

O'REILLY®

Generative Deep Learning

Teaching Machines to Paint, Write,
Compose and Play



David Foster

Generative Deep Learning

Generative modeling is one of the hottest topics in AI. It's now possible to teach a machine to excel at human endeavors such as painting, writing, and composing music. With this practical book, machine learning engineers and data scientists will discover how to re-create some of the most impressive examples of generative deep learning models, such as variational autoencoders, generative adversarial networks (GANs), encoder-decoder models, and world models.

Author David Foster demonstrates the inner workings of each technique, starting with the basics of deep learning using Keras before advancing to some of the most cutting-edge algorithms in the field. Through tips and tricks, you'll understand how to make your models learn more efficiently and become more creative.

- Discover how variational autoencoders can change facial expressions in photos
- Build practical GAN examples from scratch, including CycleGAN for style transfer and MuseGAN for music generation
- Create recurrent generative models for text generation and learn how to improve the models using attention
- Understand how generative models can help agents accomplish tasks within a reinforcement learning setting
- Explore the architecture of the Transformer (BERT, GPT-2) and image generation models such as ProGAN and StyleGAN

"Generative Deep Learning is an accessible introduction to the deep learning toolkit for generative modeling. If you are a creative practitioner who loves to tinker with code and want to apply deep learning to your work, then this book is for you."

—David Ha
Research Scientist, Google Brain

David Foster is the cofounder of Applied Data Science, a data science consultancy delivering innovative solutions for clients. He holds an MA in mathematics from Trinity College, Cambridge, UK, and an MSc in operational research from the University of Warwick.

COMPUTER SCIENCE / DEEP LEARNING

US \$69.99

CAN \$92.99

ISBN: 978-1-492-04194-8



Twitter: @oreillymedia
facebook.com/oreilly

Generative Deep Learning

*Teaching Machines to Paint, Write,
Compose, and Play*

David Foster

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Generative Deep Learning

by David Foster

Copyright © 2019 Applied Data Science Partners Ltd. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Development Editor: Michele Cronin

Indexer: Judith McConville

Acquisitions Editor: Jonathan Hassell

Interior Designer: David Futato

Production Editor: Katherine Tozer

Cover Designer: Karen Montgomery

Copyeditor: Rachel Head

Illustrator: Rebecca Demarest

Proofreader: Charles Roumeliotis

July 2019: First Edition

Revision History for the First Edition

2019-06-26: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492041948> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Generative Deep Learning*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-04194-8

[LSI]

Table of Contents

Preface.....	ix
--------------	----

Part I. Introduction to Generative Deep Learning

1. Generative Modeling.....	1
What Is Generative Modeling?	1
Generative Versus Discriminative Modeling	2
Advances in Machine Learning	4
The Rise of Generative Modeling	5
The Generative Modeling Framework	7
Probabilistic Generative Models	10
Hello Wrod!	13
Your First Probabilistic Generative Model	14
Naive Bayes	17
Hello Wrod! Continued	20
The Challenges of Generative Modeling	22
Representation Learning	23
Setting Up Your Environment	27
Summary	29
2. Deep Learning.....	31
Structured and Unstructured Data	31
Deep Neural Networks	33
Keras and TensorFlow	34
Your First Deep Neural Network	35
Loading the Data	35

Building the Model	37
Compiling the Model	41
Training the Model	43
Evaluating the Model	44
Improving the Model	46
Convolutional Layers	46
Batch Normalization	51
Dropout Layers	54
Putting It All Together	55
Summary	59
3. Variational Autoencoders.....	61
The Art Exhibition	61
Autoencoders	64
Your First Autoencoder	66
The Encoder	66
The Decoder	68
Joining the Encoder to the Decoder	71
Analysis of the Autoencoder	72
The Variational Art Exhibition	75
Building a Variational Autoencoder	78
The Encoder	78
The Loss Function	84
Analysis of the Variational Autoencoder	85
Using VAEs to Generate Faces	86
Training the VAE	87
Analysis of the VAE	91
Generating New Faces	92
Latent Space Arithmetic	93
Morphing Between Faces	94
Summary	95
4. Generative Adversarial Networks.....	97
Ganimals	97
Introduction to GANs	99
Your First GAN	100
The Discriminator	101
The Generator	103
Training the GAN	107
GAN Challenges	112
Oscillating Loss	112

Mode Collapse	113
Uninformative Loss	114
Hyperparameters	114
Tackling the GAN Challenges	115
Wasserstein GAN	115
Wasserstein Loss	115
The Lipschitz Constraint	117
Weight Clipping	118
Training the WGAN	119
Analysis of the WGAN	120
WGAN-GP	121
The Gradient Penalty Loss	121
Analysis of WGAN-GP	125
Summary	127

Part II. Teaching Machines to Paint, Write, Compose, and Play

5. Paint.....	131
Apples and Organges	132
CycleGAN	135
Your First CycleGAN	137
Overview	137
The Generators (U-Net)	139
The Discriminators	142
Compiling the CycleGAN	144
Training the CycleGAN	146
Analysis of the CycleGAN	147
Creating a CycleGAN to Paint Like Monet	149
The Generators (ResNet)	150
Analysis of the CycleGAN	151
Neural Style Transfer	153
Content Loss	154
Style Loss	156
Total Variance Loss	160
Running the Neural Style Transfer	160
Analysis of the Neural Style Transfer Model	161
Summary	162
6. Write.....	165
The Literary Society for Troublesome Miscreants	166

Long Short-Term Memory Networks	167
Your First LSTM Network	168
Tokenization	168
Building the Dataset	171
The LSTM Architecture	172
The Embedding Layer	172
The LSTM Layer	174
The LSTM Cell	176
Generating New Text	179
RNN Extensions	183
Stacked Recurrent Networks	183
Gated Recurrent Units	185
Bidirectional Cells	187
Encoder–Decoder Models	187
A Question and Answer Generator	190
A Question–Answer Dataset	191
Model Architecture	192
Inference	196
Model Results	198
Summary	200
7. Compose.....	201
Preliminaries	202
Musical Notation	202
Your First Music-Generating RNN	205
Attention	206
Building an Attention Mechanism in Keras	208
Analysis of the RNN with Attention	213
Attention in Encoder–Decoder Networks	217
Generating Polyphonic Music	221
The Musical Organ	221
Your First MuseGAN	223
The MuseGAN Generator	226
Chords, Style, Melody, and Groove	227
The Bar Generator	229
Putting It All Together	230
The Critic	232
Analysis of the MuseGAN	233
Summary	235

8. Play.....	237
Reinforcement Learning	238
OpenAI Gym	239
World Model Architecture	241
The Variational Autoencoder	242
The MDN-RNN	243
The Controller	243
Setup	244
Training Process Overview	245
Collecting Random Rollout Data	245
Training the VAE	248
The VAE Architecture	249
Exploring the VAE	252
Collecting Data to Train the RNN	255
Training the MDN-RNN	257
The MDN-RNN Architecture	258
Sampling the Next z and Reward from the MDN-RNN	259
The MDN-RNN Loss Function	259
Training the Controller	261
The Controller Architecture	262
CMA-ES	262
Parallelizing CMA-ES	265
Output from the Controller Training	267
In-Dream Training	268
In-Dream Training the Controller	270
Challenges of In-Dream Training	272
Summary	273
9. The Future of Generative Modeling.....	275
Five Years of Progress	275
The Transformer	277
Positional Encoding	279
Multihead Attention	280
The Decoder	283
Analysis of the Transformer	283
BERT	285
GPT-2	285
MuseNet	286
Advances in Image Generation	287
ProGAN	287
Self-Attention GAN (SAGAN)	289

BigGAN	291
StyleGAN	292
Applications of Generative Modeling	296
AI Art	296
AI Music	297
10. Conclusion.....	299
Index.....	303

Preface

What I cannot create, I do not understand.

—Richard Feynman

An undeniable part of the human condition is our ability to create. Since our earliest days as cave people, we have sought opportunities to generate original and beautiful creations. For early man, this took the form of cave paintings depicting wild animals and abstract patterns, created with pigments placed carefully and methodically onto rock. The Romantic Era gave us the mastery of Tchaikovsky symphonies, with their ability to inspire feelings of triumph and tragedy through sound waves, woven together to form beautiful melodies and harmonies. And in recent times, we have found ourselves rushing to bookshops at midnight to buy stories about a fictional wizard, because the combination of letters creates a narrative that wills us to turn the page and find out what happens to our hero.

It is therefore not surprising that humanity has started to ask the ultimate question of creativity: can we create something that is in itself creative?

This is the question that generative modeling aims to answer. With recent advances in methodology and technology, we are now able to build machines that can paint original artwork in a given style, write coherent paragraphs with long-term structure, compose music that is pleasant to listen to, and develop winning strategies for complex games by generating imaginary future scenarios. This is just the start of a generative revolution that will leave us with no choice but to find answers to some of the biggest questions about the mechanics of creativity, and ultimately, what it means to be human.

In short, there has never been a better time to learn about generative modeling—so let's get started!

Objective and Approach

This book covers the key techniques that have dominated the generative modeling landscape in recent years and have allowed us to make impressive progress in creative tasks. As well as covering core generative modeling theory, we will be building full working examples of some of the key models from the literature and walking through the codebase for each, step by step.

Throughout the book, you will find short, allegorical stories that help explain the mechanics of some of the models we will be building. I believe that one of the best ways to teach a new abstract theory is to first convert it into something that isn't quite so abstract, such as a story, before diving into the technical explanation. The individual steps of the theory are clearer within this context because they involve people, actions, and emotions, all of which are well understood, rather than neural networks, backpropagation, and loss functions, which are abstract constructs.

The story and the model explanation are just the same mechanics explained in two different domains. You might therefore find it useful to refer back to the relevant story while learning about each model. If you are already familiar with a particular technique, then have fun finding the parallels of each model element within the story!

In Part I of this book I shall introduce the key techniques that we will be using to build generative models, including an overview of deep learning, variational autoencoders, and generative adversarial networks. In Part II, we will be building on these techniques to tackle several creative tasks, such as painting, writing, and composing music through models such as CycleGAN, encoder-decoder models, and MuseGAN. In addition, we shall see how generative modeling can be used to optimize playing strategy for a game (World Models) and take a look at the most cutting-edge generative architectures available today, such as StyleGAN, BigGAN, BERT, GPT-2, and MuseNet.

Prerequisites

This book assumes that you have experience coding in Python. If you are not familiar with Python, the best place to start is through [LearningPython.org](#). There are many free resources online that will allow you to develop enough Python knowledge to work with the examples in this book.

Also, since some of the models are described using mathematical notation, it will be useful to have a solid understanding of linear algebra (for example, matrix multiplication, etc.) and general probability theory.

Finally, you will need an environment in which to run the code examples from the book's GitHub [repository](#). I have deliberately ensured that all of the examples in this book do not require prohibitively large amounts of computational resources to train.

There is a myth that you need a GPU in order to start training deep learning models—while this is of course helpful and will speed up training, it is not essential. In fact, if you are new to deep learning, I encourage you to first get to grips with the essentials by experimenting with small examples on your laptop, before spending money and time researching hardware to speed up training.

Other Resources

Two books I highly recommend as a general introduction to machine learning and deep learning are as follows:

- *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems* by Aurelien Geron (O'Reilly)
- *Deep Learning with Python* by Francois Chollet (Manning)

Most of the papers in this book are sourced through [arXiv](#), a free repository of scientific research papers. It is now common for authors to post papers to arXiv before they are fully peer-reviewed. Reviewing the recent submissions is a great way to keep on top of the most cutting-edge developments in the field.

I also highly recommend the website [Papers with Code](#), where you can find the latest state-of-the-art results in a variety of machine learning tasks, alongside links to the papers and official GitHub repositories. It is an excellent resource for anyone wanting to quickly understand which techniques are currently achieving the highest scores in a range of tasks and has certainly helped me to decide which techniques to cover in this book.

Finally, a useful resource for training deep learning models on accelerated hardware is [Google Colaboratory](#). This is a free Jupyter Notebook environment that requires no setup and runs entirely in the cloud. You can tell the notebook to run on a GPU that is provided for free, for up to 12 hours of runtime. While it is not essential to run the examples in this book on a GPU, it may help to speed up the training process. Either way, Colab is a great way to access GPU resources for free.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a general note.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/davidADSP/GDL_code.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Generative Deep Learning* by David Foster (O'Reilly). Copyright 2019 Applied Data Science Partners Ltd., 978-1-492-04194-8.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/generative-dl>.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

There are so many people I would like to thank for helping me write this book.

First, I would like to thank everyone who has taken time to technically review the book—in particular, Luba Elliott, Darren Richardson, Eric George, Chris Schon, Sigurður Skúli Sigurgeirsson, Hao-Wen Dong, David Ha, and Lorna Barclay.

Also, a huge thanks to my colleagues at Applied Data Science Partners, Ross Witeszczak, Chris Schon, Daniel Sharp, and Amy Bull. Your patience with me while I have taken time to finish the book is hugely appreciated, and I am greatly looking forward to all the machine learning projects we will complete together in the future! Particular thanks to Ross—had we not decided to start a business together, this book might never have taken shape, so thank you for believing in me as your business partner!

I also want to thank anyone who has ever taught me anything mathematical—I was extremely fortunate to have fantastic math teachers at school, who developed my interest in the subject and encouraged me to pursue it further at university. I would like to thank you for your commitment and for going out of your way to share your knowledge of the subject with me.

A huge thank you goes to the staff at O'Reilly for guiding me through the process of writing this book. A special thanks goes to Michele Cronin, who has been there at each step, providing useful feedback and sending me friendly reminders to keep completing chapters! Also to Katie Tozer, Rachel Head, and Melanie Yarbrough for getting the book into production, and Mike Loukides for first reaching out to ask if I'd be interested in writing a book. You have all been so supportive of this project from the start, and I want to thank you for providing me with a platform on which to write about something that I love.

Throughout the writing process, my family has been a constant source of encouragement and support. A huge thank you goes to my mum, Gillian Foster, for checking every single line of text for typos and for teaching me how to add up in the first place! Your attention to detail has been extremely helpful while proofreading this book, and I'm really grateful for all the opportunities that both you and dad have given me. My dad, Clive Foster, originally taught me how to program a computer—this book is full of practical examples, and that's thanks to his early patience while I fumbled around in BASIC trying to make football games as a teenager. My brother, Rob Foster, is the most modest genius you will ever find, particularly within linguistics—chatting with him about AI and the future of text-based machine learning has been amazingly helpful. Last, I would like to thank my Nana, who is a constant source of inspiration and fun for all of us. Her love of literature is one of the reasons I first decided that writing a book would be an exciting thing to do.

Finally, I would like to thank my fiancée (and soon to be wife) Lorna Barclay. As well as technically reviewing every word of this book, she has provided endless support to me throughout the writing process, making me tea, bringing me various snacks, and generally helping me to make this a better guide to generative modeling through her meticulous attention to detail and deep expertise in statistics and machine learning. I certainly couldn't have completed this project without you, and I'm grateful for the time you have invested in helping me restructure and expand parts of the book that needed more explanation. I promise I won't talk about generative modeling at the dinner table for at least a few weeks after it is published.

PART I

Introduction to Generative Deep Learning

The first four chapters of this book aim to introduce the core techniques that you'll need to start building generative deep learning models.

In [Chapter 1](#), we shall first take a broad look at the field of generative modeling and consider the type of problem that we are trying to solve from a probabilistic perspective. We will then explore our first example of a basic probabilistic generative model and analyze why deep learning techniques may need to be deployed as the complexity of the generative task grows.

[Chapter 2](#) provides a guide to the deep learning tools and techniques that you will need to start building more complex generative models. This is intended to be a practical guide to deep learning rather than a theoretical analysis of the field. In particular, I will introduce Keras, a framework for building neural networks that can be used to construct and train some of the most cutting-edge deep neural network architectures published in the literature.

In [Chapter 3](#), we shall take a look at our first generative deep learning model, the variational autoencoder. This powerful technique will allow us to not only generate realistic faces, but also alter existing images—for example, by adding a smile or changing the color of someone's hair.

Chapter 4 explores one of the most successful generative modeling techniques of recent years, the generative adversarial network. This elegant framework for structuring a generative modeling problem is the underlying engine behind most state-of-the-art generative models. We shall see the ways that it has been fine-tuned and adapted to continually push the boundaries of what generative modeling is able to achieve.

CHAPTER 1

Generative Modeling

This chapter is a general introduction to the field of generative modeling. We shall first look at what it means to say that a model is *generative* and learn how it differs from the more widely studied *discriminative* modeling. Then I will introduce the framework and core mathematical ideas that will allow us to structure our general approach to problems that require a generative solution.

With this in place, we will then build our first example of a generative model (Naive Bayes) that is probabilistic in nature. We shall see that this allows us to generate novel examples that are outside of our training dataset, but shall also explore the reasons why this type of model may fail as the size and complexity of the space of possible creations increases.

What Is Generative Modeling?

A generative model can be broadly defined as follows:

A generative model describes how a dataset is generated, in terms of a probabilistic model. By sampling from this model, we are able to generate new data.

Suppose we have a dataset containing images of horses. We may wish to build a model that can generate a new image of a horse that has never existed but still looks real because the model has learned the general rules that govern the appearance of a horse. This is the kind of problem that can be solved using generative modeling. A summary of a typical generative modeling process is shown in [Figure 1-1](#).

First, we require a dataset consisting of many examples of the entity we are trying to generate. This is known as the *training data*, and one such data point is called an *observation*.

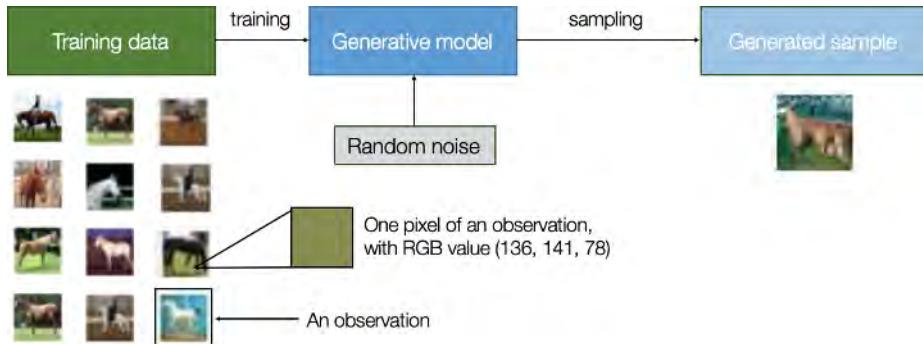


Figure 1-1. The generative modeling process

Each observation consists of many *features*—for an image generation problem, the features are usually the individual pixel values. It is our goal to build a model that can generate new sets of features that look as if they have been created using the same rules as the original data. Conceptually, for image generation this is an incredibly difficult task, considering the vast number of ways that individual pixel values can be assigned and the relatively tiny number of such arrangements that constitute an image of the entity we are trying to simulate.

A generative model must also be *probabilistic* rather than *deterministic*. If our model is merely a fixed calculation, such as taking the average value of each pixel in the dataset, it is not generative because the model produces the same output every time. The model must include a *stochastic* (random) element that influences the individual samples generated by the model.

In other words, we can imagine that there is some unknown probabilistic distribution that explains why some images are likely to be found in the training dataset and other images are not. It is our job to build a model that mimics this distribution as closely as possible and then sample from it to generate new, distinct observations that look as if they could have been included in the original training set.

Generative Versus Discriminative Modeling

In order to truly understand what generative modeling aims to achieve and why this is important, it is useful to compare it to its counterpart, *discriminative modeling*. If you have studied machine learning, most problems you will have faced will have most likely been discriminative in nature. To understand the difference, let's look at an example.

Suppose we have a dataset of paintings, some painted by Van Gogh and some by other artists. With enough data, we could train a discriminative model to predict if a given painting was painted by Van Gogh. Our model would learn that certain colors,

shapes, and textures are more likely to indicate that a painting is by the Dutch master, and for paintings with these features, the model would upweight its prediction accordingly. [Figure 1-2](#) shows the discriminative modeling process—note how it differs from the generative modeling process shown in [Figure 1-1](#).

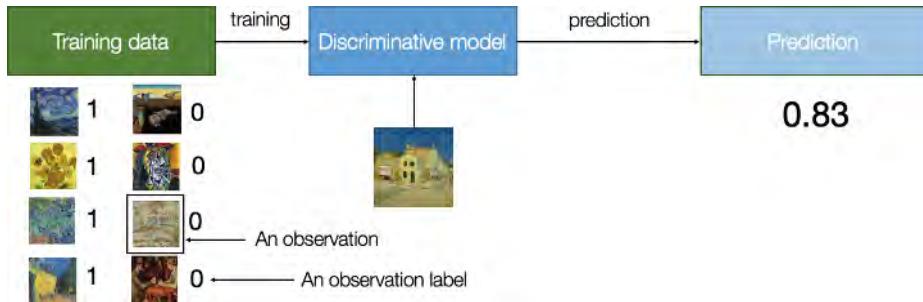


Figure 1-2. The discriminative modeling process

One key difference is that when performing discriminative modeling, each observation in the training data has a *label*. For a binary classification problem such as our artist discriminator, Van Gogh paintings would be labeled *1* and non–Van Gogh paintings labeled *0*. Our model then learns how to discriminate between these two groups and outputs the probability that a new observation has label *1*—i.e., that it was painted by Van Gogh.

For this reason, discriminative modeling is synonymous with *supervised learning*, or learning a function that maps an input to an output using a labeled dataset. Generative modeling is usually performed with an unlabeled dataset (that is, as a form of unsupervised learning), though it can also be applied to a labeled dataset to learn how to generate observations from each distinct class.

Let’s take a look at some mathematical notation to describe the difference between generative and discriminative modeling.

Discriminative modeling estimates $p(y|x)$ —the probability of a label y given observation x .

Generative modeling estimates $p(x)$ —the probability of observing observation x .

If the dataset is labeled, we can also build a generative model that estimates the distribution $p(x|y)$.

In other words, discriminative modeling attempts to estimate the probability that an observation x belongs to category y . Generative modeling doesn't care about labeling observations. Instead, it attempts to estimate the probability of seeing the observation at all.

The key point is that even if we were able to build a perfect discriminative model to identify Van Gogh paintings, it would still have no idea how to create a painting that looks like a Van Gogh. It can only output probabilities against existing images, as this is what it has been trained to do. We would instead need to train a generative model, which can output sets of pixels that have a high chance of belonging to the original training dataset.

Advances in Machine Learning

To understand why generative modeling can be considered the next frontier for machine learning, we must first look at why discriminative modeling has been the driving force behind most progress in machine learning methodology in the last two decades, both in academia and in industry.

From an academic perspective, progress in discriminative modeling is certainly easier to monitor, as we can measure performance metrics against certain high-profile classification tasks to determine the current best-in-class methodology. Generative models are often more difficult to evaluate, especially when the quality of the output is largely subjective. Therefore, much emphasis in recent years has been placed on training discriminative models to reach human or superhuman performance in a variety of image or text classification tasks.

For example, for image classification, the key breakthrough came in 2012 when a team led by Geoff Hinton at the University of Toronto won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) with a deep convolutional neural network. The competition involves classifying images into one of a thousand categories and is used as a benchmark to compare the latest state-of-the-art techniques. The deep learning model had an error rate of 16%—a massive improvement on the next best model, which only achieved a 26.2% error rate. This sparked a deep learning boom that has resulted in the error rate falling even further year after year. The 2015 winner achieved superhuman performance for the first time, with an error rate of 4%, and the current state-of-the-art model achieves an error rate of just 2%. Many would now consider the challenge a solved problem.

As well as it being easier to publish measurable results within an academic setting, discriminative modeling has historically been more readily applicable to business problems than generative modeling. Generally, in a business setting, we don't care *how* the data was generated, but instead want to know how a new example should be categorized or valued. For example:

- Given a satellite image, a government defense official would only care about the probability that it contains enemy units, not the probability that this particular image should appear.
- A customer relations manager would only be interested in knowing if the sentiment of an incoming email is positive or negative and wouldn't find much use in a generative model that could output examples of customer emails that don't yet exist.
- A doctor would want to know the chance that a given retinal image indicates glaucoma, rather than have access to a model that can generate novel pictures of the back of an eye.

As most solutions required by businesses are in the domain of discriminative modeling, there has been a rise in the number of *Machine-Learning-as-a-Service* (MLaaS) tools that aim to commoditize the use of discriminative modeling within industry, by largely automating the build, validation, and monitoring processes that are common to almost all discriminative modeling tasks.

The Rise of Generative Modeling

While discriminative modeling has so far provided the bulk of the impetus behind advances in machine learning, in the last three to five years many of the most interesting advancements in the field have come through novel applications of deep learning to generative modeling tasks.

In particular, there has been increased media attention on generative modeling projects such as StyleGAN from NVIDIA,¹ which is able to create hyper-realistic images of human faces, and the GPT-2 language model from OpenAI,² which is able to complete a passage of text given a short introductory paragraph.

