1. Double click the downloaded file.
- 2. Follow the installation wizard.

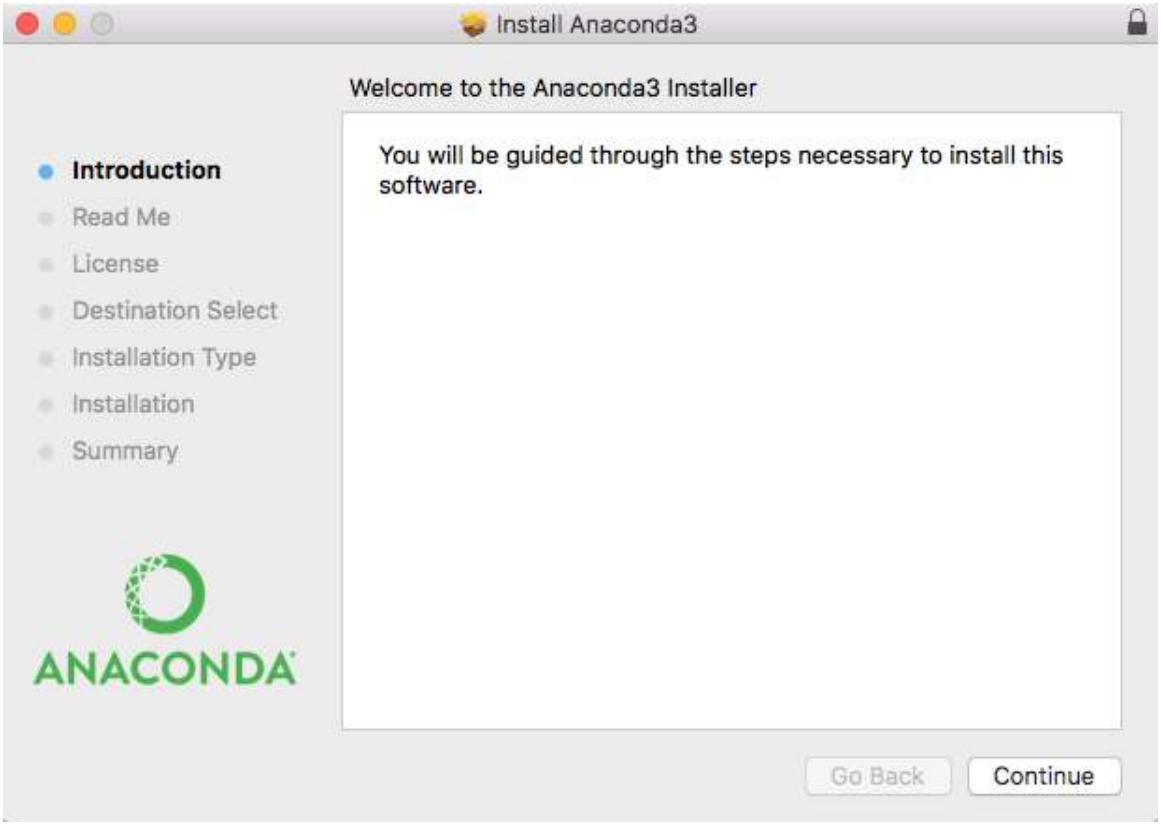

Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.