[Figure 1-3](#) shows the striking progress that has already been made in facial image generation since 2014.³ There are clear positive applications here for industries such as game design and cinematography, and improvements in automatic music generation will also surely start to resonate within these domains. It remains to be seen whether we will one day read news articles or novels written by a generative model, but the recent progress in this area is staggering and it is certainly not outrageous to suggest that this one day may be the case. While exciting, this also raises ethical

¹ Tero Karras, Samuli Laine, and Timo Aila, “A Style-Based Generator Architecture for Generative Adversarial Networks,” 12 December 2018, <https://arxiv.org/abs/1812.04948>.

² Alec Radford et al., “Language Models Are Unsupervised Multitask Learners,” 2019, <https://paperswithcode.com/paper/language-models-are-unsupervised-multitask>.

³ Miles Brundage et al., “The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation,” February 2018, https://www.eff.org/files/2018/02/20/malicious_ai_report_final.pdf.

questions around the proliferation of fake content on the internet and means it may become ever harder to trust what we see and read through public channels of communication.



Figure 1-3. Face generation using generative modeling has improved significantly in the last four years⁴

As well as the practical uses of generative modeling (many of which are yet to be discovered), there are three deeper reasons why generative modeling can be considered the key to unlocking a far more sophisticated form of artificial intelligence, that goes beyond what discriminative modeling alone can achieve.

First, purely from a theoretical point of view, we should not be content with only being able to excel at categorizing data but should also seek a more complete understanding of how the data was generated in the first place. This is undoubtedly a more difficult problem to solve, due to the high dimensionality of the space of feasible outputs and the relatively small number of creations that we would class as belonging to the dataset. However, as we shall see, many of the same techniques that have driven development in discriminative modeling, such as deep learning, can be utilized by generative models too.

Second, it is highly likely that generative modeling will be central to driving future developments in other fields of machine learning, such as reinforcement learning (the study of teaching agents to optimize a goal in an environment through trial and error). For example, we could use reinforcement learning to train a robot to walk across a given terrain. The general approach would be to build a computer simulation of the terrain and then run many experiments where the agent tries out different strategies. Over time the agent would learn which strategies are more successful than others and therefore gradually improve. A typical problem with this approach is that the physics of the environment is often highly complex and would need to be calculated at each timestep in order to feed the information back to the agent to decide its next move. However, if the agent were able to simulate its environment through a generative model, it wouldn't need to test out the strategy in the computer simulation or in

⁴ Source: Brundage et al., 2018.

the real world, but instead could learn in its own *imaginary* environment. In [Chapter 8](#) we shall see this idea in action, training a car to drive as fast as possible around a track by allowing it to learn directly from its own hallucinated environment.

Finally, if we are to truly say that we have built a machine that has acquired a form of intelligence that is comparable to a human's, generative modeling must surely be part of the solution. One of the finest examples of a generative model in the natural world is the person reading this book. Take a moment to consider what an incredible generative model you are. You can close your eyes and imagine what an elephant would look like from any possible angle. You can imagine a number of plausible different endings to your favorite TV show, and you can plan your week ahead by working through various futures in your mind's eye and taking action accordingly. Current neuroscientific theory suggests that our perception of reality is not a highly complex discriminative model operating on our sensory input to produce predictions of what we are experiencing, but is instead a generative model that is trained from birth to produce simulations of our surroundings that accurately match the future. Some theories even suggest that the output from this generative model is what we directly perceive as reality. Clearly, a deep understanding of how we can build machines to acquire this ability will be central to our continued understanding of the workings of the brain and general artificial intelligence.

With this in mind, let's begin our journey into the exciting world of generative modeling. To begin with we shall look at the simplest examples of generative models and some of the ideas that will help us to work through the more complex architectures that we will encounter later in the book.

The Generative Modeling Framework

Let's start by playing a generative modeling game in just two dimensions. I've chosen a rule that has been used to generate the set of points X in [Figure 1-4](#). Let's call this rule p_{data} . Your challenge is to choose a different point $\mathbf{x} = (x_1, x_2)$ in the space that looks like it has been generated by the same rule.

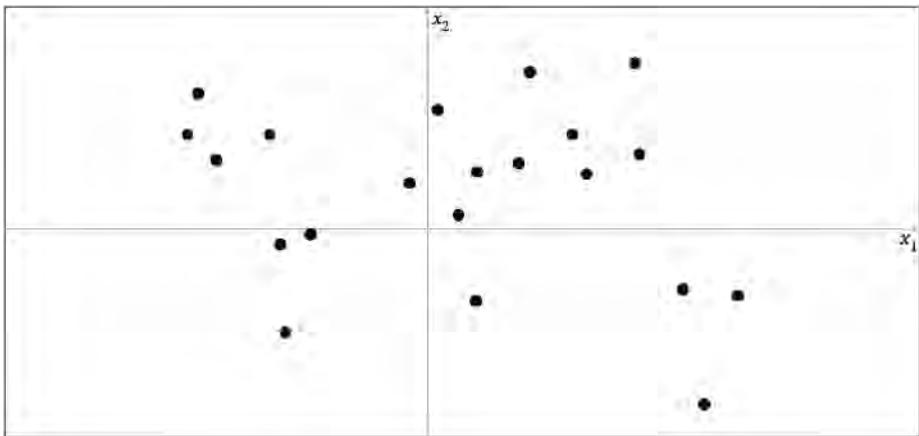


Figure 1-4. A set of points in two dimensions, generated by an unknown rule p_{data}

Where did you choose? You probably used your knowledge of the existing data points to construct a mental model, p_{model} , of whereabouts in the space the point is more likely to be found. In this respect, p_{model} is an *estimate* of p_{data} . Perhaps you decided that p_{model} should look like [Figure 1-5](#)—a rectangular box where points may be found, and an area outside of the box where there is no chance of finding any points. To generate a new observation, you can simply choose a point at random within the box, or more formally, *sample* from the distribution p_{model} . Congratulations, you have just devised your first generative model!

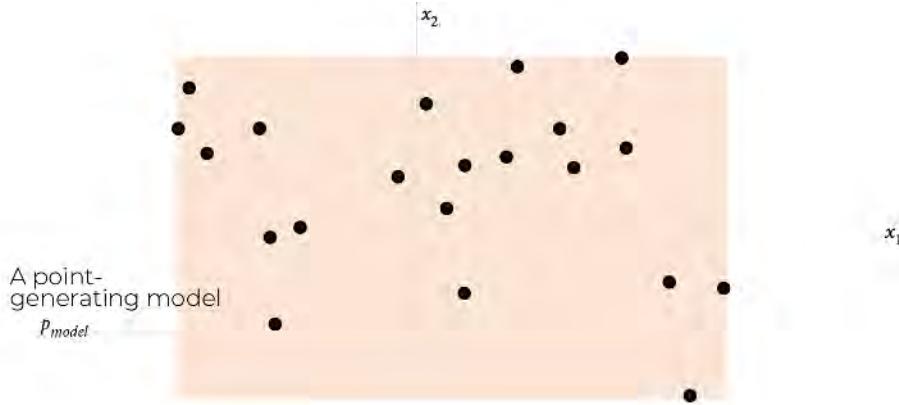


Figure 1-5. The orange box, p_{model} , is an estimate of the true data-generating distribution, p_{data}

While this isn't the most complex example, we can use it to understand what generative modeling is trying to achieve. The following framework sets out our motivations.

The Generative Modeling Framework

- We have a dataset of observations \mathbf{X} .
- We assume that the observations have been generated according to some unknown distribution, p_{data} .
- A generative model p_{model} tries to mimic p_{data} . If we achieve this goal, we can sample from p_{model} to generate observations that appear to have been drawn from p_{data} .
- We are impressed by p_{model} if:
 - Rule 1: It can generate examples that appear to have been drawn from p_{data} .
 - Rule 2: It can generate examples that are suitably different from the observations in \mathbf{X} . In other words, the model shouldn't simply reproduce things it has already seen.

Let's now reveal the true data-generating distribution, p_{data} , and see how the framework applies to this example.

As we can see from [Figure 1-6](#), the data-generating rule is simply a uniform distribution over the land mass of the world, with no chance of finding a point in the sea.

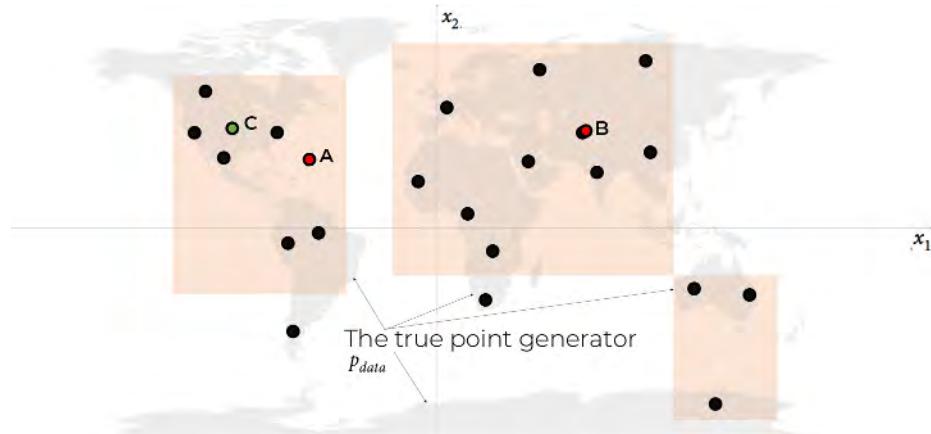


Figure 1-6. The orange box, p_{model} , is an estimate of the true data-generating distribution p_{data} (the gray area)

Clearly, our model p_{model} is an oversimplification of p_{data} . Points A, B, and C show three observations generated by p_{model} with varying degrees of success:

- **Point A** breaks Rule 1 of the Generative Modeling Framework—it does not appear to have been generated by p_{data} as it's in the middle of the sea.
- **Point B** is so close to a point in the dataset that we shouldn't be impressed that our model can generate such a point. If all the examples generated by the model were like this, it would break Rule 2 of the Generative Modeling Framework.
- **Point C** can be deemed a success because it could have been generated by p_{data} and is suitably different from any point in the original dataset.

The field of generative modeling is diverse and the problem definition can take a great variety of forms. However, in most scenarios the Generative Modeling Framework captures how we should broadly think about tackling the problem.

Let's now build our first nontrivial example of a generative model.

Probabilistic Generative Models

Firstly, if you have never studied probability, don't worry. To build and run many of the deep learning models that we shall see later in this book, it is not essential to have a deep understanding of statistical theory. However, to gain a full appreciation of the history of the task that we are trying to tackle, it's worth trying to build a generative model that doesn't rely on deep learning and instead is grounded purely in probabilistic theory. This way, you will have the foundations in place to understand all generative models, whether based on deep learning or not, from the same probabilistic standpoint.



If you already have a good understanding of probability, that's great and much of the next section may already be familiar to you. However, there is a fun example in the middle of this chapter, so be sure not to miss out on that!

As a first step, we shall define four key terms: *sample space*, *density function*, *parametric modeling*, and *maximum likelihood estimation*.

Sample Space

The *sample space* is the complete set of all values an observation \mathbf{x} can take.

In our previous example, the sample space consists of all points of latitude and longitude $\mathbf{x} = (x_1, x_2)$ on the world map.

For example, $\mathbf{x} = (40.7306, -73.9352)$ is a point in the sample space (New York City).

Probability Density Function

A *probability density function* (or simply *density function*), $p(\mathbf{x})$, is a function that maps a point \mathbf{x} in the sample space to a number between 0 and 1. The sum⁵ of the density function over all points in the sample space must equal 1, so that it is a well-defined probability distribution.⁶

In the world map example, the density function of our model is 0 outside of the orange box and constant inside of the box.

While there is only one true density function p_{data} that is assumed to have generated the observable dataset, there are infinitely many density functions p_{model} that we can use to estimate p_{data} . In order to structure our approach to finding a suitable $p_{model}(\mathbf{X})$ we can use a technique known as *parametric modeling*.

Parametric Modeling

A *parametric model*, $p_\theta(\mathbf{x})$, is a family of density functions that can be described using a finite number of parameters, θ .

The family of all possible boxes you could draw on [Figure 1-5](#) is an example of a parametric model. In this case, there are four parameters: the coordinates of the bottom-left (θ_1, θ_2) and top-right (θ_3, θ_4) corners of the box.

⁵ Or integral if the sample space is continuous.

⁶ If the sample space is discrete, $p(\mathbf{x})$ is simply the probability assigned to observing point \mathbf{x} .

Thus, each density function $p_\theta(\mathbf{x})$ in this parametric model (i.e., each box) can be uniquely represented by four numbers, $\theta = (\theta_1, \theta_2, \theta_3, \theta_4)$.

Likelihood

The *likelihood* $\mathcal{L}(\theta | \mathbf{x})$ of a parameter set θ is a function that measures the plausibility of θ , given some observed point \mathbf{x} .

It is defined as follows:

$$\mathcal{L}(\theta | \mathbf{x}) = p_\theta(\mathbf{x})$$

That is, the likelihood of θ given some observed point \mathbf{x} is defined to be the value of the density function parameterized by θ , at the point \mathbf{x} .

If we have a whole dataset \mathbf{X} of independent observations then we can write:

$$\mathcal{L}(\theta | \mathbf{X}) = \prod_{\mathbf{x} \in \mathbf{X}} p_\theta(\mathbf{x})$$

Since this product can be quite computationally difficult to work with, we often use the *log-likelihood* ℓ instead:

$$\ell(\theta | \mathbf{X}) = \sum_{\mathbf{x} \in \mathbf{X}} \log p_\theta(\mathbf{x})$$

There are statistical reasons why the likelihood is defined in this way, but it is enough for us to understand why, intuitively, this makes sense. We are simply defining the likelihood of a set of parameters θ to be equal to the probability of seeing the data under the model parameterized by θ .

In the world map example, an orange box that only covered the left half of the map would have a likelihood of 0—it couldn't possibly have generated the dataset as we have observed points in the right half of the map. The orange box in [Figure 1-5](#) has a positive likelihood as the density function is positive for all data points under this model.

It therefore makes intuitive sense that the focus of parametric modeling should be to find the optimal value $\hat{\theta}$ of the parameter set that maximizes the likelihood of observing the dataset \mathbf{X} . This technique is quite appropriately called *maximum likelihood estimation*.

Maximum Likelihood Estimation

Maximum likelihood estimation is the technique that allows us to estimate $\hat{\theta}$ —the set of parameters θ of a density function, $p_{\theta}(\mathbf{x})$, that are most likely to explain some observed data \mathbf{X} .

More formally:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \mathcal{L}(\theta | \mathbf{X})$$

$\hat{\theta}$ is also called the *maximum likelihood estimate* (MLE).

We now have all the necessary terminology to start describing how we can build a probabilistic generative model.

Most chapters in this book will contain a short story that helps to describe a particular technique. In this chapter, we shall start by taking a trip to planet Wrodl, where our first generative modeling assignment awaits...

Hello Wrodl!

The year is 2047 and you are delighted to have been appointed as the new Chief Fashion Officer (CFO) of Planet Wrodl. As CFO, it is your sole responsibility to create new and exciting fashion trends for the inhabitants of the planet to follow.

The Wrodlers are known to be quite particular when it comes to fashion, so your task is to generate new styles that are similar to those that already exist on the planet, but not identical.

On arrival, you are presented with a dataset featuring 50 observations of Wrodler fashion (Figure 1-7) and told that you have a day to come up with 10 new styles to present to the Fashion Police for inspection. You're allowed to play around with hairstyles, hair color, glasses, clothing type, and clothing color to create your masterpieces.



Figure 1-7. Headshots of 50 Wrodlers⁷

As you're a data scientist at heart, you decide to deploy a generative model to solve the problem. After a brief visit to the Intergalactic Library, you pick up a book called *Generative Deep Learning* and begin to read...

To be continued...

Your First Probabilistic Generative Model

Let's take a closer look at the Wrodl dataset. It consists of $N = 50$ observations of fashions currently seen on the planet. Each observation can be described by five features, (*accessoriesType*, *clothingColor*, *clothingType*, *hairColor*, *topType*), as shown in Table 1-1.

Table 1-1. The first 10 observations in the Wrodler face dataset

face_id	accessoriesType	clothingColor	clothingType	hairColor	topType
0	Round	White	ShirtScoopNeck	Red	ShortHairShortFlat
1	Round	White	Overall	SilverGray	ShortHairFrizzle
2	Sunglasses	White	ShirtScoopNeck	Blonde	ShortHairShortFlat
3	Round	White	ShirtScoopNeck	Red	LongHairStraight
4	Round	White	Overall	SilverGray	NoHair
5	Blank	White	Overall	Black	LongHairStraight
6	Sunglasses	White	Overall	SilverGray	LongHairStraight
7	Round	White	ShirtScoopNeck	SilverGray	ShortHairShortFlat

⁷ Images sourced from <https://getavataars.com>.

face_id	accessoriesType	clothingColor	clothingType	hairColor	topType
8	Round	Pink	Hoodie	SilverGray	LongHairStraight
9	Round	PastelOrange	ShirtScoopNeck	Blonde	LongHairStraight

The possible values for each feature include:

- 7 different hairstyles (*topType*):
 - *NoHair, LongHairBun, LongHairCurly, LongHairStraight, ShortHairShort-Waved, ShortHairShortFlat, ShortHairFrizzle*
- 6 different hair colors (*hairColor*):
 - *Black, Blonde, Brown, PastelPink, Red, SilverGray*
- 3 different kinds of glasses (*accessoriesType*):
 - *Blank, Round, Sunglasses*
- 4 different kinds of clothing (*clothingType*):
 - *Hoodie, Overall, ShirtScoopNeck, ShirtVNeck*
- 8 different clothing colors (*clothingColor*):
 - *Black, Blue01, Gray01, PastelGreen, PastelOrange, Pink, Red, White*

There are $7 \times 6 \times 3 \times 4 \times 8 = 4,032$ different combinations of these features, so there are 4,032 points in the sample space.

We can imagine that our dataset has been generated by some distribution p_{data} that favors some feature values over others. For example, we can see from the images in [Figure 1-7](#) that white clothing seems to be a popular choice, as are silver-gray hair and scoop-neck T-shirts.

The problem is that we do not know p_{data} explicitly—all we have to work with is the sample of observations \mathbf{X} generated by p_{data} . The goal of generative modeling is to use these observations to build a p_{model} that can accurately mimic the observations produced by p_{data} .

To achieve this, we could simply assign a probability to each possible combination of features, based on the data we have seen. Therefore, this parametric model would have $d = 4,031$ parameters—one for each point in the sample space of possibilities, minus one since the value of the last parameter would be forced so that the total sums to 1. Thus the parameters of the model that we are trying to estimate are $(\theta_1, \dots, \theta_{4031})$.

This particular class of parametric model is known as a *multinomial distribution*, and the maximum likelihood estimate $\hat{\theta}_j$ of each parameter is given by:

$$\hat{\theta}_j = \frac{n_j}{N}$$

where n_j is the number of times that combination j was observed in the dataset and $N = 50$ is the total number of observations.

In other words, the estimate for each parameter is just the proportion of times that its corresponding combination was observed in the dataset.

For example, the following combination (let's call it combination 1) appears twice in the dataset:

- (*LongHairStraight, Red, Round, ShirtScoopNeck, White*)

Therefore:

$$\hat{\theta}_1 = 2/50 = 0.04$$

As another example, the following combination (let's call it combination 2) doesn't appear at all in the dataset:

- (*LongHairStraight, Red, Round, ShirtScoopNeck, Blue01*)

Therefore:

$$\hat{\theta}_2 = 0/50 = 0$$

We can calculate all of the $\hat{\theta}_j$ values in this way, to define a distribution over our sample space. Since we can sample from this distribution, our list could potentially be called a generative model. However, it fails in one major respect: it can never generate anything that it hasn't already seen, since $\hat{\theta}_j = 0$ for any combination that wasn't in the original dataset X .

To address this, we could assign an additional *pseudocount* of 1 to each possible combination of features. This is known as *additive smoothing*. Under this model, our MLE for the parameters would be:

$$\hat{\theta}_j = \frac{n_j + 1}{N + d}$$

Now, every single combination has a nonzero probability of being sampled, including those that were not in the original dataset. However, this still fails to be a satisfactory generative model, because the probability of observing a point not in the original dataset is just a constant. If we tried to use such a model to generate Picasso paintings, it would assign just as much weight to a random collection of colorful pixels as to a replica of a Picasso painting that differs only very slightly from a genuine painting.

We would ideally like our generative model to upweight areas of the sample space that it believes are more likely, due to some inherent structure learned from the data, rather than just placing all probabilistic weight on the points that are present in the dataset.

To achieve this, we need to choose a different parametric model.

Naive Bayes

The Naive Bayes parametric model makes use of a simple assumption that drastically reduces the number of parameters we need to estimate.

We make the *naive* assumption that each feature x_j is *independent* of every other feature x_k .⁸ Relating this to the Wrld dataset, we are assuming that the choice of hair color has no impact on the choice of clothing type, and the type of glasses that someone wears has no impact on their hairstyle, for example. More formally, for all features x_j, x_k :

$$p(x_j | x_k) = p(x_j)$$

This is known as the *Naive Bayes* assumption. To apply this assumption, we first make use of the chain rule of probability to write the density function as a product of conditional probabilities:

$$\begin{aligned} p(\mathbf{x}) &= p(x_1, \dots, x_K) \\ &= p(x_2, \dots, x_K | x_1)p(x_1) \\ &= p(x_3, \dots, x_K | x_1, x_2)p(x_2 | x_1)p(x_1) \\ &= \prod_{k=1}^K p(x_k | x_1, \dots, x_{k-1}) \end{aligned}$$

⁸ When a response variable y is present, the Naive Bayes assumption states that there is *conditional* independence between each pair of features x_j, x_k given y .

where K is the total number of features (i.e., 5 for the Wrodl example).

We now apply the Naive Bayes assumption to simplify the last line:

$$p(\mathbf{x}) = \prod_{k=1}^K p(x_k)$$

This is the Naive Bayes model. The problem is reduced to estimating the parameters $\theta_{kl} = p(x_k = l)$ for each feature separately and multiplying these to find the probability for any possible combination.

How many parameters do we now need to estimate? For each feature, we need to estimate a parameter for each value that the feature can take. Therefore, in the Wrodl example, this model is defined by only $7 + 6 + 3 + 4 + 8 - 5 = 23$ parameters.⁹

The maximum likelihood estimates $\widehat{\theta}_{kl}$ are as follows:

$$\widehat{\theta}_{kl} = \frac{n_{kl}}{N}$$

where $\widehat{\theta}_{kl}$ is the number of times that the feature k takes on the value l in the dataset and $N = 50$ is the total number of observations.

Table 1-2 shows the calculated parameters for the Wrodl dataset.

Table 1-2. The MLEs for the parameters under the Naive Bayes model

topType	n	$\widehat{\theta}$	hairColor	n	$\widehat{\theta}$	clothingColor	n	$\widehat{\theta}$	
NoHair	7	0.14	Black	7	0.14	Black	0	0.00	
LongHairBun	0	0.00	Blonde	6	0.12	Blue01	4	0.08	
LongHairCurly	1	0.02	Brown	2	0.04	Grey01	10	0.20	
LongHairStraight	23	0.46	PastelPink	3	0.06	PastelGreen	5	0.10	
ShortHairShortWaved	1	0.02	Red	8	0.16	PastelOrange	2	0.04	
ShortHairShortFlat	11	0.22	SilverGrey	24	0.48	Pink	4	0.08	
ShortHairFrizz	7	0.14	<i>Grand Total</i>		50	1.00	Red	3	0.06
<i>Grand Total</i>	50	1.00				White	22	0.44	
						<i>Grand Total</i>	50	1.00	

⁹ The -5 is due to the fact that the last parameter for each feature is forced to ensure that the sum of the parameters for this feature sums to 1.

accessoriesType	n	$\hat{\theta}$	clothingType	n	$\hat{\theta}$
Blank	11	0.22	Hoodie	7	0.14
Round	22	0.44	Overall	18	0.36
Sunglasses	17	0.34	ShirtScoopNeck	19	0.38
<i>Grand Total</i>	<i>50</i>	<i>1.00</i>	ShirtVNeck	6	0.12
			<i>Grand Total</i>	<i>50</i>	<i>1.00</i>

To calculate the probability of the model generating some observation \mathbf{x} , we simply multiply together the individual feature probabilities. For example:

$$\begin{aligned}
 & p(\text{LongHairStraight}, \text{Red}, \text{Round}, \text{ShirtScoopNeck}, \text{White}) \\
 &= p(\text{LongHairStraight}) \times p(\text{Red}) \times p(\text{Round}) \times p(\text{ShirtScoopNeck}) \times p(\text{White}) \\
 &= 0.46 \times 0.16 \times 0.44 \times 0.38 \times 0.44 \\
 &= 0.0054
 \end{aligned}$$

Notice that this combination doesn't appear in the original dataset, but our model still allocates it a nonzero probability, so it is still able to be generated. Also, it has a higher probability of being sampled than, say, $(\text{LongHairStraight}, \text{Red}, \text{Round}, \text{ShirtScoopNeck}, \text{White})$, because white clothing appears more often than blue clothing in the dataset.

Therefore, a Naive Bayes model is able to learn some structure from the data and use this to generate new examples that were not seen in the original dataset. The model has estimated the probability of seeing each feature value independently, so that under the Naive Bayes assumption we can multiply these probabilities to build our full density function, $p_{\theta}(\mathbf{x})$.

Figure 1-8 shows 10 observations sampled from the model.



Figure 1-8. Ten new Wrodl styles, generated using the Naive Bayes model

For this simple problem, the Naive Bayes assumption that each feature is independent of every other feature is reasonable and therefore produces a good generative model.

Now let's see what happens when this assumption breaks down

Hello Wrodl! Continued

You feel a certain sense of pride as you look upon the 10 new creations generated by your Naive Bayes model. Glowing with success, you turn your attention to another planet's fashion dilemma—but this time the problem isn't quite as simple.

On the conveniently named Planet Pixel, the dataset you are provided with doesn't consist of the five high-level features that you saw on Wrodl (*hairColor*, *accessories-Type*, etc.), but instead contains just the values of the 32×32 pixels that make up each image. Thus each observation now has $32 \times 32 = 1,024$ features and each feature can take any of 256 values (the individual colors in the palette).

Images from the new dataset are shown in Figure 1-9, and a sample of the pixel values for the first 10 observations appears in Table 1-3.



Figure 1-9. Fashions on Planet Pixel

Table 1-3. The values of pixels 458–467 from the first 10 observations on Planet Pixel

face_id	px_458	px_459	px_460	px_461	px_462	px_463	px_464	px_465	px_466	px_467
0	49	14	14	19	7	5	5	12	19	14
1	43	10	10	17	9	3	3	18	17	10
2	37	12	12	14	11	4	4	6	14	12
3	54	9	9	14	10	4	4	16	14	9
4	2	2	5	2	4	4	4	4	2	5
5	44	15	15	21	14	3	3	4	21	15
6	12	9	2	31	16	3	3	16	31	2
7	36	9	9	13	11	4	4	12	13	9
8	54	11	11	16	10	4	4	19	16	11
9	49	17	17	19	12	6	6	22	19	17

You decide to try your trusty Naive Bayes model once more, this time trained on the pixel dataset. The model will estimate the maximum likelihood parameters that govern the distribution of the color of each pixel so that you are able to sample from this distribution to generate new observations. However, when you do so, it is clear that something has gone very wrong.

Rather than producing novel fashions, the model outputs 10 very similar images that have no distinguishable accessories or clear blocks of hair or clothing color ([Figure 1-10](#)). Why is this?



Figure 1-10. Ten new Planet Pixel styles, generated by the Naive Bayes model

The Challenges of Generative Modeling

First, since the Naive Bayes model is sampling pixels independently, it has no way of knowing that two adjacent pixels are probably quite similar in shade, as they are part of the same item of clothing, for example. The model can generate the facial color and mouth, as all of these pixels in the training set are roughly the same shade in each observation; however for the T-shirt pixels, each pixel is sampled at random from a variety of different colors in the training set, with no regard to the colors that have been sampled in neighboring pixels. Additionally, there is no mechanism for pixels near the eyes to form circular glasses shapes, or red pixels near the top of the image to exhibit a wavy pattern to represent a particular hairstyle, for example.

Second, there are now an incomprehensibly vast number of possible observations in the sample space. Only a tiny proportion of these are recognizable faces, and an even smaller subset are faces that adhere to the fashion rules on Planet Pixel. Therefore, if our Naive Bayes model is working directly with the highly correlated pixel values, the chance of it finding a satisfying combination of values is incredibly small.

In summary, on Planet Wrodl individual features are independent and the sample space is relatively small, so Naive Bayes works well. On Planet Pixel, the assumption that every pixel value is independent of every other pixel value clearly doesn't hold. Pixel values are highly correlated and the sample space is vast, so finding a valid face by sampling pixels independently is almost impossible. This explains why Naive Bayes models cannot be expected to work well on raw image data.

This example highlights the two key challenges that a generative model must overcome in order to be successful.

Generative Modeling Challenges

- How does the model cope with the high degree of conditional dependence between features?
- How does the model find one of the tiny proportion of satisfying possible generated observations among a high-dimensional sample space?

Deep learning is the key to solving both of these challenges.

We need a model that can infer relevant structure from the data, rather than being told which assumptions to make in advance. This is exactly where deep learning excels and is one of the key reasons why the technique has driven the major recent advances in generative modeling.

The fact that deep learning can form its own features in a lower-dimensional space means that it is a form of *representation learning*. It is important to understand the key concepts of representation learning before we tackle deep learning in the next chapter.

Representation Learning

The core idea behind representation learning is that instead of trying to model the high-dimensional sample space directly, we should instead describe each observation in the training set using some low-dimensional *latent* space and then learn a mapping function that can take a point in the latent space and map it to a point in the original domain. In other words, each point in the latent space is the *representation* of some high-dimensional image.

What does this mean in practice? Let's suppose we have a training set consisting of grayscale images of biscuit tins ([Figure 1-11](#)).

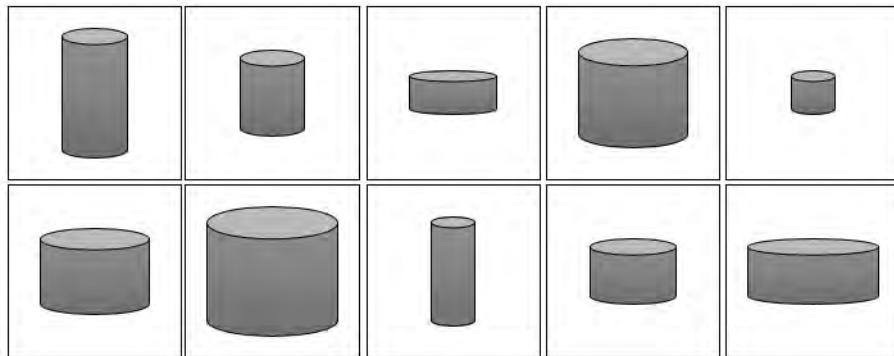


Figure 1-11. The biscuit tin dataset

To us, it is obvious that there are two features that can uniquely represent each of these tins: the height and width of the tin. Given a height and width, we could draw the corresponding tin, even if its image wasn't in the training set. However, this is not so easy for a machine—it would first need to establish that height and width are the two latent space dimensions that best describe this dataset, then learn the mapping function, f , that can take a point in this space and map it to a grayscale biscuit tin image. The resulting latent space of biscuit tins and generation process are shown in Figure 1-12.

Deep learning gives us the ability to learn the often highly complex mapping function f in a variety of ways. We shall explore some of the most important techniques in later chapters of this book. For now, it is enough to understand at a high level what representation learning is trying to achieve.

One of the advantages of using representation learning is that we can perform operations within the more manageable latent space that affect high-level properties of the image. It is not obvious how to adjust the shading of every single pixel to make a given biscuit tin image *taller*. However, in the latent space, it's simply a case of adding 1 to the *height* latent dimension, then applying the mapping function to return to the image domain. We shall see an explicit example of this in the next chapter, applied not to biscuit tins but to faces.

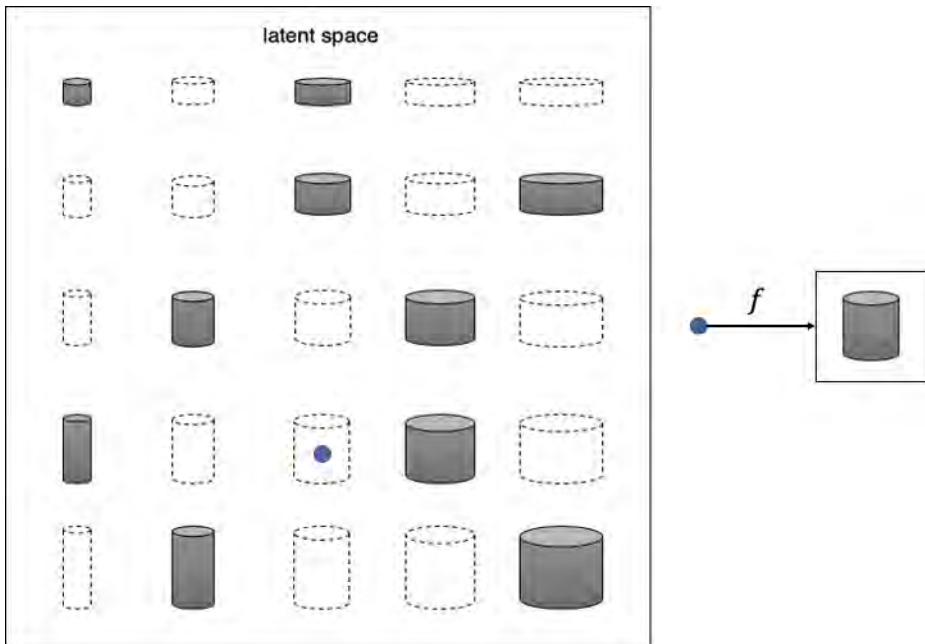


Figure 1-12. The latent space of biscuit tins and the function f that maps a point in the latent space to the original image domain

Representation learning comes so naturally to us as humans that you may never have stopped to think just how amazing it is that we can do it so effortlessly. Suppose you wanted to describe your appearance to someone who was looking for you in a crowd of people and didn't know what you looked like. You wouldn't start by stating the color of pixel 1 of your hair, then pixel 2, then pixel 3, etc. Instead, you would make the reasonable assumption that the other person has a general idea of what an average human looks like, then amend this baseline with features that describe groups of pixels, such as *I have very blonde hair* or *I wear glasses*. With no more than 10 or so of these statements, the person would be able to map the description back into pixels to generate an image of you in their head. The image wouldn't be perfect, but it would be a close enough likeness to your actual appearance for them to find you among possibly hundreds of other people, even if they've never seen you before.

Note that representation learning doesn't just assign values to a given set of features such as *blondeness of hair*, *height*, etc., for some given image. The power of representation learning is that it actually learns which features are most important for it to describe the given observations and how to generate these features from the raw data. Mathematically speaking, it tries to find the highly nonlinear *manifold* on which the data lies and then establish the dimensions required to fully describe this space. This is shown in Figure 1-13.

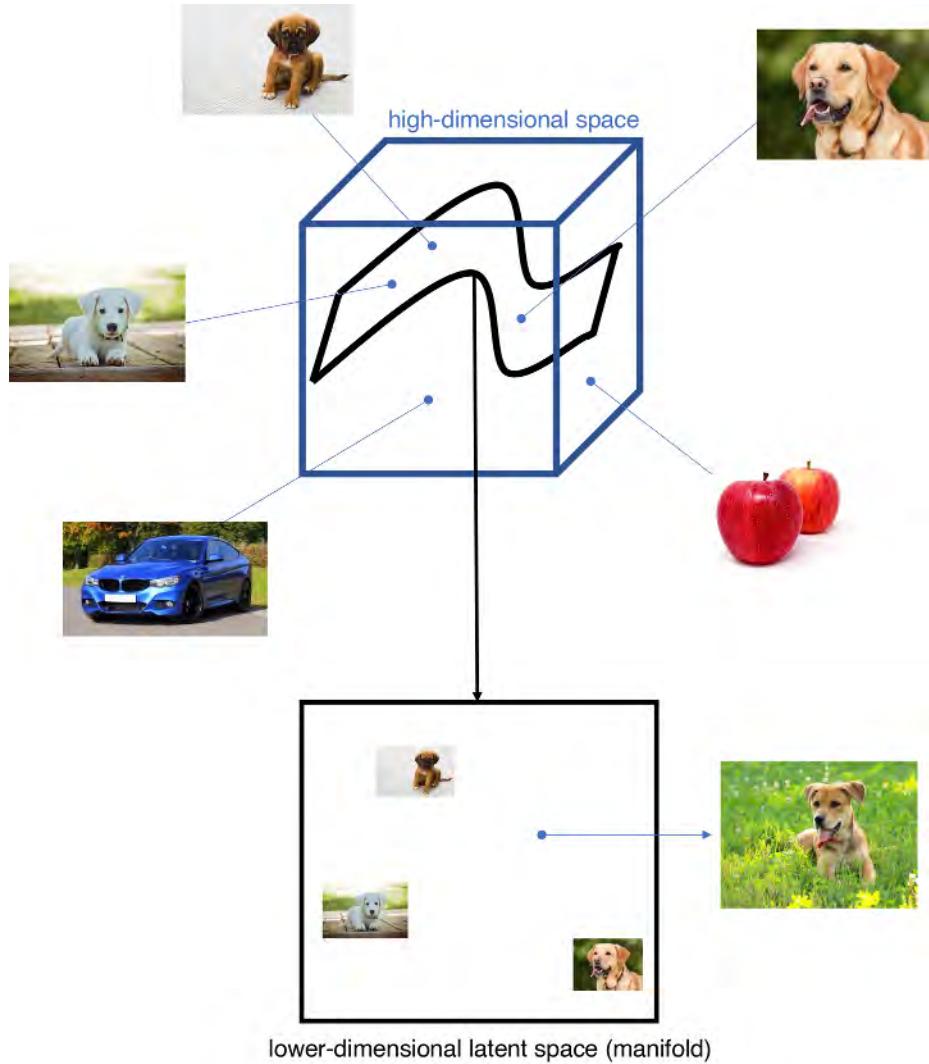


Figure 1-13. The cube represents the extremely high-dimensional space of all images; representation learning tries to find the lower-dimensional latent subspace or manifold on which particular kinds of image lie (for example, the dog manifold)

In summary, representation learning establishes the most relevant high-level features that describe how groups of pixels are displayed so that it is likely that any point in the latent space is the representation of a well-formed image. By tweaking the values of features in the latent space we can produce novel representations that, when

mapped back to the original image domain, have a much better chance of looking *real* than if we'd tried to work directly with the individual raw pixels.

Now that you have an understanding of representation learning, which forms the backbone of many of the generative deep learning examples in this book, all that remains is to set up your environment so that you can begin building generative deep learning models of your own.

Setting Up Your Environment

Throughout this book, there are many worked examples of how to build the models that we will be discussing in the text.

To get access to these examples, you'll need to clone the Git repository that accompanies this book. Git is an open source version control system and will allow you to copy the code locally so that you can run the notebooks on your own machine, or perhaps in a cloud-based environment. You may already have this installed, but if not, follow the [instructions relevant to your operating system](#).

To clone the repository for this book, navigate to the folder where you would like to store the files and type the following into your terminal:

```
git clone https://github.com/davidADSP/GDL_code.git
```

Always make sure that you have the most up-to-date version of the codebase by running the following command:

```
git pull
```

You should now be able to see the files in a folder on your machine.

Next, you need to set up a virtual environment. This is simply a folder into which you'll install a fresh copy of Python and all of the packages that we will be using in this book. This way, you can be sure that your system version of Python isn't affected by any of the libraries that we will be using.

If you are using Anaconda, you can set up a virtual environment as follows:

```
conda create -n generative python=3.6 ipykernel
```

If not, you can install *virtualenv* and *virtualenvwrapper* with the command:¹⁰

```
pip install virtualenv virtualenvwrapper
```

¹⁰ For full instructions on installing *virtualenvwrapper*, consult the [documentation](#).

You will also need to add the following lines to your shell startup script (e.g., `.bash_profile`):

```
export WORKON_HOME=$HOME/.virtualenvs ❶
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3 ❷
source /usr/local/bin/virtualenvwrapper.sh ❸
```

- ❶ The location where your virtual environments will be stored
- ❷ The default version of Python to use when a virtual environment is created—make sure this points at Python 3, rather than Python 2.
- ❸ Reloads the `virtualenvwrapper` initialization script

To create a virtual environment called `generative`, simply enter the following into your terminal:

```
mkvirtualenv generative
```

You'll know that you're inside the virtual environment because your terminal will show (`generative`) at the start of the prompt.

Now you can go ahead and install all the packages that we'll be using in this book with the following command:

```
pip install -r requirements.txt
```

Throughout this book, we will use Python 3. The `requirements.txt` file contains the names and version numbers of all the packages that you will need to run the examples.

To check everything works as expected, from inside your virtual environment type `python` into your terminal and then try to import Keras (a deep learning library that we will be using extensively in this book). You should see a Python 3 prompt, with Keras reporting that it is using the TensorFlow backend as shown in [Figure 1-14](#).

```
Python 3.6.5 (default, Oct  6 2018, 09:49:35)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.11.45.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import keras
Using TensorFlow backend.
>>> keras.__version__
'2.2.4'
```

Figure 1-14. Setting up your environment

Finally, you will need to ensure you are set up to access your virtual environment through Jupyter notebooks on your machine. Jupyter is a way to interactively code in

Python through your browser and is a great option for developing new ideas and sharing code. Most of the examples in this book are written using Jupyter notebooks.

To do this, run the following command from your terminal inside your virtual environment:

```
python -m ipykernel install --user --name generative ①
```

- ① This gives you access to the virtual environment that you've just set up (`generative`) inside Jupyter notebooks.

To check that it has installed correctly, navigate in your terminal to the folder where you have cloned the book repository and type:

```
jupyter notebook
```

A window should open in your browser showing a screen similar to [Figure 1-15](#). Click the notebook you wish to run and, from the *Kernel → Change kernel* dropdown, select the `generative` virtual environment.



Figure 1-15. Jupyter notebook

You are now ready to start building generative deep neural networks.

Summary

This chapter introduced the field of generative modeling, an important branch of machine learning that complements the more widely studied discriminative modeling. Our first basic example of a generative model utilized the Naive Bayes assumption to produce a probability distribution that was able to represent inherent structure in the data and generate examples outside of the training set. We also saw how these kinds of basic models can fail as the complexity of the generative task grows, and analyzed the general challenges associated with generative modeling. Finally, we took our first look at representation learning, an important concept that forms the core of many generative models.

In [Chapter 2](#), we will begin our exploration of deep learning and see how to use Keras to build models that can perform discriminative modeling tasks. This will give us the necessary foundations to go on to tackle generative deep learning in later chapters.

Deep Learning

Let's start with a basic definition of deep learning:

Deep learning is a class of machine learning algorithm that uses multiple stacked layers of processing units to learn high-level representations from unstructured data.

To understand deep learning fully, and particularly why it is so useful within generative modeling, we need to delve into this definition a bit further. First, what do we mean by “unstructured data” and its counterpart, “structured data”?

Structured and Unstructured Data

Many types of machine learning algorithm require *structured*, tabular data as input, arranged into columns of features that describe each observation. For example, a person's age, income, and number of website visits in the last month are all features that could help to predict if the person will subscribe to a particular online service in the coming month. We could use a structured table of these features to train a logistic regression, random forest, or XGBoost model to predict the binary response variable —did the person subscribe (1) or not (0)? Here, each individual feature contains a nugget of information about the observation, and the model would learn how these features interact to influence the response.

Unstructured data refers to any data that is not naturally arranged into columns of features, such as images, audio, and text. There is of course spatial structure to an image, temporal structure to a recording, and both spatial and temporal structure to video data, but since the data does not arrive in columns of features, it is considered unstructured, as shown in [Figure 2-1](#).

STRUCTURED DATA				
id	age	gender	height (cm)	location
0001	54	M	186	London
0002	35	F	166	New York
0003	62	F	170	Amsterdam
0004	23	M	164	London
0005	25	M	180	Cairo
0006	29	F	181	Beijing
0007	46	M	172	Chicago

UNSTRUCTURED DATA		
images	audio	text
		This service is terrible!
		Your website is great!

Figure 2-1. The difference between structured and unstructured data

When our data is unstructured, individual pixels, frequencies, or characters are almost entirely uninformative. For example, knowing that pixel 234 of an image is a muddy shade of brown doesn't really help identify if the image is of a house or a dog, and knowing that character 24 of a sentence is an *e* doesn't help predict if the text is about football or politics.

Pixels or characters are really just the dimples of the canvas into which higher-level informative features, such as an image of a chimney or the word *striker*, are embedded. If the chimney in the image were placed on the other side of the house, the image would still contain a chimney, but this information would now be carried by completely different pixels. If the word *striker* appeared slightly earlier or later in the text, the text would still be about football, but different character positions would provide this information. The granularity of the data combined with the high degree of spatial dependence destroys the concept of the pixel or character as an informative feature in its own right.

For this reason, if we train logistic regression, random forest, or XGBoost algorithms on raw pixel values, the trained model will often perform poorly for all but the simplest of classification tasks. These models rely on the input features to be informative and not spatially dependent. A deep learning model, on the other hand, can learn how to build high-level informative features by itself, directly from the unstructured data.

Deep learning can be applied to structured data, but its real power, especially with regard to generative modeling, comes from its ability to work with unstructured data. Most often, we want to generate unstructured data such as new images or original strings of text, which is why deep learning has had such a profound impact on the field of generative modeling.

Deep Neural Networks

The majority of deep learning systems are *artificial neural networks* (ANNs, or just *neural networks* for short) with multiple stacked hidden layers. For this reason, *deep learning* has now almost become synonymous with *deep neural networks*. However, it is important to point out that any system that employs many layers to learn high level representations of the input data is also a form of deep learning (e.g., deep belief networks and deep Boltzmann machines).

Let's start by taking a high-level look at how a deep neural network can make a prediction about a given input.

A deep neural network consists of a series of stacked *layers*. Each layer contains *units*, that are connected to the previous layer's units through a set of *weights*. As we shall see, there are many different types of layer, but one of the most common is the *dense* layer that connects all units in the layer directly to every unit in the previous layer. By stacking layers, the units in each subsequent layer can represent increasingly sophisticated aspects of the original input.

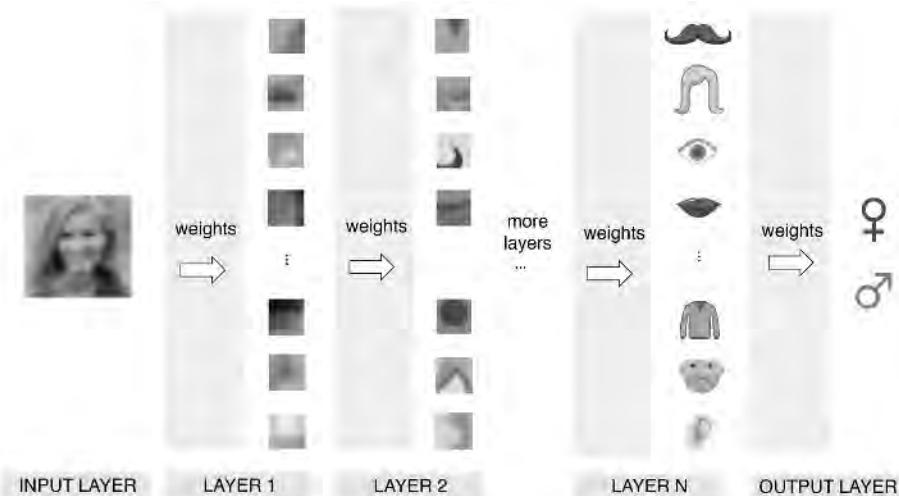


Figure 2-2. Deep learning conceptual diagram

For example, in Figure 2-2, layer 1 consists of units that activate more strongly when they detect particular basic properties of the input image, such as edges. The output from these units is then passed to the units of layer 2, which are able to use this information to detect slightly more complex features—and so on, through the network. The final output layer is the culmination of this process, where the network outputs a set of numbers that can be converted into probabilities, to represent the chance that the original input belongs to one of n categories.

The magic of deep neural networks lies in finding the set of weights for each layer that results in the most accurate predictions. The process of finding these weights is what we mean by *training* the network.

During the training process, batches of images are passed through the network and the output is compared to the ground truth. The error in the prediction is then propagated backward through the network, adjusting each set of weights a small amount in the direction that improves the prediction most significantly. This process is appropriately called *backpropagation*. Gradually, each unit becomes skilled at identifying a particular feature that ultimately helps the network to make better predictions.

Deep neural networks can have any number of middle or *hidden* layers. For example, ResNet,¹ designed for image recognition, contains 152 layers. We shall see in [Chapter 3](#) that we can use deep neural networks to influence high-level features of an image, such as hair color or expression of a face, by manually tweaking the values of these hidden layers. This is only possible because the deeper layers of the network are capturing high-level features that we can work with directly.

Next, we'll dive straight into the practical side of deep learning and get set up with Keras and TensorFlow, the two libraries that will enable you to start building your own generative deep neural networks.