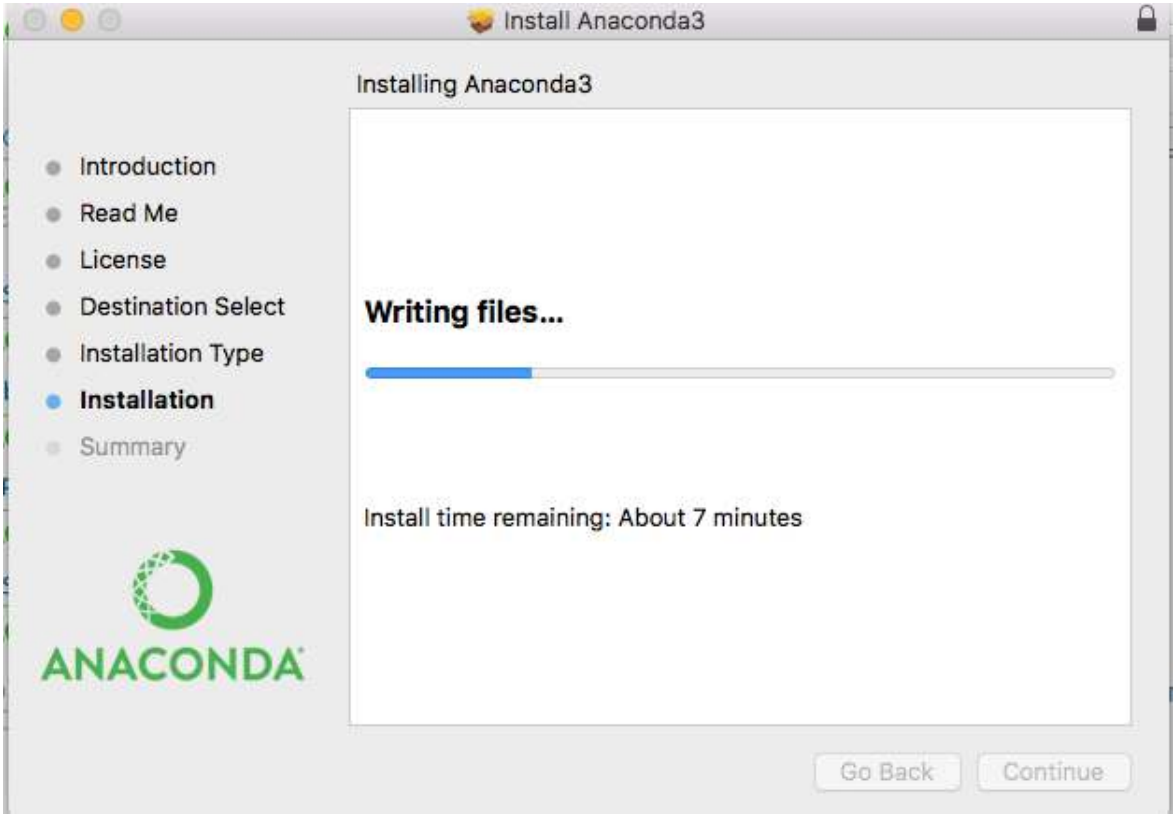

Figure B.4: Anaconda Python Installation Wizard Writing Files.

The installation should take less than 10 minutes and take up a little more than 1 GB of space on your hard drive.

B.4 Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.

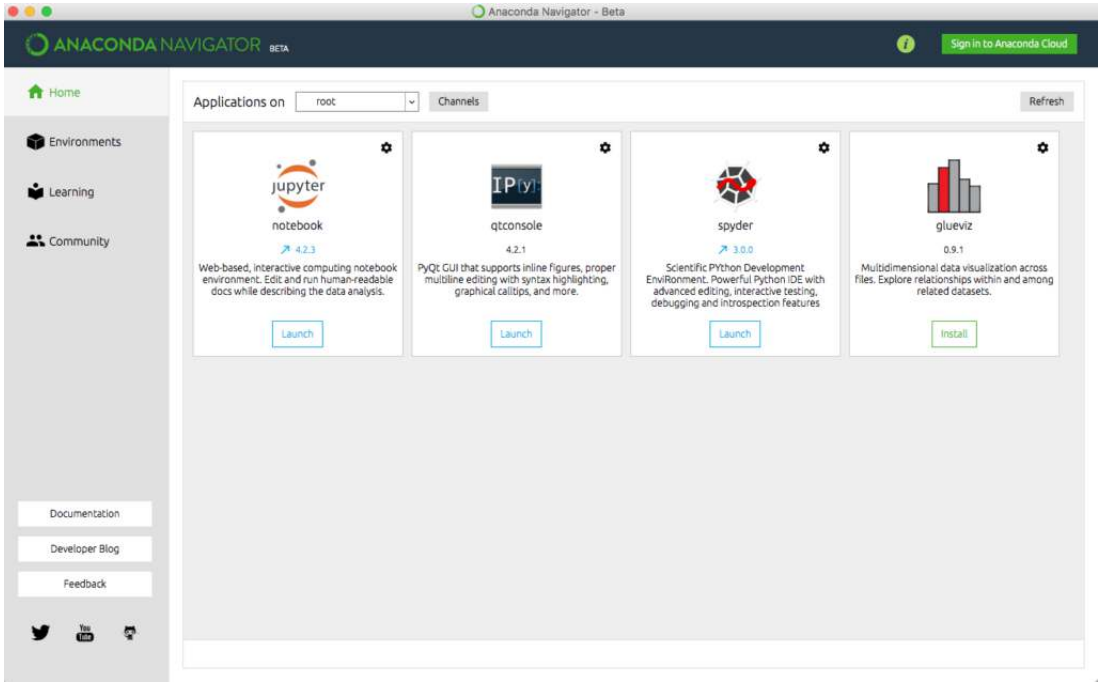

Figure B.5: Anaconda Navigator GUI.

You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

- 1. Open a terminal (command line window).
- 2. Confirm conda is installed correctly, by typing:

```
conda -V
```

Listing B.2: Check the conda version.

You should see the following (or something similar):

```
conda 4.3.21
```

Listing B.3: Example conda version.

- 3. Confirm Python is installed correctly by typing:

```
python -V
```

Listing B.4: Check the Python version.

You should see the following (or something similar):

```
Python 3.6.1 :: Anaconda 4.4.0 (x86_64)
```

Listing B.5: Example Python version.

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the *Further Reading* section.

- 4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

Listing B.6: Update conda and anaconda.

You may need to install some packages and confirm the updates.

- 5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type `python` and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```
# check library version numbers
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.7: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

Listing B.8: Run the script from the command line.

You should see output like the following:

```
scipy: 1.5.2
numpy: 1.19.1
matplotlib: 3.3.0
pandas: 1.1.0
statsmodels: 0.11.1
sklearn: 0.23.2
```

Listing B.9: Sample output of versions script.

B.5 Install Deep Learning Libraries

In this step, we will install Python libraries used for deep learning, specifically: Theano, TensorFlow, and Keras. Note: I recommend using Keras for deep learning and Keras only requires one of Theano or TensorFlow to be installed. You do not need both. There may be problems installing TensorFlow on some Windows machines.

- 1. Install the Theano deep learning library by typing:

```
conda install theano
```

Listing B.10: Install Theano with conda.

- 2. Install the TensorFlow deep learning library by typing:

```
conda install -c conda-forge tensorflow
```

Listing B.11: Install TensorFlow with conda.

Alternatively, you may choose to install using `pip` and a specific version of TensorFlow for your platform.

- 3. Install Keras by typing:

```
pip install keras
```

Listing B.12: Install Keras with `pip`.

- 4. Confirm your deep learning environment is installed and working correctly.

Create a script that prints the version numbers of each library, as we did before for the SciPy environment.

```
# check deep learning version numbers
# theano
import theano
print('theano: %s' % theano.__version__)
# tensorflow
import tensorflow
print('tensorflow: %s' % tensorflow.__version__)
# keras
import keras
print('keras: %s' % keras.__version__)
```

Listing B.13: Code to check that key deep learning libraries are installed.

Save the script to a file `deep_versions.py`. Run the script by typing:

```
python deep_versions.py
```

Listing B.14: Run script from the command line.

You should see output like:

```
theano: 1.0.5
tensorflow: 2.3.0
keras: 2.4.3
```

Listing B.15: Sample output of the deep learning versions script.

B.6 Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda homepage.
  https://www.continuum.io/
- Anaconda Navigator.
  https://docs.continuum.io/anaconda/navigator.html
- The conda command line tool.
  http://conda.pydata.org/docs/index.html
- Instructions for installing TensorFlow in Anaconda.
  https://www.tensorflow.org/get_started/os_setup#anaconda_installation

B.7 Summary

Congratulations, you now have a working Python development environment for machine learning and deep learning. You can now learn and practice machine learning and deep learning on your workstation.

Appendix C

How to Setup Amazon EC2 for Deep Learning on GPUs

Large deep learning models require a lot of compute time to run. You can run them on your CPU but it can take hours or days to get a result. If you have access to a GPU on your desktop, you can drastically speed up the training time of your deep learning models. In this project you will discover how you can get access to GPUs to speed up the training of your deep learning models by using the Amazon Web Service (AWS) infrastructure. For less than a dollar per hour and often a lot cheaper you can use this service from your workstation or laptop. After working through this project you will know:

- How to create an account and log-in to Amazon Web Service.
- How to launch a server instance for deep learning.
- How to configure a server instance for faster deep learning on the GPU.

Let's get started.

C.1 Overview

The process is quite simple because most of the work has already been done for us. Below is an overview of the process.

- Setup Your AWS Account.
- Launch Your Server Instance.
- Login and Run Your Code.
- Close Your Server Instance.

Note, it costs money to use a virtual server instance on Amazon. The cost is low for model development (e.g. less than one US dollar per hour), which is why this is so attractive, but it is not free. The server instance runs Linux. It is desirable although not required that you know how to navigate Linux or a Unix-like environment. We're just running our Python scripts, so no advanced skills are needed.