Keras and TensorFlow

Keras is a high-level Python library for building neural networks and is the core library that we shall be using in this book. It is extremely flexible and has a very user-friendly API, making it an ideal choice for getting started with deep learning. Moreover, Keras provides numerous useful building blocks that can be plugged together to create highly complex deep learning architectures through its functional API.

Keras is not the library that performs the low-level array operations required to train neural networks. Instead Keras utilizes one of three backend libraries for this purpose: TensorFlow, CNTK, or Theano. You are free to choose whichever you are most comfortable with, or whichever library works fastest for a particular network architecture. For most purposes, it doesn't matter which you choose as you usually won't be coding directly using the underlying backend framework. In this book we use TensorFlow as it is the most widely adopted and best documented of the three.

TensorFlow is an open-source Python library for machine learning, developed by Google. It is now one of the most utilized frameworks for building machine learning solutions, with particular emphasis on the manipulation of tensors (hence the name).

¹ Kaiming He et al., "Deep Residual Learning for Image Recognition," 10 December 2015, <https://arxiv.org/abs/1512.03385>.

Within the context of deep learning, tensors are simply multidimensional arrays that store the data as it flows through the network. As we shall see, understanding how each layer of a neural network changes the shape of the data as it flows through the network is a key part of truly understanding the mechanics of deep learning.

If you are just getting started with deep learning, I highly recommend that you choose Keras with a TensorFlow backend as your toolkit. These two libraries are a powerful combination that will allow you to build any network that you can think of in a production environment, while also giving you the easy-to-learn API that is so important for rapid development of new ideas and concepts.

Your First Deep Neural Network

Let's start by seeing how easy it is to build a deep neural network in Keras.

We will be working through the Jupyter notebook in the book repository called *02_01_deep_learning_deep_neural_network.ipynb*.

Loading the Data

For this example we will be using the CIFAR-10 dataset, a collection of 60,000 32×32 -pixel color images that comes bundled with Keras out of the box. Each image is classified into exactly one of 10 classes, as shown in [Figure 2-3](#).

The following code loads and scales the data:

```
import numpy as np
from keras.utils import to_categorical
from keras.datasets import cifar10

(x_train, y_train), (x_test, y_test) = cifar10.load_data() ❶

NUM_CLASSES = 10

x_train = x_train.astype('float32') / 255.0 ❷
x_test = x_test.astype('float32') / 255.0

y_train = to_categorical(y_train, NUM_CLASSES) ❸
y_test = to_categorical(y_test, NUM_CLASSES)
```

- ❶ Loads the CIFAR-10 dataset. `x_train` and `x_test` are `numpy` arrays of shape `[50000, 32, 32, 3]` and `[10000, 32, 32, 3]`, respectively. `y_train` and `y_test` are `numpy` arrays with shape `[50000, 1]` and `[10000, 1]`, respectively, containing the integer labels in the range 0 to 9 for the class of each image.

- ② By default the image data consists of integers between 0 and 255 for each pixel channel. Neural networks work best when each input is inside the range -1 to 1, so we need to divide by 255.
- ③ We also need to change the integer labeling of the images to one-hot-encoded vectors. If the class integer label of an image is i , then its one-hot encoding is a vector of length 10 (the number of classes) that has 0s in all but the i th element, which is 1. The new shapes of y_{train} and y_{test} are therefore [50000, 10] and [10000, 10] respectively.



Figure 2-3. Example images from the CIFAR-10 dataset²

² Source: Alex Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” 2009, <https://www.cs.toronto.edu/~kriz/cifar.html>.

It's worth noting the shape of the image data in `x_train`: [50000, 32, 32, 3]. The first dimension of this array references the index of the image in the dataset, the second and third relate to the size of the image, and the last is the channel (i.e., red, green, or blue, since these are RGB images). There are no *columns* or *rows* in this dataset; instead, this is a *tensor* with four dimensions. For example, the following entry refers to the green channel (1) value of the pixel in the (12,13) position of image 54:

```
x_train[54, 12, 13, 1]
# 0.36862746
```

Building the Model

In Keras there are two ways to define the structure of your neural network: as a Sequential model or using the Functional API.

A Sequential model is useful for quickly defining a linear stack of layers (i.e., where one layer follows on directly from the previous layer without any branching). However, many of the models in this book require that the output from a layer is passed to multiple separate layers beneath it, or conversely, that a layer receives input from multiple layers above it.

To be able to build networks with branches, we need to use the Functional API, which is a lot more flexible. I recommend that even if you are just starting out building linear models with Keras, you still use the Functional API rather than Sequential models, since it will serve you better in the long run as your neural networks become more architecturally complex. The Functional API will give you complete freedom over the design of your deep neural network.

To demonstrate the difference between the two methods, Examples 2-1 and 2-2 show the same network coded using a Sequential model and the Functional API. Feel free to try both and observe that they give the same result.

Example 2-1. The architecture using a Sequential model

```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential([
    Dense(200, activation = 'relu', input_shape=(32, 32, 3)),
    Flatten(),
    Dense(150, activation = 'relu'),
    Dense(10, activation = 'softmax'),
])
```

Example 2-2. The architecture using the Functional API

```
from keras.layers import Input, Flatten, Dense
from keras.models import Model

input_layer = Input(shape=(32,32, 3))

x = Flatten()(input_layer)

x = Dense(units=200, activation = 'relu')(x)
x = Dense(units=150, activation = 'relu')(x)

output_layer = Dense(units=10, activation = 'softmax')(x)

model = Model(input_layer, output_layer)>
```

Here, we are using three different types of layer: `Input`, `Flatten`, and `Dense`.

The `Input` layer is an entry point into the network. We tell the network the shape of each data element to expect as a tuple. Notice that we do not specify the batch size; this isn't necessary as we can pass any number of images into the `Input` layer simultaneously. We do not need to explicitly state the batch size in the `Input` layer definition.

Next we flatten this input into a vector, using a `Flatten` layer. This results in a vector of length 3,072 ($= 32 \times 32 \times 3$). The reason we do this is because the subsequent `Dense` layer requires that its input is flat, rather than a multidimensional array. As we shall see later, other layer types require multidimensional arrays as input, so you need to be aware of the required input and output shape of each layer type to understand when it is necessary to use `Flatten`.

The `Dense` layer is perhaps the most fundamental layer type in any neural network. It contains a given number of units that are densely connected to the previous layer—that is, every unit in the layer is connected to every unit in the previous layer, through a single connection that carries a weight (which can be positive or negative). The output from a given unit is the weighted sum of the input it receives from the previous layer, which is then passed through a nonlinear *activation function* before being sent to the following layer. The activation function is critical to ensure the neural network is able to learn complex functions and doesn't just output a linear combination of its input.

There are many kinds of activation function, but the three most important are ReLU, sigmoid, and softmax.

The *ReLU* (rectified linear unit) activation function is defined to be zero if the input is negative and is otherwise equal to the input. The *LeakyReLU* activation function is very similar to ReLU, with one key difference: whereas the ReLU activation function returns zero for input values less than zero, the LeakyReLU function returns a small

negative number proportional to the input. ReLU units can sometimes *die* if they always output zero, because of a large bias toward negative values preactivation. In this case, the gradient is zero and therefore no error is propagated back through this unit. LeakyReLU activations fix the issue by always ensuring the gradient is nonzero. ReLU-based functions are now established to be the most reliable activations to use between the layers of a deep network to encourage stable training.

The *sigmoid* activation is useful if you wish the output from the layer to be scaled between 0 and 1—for example, for binary classification problems with one output unit or multilabel classification problems, where each observation can belong to more than one class. [Figure 2-4](#) shows ReLU, LeakyReLU, and sigmoid activation functions side by side for comparison.

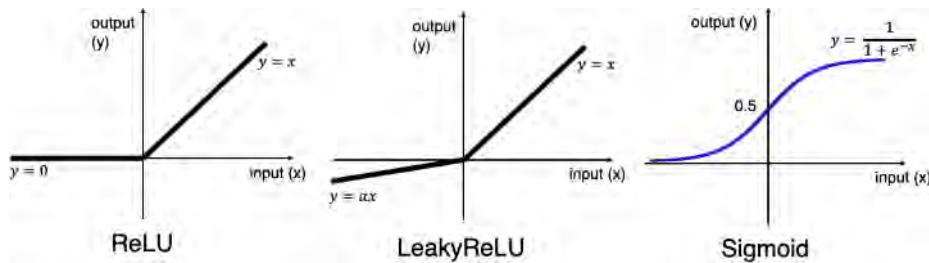


Figure 2-4. The ReLU, LeakyReLU, and sigmoid activation functions

The *softmax* activation is useful if you want the total sum of the output from the layer to equal 1, for example, for multiclass classification problems where each observation only belongs to exactly one class. It is defined as:

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}}$$

Here, J is the total number of units in the layer. In our neural network, we use a softmax activation in the final layer to ensure that the output is a set of 10 probabilities that sum to 1, which can be interpreted as the chance that the image belongs to each class.

In Keras, activation functions can also be defined in a separate layer as follows:

```
x = Dense(units=200)(x)
x = Activation('relu')(x)
```

This is equivalent to:

```
x = Dense(units=200, activation = 'relu')(x)
```

In our example, we pass the input through two dense hidden layers, the first with 200 units and the second with 150, both with ReLU activation functions. A diagram of the total network is shown in [Figure 2-5](#).

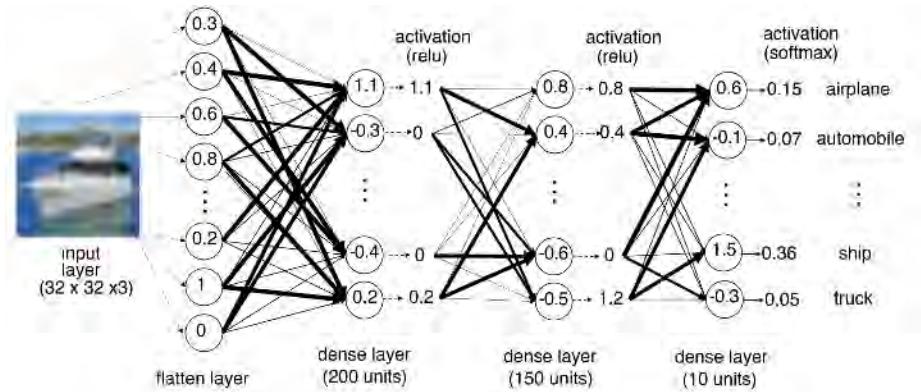


Figure 2-5. A diagram of the neural network trained on CIFAR-10 data

The final step is to define the model itself, using the `Model` class. In Keras a model is defined by the input and output layers. In our case, we have one input layer that we defined earlier, and the output layer is the final `Dense` layer of 10 units. It is also possible to define models with multiple input and output layers; we shall see this in action later in the book.

In our example, as required, the shape of our `Input` layer matches the shape of `x_train` and the shape of our `Dense` output layer matches the shape of `y_train`. To illustrate this, we can use the `model.summary()` method to see the shape of the network at each layer as shown in [Figure 2-6](#).

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 32, 32, 3)	0
flatten_1 (Flatten)	(None, 3072)	0
dense_1 (Dense)	(None, 200)	614600
dense_2 (Dense)	(None, 150)	30150
dense_3 (Dense)	(None, 10)	1510
<hr/>		
Total params:	646,260	
Trainable params:	646,260	
Non-trainable params:	0	

Figure 2-6. The summary of the model

Notice how Keras uses `None` as a marker to show that it doesn't yet know the number of observations that will be passed into the network. In fact, it doesn't need to; we could just as easily pass one observation through the network at a time as 1,000. That's because tensor operations are conducted across all observations simultaneously using linear algebra—this is the part handled by TensorFlow. It is also the reason why you get a performance increase when training deep neural networks on GPUs instead of CPUs: GPUs are optimized for large tensor multiplications since these calculations are also necessary for complex graphics manipulation.

The `summary` method also gives the number of parameters (weights) that will be trained at each layer. If ever you find that your model is training too slowly, check the summary to see if there are any layers that contain a huge number of weights. If so, you should consider whether the number of units in the layer could be reduced to speed up training.

Compiling the Model

In this step, we compile the model with an optimizer and a loss function:

```
from keras.optimizers import Adam

opt = Adam(lr=0.0005)
model.compile(loss='categorical_crossentropy', optimizer=opt,
               metrics=['accuracy'])
```

The loss function is used by the neural network to compare its predicted output to the ground truth. It returns a single number for each observation; the greater this number, the worse the network has performed for this observation.

Keras provides many built-in loss functions to choose from, or you can create your own. Three of the most commonly used are mean squared error, categorical cross-entropy, and binary cross-entropy. It is important to understand when it is appropriate to use each.

If your neural network is designed to solve a regression problem (i.e., the output is continuous), then you can use the *mean squared error* loss. This is the mean of the squared difference between the ground truth y_i and predicted value p_i of each output unit, where the mean is taken over all n output units:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - p_i)^2$$

If you are working on a classification problem where each observation only belongs to one class, then *categorical cross-entropy* is the correct loss function. This is defined as follows:

$$-\sum_{i=1}^n y_i \log(p_i)$$

Finally, if you are working on a binary classification problem with one output unit, or a multilabel problem where each observation can belong to multiple classes simultaneously, you should use *binary cross-entropy*:

$$-\frac{1}{n} \sum_{i=1}^n (y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

The optimizer is the algorithm that will be used to update the weights in the neural network based on the gradient of the loss function. One of the most commonly used and stable optimizers is *Adam*.³ In most cases, you shouldn't need to tweak the default parameters of the Adam optimizer, except for the learning rate. The greater the learning rate, the larger the change in weights at each training step. While training is initially faster with a large learning rate, the downside is that it may result in less stable training and may not find the minima of the loss function. This is a parameter that you may want to tune or adjust during training.

³ Diederik Kingma and Jimmy Ba, “Adam: A Method for Stochastic Optimization,” 22 December 2014, <https://arxiv.org/abs/1412.6980v8>.

Another common optimizer that you may come across is *RMSProp*. Again, you shouldn't need to adjust the parameters of this optimizer too much, but it is worth reading the [Keras documentation](#) to understand the role of each parameter.

We pass both the loss function and the optimizer into the `compile` method of the model, as well as a `metrics` parameter where we can specify any additional metrics that we would like reporting on during training, such as accuracy.

Training the Model

Thus far, we haven't shown the model any data and have just set up the architecture and compiled the model with a loss function and optimizer.

To train the model, simply call the `fit` method, as shown here:

```
model.fit(x_train ❶
          , y_train ❷
          , batch_size = 32 ❸
          , epochs = 10 ❹
          , shuffle = True ❺
        )
```

- ❶ The raw image data.
- ❷ The one-hot-encoded class labels.
- ❸ The `batch_size` determines how many observations will be passed to the network at each training step.
- ❹ The `epochs` determine how many times the network will be shown the full training data.
- ❺ If `shuffle = True`, the batches will be drawn randomly without replacement from the training data at each training step.

This will start training a deep neural network to predict the category of an image from the CIFAR-10 dataset.

The training process works as follows. First, the weights of the network are initialized to small random values. Then the network performs a series of training steps.

At each training step, one batch of images is passed through the network and the errors are backpropagated to update the weights. The `batch_size` determines how many images are in each training step batch. The larger the batch size, the more stable the gradient calculation, but the slower each training step. It would be far too time-consuming and computationally intensive to use the entire dataset to calculate the gradient at each training step, so generally a batch size between 32 and 256 is

used. It is also now recommended practice to increase the batch size as training progresses.⁴

This continues until all observations in the dataset have been seen once. This completes the first *epoch*. The data is then passed through the network again in batches as part of the second epoch. This process repeats until the specified number of epochs have elapsed.

During training, Keras outputs the progress of the procedure, as shown in Figure 2-7. We can see that the training dataset of 50,000 observations has been shown to the network 10 times (i.e., over 10 epochs), at a rate of approximately 160 microseconds per observation. The categorical cross-entropy loss has fallen from 1.842 to 1.357, resulting in an accuracy increase from 33.5% after the first epoch to 51.9% after the tenth epoch.

```
model.fit(x_train
          , y_train
          , batch_size=BATCH_SIZE
          , epochs=EPOCHS
          , shuffle=True)

Epoch 1/10
50000/50000 [=====] - 8s 164us/step - loss: 1.8424 - acc: 0.3354
Epoch 2/10
50000/50000 [=====] - 8s 154us/step - loss: 1.6592 - acc: 0.4048
Epoch 3/10
50000/50000 [=====] - 8s 153us/step - loss: 1.5733 - acc: 0.4381
Epoch 4/10
50000/50000 [=====] - 8s 154us/step - loss: 1.5232 - acc: 0.4379
Epoch 5/10
50000/50000 [=====] - 8s 155us/step - loss: 1.4874 - acc: 0.4698
Epoch 6/10
50000/50000 [=====] - 8s 165us/step - loss: 1.4569 - acc: 0.4799
Epoch 7/10
50000/50000 [=====] - 10s 208us/step - loss: 1.4281 - acc: 0.4887
Epoch 8/10
50000/50000 [=====] - 8s 165us/step - loss: 1.4038 - acc: 0.4984
Epoch 9/10
50000/50000 [=====] - 8s 153us/step - loss: 1.3797 - acc: 0.5084
Epoch 10/10
50000/50000 [=====] - 8s 155us/step - loss: 1.3571 - acc: 0.5187
```

Figure 2-7. The output from the fit method

Evaluating the Model

We know the model achieves an accuracy of 51.9% on the training set, but how does it perform on data it has never seen?

To answer this question we can use the evaluate method provided by Keras:

```
model.evaluate(x_test, y_test)
```

⁴ Samuel L. Smith et al., “Don’t Decay the Learning Rate, Increase the Batch Size,” 1 November 2017, <https://arxiv.org/abs/1711.00489>.

Figure 2-8 shows the output from this method.

```
10000/10000 [=====] - 1s 55us/step  
[1.4358007415771485, 0.4896]
```

Figure 2-8. The output from the evaluate method

The output from this method is a list of the metrics we are monitoring: categorical cross-entropy and accuracy. We can see that model accuracy is still 49.0% even on images that it has never seen before. Note that if the model was guessing randomly, it would achieve approximately 10% accuracy (because there are 10 classes), so 50% is a good result given that we have used a very basic neural network.

We can view some of the predictions on the test set using the `predict` method:

```
CLASSES = np.array(['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog'  
    , 'frog', 'horse', 'ship', 'truck'])  
  
preds = model.predict(x_test) ❶  
preds_single = CLASSES[np.argmax(preds, axis = -1)] ❷  
actual_single = CLASSES[np.argmax(y_test, axis = -1)]
```

- ❶ `preds` is an array of shape `[10000, 10]`—i.e., a vector of 10 class probabilities for each observation.
- ❷ We convert this array of probabilities back into a single prediction using `numpy`'s `argmax` function. Here, `axis = -1` tells the function to collapse the array over the last dimension (the classes dimension), so that the shape of `preds_single` is then `[10000, 1]`.

We can view some of the images alongside their labels and predictions with the following code. As expected, around half are correct:

```
import matplotlib.pyplot as plt  
  
n_to_show = 10  
indices = np.random.choice(range(len(x_test)), n_to_show)  
  
fig = plt.figure(figsize=(15, 3))  
fig.subplots_adjust(hspace=0.4, wspace=0.4)  
  
for i, idx in enumerate(indices):  
    img = x_test[idx]  
    ax = fig.add_subplot(1, n_to_show, i+1)  
    ax.axis('off')  
    ax.text(0.5, -0.35, 'pred = ' + str(preds_single[idx]), fontsize=10  
        , ha='center', transform=ax.transAxes)
```

```
ax.text(0.5, -0.7, 'act = ' + str(actual_single[idx]), fontsize=10  
       , ha='center', transform=ax.transAxes)  
ax.imshow(img)
```

Figure 2-9 shows a randomly chosen selection of predictions made by the model, alongside the true labels.



Figure 2-9. Some predictions made by the model, alongside the actual labels

Congratulations! You've just built your first deep neural network using Keras and used it to make predictions on new data. Even though this is a supervised learning problem, when we come to building generative models in future chapters many of the core ideas from this network (such as loss functions, activation functions, and understanding layer shapes) will still be extremely important. Next we'll look at ways of improving this model, by introducing a few new layer types.

Improving the Model

One of the reasons our network isn't yet performing as well as it might is because there isn't anything in the network that takes into account the spatial structure of the input images. In fact, our first step is to flatten the image into a single vector, so that we can pass it to the first `Dense` layer!

To achieve this we need to use a *convolutional* layer.

Convolutional Layers

First, we need to understand what is meant by a *convolution* in the context of deep learning.

Figure 2-10 shows a $3 \times 3 \times 1$ portion of a grayscale image being convoluted with a $3 \times 3 \times 1$ filter (or *kernel*).

3x3 portion of an image	filter		
0.6 0.2 0.6 0.1 -0.2 -0.3 -0.5 -0.1 -0.3	*	1 1 1 0 0 0 -1 -1 -1	= 2.3
-0.6 -0.2 -0.6 -0.1 0.2 0.3 0.5 0.1 0.3	*	1 1 1 0 0 0 -1 -1 -1	= -2.3

Figure 2-10. The convolution operation

The convolution is performed by multiplying the filter pixelwise with the portion of the image, and summing the result. The output is more positive when the portion of the image closely matches the filter and more negative when the portion of the image is the inverse of the filter.

If we move the filter across the entire image, from left to right and top to bottom, recording the convolutional output as we go, we obtain a new array that picks out a particular feature of the input, depending on the values in the filter.

This is exactly what a convolutional layer is designed to do, but with multiple filters rather than just one. For example, [Figure 2-11](#) shows two filters that highlight horizontal and vertical edges. You can see this convolutional process worked through manually in the notebook `02_02_deep_learning_convolutions.ipynb` in the book repository.

If we are working with color images, then each filter would have three channels rather than one (i.e. each having shape $3 \times 3 \times 3$) to match the three channels (red, green, blue) of the image.

In Keras, the Conv2D layer applies convolutions to an input tensor with two spatial dimensions (such as an image). For example, the Keras code corresponding to the diagram in [Figure 2-11](#) is:

```
input_layer = Input(shape=(64,64,1))

conv_layer_1 = Conv2D(
    filters = 2
    , kernel_size = (3,3)
    , strides = 1
    , padding = "same"
)(input_layer)
```

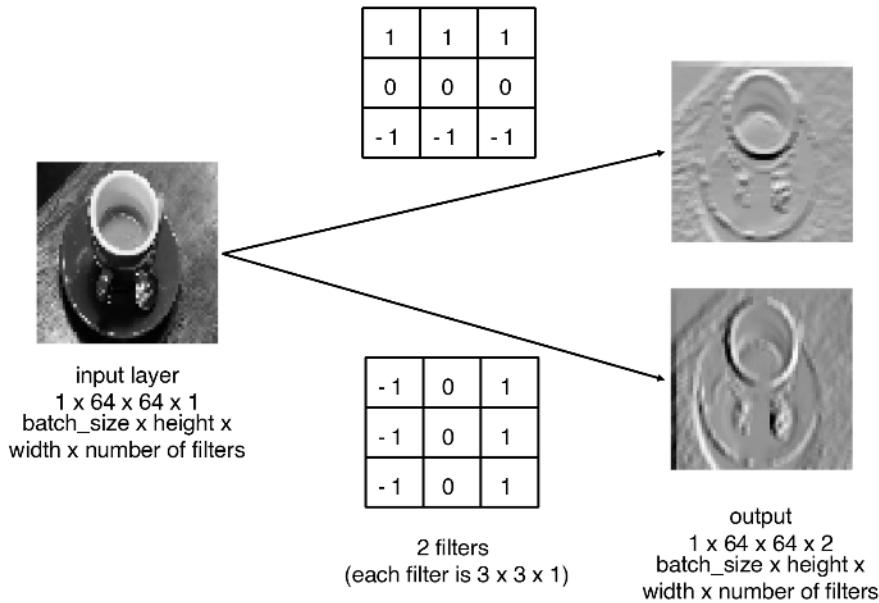


Figure 2-11. Two convolutional filters applied to a grayscale image

Strides

The `strides` parameter is the step size used by the layer to move the filters across the input. Increasing the stride therefore reduces the size of the output tensor. For example, when `strides = 2`, the height and width of the output tensor will be half the size of the input tensor. This is useful for reducing the spatial size of the tensor as it passes through the network, while increasing the number of channels.

Padding

The `padding = "same"` input parameter pads the input data with zeros so that the output size from the layer is exactly the same as the input size when `strides = 1`.

Figure 2-12 shows a 3×3 kernel being passed over a 5×5 input image, with `padding = "same"` and `strides = 1`. The output size from this convolutional layer would also be 5×5 , as the padding allows the kernel to extend over the edge of the image, so that it fits five times in both directions. Without padding, the kernel could only fit three times along each direction, giving an output size of 3×3 .