Note: The specific versions may differ as the software and libraries are updated frequently.

C.2 Setup Your AWS Account

You need an account on Amazon Web Services[1].

- 1. You can create account by the Amazon Web Services portal and click *Sign in to the Console*. From there you can sign in using an existing Amazon account or create a new account.

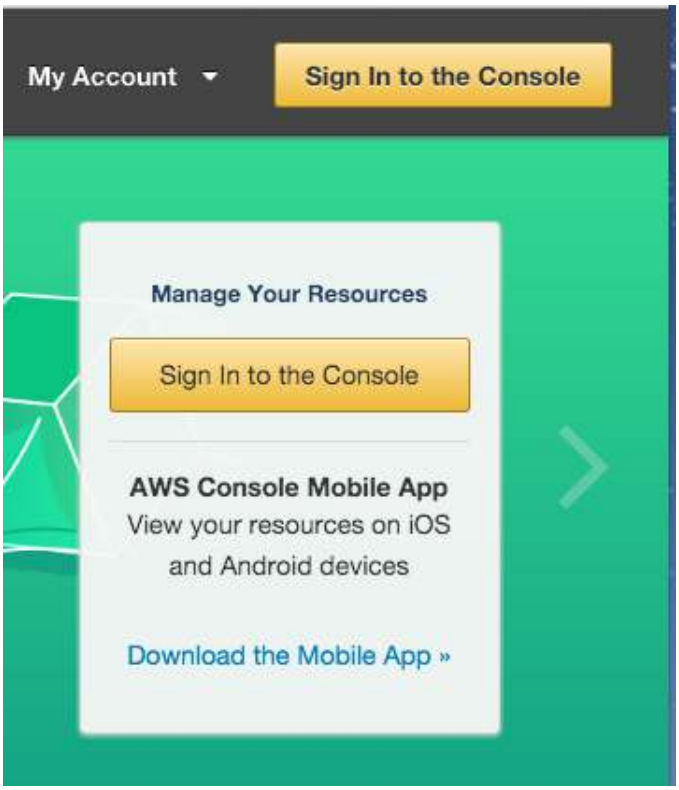

Figure C.1: AWS Sign-in Button

- 2. You will need to provide your details as well as a valid credit card that Amazon can charge. The process is a lot quicker if you are already an Amazon customer and have your credit card on file.

[1] https://aws.amazon.com

Figure C.2: AWS Sign-In Form

Once you have an account you can log into the Amazon Web Services console. You will see a range of different services that you can access.

C.3 Launch Your Server Instance

Now that you have an AWS account, you want to launch an EC2 virtual server instance on which you can run Keras. Launching an instance is as easy as selecting the image to load and starting the virtual server. Thankfully there is already an image available that has almost everything we need it is called the **Deep Learning AMI (Amazon Linux)** and was created and is maintained by Amazon. Let's launch it as an instance.

- 1. Login to your AWS console if you have not already.
  https://console.aws.amazon.com/console/home