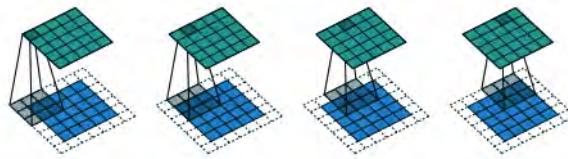


Figure 2-12. A $3 \times 3 \times 1$ kernel (gray) being passed over a $5 \times 5 \times 1$ input image (blue), with `padding = "same"` and `strides = 1`, to generate the $5 \times 5 \times 1$ output (green)⁵

Setting `padding = "same"` is a good way to ensure that you are able to easily keep track of the size of the tensor as it passes through many convolutional layers.

The values stored in the filters are the weights that are learned by the neural network through training. Initially these are random, but gradually the filters adapt their weights to start picking out interesting features such as edges or particular color combinations.

The output of a Conv2D layer is another four-dimensional tensor, now of shape (`batch_size, height, width, filters`), so we can stack Conv2D layers on top of each other to grow the depth of our neural network. It's really important to understand how the shape of the tensor changes as data flows through from one convolutional layer to the next. To demonstrate this, let's imagine we are applying Conv2D layers to the CIFAR-10 dataset. This time, instead of one input channel (grayscale) we have three (red, green, and blue).

Figure 2-13 represents the following network in Keras:

```
input_layer = Input(shape=(32, 32, 3))

conv_layer_1 = Conv2D(
    filters = 10
    , kernel_size = (4,4)
    , strides = 2
    , padding = 'same'
)(input_layer)

conv_layer_2 = Conv2D(
    filters = 20
    , kernel_size = (3,3)
    , strides = 2
    , padding = 'same'
)(conv_layer_1)
```

⁵ Source: Vincent Dumoulin and Francesco Visin, “A Guide to Convolution Arithmetic for Deep Learning,” 12 January 2018, <https://arxiv.org/pdf/1603.07285.pdf>.

```

flatten_layer = Flatten()(conv_layer_2)

output_layer = Dense(units=10, activation = 'softmax')(flatten_layer)

model = Model(input_layer, output_layer)

```

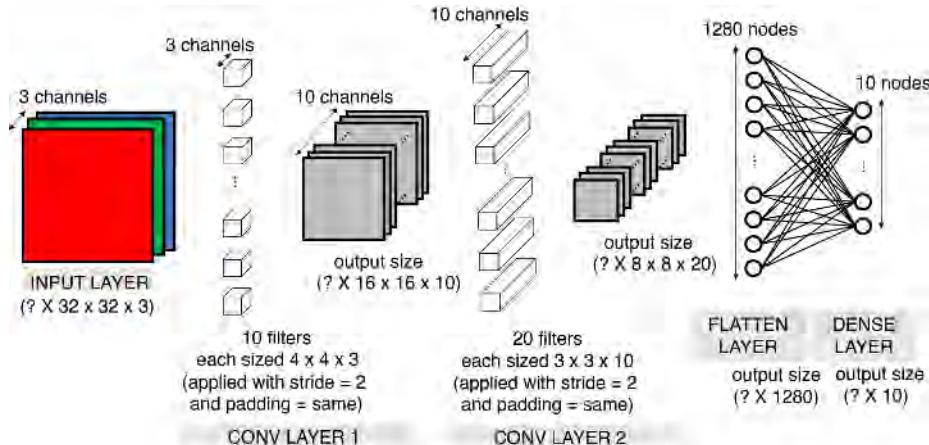


Figure 2-13. A diagram of a convolutional neural network

We can use the `model.summary()` method to see the shape of the tensor as it passes through the network (Figure 2-14).

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	(None, 32, 32, 3)	0
conv2d_1 (Conv2D)	(None, 16, 16, 10)	490
conv2d_2 (Conv2D)	(None, 8, 8, 20)	1820
flatten_1 (Flatten)	(None, 1280)	0
dense_1 (Dense)	(None, 10)	12810
<hr/>		
Total params: 15,120		
Trainable params: 15,120		
Non-trainable params: 0		

Figure 2-14. A convolutional neural network summary

Let's analyze this network from input through to output. The input shape is $(\text{None}, 32, 32, 3)$ —Keras uses `None` to represent the fact that we can pass any number of images through the network simultaneously. Since the network is just performing

tensor algebra, we don't need to pass images through the network individually, but instead can pass them through together as a *batch*.

The shape of the filters in the first convolutional layer is $4 \times 4 \times 3$. This is because we have chosen the filter to have height and width of 4 (`kernel_size = (4,4)`) and there are three channels in the preceding layer (red, green, and blue). Therefore, the number of parameters (or weights) in the layer is $(4 \times 4 \times 3 + 1) \times 10 = 490$, where the $+ 1$ is due to the inclusion of a bias term attached to each of the filters. It's worth remembering that the depth of the filters in a layer is *always* the same as the number of channels in the preceding layer.

As before, the output from each filter when applied to each $4 \times 4 \times 3$ section of the input image will be the pixelwise multiplication of the filter weights and the area of the image it is covering. As `strides = 2` and `padding = "same"`, the width and height of the output are both halved to 16, and since there are 10 filters the output of the first layer is a batch of tensors each having shape [16, 16, 10].

In general, the shape of the output from a convolutional layer with `padding="same"` is:

$$\left(\text{None}, \frac{\text{input height}}{\text{stride}}, \frac{\text{input width}}{\text{stride}}, \text{filters} \right)$$

In the second convolutional layer, we choose the filters to be 3×3 and they now have depth 10, to match the number of channels in the previous layer. Since there are 20 filters in this layer, this gives a total number of parameters (weights) of $(3 \times 3 \times 10 + 1) \times 20 = 1,820$. Again, we use `strides = 2` and `padding = "same"`, so the width and height both halve. This gives us an overall output shape of (None, 8, 8, 20).

After applying a series of `Conv2D` layers, we need to flatten the tensor using the Keras `Flatten` layer. This results in a set of $8 \times 8 \times 20 = 1,280$ units that we can connect to a final 10-unit `Dense` layer with softmax activation, which represents the probability of each category in a 10-category classification task.

This example demonstrates how we can chain convolutional layers together to create a convolutional neural network. Before we see how this compares in accuracy to our densely connected neural network, I'm going to introduce two more layer types that can also improve performance: `BatchNormalization` and `Dropout`.

Batch Normalization

One common problem when training a deep neural network is ensuring that the weights of the network remain within a reasonable range of values—if they start to become too large, this is a sign that your network is suffering from what is known as the *exploding gradient* problem. As errors are propagated backward through the

network, the calculation of the gradient in the earlier layers can sometimes grow exponentially large, causing wild fluctuations in the weight values. If your loss function starts to return NaN, chances are that your weights have grown large enough to cause an overflow error.

This doesn't necessarily happen immediately as you start training the network. Sometimes your network can be happily training for hours when suddenly the loss function returns NaN and your network has exploded. This can be incredibly annoying, especially when the network has seemingly been training well for a long time. To prevent this from happening, you need to understand the root cause of the exploding gradient problem.

One of the reasons for scaling input data into a neural network is to ensure a stable start to training over the first few iterations. Since the weights of the network are initially randomized, unscaled input could potentially create huge activation values that immediately lead to exploding gradients. For example, instead of passing pixel values from 0–255 into the input layer, we usually scale these values to between –1 and 1.

Because the input is scaled, it's natural to expect the activations from all future layers to be relatively well scaled as well. Initially, this may be true, but as the network trains and the weights move further away from their random initial values, this assumption can start to break down. This phenomenon is known as *covariate shift*.

Imagine you're carrying a tall pile of books, and you get hit by a gust of wind. You move the books in a direction opposite to the wind to compensate, but in doing so, some of the books shift so that the tower is slightly more unstable than before. Initially, this is OK, but with every gust the pile becomes more and more unstable, until eventually the books have shifted so much that the pile collapses. This is covariate shift.

Relating this to neural networks, each layer is like a book in the pile. To remain stable, when the network updates the weights, each layer implicitly assumes that the distribution of its input from the layer beneath is approximately consistent across iterations. However, since there is nothing to stop any of the activation distributions shifting significantly in a certain direction, this can sometimes lead to runaway weight values and an overall collapse of the network.

Batch normalization is a solution that drastically reduces this problem. The solution is surprisingly simple. A batch normalization layer calculates the mean and standard deviation of each of its input channels across the batch and normalizes by subtracting the mean and dividing by the standard deviation. There are then two learned parameters for each channel, the scale (gamma) and shift (beta). The output is simply the normalized input, scaled by gamma and shifted by beta. [Figure 2-15](#) shows the whole process.

Input:	Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
	Parameters to be learned: γ, β
Output:	$\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$	// mini-batch mean
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$	// mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$	// normalize
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$	// scale and shift

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Figure 2-15. The batch normalization process⁶

We can place batch normalization layers after dense or convolutional layers to normalize the output from those layers. It's a bit like connecting the layers of books with small sets of adjustable springs that ensure there aren't any overall huge shifts in their positions over time.

You might be wondering how this layer works at test time. When it comes to prediction, we may only want to predict a single observation, so there is no *batch* over which to take averages. To get around this problem, during training a batch normalization layer also calculates the moving average of the mean and standard deviation of each channel and stores this value as part of the layer to use at test time.

How many parameters are contained within a batch normalization layer? For every channel in the preceding layer, two weights need to be learned: the scale (gamma) and shift (beta). These are the *trainable* parameters. The moving average and standard deviation also need to be calculated for each channel but since they are derived from the data passing through the layer rather than trained through backpropagation, they are called *nontrainable* parameters. In total, this gives four parameters for each channel in the preceding layer, where two are trainable and two are nontrainable.

In Keras, the `BatchNormalization` layer implements the batch normalization functionality:

```
BatchNormalization(momentum = 0.9)
```

The `momentum` parameter is the weight given to the previous value when calculating the moving average and moving standard deviation.

⁶ Source: Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," 11 February 2015, <https://arxiv.org/abs/1502.03167>.

Dropout Layers

When studying for an exam, it is common practice for students to use past papers and sample questions to improve their knowledge of the subject material. Some students try to memorize the answers to these questions, but then come unstuck in the exam because they haven't truly understood the subject matter. The best students use the practice material to further their general understanding, so that they are still able to answer correctly when faced with new questions that they haven't seen before.

The same principle holds for machine learning. Any successful machine learning algorithm must ensure that it generalizes to unseen data, rather than simply *remembering* the training dataset. If an algorithm performs well on the training dataset, but not the test dataset, we say that it is suffering from *overfitting*. To counteract this problem, we use *regularization* techniques, which ensure that the model is penalized if it starts to overfit.

There are many ways to regularize a machine learning algorithm, but for deep learning, one of the most common is by using *dropout* layers. This idea was introduced by Geoffrey Hinton in 2012 and presented in a 2014 paper by Srivastava et al.⁷

Dropout layers are very simple. During training, each dropout layer chooses a random set of units from the preceding layer and sets their output to zero, as shown in Figure 2-16.

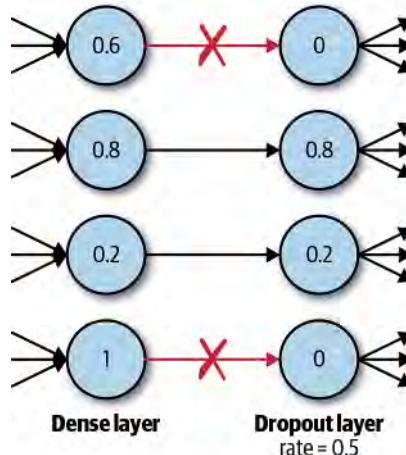


Figure 2-16. A dropout layer

⁷ Nitish Srivastava et al., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research* 15 (2014): 1929–1958, <http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>.

Incredibly, this simple addition drastically reduces overfitting, by ensuring that the network doesn't become overdependent on certain units or groups of units that, in effect, just remember observations from the training set. If we use dropout layers, the network cannot rely too much on any one unit and therefore knowledge is more evenly spread across the whole network. This makes the model much better at generalizing to unseen data, because the network has been trained to produce accurate predictions even under unfamiliar conditions, such as those caused by dropping random units. There are no weights to learn within a dropout layer, as the units to drop are decided stochastically. At test time, the dropout layer doesn't drop any units, so that the full network is used to make predictions.

Returning to our analogy, it's a bit like a math student practicing past papers with a random selection of key formulae missing from their formula book. This way, they learn how to answer questions through an understanding of the core principles, rather than always looking up the formulae in the same places in the book. When it comes to test time, they will find it much easier to answer questions that they have never seen before, due to their ability to generalize beyond the training material.

The `Dropout` layer in Keras implements this functionality, with the `rate` parameter specifying the proportion of units to drop from the preceding layer:

```
Dropout(rate = 0.25)
```

Dropout layers are used most commonly after Dense layers since these are most prone to overfitting due to the higher number of weights, though you can also use them after convolutional layers.



Batch normalization also has been shown to reduce overfitting, and therefore many modern deep learning architectures don't use dropout at all, and rely solely on batch normalization for regularization. As with most deep learning principles, there is no golden rule that applies in every situation—the only way to know for sure what's best is to test different architectures and see which performs best on a holdout set of data.

Putting It All Together

You've now seen three new Keras layer types: `Conv2D`, `BatchNormalization`, and `Dropout`. Let's put these pieces together into a new deep learning architecture and see how it performs on the CIFAR-10 dataset.

You can run the following example in the Jupyter notebook in the book repository called `02_03_deep_learning_conv_neural_network.ipynb`.

The model architecture we shall test is shown here:

```

input_layer = Input((32,32,3))

x = Conv2D(filters = 32, kernel_size = 3
           , strides = 1, padding = 'same')(input_layer)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 32, kernel_size = 3, strides = 2, padding = 'same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 64, kernel_size = 3, strides = 1, padding = 'same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 64, kernel_size = 3, strides = 2, padding = 'same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Flatten()(x)

x = Dense(128)(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)
x = Dropout(rate = 0.5)(x)

x = Dense(NUM_CLASSES)(x)
output_layer = Activation('softmax')(x)

model = Model(input_layer, output_layer)

```

We use four stacked Conv2D layers, each followed by a BatchNormalization and a LeakyReLU layer. After flattening the resulting tensor, we pass the data through a Dense layer of size 128, again followed by a BatchNormalization and a LeakyReLU layer. This is immediately followed by a Dropout layer for regularization, and the network is concluded with an output Dense layer of size 10.



The order in which to use the BatchNormalization and Activation layers is a matter of preference. I like to place the BatchNormalization before the Activation, but some successful architectures use these layers the other way around. If you do choose to use BatchNormalization before Activation then you can remember the order using the acronym *BAD* (BatchNormalization, Activation then Dropout)!

The model summary is shown in Figure 2-17.

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 32, 32, 3)	0
conv2d_3 (Conv2D)	(None, 32, 32, 32)	896
batch_normalization_1 (Batch Normalization)	(None, 32, 32, 32)	128
leaky_re_lu_1 (LeakyReLU)	(None, 32, 32, 32)	0
conv2d_4 (Conv2D)	(None, 16, 16, 32)	9248
batch_normalization_2 (Batch Normalization)	(None, 16, 16, 32)	128
leaky_re_lu_2 (LeakyReLU)	(None, 16, 16, 32)	0
conv2d_5 (Conv2D)	(None, 16, 16, 64)	18496
batch_normalization_3 (Batch Normalization)	(None, 16, 16, 64)	256
leaky_re_lu_3 (LeakyReLU)	(None, 16, 16, 64)	0
conv2d_6 (Conv2D)	(None, 8, 8, 64)	36928
batch_normalization_4 (Batch Normalization)	(None, 8, 8, 64)	256
leaky_re_lu_4 (LeakyReLU)	(None, 8, 8, 64)	0
flatten_2 (Flatten)	(None, 4096)	0
dense_2 (Dense)	(None, 128)	524416
batch_normalization_5 (Batch Normalization)	(None, 128)	512
leaky_re_lu_5 (LeakyReLU)	(None, 128)	0
dropout_1 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1290
activation_1 (Activation)	(None, 10)	0
<hr/>		
Total params:	592,554	
Trainable params:	591,914	
Non-trainable params:	640	

Figure 2-17. Convolutional neural network (CNN) for CIFAR-10



Before moving on, make sure you are able to calculate the output shape and number of parameters for each layer by hand. It's a good exercise to prove to yourself that you have fully understood how each layer is constructed and how it is connected to the preceding layer! Don't forget to include the bias weights that are included as part of the Conv2D and Dense layers.

We compile and train the model in exactly the same way as before and call the evaluate method to determine its accuracy on the holdout set ([Figure 2-18](#)).

```
model.evaluate(x_test, y_test, batch_size=1000)  
10000/10000 [=====] - 15s 1ms/step  
[0.8423407137393951, 0.7155999958515167]
```

Figure 2-18. CNN performance

As you can see, this model is now achieving 71.5% accuracy, up from 49.0% previously. Much better! [Figure 2-19](#) shows some predictions from our new convolutional model.



Figure 2-19. CNN predictions

This improvement has been achieved simply by changing the architecture of the model to include convolutional, batch normalization, and dropout layers. Notice that the number of parameters is actually fewer in our new model than the previous model, even though the number of layers is far greater. This demonstrates the importance of being experimental with your model design and being comfortable with how the different layer types can be used to your advantage. When building generative models, it becomes even more important to understand the inner workings of your model since it is the middle layers of your network that capture the high-level features that you are most interested in.

Summary

This chapter introduced the core deep learning concepts that you will need to start building your first deep generative models.

A really important point to take away from this chapter is that deep neural networks are completely flexible by design, and there really are no fixed rules when it comes to model architecture. There are guidelines and best practices but you should feel free to experiment with layers and the order in which they appear. You will need to bear in mind that, like a set of building blocks, some layers will not fit together, simply because the input shape of one does not conform to the output shape of the other. This knowledge will come with experience and a solid understanding of how each layer changes the tensor shape as data flows through the network.

Another point to remember is that it is the *layers* in a deep neural network that are convolutional, rather than the network itself. When people talk about “convolutional neural networks,” they really mean “neural networks that contain convolutional layers.” It is important to make this distinction, because you shouldn’t feel constrained to only use the architectures that you have read about in this book or elsewhere; instead, you should see them as examples of how you can piece together the different layer types. Like a child with a set of building blocks, the design of your neural network is only limited by your own imagination—and, crucially, your understanding of how the various layers fit together.

In the next chapter, we shall see how we can use these building blocks to design a network that can generate images.

CHAPTER 3

Variational Autoencoders

In 2013, Diederik P. Kingma and Max Welling published a paper that laid the foundations for a type of neural network known as a *variational autoencoder* (VAE).¹ This is now one of the most fundamental and well-known deep learning architectures for generative modeling. In this chapter, we shall start by building a standard autoencoder and then see how we can extend this framework to develop a variational autoencoder—our first example of a generative deep learning model.

Along the way, we will pick apart both types of model, to understand how they work at a granular level. By the end of the chapter you should have a complete understanding of how to build and manipulate autoencoder-based models and, in particular, how to build a variational autoencoder from scratch to generate images based on your own training set.

Let's start by paying a visit to a strange art exhibition...

The Art Exhibition

Two brothers, Mr. N. Coder and Mr. D. Coder, run an art gallery. One weekend, they host an exhibition focused on monochrome studies of single-digit numbers. The exhibition is particularly strange because it contains only one wall and no physical artwork. When a new painting arrives for display, Mr. N. Coder simply chooses a point on the wall to represent the painting, places a marker at this point, then throws the original artwork away. When a customer requests to see the painting, Mr. D. Coder attempts to re-create the artwork using just the coordinates of the relevant marker on the wall.

¹ Diederik P. Kingma and Max Welling, “Auto-Encoding Variational Bayes,” 20 December 2013, <https://arxiv.org/abs/1312.6114>.

The exhibition wall is shown in [Figure 3-1](#), where each black dot is a marker placed by Mr. N. Coder to represent a painting. We also show one of the paintings that has been marked on the wall at the point $[-3.5, -0.5]$ by Mr. N. Coder and then reconstructed using just these two numbers by Mr. D. Coder.

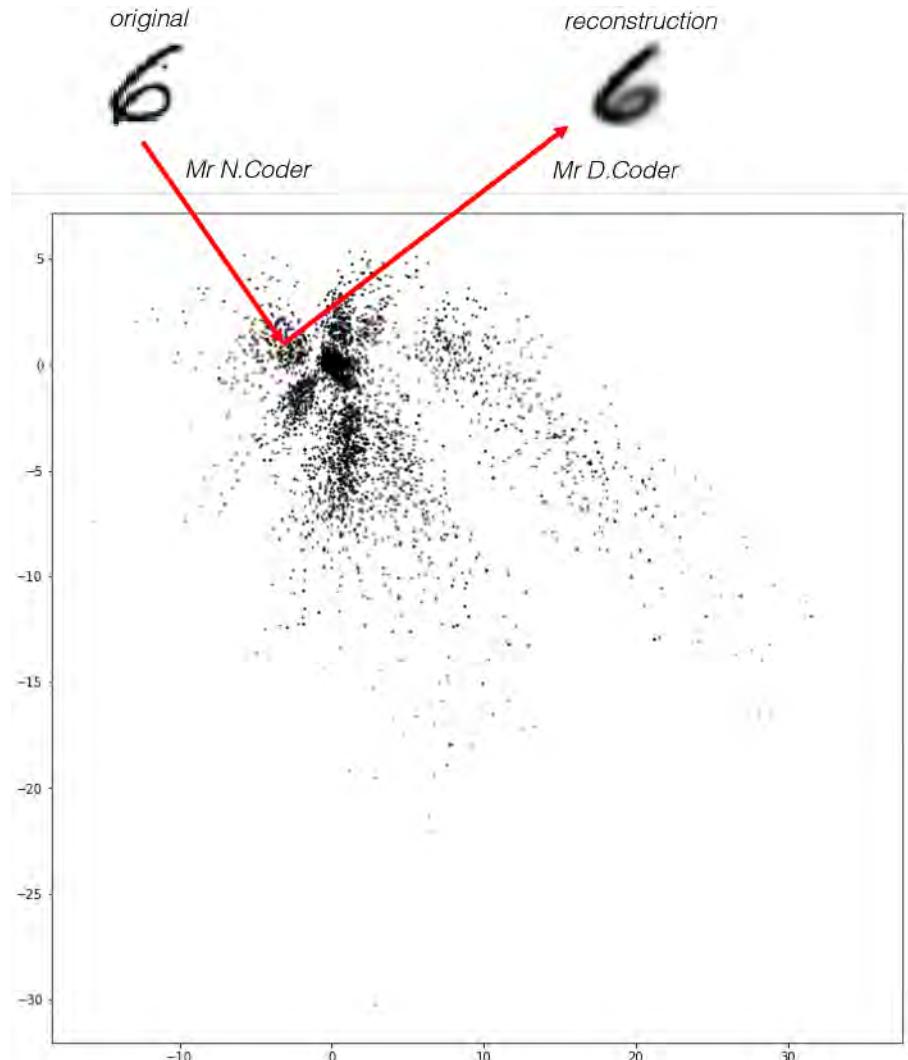


Figure 3-1. The wall at the art exhibition

In [Figure 3-2](#) you can see examples of other original paintings (top row), the coordinates of the point on the wall given by Mr. N. Coder, and the reconstructed paintings produced by Mr. D. Coder (bottom row).



Figure 3-2. More examples of reconstructed paintings

So how does Mr. N. Coder decide where to place the markers? The system evolves naturally through years of training and working together, gradually tweaking the processes for marker placement and reconstruction. The brothers carefully monitor the loss of revenue at the ticket office caused by customers asking for money back because of badly reconstructed paintings, and are constantly trying to find a system that minimizes this loss of earnings. As you can see from [Figure 3-2](#), it works remarkably well—customers who come to view the artwork very rarely complain that Mr. D. Coder's re-created paintings are significantly different from the original pieces they came to see.

One day, Mr. N. Coder has an idea. What if he randomly placed markers on parts of the wall that currently do not have a marker? Mr. D. Coder could then re-create the artwork corresponding to these points, and within a few days they would have their own exhibition of completely original, generated paintings.

The brothers set about their plan and open their new exhibition to the public. Some of the exhibits and corresponding markers are displayed in [Figure 3-3](#).