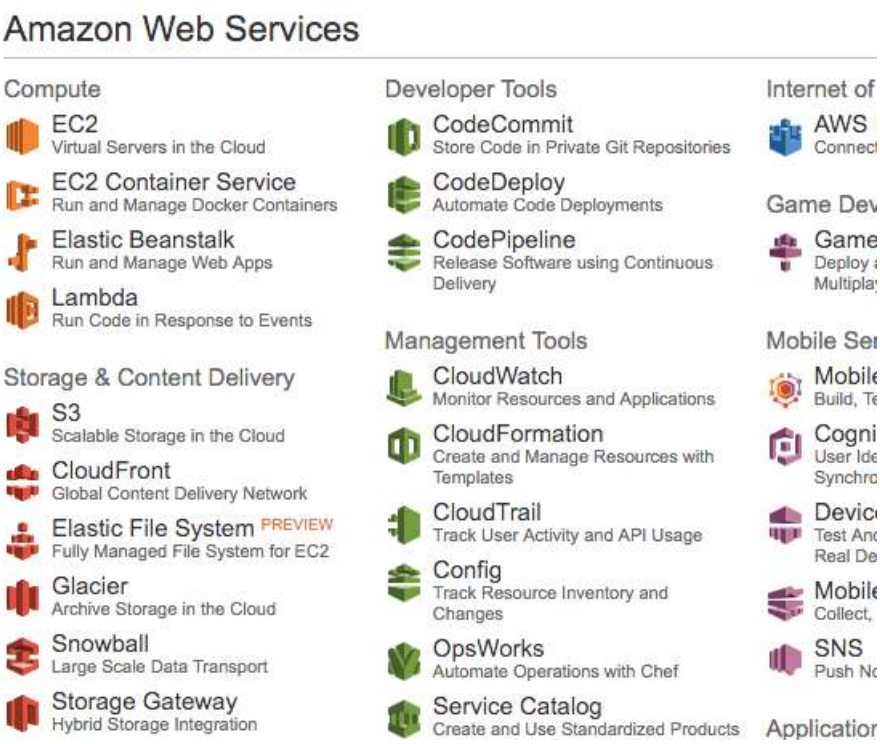

Figure C.3: AWS Console

- 2. Click on EC2 for launching a new virtual server.
- 3. Select *US West Orgeon* from the drop-down in the top right hand corner. This is important otherwise you will not be able to find the image we plan to use.
- 4. Click the *Launch Instance* button.
- 5. Click *Community AMIs*. An AMI is an Amazon Machine Image. It is a frozen instance of a server that you can select and instantiate on a new virtual server.

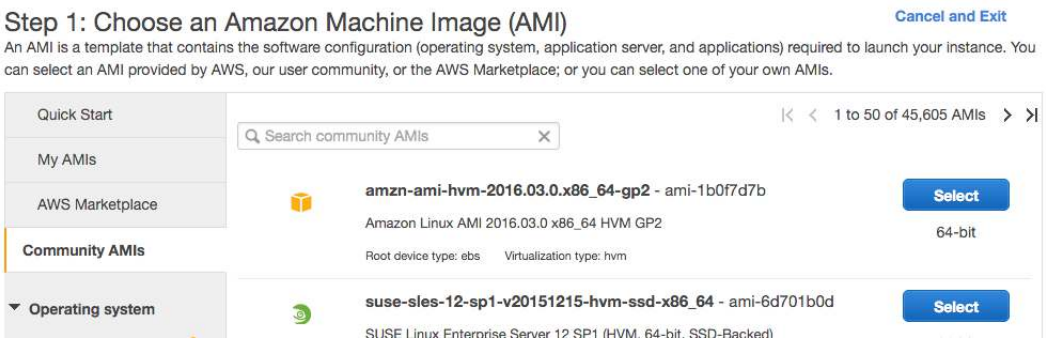

Figure C.4: Community AMIs

- 6. Enter `Deep Learning AMI` in the *Search community AMIs* search box and press enter.

Figure C.5: Select a Specific AMI

- 7. Click *Select* to choose the AMI in the search result.
- 8. Now you need to select the hardware on which to run the image. Scroll down and select the `p3.2xlarge` hardware (I used to recommend g2 or g3 instances and p2 instances, but the p3 instances[2] are newer and faster). This includes a Tesla V100 GPU that we can use to significantly increase the training speed of our models. It also includes 8 CPU Cores, 61GB of RAM and 16GB of GPU RAM. Note: using this instance will cost approximately $3 USD/hour.

Figure C.6: Select g2.2xlarge Hardware

- 9. Click *Review and Launch* to finalize the configuration of your server instance.
- 10. Click the *Launch* button.
- 11. Select Your Key Pair.

If you have a key pair because you have used EC2 before, select *Choose an existing key pair* and choose your key pair from the list. Then check *I acknowledge....* If you do not have a key pair, select the option *Create a new key pair* and enter a *Key pair name* such as keras-keypair. Click the *Download Key Pair* button.