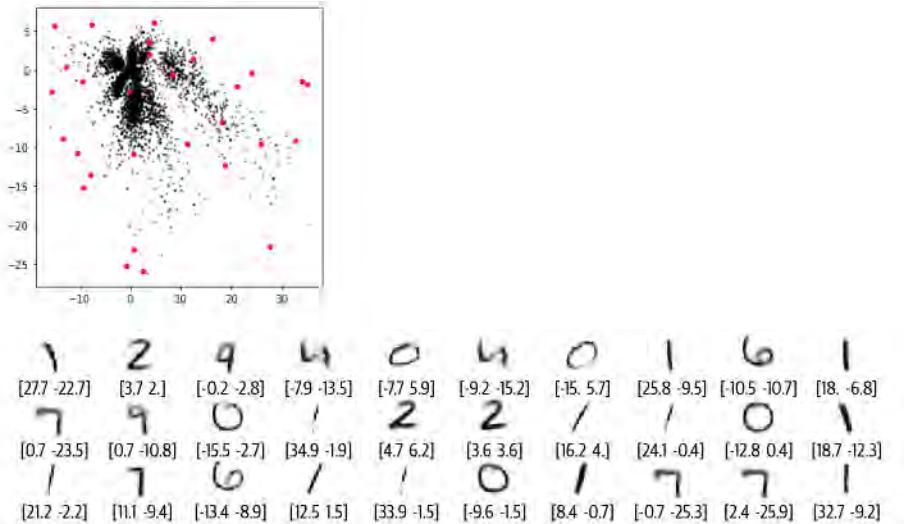


Figure 3-3. The new generative art exhibition

As you can see, the plan was not a great success. The overall variety is poor and some pieces of artwork do not really resemble single-digit numbers.

So, what went wrong and how can the brothers improve their scheme?

Autoencoders

The preceding story is an analogy for an *autoencoder*, which is a neural network made up of two parts:

- An *encoder* network that compresses high-dimensional input data into a lower-dimensional representation vector
- A *decoder* network that decompresses a given representation vector back to the original domain

This process is shown in [Figure 3-4](#).

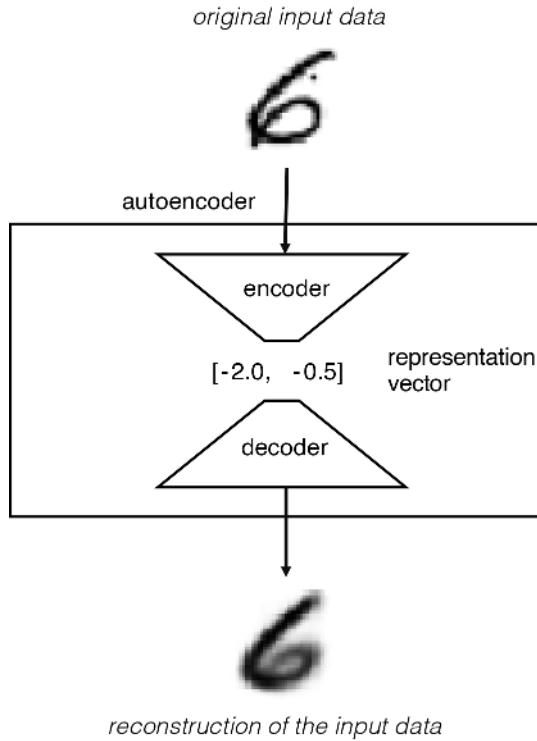


Figure 3-4. Diagram of an autoencoder

The network is trained to find weights for the encoder and decoder that minimize the loss between the original input and the reconstruction of the input after it has passed through the encoder and decoder.

The representation vector is a compression of the original image into a lower-dimensional, latent space. The idea is that by choosing *any* point in the latent space, we should be able to generate novel images by passing this point through the decoder, since the decoder has learned how to convert points in the latent space into viable images.

In our analogy, Mr. N. Coder and Mr. D. Coder are using representation vectors inside a two-dimensional latent space (the wall) to encode each image. This helps us to visualize the latent space, since we can easily plot points in 2D. In practice, autoencoders usually have more than two dimensions in order to have more freedom to capture greater nuance in the images.

Autoencoders can also be used to clean noisy images, since the encoder learns that it is not useful to capture the position of the random noise inside the latent space. For tasks such as this, a 2D latent space is probably too small to encode sufficient relevant

information from the input. However, as we shall see, increasing the dimensionality of the latent space quickly leads to problems if we want to use the autoencoder as a generative model.

Your First Autoencoder

Let's now build an autoencoder in Keras. This example follows the Jupyter notebook `03_01_autoencoder_train.ipynb` in the book repository.

Generally speaking, it is a good idea to create a class for your model in a separate file. This way, you can instantiate an `Autoencoder` object with parameters that define a particular model architecture in the notebook, as shown in [Example 3-1](#). This makes your model very flexible and able to be easily tested and ported to other projects as necessary.

Example 3-1. Defining the autoencoder

```
from models.AE import Autoencoder

AE = Autoencoder(
    input_dim = (28,28,1)
    , encoder_conv_filters = [32,64,64, 64]
    , encoder_conv_kernel_size = [3,3,3,3]
    , encoder_conv_strides = [1,2,2,1]
    , decoder_conv_t_filters = [64,64,32,1]
    , decoder_conv_t_kernel_size = [3,3,3,3]
    , decoder_conv_t_strides = [1,2,2,1]
    , z_dim = 2)
```

Let's now take a look at the architecture of an autoencoder in more detail, starting with the encoder.

The Encoder

In an autoencoder, the encoder's job is to take the input image and map it to a point in the latent space. The architecture of the encoder we will be building is shown in [Figure 3-5](#).

Layer (type)	Output Shape	Param #
encoder_input (InputLayer)	(None, 28, 28, 1)	0
encoder_conv_0 (Conv2D)	(None, 28, 28, 32)	320
leaky_re_lu_1 (LeakyReLU)	(None, 28, 28, 32)	0
encoder_conv_1 (Conv2D)	(None, 14, 14, 64)	18496
leaky_re_lu_2 (LeakyReLU)	(None, 14, 14, 64)	0
encoder_conv_2 (Conv2D)	(None, 7, 7, 64)	36928
leaky_re_lu_3 (LeakyReLU)	(None, 7, 7, 64)	0
encoder_conv_3 (Conv2D)	(None, 7, 7, 64)	36928
leaky_re_lu_4 (LeakyReLU)	(None, 7, 7, 64)	0
flatten_1 (Flatten)	(None, 3136)	0
encoder_output (Dense)	(None, 2)	6274

Total params: 98,946
Trainable params: 98,946
Non-trainable params: 0

Figure 3-5. Architecture of the encoder

To achieve this, we first create an input layer for the image and pass this through four Conv2D layers in sequence, each capturing increasingly high-level features. We use a stride of 2 on some of the layers to reduce the size of the output. The last convolutional layer is then flattened and connected to a Dense layer of size 2, which represents our two-dimensional latent space.

Example 3-2 shows how to build this in Keras.

Example 3-2. The encoder

```
### THE ENCODER
encoder_input = Input(shape=self.input_dim, name='encoder_input') ①

x = encoder_input

for i in range(self.n_layers_encoder):
    conv_layer = Conv2D(
        filters = self.encoder_conv_filters[i]
```

```

        , kernel_size = self.encoder_conv_kernel_size[i]
        , strides = self.encoder_conv_strides[i]
        , padding = 'same'
        , name = 'encoder_conv_' + str(i)
    )

x = conv_layer(x) ❷
x = LeakyReLU()(x)

shape_before_flattening = K.int_shape(x)[1:]
x = Flatten()(x) ❸

encoder_output= Dense(self.z_dim, name='encoder_output')(x) ❹

self.encoder = Model(encoder_input, encoder_output) ❺

```

- ❶ Define the input to the encoder (the image).
- ❷ Stack convolutional layers sequentially on top of each other.
- ❸ Flatten the last convolutional layer to a vector.
- ❹ Dense layer that connects this vector to the 2D latent space.
- ❺ The Keras model that defines the encoder—a model that takes an input image and encodes it into the 2D latent space.

You can change the number of convolutional layers in the encoder simply by adding elements to the lists that define the model architecture in the notebook. I strongly recommend experimenting with the parameters that define the models in this book, to understand how the architecture affects the number of weights in each layer, model performance, and model runtime.

The Decoder

The decoder is a mirror image of the encoder, except instead of convolutional layers, we use *convolutional transpose* layers, as shown in [Figure 3-6](#).

Layer (type)	Output Shape	Param #
decoder_input (InputLayer)	(None, 2)	0
dense_1 (Dense)	(None, 3136)	9408
reshape_1 (Reshape)	(None, 7, 7, 64)	0
decoder_conv_t_0 (Conv2DTran)	(None, 7, 7, 64)	36928
leaky_re_lu_5 (LeakyReLU)	(None, 7, 7, 64)	0
decoder_conv_t_1 (Conv2DTran)	(None, 14, 14, 64)	36928
leaky_re_lu_6 (LeakyReLU)	(None, 14, 14, 64)	0
decoder_conv_t_2 (Conv2DTran)	(None, 28, 28, 32)	18464
leaky_re_lu_7 (LeakyReLU)	(None, 28, 28, 32)	0
decoder_conv_t_3 (Conv2DTran)	(None, 28, 28, 1)	289
activation_1 (Activation)	(None, 28, 28, 1)	0

Total params: 102,017
Trainable params: 102,017
Non-trainable params: 0

Figure 3-6. Architecture of the decoder

Note that the decoder doesn't have to be a mirror image of the encoder. It can be anything you want, as long as the output from the last layer of the decoder is the same size as the input to the encoder (since our loss function will be comparing these pixel-wise).

Convolutional Transpose Layers

Standard convolutional layers allow us to halve the size of an input tensor in both height and width, by setting `strides = 2`.

The convolutional transpose layer uses the same principle as a standard convolutional layer (passing a filter across the image), but is different in that setting `strides = 2` *doubles* the size of the input tensor in both height and width.

In a convolutional transpose layer, the `strides` parameter determines the internal zero padding between pixels in the image as shown in Figure 3-7.

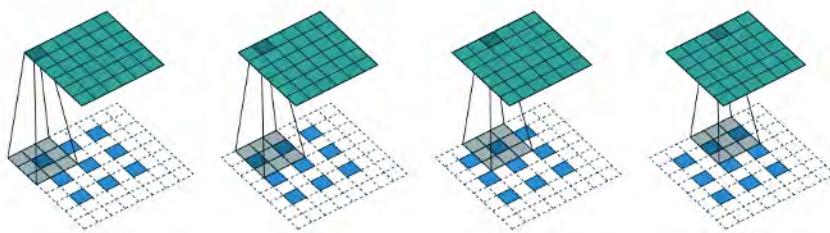


Figure 3-7. A convolutional transpose layer example—here, a $3 \times 3 \times 1$ filter (gray) is being passed across a $3 \times 3 \times 1$ image (blue) with strides = 2, to produce a $6 \times 6 \times 1$ output tensor (green)²

In Keras, the `Conv2DTranspose` layer allows us to perform convolutional transpose operations on tensors. By stacking these layers, we can gradually expand the size of each layer, using strides of 2, until we get back to the original image dimension of 28×28 .

Example 3-3 shows how we build the decoder in Keras.

Example 3-3. The decoder

```
#### THE DECODER
decoder_input = Input(shape=(self.z_dim,), name='decoder_input') ①

x = Dense(np.prod(shape_before_flattening))(decoder_input) ②
x = Reshape(shape_before_flattening)(x) ③

for i in range(self.n_layers_decoder):
    conv_t_layer = Conv2DTranspose(
        filters = self.decoder_conv_t_filters[i]
        , kernel_size = self.decoder_conv_t_kernel_size[i]
        , strides = self.decoder_conv_t_strides[i]
        , padding = 'same'
        , name = 'decoder_conv_t_' + str(i)
    )

    x = conv_t_layer(x) ④

    if i < self.n_layers_decoder - 1:
        x = LeakyReLU()(x)
    else:
```

² Source: Vincent Dumoulin and Francesco Visin, “A Guide to Convolution Arithmetic for Deep Learning,” 12 January 2018, <https://arxiv.org/pdf/1603.07285.pdf>.

```
x = Activation('sigmoid')(x)

decoder_output = x

self.decoder = Model(decoder_input, decoder_output) # 5
```

- ➊ Define the input to the decoder (the point in the latent space).
- ➋ Connect the input to a `Dense` layer.
- ➌ Reshape this vector into a tensor that can be fed as input into the first convolutional transpose layer.
- ➍ Stack convolutional transpose layers on top of each other.
- ➎ The Keras model that defines the decoder—a model that takes a point in the latent space and decodes it into the original image domain.

Joining the Encoder to the Decoder

To train the encoder and decoder simultaneously, we need to define a model that will represent the flow of an image through the encoder and back out through the decoder. Luckily, Keras makes it extremely easy to do this, as you can see in [Example 3-4](#).

Example 3-4. The full autoencoder

```
### THE FULL AUTOENCODER
model_input = encoder_input # 1
model_output = decoder(encoder_output) # 2

self.model = Model(model_input, model_output) # 3
```

- ➊ The input to the autoencoder is the same as the input to the encoder.
- ➋ The output from the autoencoder is the output from the encoder passed through the decoder.
- ➌ The Keras model that defines the full autoencoder—a model that takes an image, and passes it through the encoder and back out through the decoder to generate a reconstruction of the original image.

Now that we've defined our model, we just need to compile it with a loss function and optimizer, as shown in [Example 3-5](#). The loss function is usually chosen to be either the root mean squared error (RMSE) or binary cross-entropy between the individual pixels of the original image and the reconstruction. Binary cross-entropy places

heavier penalties on predictions at the extremes that are badly wrong, so it tends to push pixel predictions to the middle of the range. This results in less vibrant images. For this reason, I generally prefer to use RMSE as the loss function. However, there is no right or wrong choice—you should choose whichever works best for your use case.

Example 3-5. Compilation

```
### COMPILED
optimizer = Adam(lr=learning_rate)

def r_loss(y_true, y_pred):
    return K.mean(K.square(y_true - y_pred), axis = [1,2,3])

self.model.compile(optimizer=optimizer, loss = r_loss)
```

We can now train the autoencoder by passing in the input images as both the input and output, as shown in [Example 3-6](#).

Example 3-6. Training the autoencoder

```
self.model.fit(
    x = x_train
    , y = x_train
    , batch_size = batch_size
    , shuffle = True
    , epochs = 10
    , callbacks = callbacks_list
)
```

Analysis of the Autoencoder

Now that our autoencoder is trained, we can start to investigate how it is representing images in the latent space. We'll then see how variational autoencoders are a natural extension that fixes the issues faced by autoencoders. The relevant code is included in the `03_02_autoencoder_analysis.ipynb` notebook in the book repository.

First, let's take a set of new images that the model hasn't seen, pass them through the encoder, and plot the 2D representations in a scatter plot. In fact, we've already seen this plot: it's just Mr. N. Coder's wall from [Figure 3-1](#). Coloring this plot by digit produces the chart in [Figure 3-8](#). It's worth noting that even though the digit labels were never shown to the model during training, the autoencoder has naturally grouped digits that look alike into the same part of the latent space.

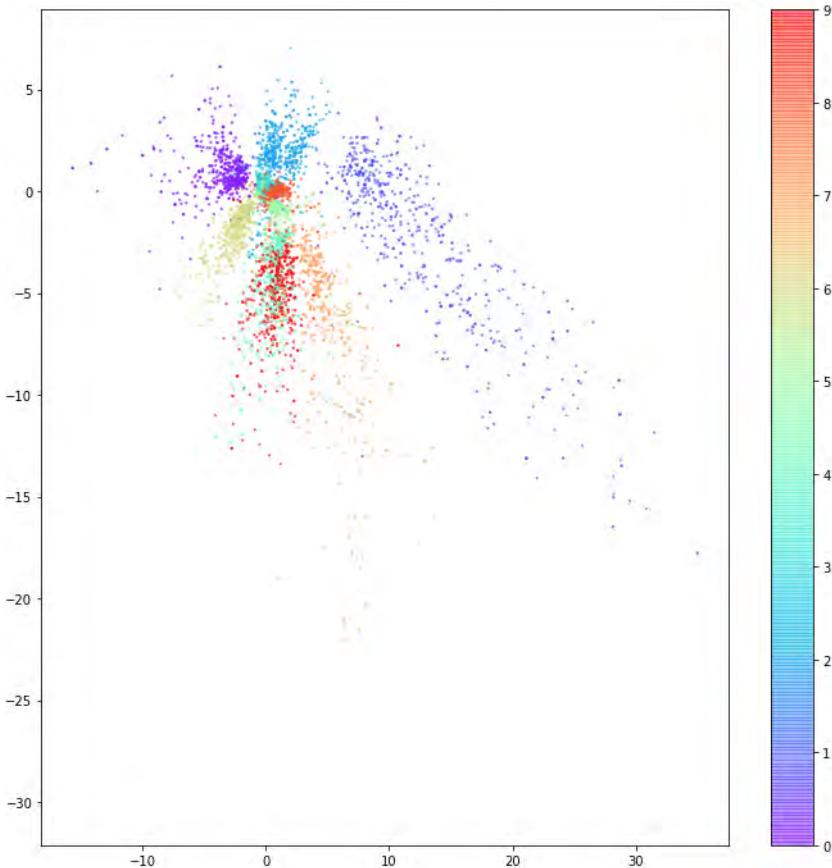


Figure 3-8. Plot of the latent space, colored by digit

There are a few interesting points to note:

1. The plot is not symmetrical about the point (0, 0), or bounded. For example, there are far more points with negative y-axis values than positive, and some points even extend to a y-axis value of < -30.
2. Some digits are represented over a very small area and others over a much larger area.
3. There are large gaps between colors containing few points.

Remember, our goal is to be able to choose a random point in the latent space, pass this through the decoder, and obtain an image of a digit that looks real. If we do this multiple times, we would also ideally like to get a roughly equal mixture of different kinds of digit (i.e., it shouldn't always produce the same digit). This was also the aim

of the Coder brothers when they were choosing random points on their wall to generate new artwork for their exhibition.

Point 1 explains why it's not obvious how we should even go about choosing a *random* point in the latent space, since the distribution of these points is undefined. Technically, we would be justified in choosing any point in the 2D plane! It's not even guaranteed that points will be centered around (0,0). This makes sampling from our latent space extremely problematic.

Point 2 explains the lack of diversity in the generated images. Ideally, we'd like to obtain a roughly equal spread of digits when sampling randomly from our latent space. However, with an autoencoder this is not guaranteed. For example, the area of 1's is far bigger than the area for 8's, so when we pick points randomly in the space, we're more likely to sample something that decodes to look like a 1 than an 8.

Point 3 explains why some generated images are poorly formed. In [Figure 3-9](#) we can see three points in the latent space and their decoded images, none of which are particularly well formed.

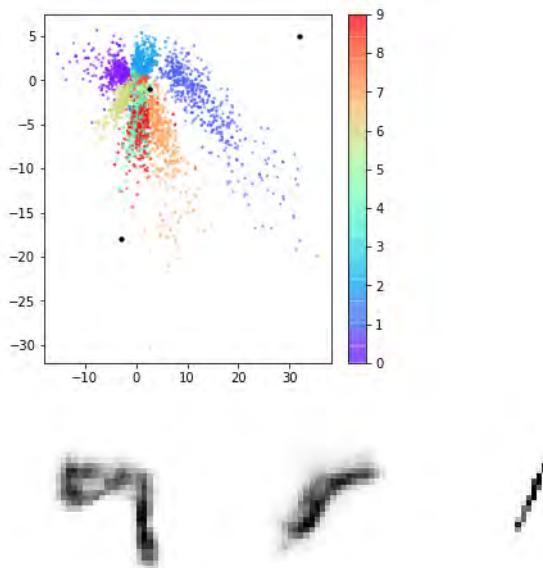


Figure 3-9. Some poorly generated images

Partly, this is because of the large spaces at the edge of the domain where there are few points—the autoencoder has no reason to ensure that points here are decoded to legible digits as very few images are encoded here. However, more worryingly, even points that are right in the middle of the domain may not be decoded into well-formed images. This is because the autoencoder is not forced to ensure that the space

is *continuous*. For example, even though the point $(2, -2)$ might be decoded to give a satisfactory image of a 4, there is no mechanism in place to ensure that the point $(2.1, -2.1)$ also produces a satisfactory 4.

In 2D this issue is subtle; the autoencoder only has a small number of dimensions to work with, so naturally it has to squash digit groups together, resulting in the space between digit groups being relatively small. However, as we start to use more dimensions in the latent space to generate more complex images, such as faces, this problem becomes even more apparent. If we give the autoencoder free rein in how it uses the latent space to encode images, there will be huge gaps between groups of similar points with no incentive for the space between to generate well-formed images.

So how can we solve these three problems, so that our autoencoder framework is ready to be used as a generative model? To explain, let's revisit the Coder brothers' art exhibition, where a few changes have taken place since our last visit...

The Variational Art Exhibition

Determined to make the generative art exhibition work, Mr. N. Coder recruits the help of his daughter, Epsilon. After a brief discussion, they decide to change the way that new paintings are marked on the wall. The new process works as follows.

When a new painting arrives at the exhibition, Mr. N. Coder chooses a point on the wall where he would like to place the marker to represent the artwork, as before. However, now, instead of placing the marker on the wall himself, he passes his opinion of where it should go to Epsilon, who decides where the marker will be placed. She of course takes her father's opinion into account, so she usually places the marker somewhere near the point that he suggests. Mr. D. Coder then finds the marker where Epsilon placed it and never hears Mr. N. Coder's original opinion.

Mr. N. Coder also provides his daughter with an indication of how sure he is that the marker should be placed at the given point. The more certain he is, the closer Epsilon will generally place the point to his suggestion.

There is one final change to the old system. Before, the only feedback mechanism was the loss of earnings at the ticket office resulting from poorly reconstructed images. If the brothers saw that particular paintings weren't being re-created accurately, they would adjust their understanding of marker placement and image regeneration to ensure revenue loss was minimized.

Now, there is another source of feedback. Epsilon is quite lazy and gets annoyed whenever her father tells her to place markers far away from the center of the wall, where the ladder rests. She also doesn't like it when he is too strict about where the markers should be placed, as then she feels she doesn't have enough responsibility. Equally, if her father professes little confidence in where the markers should go, she

feels like she's the one doing all the work! His confidence in the marker placements that he provides has to be just right for her to be happy.

To compensate for her annoyance, her father pays her more to do the job whenever he doesn't stick to these rules. On the balance sheet, this expense is listed as his kitty-loss (KL) divulgence. He therefore needs to be careful that he doesn't end up paying his daughter too much while also monitoring the loss of revenue at the ticket office. After training with these simple changes, Mr. N. Coder once again tries his strategy of placing markers on portions of the wall that are empty, so that Mr. D. Coder can regenerate these points as original artwork.

Some of these points are shown in [Figure 3-10](#), along with the generated images.

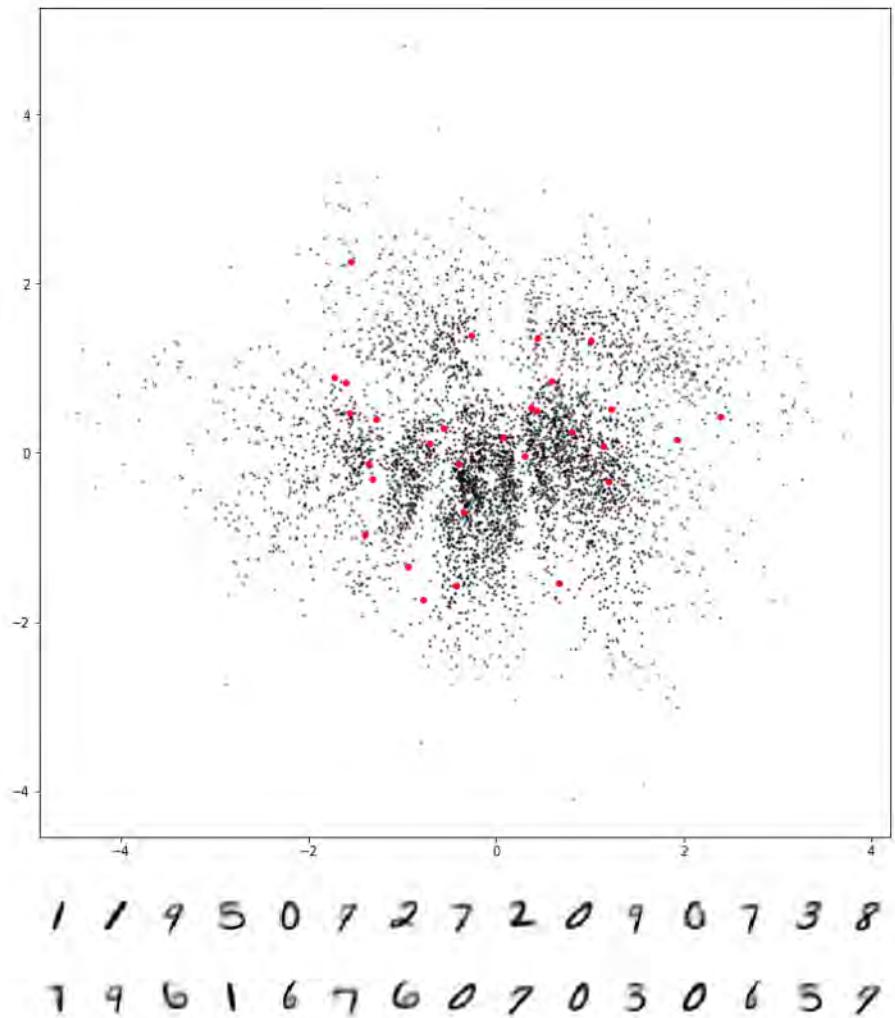


Figure 3-10. Artwork from the new exhibition

Much better! The crowds arrive in great waves to see this new, exciting generative art and are amazed by the originality and diversity of the paintings.

Building a Variational Autoencoder

The previous story showed how, with a few simple changes, the art exhibition could be transformed into a successful generative process. Let's now try to understand mathematically what we need to do to our autoencoder to convert it into a variational autoencoder and thus make it a truly generative model.

There are actually only two parts that we need to change: the encoder and the loss function.

The Encoder

In an autoencoder, each image is mapped directly to one point in the latent space. In a variational autoencoder, each image is instead mapped to a multivariate normal distribution around a point in the latent space, as shown in [Figure 3-11](#).

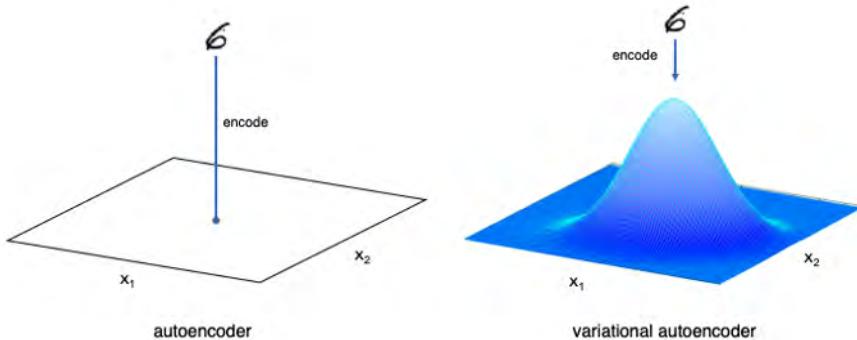


Figure 3-11. The difference between the encoder in an autoencoder and a variational autoencoder

The Normal Distribution

A normal distribution is a probability distribution characterized by a distinctive *bell curve* shape. In one dimension, it is defined by two variables: the *mean* (μ) and the *variance* (σ^2). The *standard deviation* (σ) is the square root of the variance.

The probability density function of the normal distribution in one dimension is:

$$f(x | \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Figure 3-12 shows several normal distributions in one dimension, for different values of the mean and variance. The red curve is the *standard normal*—the normal distribution with mean equal to 0 and variance equal to 1.

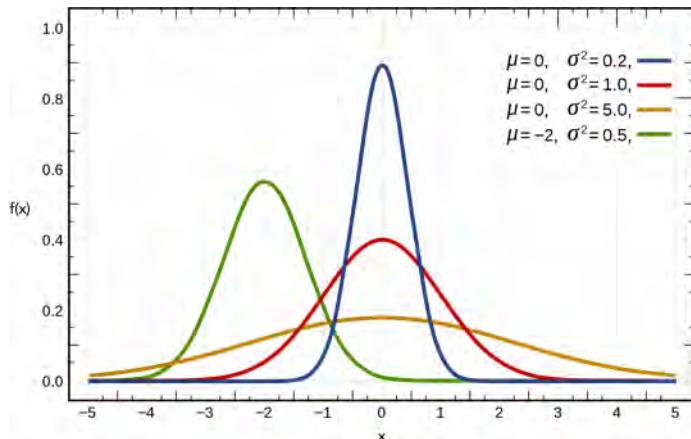


Figure 3-12. The normal distribution in one dimension³

We can sample a point z from a normal distribution with mean μ and standard deviation σ using the following equation:

$$z = \mu + \sigma e$$

where e is sampled from a standard normal distribution.

³ Source: Wikipedia, <http://bit.ly/2ZDWRJv>.

The concept of a normal distribution extends to more than one dimension—the probability density function for a general multivariate normal distribution in k dimensions is as follows:

$$f(x_1, \dots, x_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}}$$

In 2D, the mean vector $\boldsymbol{\mu}$ and the symmetric covariance matrix $\boldsymbol{\Sigma}$ are defined as:

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix}$$

where ρ is the correlation between the two dimensions x_1 and x_2 .

Variational autoencoders assume that there is no correlation between any of the dimensions in the latent space and therefore that the covariance matrix is diagonal. This means the encoder only needs to map each input to a mean vector and a variance vector and does not need to worry about covariance between dimensions. We also choose to map to the *logarithm* of the variance, as this can take any real number in the range $(-\infty, \infty)$, matching the natural output range from a neural network unit, whereas variance values are always positive.

To summarize, the encoder will take each input image and encode it to two vectors, `mu` and `log_var` which together define a multivariate normal distribution in the latent space:

`mu`

The mean point of the distribution.

`log_var`

The logarithm of the variance of each dimension.

To encode an image into a specific point z in the latent space, we can sample from this distribution, using the following equation:

`z = mu + sigma * epsilon`

where⁴

`sigma = exp(log_var / 2)`

⁴ $\sigma = \exp(\log(\sigma)) = \exp(2 \log(\sigma)/2) = \exp(\log(\sigma^2)/2)$

`epsilon` is a point sampled from the standard normal distribution.

Relating this back to our story, `mu` represents Mr. N. Coder's opinion of where the marker should appear on the wall. `epsilon` is his daughter's random choice of how far away from `mu` the marker should be placed, scaled by `sigma`, Mr. N. Coder's confidence in the marker's position.

So why does this small change to the encoder help?

Previously, we saw how there was no requirement for the latent space to be continuous—even if the point $(-2, 2)$ decodes to a well-formed image of a 4, there was no requirement for $(-2.1, 2.1)$ to look similar. Now, since we are sampling a random point from an area around `mu`, the decoder must ensure that all points in the same neighborhood produce very similar images when decoded, so that the reconstruction loss remains small. This is a very nice property that ensures that even when we choose a point in the latent space that has never been seen by the decoder, it is likely to decode to an image that is well formed.

Let's now see how we build this new version of the encoder in Keras ([Example 3-7](#)). You can train your own variational autoencoder on the digits dataset by running the notebook `03_03_vae_digits_train.ipynb` in the book repository.

Example 3-7. The variational autoencoder's encoder

```
### THE ENCODER
encoder_input = Input(shape=self.input_dim, name='encoder_input')

x = encoder_input

for i in range(self.n_layers_encoder):
    conv_layer = Conv2D(
        filters = self.encoder_conv_filters[i]
        , kernel_size = self.encoder_conv_kernel_size[i]
        , strides = self.encoder_conv_strides[i]
        , padding = 'same'
        , name = 'encoder_conv_' + str(i)
    )

    x = conv_layer(x)

    if self.use_batch_norm:
        x = BatchNormalization()(x)

    x = LeakyReLU()(x)
    if self.use_dropout:
        x = Dropout(rate = 0.25)(x)

shape_before_flattening = K.int_shape(x)[1:]
x = Flatten()(x)
```

```

self.mu = Dense(self.z_dim, name='mu')(x) ❶
self.log_var = Dense(self.z_dim, name='log_var')(x) #

encoder_mu_log_var = Model(encoder_input, (self.mu, self.log_var)) ❷

def sampling(args):
    mu, log_var = args
    epsilon = K.random_normal(shape=K.shape(mu), mean=0., stddev=1.)
    return mu + K.exp(log_var / 2) * epsilon

encoder_output = Lambda(sampling, name='encoder_output')([self.mu, self.log_var]) ❸

encoder = Model(encoder_input, encoder_output) ❹

```

- ❶ Instead of connecting the flattened layer directly to the 2D latent space, we connect it to layers `mu` and `log_var`.
- ❷ The Keras model that outputs the values of `mu` and `log_var` for a given input image.
- ❸ This `Lambda` layer samples a point z in the latent space from the normal distribution defined by the parameters `mu` and `log_var`.
- ❹ The Keras model that defines the encoder—a model that takes an input image and encodes it into the 2D latent space, by sampling a point from the normal distribution defined by `mu` and `log_var`.

Lambda layer

A `Lambda` layer simple wraps any function into Keras layer. For example, the following layer squares its input:

```
Lambda(lambda x: x ** 2)
```

They are useful when you want to apply a function to a tensor that isn't already included as one of the out-of-the-box Keras layer types.

A diagram of the encoder is shown in [Figure 3-13](#).

As mentioned previously, the decoder of a variational autoencoder is identical to the decoder of a plain autoencoder. The only other part we need to change is the loss function.

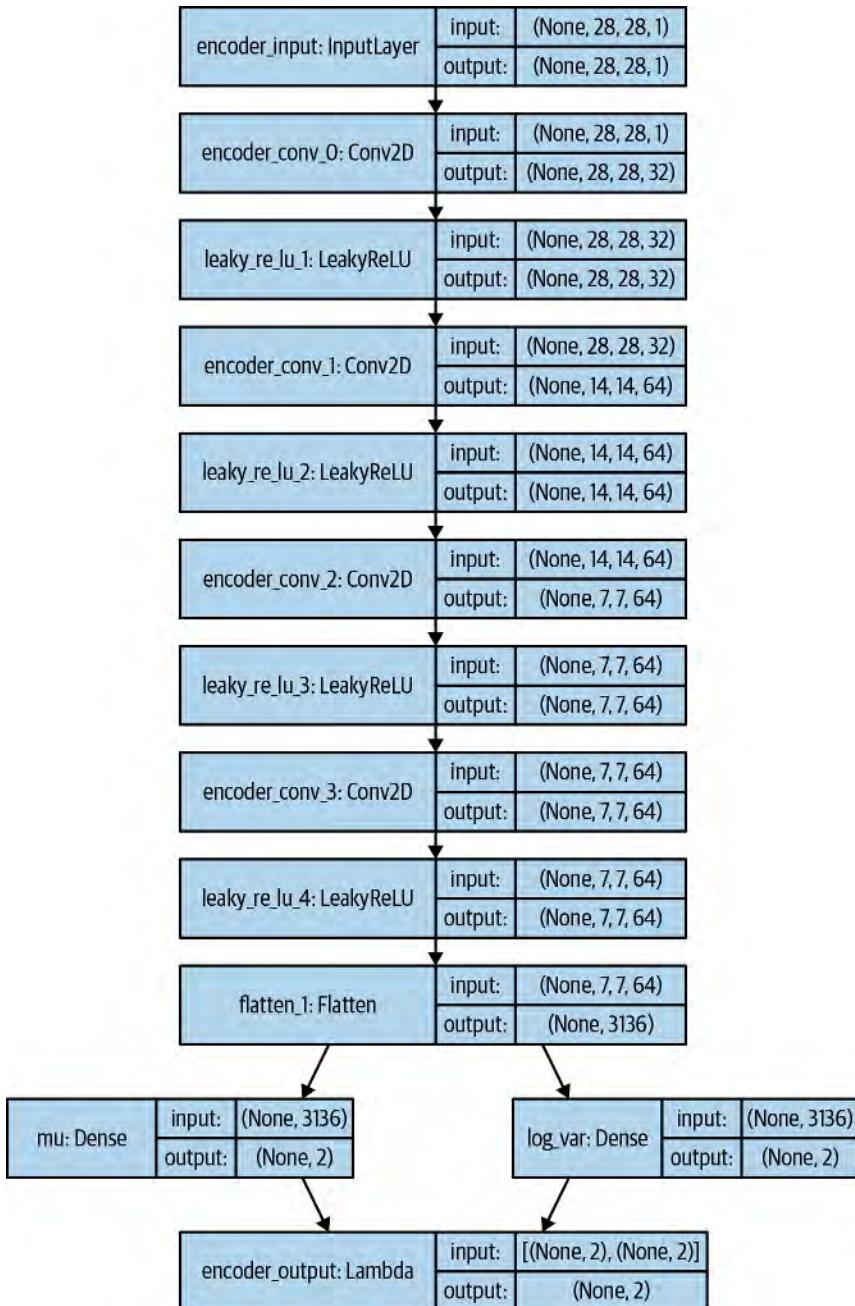


Figure 3-13. Diagram of the VAE encoder

The Loss Function

Previously, our loss function only consisted of the RMSE loss between images and their reconstruction after being passed through the encoder and decoder. This *reconstruction loss* also appears in a variational autoencoder, but we require one extra component: the *Kullback–Leibler (KL) divergence*.

KL divergence is a way of measuring how much one probability distribution differs from another. In a VAE, we want to measure how different our normal distribution with parameters `mu` and `log_var` is from the standard normal distribution. In this special case, the KL divergence has the closed form:

```
kl_loss = -0.5 * sum(1 + log_var - mu ^ 2 - exp(log_var))
```

or in mathematical notation:

$$D_{KL}[N(\mu, \sigma \parallel N(0, 1)] = \frac{1}{2} \sum (1 + \log(\sigma^2) - \mu^2 - \sigma^2)$$

The sum is taken over all the dimensions in the latent space. `kl_loss` is minimized to 0 when `mu` = 0 and `log_var` = 0 for all dimensions. As these two terms start to differ from 0, `kl_loss` increases.

In summary, the KL divergence term penalizes the network for encoding observations to `mu` and `log_var` variables that differ significantly from the parameters of a standard normal distribution, namely `mu` = 0 and `log_var` = 0.

Again, relating this back to our story, this term represents Epsilon's annoyance at having to move the ladder away from the middle of the wall (`mu` different from 0) and also if Mr. N. Coder's confidence in the marker position isn't just right (`log_var` different from 0), both of which incur a cost.

Why does this addition to the loss function help?

First, we now have a well-defined distribution that we can use for choosing points in the latent space—the standard normal distribution. If we sample from this distribution, we know that we're very likely to get a point that lies within the limits of what the VAE is used to seeing. Secondly, since this term tries to force all encoded distributions toward the standard normal distribution, there is less chance that large gaps will form between point clusters. Instead, the encoder will try to use the space around the origin symmetrically and efficiently.

In the code, the loss function for a VAE is simply the addition of the reconstruction loss and the KL divergence loss term. We weight the reconstruction loss with a term, `r_loss_factor`, that ensures that it is well balanced with the KL divergence loss. If we weight the reconstruction loss too heavily, the KL loss will not have the desired

regulatory effect and we will see the same problems that we experienced with the plain autoencoder. If the weighting term is too small, the KL divergence loss will dominate and the reconstructed images will be poor. This weighting term is one of the parameters to tune when you're training your VAE.

Example 3-8 shows how we include the KL divergence term in our loss function.

Example 3-8. Including KL divergence in the loss function

```
### COMPIILATION
optimizer = Adam(lr=learning_rate)

def vae_r_loss(y_true, y_pred):
    r_loss = K.mean(K.square(y_true - y_pred), axis = [1,2,3])
    return r_loss_factor * r_loss

def vae_kl_loss(y_true, y_pred):
    kl_loss = -0.5 * K.sum(1 + self.log_var - K.square(self.mu)
                           - K.exp(self.log_var), axis = 1)
    return kl_loss

def vae_loss(y_true, y_pred):
    r_loss = vae_r_loss(y_true, y_pred)
    kl_loss = vae_kl_loss(y_true, y_pred)
    return r_loss + kl_loss

optimizer = Adam(lr=learning_rate)
self.model.compile(optimizer=optimizer, loss = vae_loss
                   , metrics = [vae_r_loss, vae_kl_loss])
```

Analysis of the Variational Autoencoder

All of the following analysis is available in the book repository, in the notebook *03_04_vae_digits_analysis.ipynb*.

Referring back to Figure 3-10, we can see several changes in how the latent space is organized. The black dots show the *mu* values of each encoded image. The KL divergence loss term ensures that the *mu* and *sigma* values never stray too far from a standard normal. We can therefore sample from the standard normal distribution to generate new points in the space to be decoded (the red dots).

Secondly, there are not so many generated digits that are badly formed, since the latent space is now locally continuous due to fact that the encoder is now stochastic, rather than deterministic.

Finally, by coloring points in the latent space by digit (Figure 3-14), we can see that there is no preferential treatment of any one type. The righthand plot shows the space transformed into *p*-values, and we can see that each color is approximately equally represented. Again, it's important to remember that the labels were not used at all

during training—the VAE has learned the various forms of digits by itself in order to help minimize reconstruction loss.

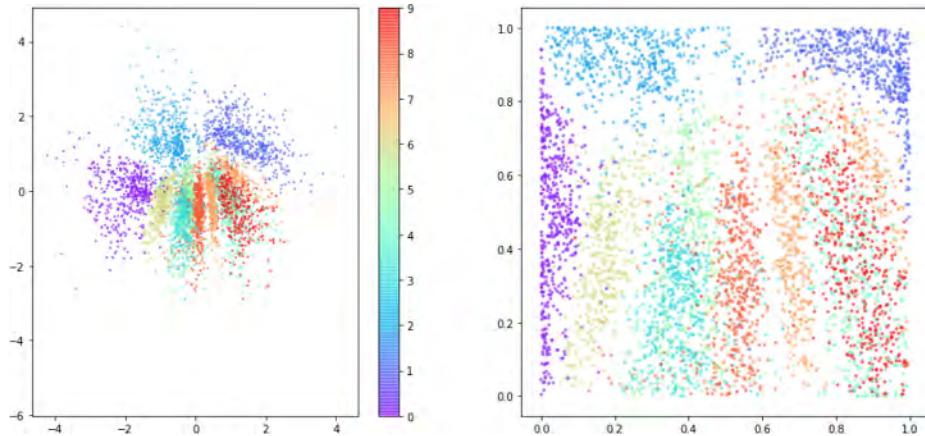


Figure 3-14. The latent space of the VAE colored by digit

So far, all of our work on autoencoders and variational autoencoders has been limited to a latent space with two dimensions. This has helped us to visualize the inner workings of a VAE on the page and understand why the small tweaks that we made to the architecture of the autoencoder helped transform it into a more powerful class of network that can be used for generative modeling.

Let's now turn our attention to a more complex dataset and see the amazing things that variational autoencoders can achieve when we increase the dimensionality of the latent space.

Using VAEs to Generate Faces

We shall be using the [CelebFaces Attributes \(CelebA\) dataset](#) to train our next variational autoencoder. This is a collection of over 200,000 color images of celebrity faces, each annotated with various labels (e.g., *wearing hat*, *smiling*, etc.). A few examples are shown in Figure 3-15.



Figure 3-15. Some examples from the CelebA dataset⁵

Of course, we don't need the labels to train the VAE, but these will be useful later when we start exploring how these features are captured in the multidimensional latent space. Once our VAE is trained, we can sample from the latent space to generate new examples of celebrity faces.

Training the VAE

The network architecture for the faces model is similar to the digits example, with a few slight differences:

1. Our data now has three input channels (RGB) instead of one (grayscale). This means we need to change the number of channels in the final convolutional transpose layer of the decoder to 3.
2. We shall be using a latent space with two hundred dimensions instead of two. Since faces are much more complex than digits, we increase the dimensionality of the latent space so that the network can encode a satisfactory amount of detail from the images.

⁵ Source: Liu et al., 2015, <http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>.

3. There are batch normalization layers after each convolution layer to speed up training. Even though each batch takes a longer time to run, the number of batches required to reach the same loss is greatly reduced. Dropout layers are also used.
4. We increase the reconstruction loss factor to ten thousand. This is a parameter that requires tuning; for this dataset and architecture this value was found to generate good results.
5. We use a *generator* to feed images to the VAE from a folder, rather than loading all the images into memory up front. Since the VAE trains in batches, there is no need to load all the images into memory first, so instead we use the built-in `fit_generator` method that Keras provides to read in images only when they are required for training.

The full architectures of the encoder and decoder are shown in Figures 3-16 and 3-17

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
encoder_input (InputLayer)	(None, 128, 128, 3)	0	
encoder_conv_0 (Conv2D)	(None, 64, 64, 32)	896	encoder_input[0][0]
batch_normalization_1 (BatchNor	(None, 64, 64, 32)	128	encoder_conv_0[0][0]
leaky_re_lu_1 (LeakyReLU)	(None, 64, 64, 32)	0	batch_normalization_1[0][0]
dropout_1 (Dropout)	(None, 64, 64, 32)	0	leaky_re_lu_1[0][0]
encoder_conv_1 (Conv2D)	(None, 32, 32, 64)	18496	dropout_1[0][0]
batch_normalization_2 (BatchNor	(None, 32, 32, 64)	256	encoder_conv_1[0][0]
leaky_re_lu_2 (LeakyReLU)	(None, 32, 32, 64)	0	batch_normalization_2[0][0]
dropout_2 (Dropout)	(None, 32, 32, 64)	0	leaky_re_lu_2[0][0]
encoder_conv_2 (Conv2D)	(None, 16, 16, 64)	36928	dropout_2[0][0]
batch_normalization_3 (BatchNor	(None, 16, 16, 64)	256	encoder_conv_2[0][0]
leaky_re_lu_3 (LeakyReLU)	(None, 16, 16, 64)	0	batch_normalization_3[0][0]
dropout_3 (Dropout)	(None, 16, 16, 64)	0	leaky_re_lu_3[0][0]
encoder_conv_3 (Conv2D)	(None, 8, 8, 64)	36928	dropout_3[0][0]
batch_normalization_4 (BatchNor	(None, 8, 8, 64)	256	encoder_conv_3[0][0]
leaky_re_lu_4 (LeakyReLU)	(None, 8, 8, 64)	0	batch_normalization_4[0][0]
dropout_4 (Dropout)	(None, 8, 8, 64)	0	leaky_re_lu_4[0][0]
flatten_1 (Flatten)	(None, 4096)	0	dropout_4[0][0]
mu (Dense)	(None, 200)	819400	flatten_1[0][0]
log_var (Dense)	(None, 200)	819400	flatten_1[0][0]
encoder_output (Lambda)	(None, 200)	0	mu[0][0] log_var[0][0]
<hr/>			
Total params:	1,732,944		
Trainable params:	1,732,496		
Non-trainable params:	448		

Figure 3-16. The VAE encoder for the CelebA dataset

Layer (type)	Output Shape	Param #
decoder_input (InputLayer)	(None, 200)	0
dense_1 (Dense)	(None, 4096)	823296
reshape_1 (Reshape)	(None, 8, 8, 64)	0
decoder_conv_t_0 (Conv2DTran	(None, 16, 16, 64)	36928
batch_normalization_5 (Batch	(None, 16, 16, 64)	256
leaky_re_lu_5 (LeakyReLU)	(None, 16, 16, 64)	0
dropout_5 (Dropout)	(None, 16, 16, 64)	0
decoder_conv_t_1 (Conv2DTran	(None, 32, 32, 64)	36928
batch_normalization_6 (Batch	(None, 32, 32, 64)	256
leaky_re_lu_6 (LeakyReLU)	(None, 32, 32, 64)	0
dropout_6 (Dropout)	(None, 32, 32, 64)	0
decoder_conv_t_2 (Conv2DTran	(None, 64, 64, 32)	18464
batch_normalization_7 (Batch	(None, 64, 64, 32)	128
leaky_re_lu_7 (LeakyReLU)	(None, 64, 64, 32)	0
dropout_7 (Dropout)	(None, 64, 64, 32)	0
decoder_conv_t_3 (Conv2DTran	(None, 128, 128, 3)	867
activation_1 (Activation)	(None, 128, 128, 3)	0
<hr/>		
Total params:	917,123	
Trainable params:	916,803	
Non-trainable params:	320	

Figure 3-17. The VAE decoder for the CelebA dataset

To train the VAE on the CelebA dataset, run the Jupyter notebook `03_05_vae_faces_train.ipynb` from the book repository. After around five epochs of training your VAE should be able to produce novel images of celebrity faces!

Analysis of the VAE

You can replicate the analysis that follows by running the notebook *03_06_vae_faces_analysis.ipynb*, once you have trained the VAE. Many of the ideas in this section were inspired by a 2016 paper by Xianxu Hou et al.⁶

First, let's take a look at a sample of reconstructed faces. The top row in [Figure 3-18](#) shows the original images and the bottom row shows the reconstructions once they have passed through the encoder and decoder.



Figure 3-18. Reconstructed faces, after passing through the encoder and decoder

We can see that the VAE has successfully captured the key features of each face—the angle of the head, the hairstyle, the expression, etc. Some of the fine detail is missing, but it is important to remember that the aim of building variational autoencoders isn't to achieve perfect reconstruction loss. Our end goal is to sample from the latent space in order to generate new faces.

For this to be possible we must check that the distribution of points in the latent space approximately resembles a multivariate standard normal distribution. Since we cannot view all dimensions simultaneously, we can instead check the distribution of each latent dimension individually. If we see any dimensions that are significantly different from a standard normal distribution, we should probably reduce the reconstruction loss factor, since the KL divergence term isn't having enough effect.