[2] https://aws.amazon.com/ec2/instance-types/p3/

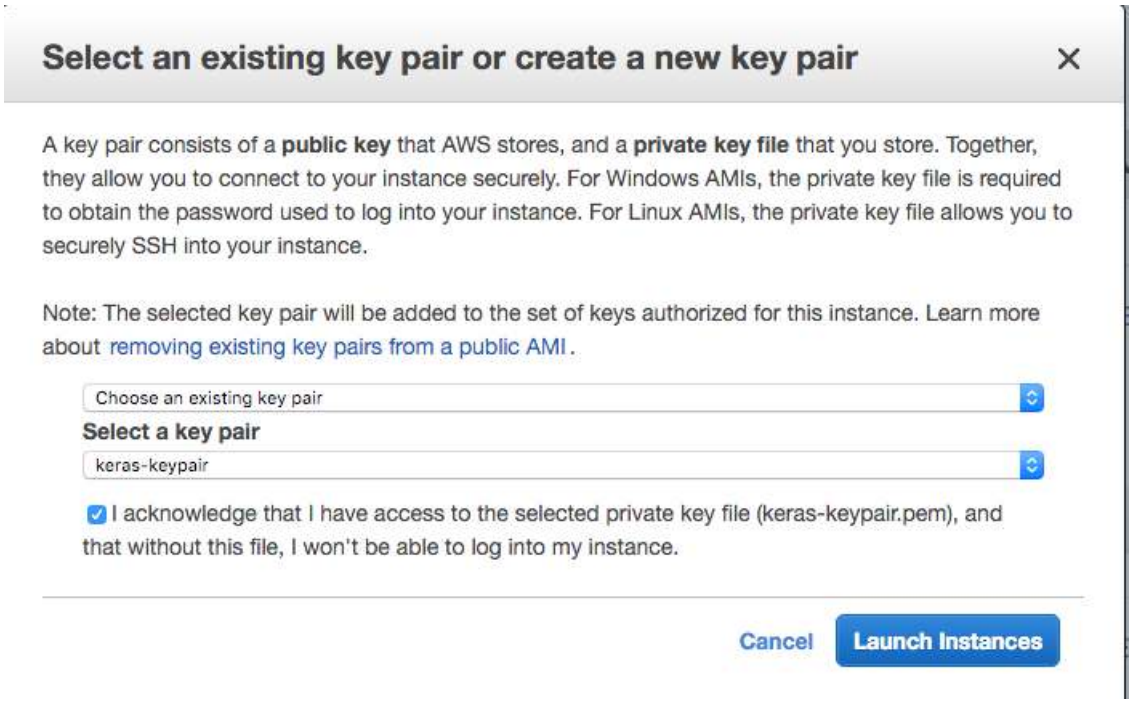

Figure C.7: Select Your Key Pair

- 12. Open a Terminal and change directory to where you downloaded your key pair.
- 13. If you have not already done so, restrict the access permissions on your key pair file. This is required as part of the SSH access to your server. For example, open a terminal on your workstation and type:

```
cd Downloads
chmod 600 keras-aws-keypair.pem
```

Listing C.1: Change Permissions of Your Key Pair File.

- 14. Click *Launch Instances*. If this is your first time using AWS, Amazon may have to validate your request and this could take up to 2 hours (often just a few minutes).
- 15. Click *View Instances* to review the status of your instance.

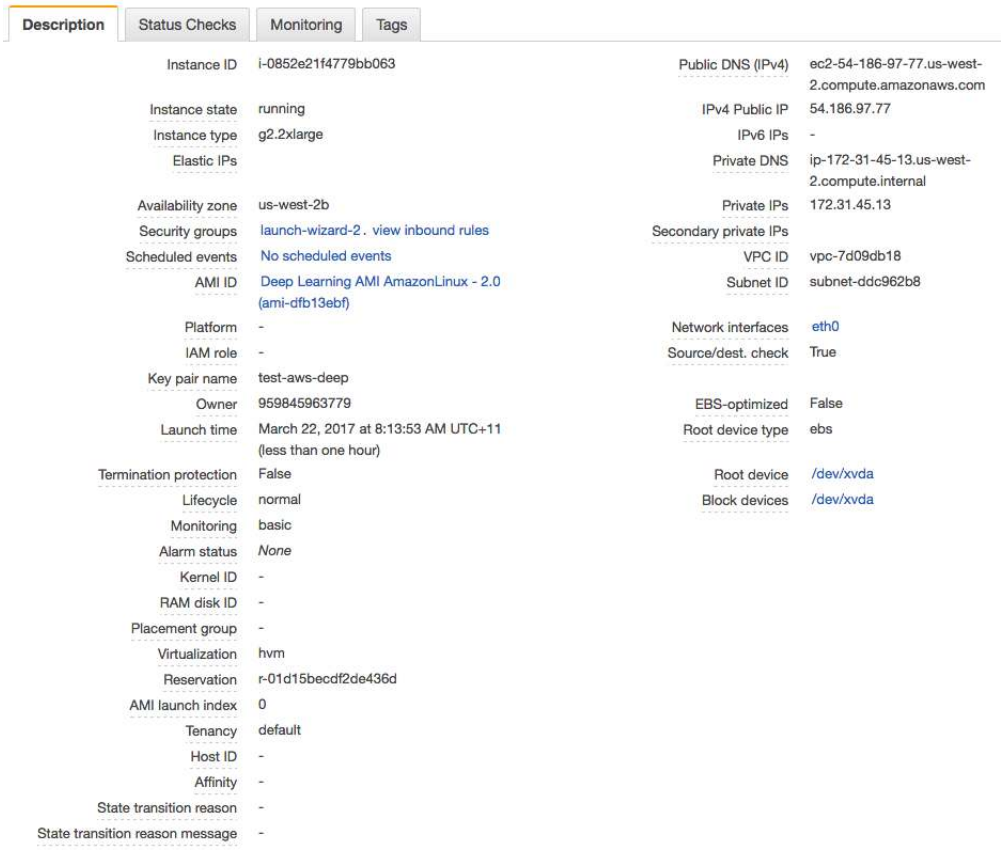

Figure C.8: Review Your Running Instance

Your server is now running and ready for you to log in.

C.4 Login, Configure and Run

Now that you have launched your server instance, it is time to log in and start using it.

- 1. Click *View Instances* in your Amazon EC2 console if you have not done so already.
- 2. Copy the *Public IP* (down the bottom of the screen in Description) to your clipboard. In this example my IP address is `54.186.97.77`. **Do not use this IP address, it will not work as your server IP address will be different**.
- 3. Open a Terminal and change directory to where you downloaded your key pair. Login to your server using SSH, for example:

```
ssh -i keras-aws-keypair.pem ec2-user@54.186.97.77
```

Listing C.2: Log-in To Your AWS Instance.

- 4. If prompted, type `yes` and press enter.

You are now logged into your server.

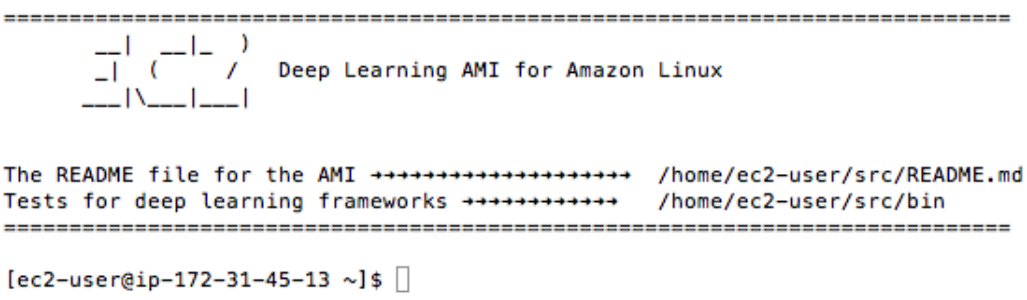

Figure C.9: Log in Screen for Your AWS Server

The instance will ask what Python environment you wish to use. I recommend using:

- **TensorFlow(+Keras2) with Python3 (CUDA 9.0 and Intel MKL-DNN)**

You can activate this virtual environment by typing:

```
source activate tensorflow_p36
```

Listing C.3: Activate the appropriate virtual environment.

You are now free to run your code.

C.5 Build and Run Models on AWS

This section offers some tips for running your code on AWS.

C.5.1 Copy Scripts and Data to AWS

You can get started quickly by copying your files to your running AWS instance. For example, you can copy the examples provided with this book to your AWS instance using the `scp` command as follows:

```
scp -i keras-aws-keypair.pem -r src ec2-user@54.186.97.77:~/
```

Listing C.4: Example for Copying Sample Code to AWS.

This will copy the entire `src/` directory to your home directory on your AWS instance. You can easily adapt this example to get your larger datasets from your workstation onto your AWS instance. Note that Amazon may impose charges for moving very large amounts of data in and out of your AWS instance. Refer to Amazon documentation for relevant charges.

C.5.2 Run Models on AWS

You can run your scripts on your AWS instance as per normal:

```
python filename.py
```

Listing C.5: Example of Running a Python script on AWS.

You are using AWS to create large neural network models that may take hours or days to train. As such, it is a better idea to run your scripts as a background job. This allows you to close your terminal and your workstation while your AWS instance continues to run your script. You can easily run your script as a background process as follows:

```
nohup /path/to/script >/path/to/script.log 2>&1 < /dev/null &
```

Listing C.6: Run Script as a Background Process.

You can then check the status and results in your `script.log` file later.

C.6 Close Your EC2 Instance

When you are finished with your work you must close your instance. Remember you are charged by the amount of time that you use the instance. It is cheap, but you do not want to leave an instance on if you are not using it.

- 1. Log out of your instance at the terminal, for example you can type:

```
exit
```

Listing C.7: Log-out of Server Instance.

- 2. Log in to your AWS account with your web browser.
- 3. Click EC2.
- 4. Click *Instances* from the left-hand side menu.

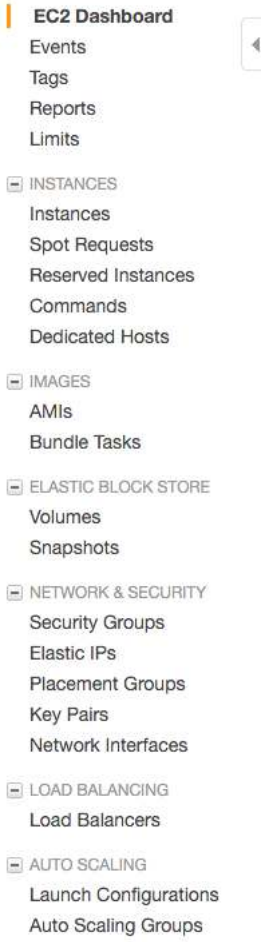

Figure C.10: Review Your List of Running Instances

- 5. Select your running instance from the list (it may already be selected if you only have one running instance).
- 6. Click the *Actions* button and select *Instance State* and choose *Terminate*. Confirm that you want to terminate your running instance.

It may take a number of seconds for the instance to close and to be removed from your list of instances.

C.7 Tips and Tricks for Using Keras on AWS

Below are some tips and tricks for getting the most out of using Keras on AWS instances.

- **Design a suite of experiments to run beforehand**. Experiments can take a long time to run and you are paying for the time you use. Make time to design a batch of experiments to run on AWS. Put each in a separate file and call them in turn from another script. This will allow you to answer multiple questions from one long run, perhaps overnight.

- **Always close your instance at the end of your experiments**. You do not want to be surprised with a very large AWS bill.
- **Try spot instances for a cheaper but less reliable option**. Amazon sell unused time on their hardware at a much cheaper price, but at the cost of potentially having your instance closed at any second. If you are learning or your experiments are not critical, this might be an ideal option for you. You can access spot instances from the *Spot Instance* option on the left hand side menu in your EC2 web console.

C.8 Further Reading

Below is a list of resources to learn more about AWS and developing deep learning models in the cloud.

- An introduction to Amazon Elastic Compute Cloud (EC2).
  http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
- An introduction to Amazon Machine Images (AMI).
  http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html
- Deep Learning AMI (Amazon Linux) on the AMI Marketplace.
  https://aws.amazon.com/marketplace/pp/B077GF11NF
- P3 EC2 Instances.
  https://aws.amazon.com/ec2/instance-types/p3/

C.9 Summary

In this lesson you discovered how you can develop and evaluate your large deep learning models in Keras using GPUs on the Amazon Web Service. You learned:

- Amazon Web Services with their Elastic Compute Cloud offers an affordable way to run large deep learning models on GPU hardware.
- How to setup and launch an EC2 server for deep learning experiments.
- How to update the Keras version on the server and confirm that the system is working correctly.
- How to run Keras experiments on AWS instances in batch as background tasks.

Part IX

Conclusions

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now know:

- How to use upsampling and inverse convolutional layers in deep convolutional neural network models.
- How to implement the training procedure for fitting GAN models with the Keras deep learning library.
- How to implement best practice heuristics for the successful configuration and training of GAN models.
- How to develop and train simple GAN models for image synthesis for black and white and color images.
- How to explore the latent space for image generation with point interpolation and vector arithmetic.
- How to evaluate GAN models using qualitative and quantitative measures, such as the inception score.
- How to train GAN models with alternate loss functions, such as least squares and Wasserstein loss.
- How to structure the latent space and influence the generation of synthetic images with conditional GANs.
- How to develop image translation models with Pix2Pix for paired images and CycleGAN for unpaired images.
- How sophisticated GAN models, such as Progressive Growing GAN, are used to achieve remarkable results.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable set of skills for developing generative adversarial network models for computer vision applications. You can now confidently:

- Design, configure, and train a GAN model.
- Use trained GAN models for image synthesis and evaluate model performance.
- Harness world-class GANs for image-to-image translation tasks.

The sky's the limit.

Thank You!

I want to take a moment and sincerely thank you for letting me help you start your journey with generative adversarial networks. I hope you keep learning and have fun as you continue to master machine learning.

Jason Brownlee
2019