The first 50 dimensions in our latent space are shown in [Figure 3-19](#). There aren't any distributions that stand out as being significantly different from the standard normal, so we can move on to generating some faces!

⁶ Xianxu Hou et al., “Deep Feature Consistent Variational Autoencoder,” 2 October 2016, <https://arxiv.org/abs/1610.00291>.

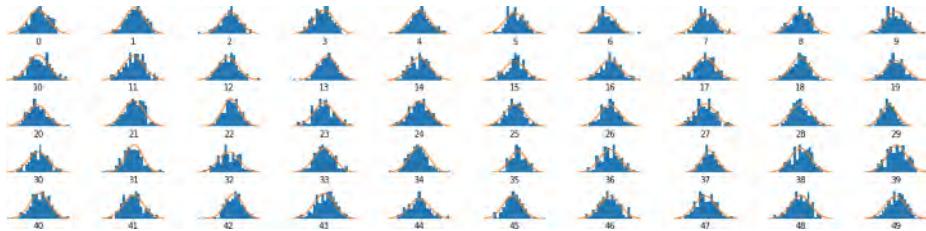


Figure 3-19. Distributions of points for the first 50 dimensions in the latent space

Generating New Faces

To generate new faces, we can use the code in [Example 3-9](#).

Example 3-9. Generating new faces from the latent space

```
n_to_show = 30

znew = np.random.normal(size = (n_to_show,VAE.z_dim)) ①

reconst = VAE.decoder.predict(np.array(znew)) ②

fig = plt.figure(figsize=(18, 5))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(n_to_show):
    ax = fig.add_subplot(3, 10, i+1)
    ax.imshow(reconst[i, :, :, :]) ③
    ax.axis('off')

plt.show()
```

- ① We sample 30 points from a standard normal distribution with 200 dimensions...
- ② ...then pass these points to the decoder.
- ③ The resulting output is a $128 \times 128 \times 3$ image that we can view.

The output is shown in [Figure 3-20](#).

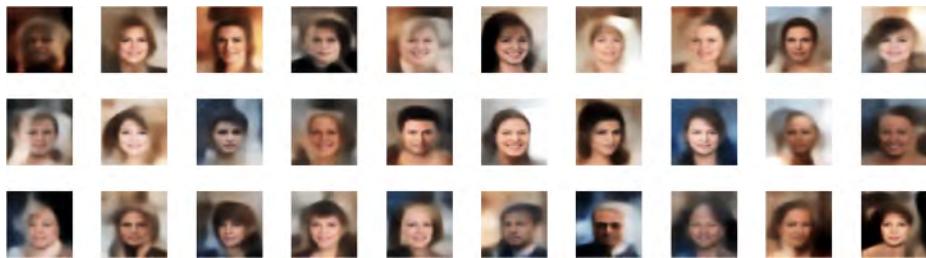


Figure 3-20. New generated faces

Amazingly, the VAE is able to take the set of points that we sampled and convert each into a convincing image of a person's face. While the images are not perfect, they are a giant leap forward from the Naive Bayes model that we started exploring in [Chapter 1](#). The Naive Bayes model faced the problem of not being able to capture dependency between adjacent pixels, since it had no notion of higher-level features such as *sunglasses* or *brown hair*. The VAE doesn't suffer from this problem, since the convolutional layers of the encoder are designed to translate low-level pixels into high-level features and the decoder is trained to perform the opposite task of translating the high-level features in the latent space back to raw pixels.

Latent Space Arithmetic

One benefit of mapping images into a lower-dimensional space is that we can perform arithmetic on vectors in this latent space that has a visual analogue when decoded back into the original image domain.

For example, suppose we want to take an image of somebody who looks sad and give them a smile. To do this we first need to find a vector in the latent space that points in the direction of increasing smile. Adding this vector to the encoding of the original image in the latent space will give us a new point which, when decoded, should give us a more smiley version of the original image.

So how can we find the *smile* vector? Each image in the CelebA dataset is labeled with attributes, one of which is *smiling*. If we take the average position of encoded images in the latent space with the attribute *smiling* and subtract the average position of encoded images that do not have the attribute *smiling*, we will obtain the vector that points from *not smiling* to *smiling*, which is exactly what we need.

Conceptually, we are performing the following vector arithmetic in the latent space, where *alpha* is a factor that determines how much of the feature vector is added or subtracted:

```
z_new = z + alpha * feature_vector
```

Let's see this in action. [Figure 3-21](#) shows several images that have been encoded into the latent space. We then add or subtract multiples of a certain vector (e.g., smile, blonde, male, eyeglasses) to obtain different versions of the image, with only the relevant feature changed.

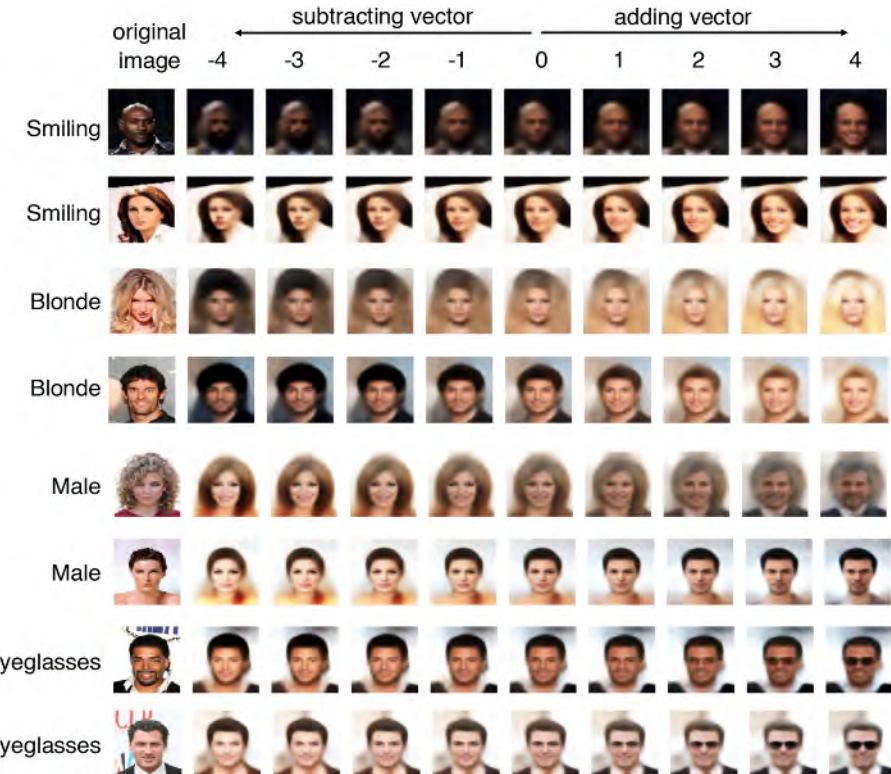


Figure 3-21. Adding and subtracting features to and from faces

It is quite remarkable that even though we are moving the point a significantly large distance in the latent space, the core image barely changes, except for the one feature that we want to manipulate. This demonstrates the power of variational autoencoders for capturing and adjusting high-level features in images.

Morphing Between Faces

We can use a similar idea to morph between two faces. Imagine two points in the latent space, A and B, that represent two images. If you started at point A and walked toward point B in a straight line, decoding each point on the line as you went, you would see a gradual transition from the starting face to the end face.

Mathematically, we are traversing a straight line, which can be described by the following equation:

$$z_{\text{new}} = z_A * (1 - \alpha) + z_B * \alpha$$

Here, α is a number between 0 and 1 that determines how far along the line we are, away from point A.

Figure 3-22 shows this process in action. We take two images, encode them into the latent space, and then decode points along the straight line between them at regular intervals.

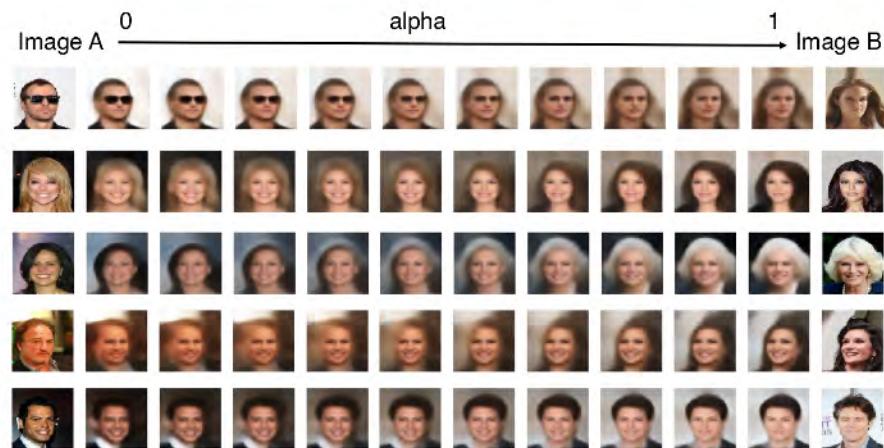


Figure 3-22. Morphing between two faces

It is worth noting the smoothness of the transition—even where there are multiple features to change simultaneously (e.g., removal of glasses, hair color, gender), the VAE manages to achieve this fluidly, showing that the latent space of the VAE is truly a continuous space that can be traversed and explored to generate a multitude of different human faces.

Summary

In this chapter we have seen how variational autoencoders are a powerful tool in the generative modeling toolbox. We started by exploring how plain autoencoders can be used to map high-dimensional images into a low-dimensional latent space, so that high-level features can be extracted from the individually uninformative pixels. However, like with the Coder brothers’ art exhibition, we quickly found that there were some drawbacks to using plain autoencoders as a generative model—sampling from the learned latent space was problematic, for a number of reasons.

Variational autoencoders solve these problems, by introducing randomness into the model and constraining how points in the latent space are distributed. We saw that with a few minor adjustments, we can transform our autoencoder into a variational autoencoder, thus giving it the power to be a generative model.

Finally, we applied our new technique to the problem of face generation and saw how we can simply choose points from a standard normal distribution to generate new faces. Moreover, by performing vector arithmetic within the latent space, we can achieve some amazing effects, such as face morphing and feature manipulation. With these features, it is easy to see why VAEs have become a prominent technique for generative modeling in recent years.

In the next chapter, we shall explore a different kind of generative model that has attracted an even greater amount of attention: the generative adversarial network.

CHAPTER 4

Generative Adversarial Networks

On Monday, December 5, 2016, at 2:30 p.m., Ian Goodfellow of Google Brain presented a tutorial entitled “[Generative Adversarial Networks](#)” to the delegates of the Neural Information Processing Systems (NIPS) conference in Barcelona.¹ The ideas presented in the tutorial are now regarded as one of the key turning points for generative modeling and have spawned a wide variety of variations on his core idea that have pushed the field to even greater heights.

This chapter will first lay out the theoretical underpinning of generative adversarial networks (GANs). You will then learn how to use the Python library Keras to start building your own GANs.

First though, we shall take a trip into the wilderness to meet Gene...

Ganimals

One afternoon, while walking through the local jungle, Gene sees a woman thumbing through a set of black and white photographs, looking worried. He goes over to ask if he can help.

The woman introduces herself as Di, a keen explorer, and explains that she is hunting for the elusive *ganimal*, a mythical creature that is said to roam around the jungle. Since the creature is nocturnal, she only has a collection of nighttime photos of the beast that she once found lying on the floor of the jungle, dropped by another ganimal enthusiast. Some of these photos are shown in [Figure 4-1](#). Di makes money by selling the images to collectors but is starting to worry, as she hasn’t actually ever seen

¹ Ian Goodfellow, “NIPS 2016 Tutorial: Generative Adversarial Networks,” 21 December 2016, <https://arxiv.org/abs/1701.00160v4>.

the creatures and is concerned that her business will falter if she can't produce more original photographs soon.

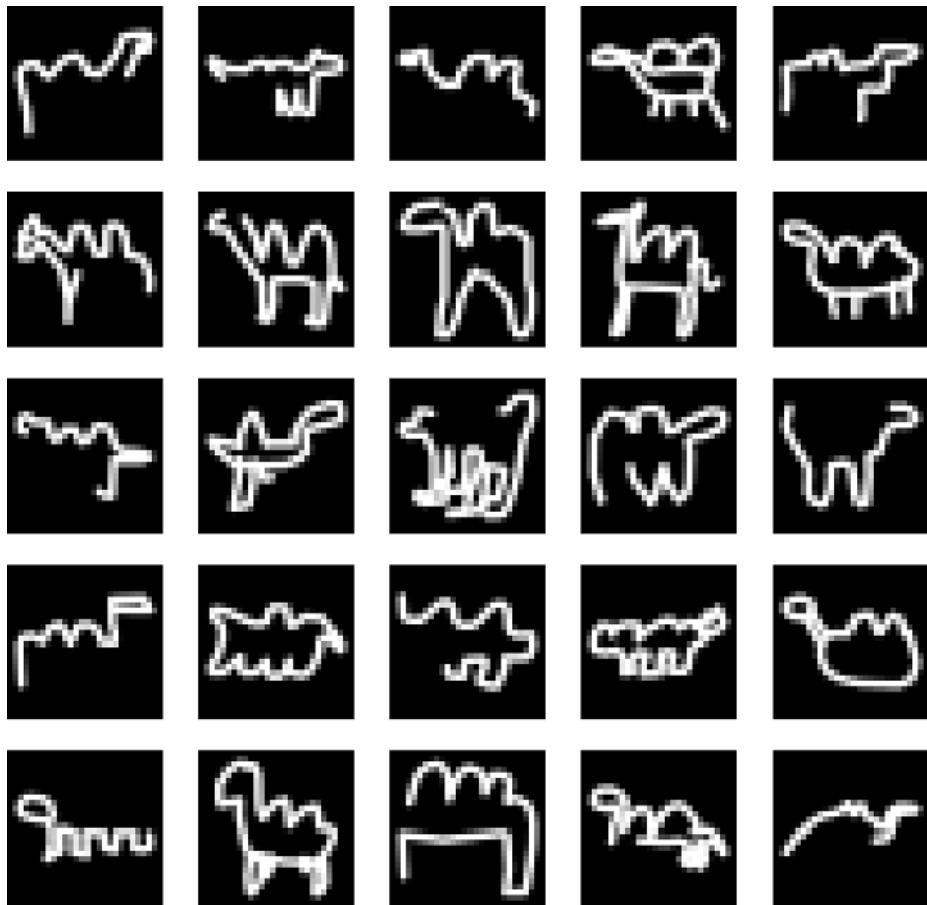


Figure 4-1. Original ganimal photographs

Being a keen photographer, Gene decides to help Di. He agrees to search for the ganimal himself and give her any photographs of the nocturnal beast that he manages to take.

However, there is a problem. Since Gene has never seen a ganimal, he doesn't know how to produce good photos of the creature, and also, since Di has only ever sold the photos she found, she cannot even tell the difference between a good photo of a ganimal and a photo of nothing at all.

Starting from this state of ignorance, how can they work together to ensure Gene is eventually able to produce impressive ganimal photographs?

They come up with following process. Each night, Gene takes 64 photographs, each in a different location with different random moonlight readings, and mixes them with 64 ganimal photos from the original collection. Di then looks at this set of photos and tries to guess which were taken by Gene and which are originals. Based on her mistakes, she updates her own understanding of how to discriminate between Gene's attempts and the original photos. Afterwards, Gene takes another 64 photos and shows them to Di. Di gives each photo a score between 0 and 1, indicating how realistic she thinks each photo is. Based on this feedback, Gene updates the settings on his camera to ensure that next time, he takes photos that Di is more likely to rate highly.

This process continues for many days. Initially, Gene doesn't get any useful feedback from Di, since she is randomly guessing which photos are genuine. However, after a few weeks of her training ritual, she gradually gets better at this, which means that she can provide better feedback to Gene so that he can adjust his camera accordingly in his training session. This makes Di's task harder, since now Gene's photos aren't quite as easy to distinguish from the real photos, so she must again learn how to improve. This back-and-forth process continues, over many days and weeks.

Over time, Gene gets better and better at producing ganimal photos, until eventually, Di is once again resigned to the fact that she cannot tell the difference between Gene's photos and the originals. They take Gene's generated photos to the auction and the experts cannot believe the quality of the new sightings—they are just as convincing as the originals. Some examples of Gene's work are shown in [Figure 4-2](#).



Figure 4-2. Samples of Gene's ganimal photography

Introduction to GANs

The adventures of Gene and Di hunting elusive nocturnal ganimals are a metaphor for one of the most important deep learning advancements of recent years: generative adversarial networks.

Simply put, a GAN is a battle between two adversaries, the generator and the discriminator. The generator tries to convert random noise into observations that look as if

they have been sampled from the original dataset and the discriminator tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries. Examples of the inputs and outputs to the two networks are shown in Figure 4-3.

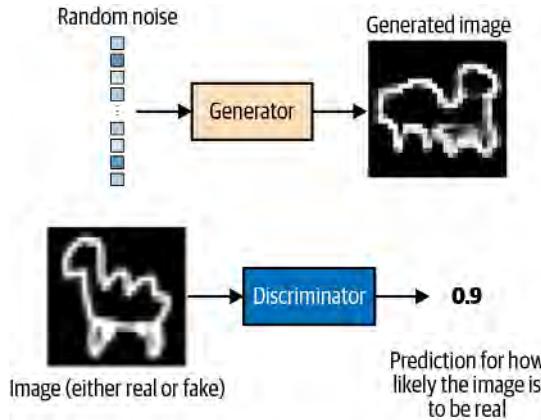


Figure 4-3. Inputs and outputs of the two networks in a GAN

At the start of the process, the generator outputs noisy images and the discriminator predicts randomly. The key to GANs lies in how we alternate the training of the two networks, so that as the generator becomes more adept at fooling the discriminator, the discriminator must adapt in order to maintain its ability to correctly identify which observations are fake. This drives the generator to find new ways to fool the discriminator, and so the cycle continues.

To see this in action, let's start building our first GAN in Keras, to generate pictures of nocturnal ganimals.

Your First GAN

First, you'll need to download the training data. We'll be using the [Quick, Draw! dataset](#) from Google. This is a crowdsourced collection of 28×28 -pixel grayscale doodles, labeled by subject. The dataset was collected as part of an online game that challenged players to draw a picture of an object or concept, while a neural network tries to guess the subject of the doodle. It's a really useful and fun dataset for learning the fundamentals of deep learning. For this task you'll need to download the `camel` `numpy` file and save it into the `./data/camel/` folder in the book repository.² The origi-

² By happy coincidence, ganimals look exactly like camels.

nal data is scaled in the range [0, 255] to denote the pixel intensity. For this GAN we rescale the data to the range [-1, 1].

Running the notebook `04_01_gan_camel_train.ipynb` in the book repository will start training the GAN. As in the previous chapter on VAEs, you can instantiate a GAN object in the notebook, as shown in [Example 4-1](#), and play around with the parameters to see how it affects the model.

Example 4-1. Defining the GAN

```
gan = GAN(input_dim = (28,28,1)
           , discriminator_conv_filters = [64,64,128,128]
           , discriminator_conv_kernel_size = [5,5,5,5]
           , discriminator_conv_strides = [2,2,2,1]
           , discriminator_batch_norm_momentum = None
           , discriminator_activation = 'relu'
           , discriminator_dropout_rate = 0.4
           , discriminator_learning_rate = 0.0008
           , generator_initial_dense_layer_size = (7, 7, 64)
           , generator_upsample = [2,2, 1, 1]
           , generator_conv_filters = [128,64, 64,1]
           , generator_conv_kernel_size = [5,5,5,5]
           , generator_conv_strides = [1,1, 1, 1]
           , generator_batch_norm_momentum = 0.9
           , generator_activation = 'relu'
           , generator_dropout_rate = None
           , generator_learning_rate = 0.0004
           , optimiser = 'rmsprop'
           , z_dim = 100
         )
```

Let's first take a look at how we build the discriminator.

The Discriminator

The goal of the discriminator is to predict if an image is real or fake. This is a supervised image classification problem, so we can use the same network architecture as in [Chapter 2](#): stacked convolutional layers, followed by a dense output layer.

In the original GAN paper, dense layers were used in place of the convolutional layers. However, since then, it has been shown that convolutional layers give greater predictive power to the discriminator. You may see this type of GAN called a DCGAN (deep convolutional generative adversarial network) in the literature, but now essentially all GAN architectures contain convolutional layers, so the “DC” is implied when we talk about GANs. It is also common to see batch normalization layers in the discriminator for vanilla GANs, though we choose not to use them here for simplicity.

The full architecture of the discriminator we will be building is shown in [Figure 4-4](#).

Layer (type)	Output Shape	Param #
<hr/>		
discriminator_input (InputLayer)	(None, 28, 28, 1)	0
discriminator_conv_0 (Conv2D)	(None, 14, 14, 64)	1664
activation_1 (Activation)	(None, 14, 14, 64)	0
dropout_1 (Dropout)	(None, 14, 14, 64)	0
discriminator_conv_1 (Conv2D)	(None, 7, 7, 64)	102464
activation_2 (Activation)	(None, 7, 7, 64)	0
dropout_2 (Dropout)	(None, 7, 7, 64)	0
discriminator_conv_2 (Conv2D)	(None, 4, 4, 128)	204928
activation_3 (Activation)	(None, 4, 4, 128)	0
dropout_3 (Dropout)	(None, 4, 4, 128)	0
discriminator_conv_3 (Conv2D)	(None, 4, 4, 128)	409728
activation_4 (Activation)	(None, 4, 4, 128)	0
dropout_4 (Dropout)	(None, 4, 4, 128)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_1 (Dense)	(None, 1)	2049
<hr/>		
Total params:	720,833	
Trainable params:	720,833	
Non-trainable params:	0	

Figure 4-4. The discriminator of the GAN

The Keras code to build the discriminator is provided in [Example 4-2](#).

Example 4-2. The discriminator

```
discriminator_input = Input(shape=self.input_dim, name='discriminator_input') ❶
x = discriminator_input

for i in range(self.n_layers_discriminator): ❷

    x = Conv2D(
        filters = self.discriminator_conv_filters[i]
        , kernel_size = self.discriminator_conv_kernel_size[i]
        , strides = self.discriminator_conv_strides[i]
        , padding = 'same'
        , name = 'discriminator_conv_' + str(i)
    )(x)
```

```

if self.discriminator_batch_norm_momentum and i > 0:
    x = BatchNormalization(momentum = self.discriminator_batch_norm_momentum)(x)

x = Activation(self.discriminator_activation)(x)

if self.discriminator_dropout_rate:
    x = Dropout(rate = self.discriminator_dropout_rate)(x)

x = Flatten()(x) ❸
discriminator_output= Dense(1, activation='sigmoid'
    , kernel_initializer = self.weight_init)(x) ❹

discriminator = Model(discriminator_input, discriminator_output) ❺

```

- ❶ Define the input to the discriminator (the image).
- ❷ Stack convolutional layers on top of each other.
- ❸ Flatten the last convolutional layer to a vector.
- ❹ Dense layer of one unit, with a sigmoid activation function that transforms the output from the dense layer to the range [0, 1].
- ❺ The Keras model that defines the discriminator—a model that takes an input image and outputs a single number between 0 and 1.

Notice how we use a stride of 2 in some of the convolutional layers to reduce the size of the tensor as it passes through the network, but increase the number of channels (1 in the grayscale input image, then 64, then 128).

The sigmoid activation in the final layer ensures that the output is scaled between 0 and 1. This will be the predicted probability that the image is real.

The Generator

Now let's build the generator. The input to the generator is a vector, usually drawn from a multivariate standard normal distribution. The output is an image of the same size as an image in the original training data.

This description may remind you of the decoder in a variational autoencoder. In fact, the generator of a GAN fulfills exactly the same purpose as the decoder of a VAE: converting a vector in the latent space to an image. The concept of mapping from a latent space back to the original domain is very common in generative modeling as it gives us the ability to manipulate vectors in the latent space to change high-level features of images in the original domain.

The architecture of the generator we will be building is shown in [Figure 4-5](#).

Layer (type)	Output Shape	Param #
<hr/>		
generator_input (InputLayer)	(None, 100)	0
dense_9 (Dense)	(None, 3136)	316736
batch_normalization_10 (BatchNormalization)	(None, 3136)	12544
activation_36 (Activation)	(None, 3136)	0
reshape_4 (Reshape)	(None, 7, 7, 64)	0
up_sampling2d_10 (UpSampling2D)	(None, 14, 14, 64)	0
generator_conv_0 (Conv2D)	(None, 14, 14, 128)	204928
batch_normalization_11 (BatchNormalization)	(None, 14, 14, 128)	512
activation_37 (Activation)	(None, 14, 14, 128)	0
up_sampling2d_11 (UpSampling2D)	(None, 28, 28, 128)	0
generator_conv_1 (Conv2D)	(None, 28, 28, 64)	204864
batch_normalization_12 (BatchNormalization)	(None, 28, 28, 64)	256
activation_38 (Activation)	(None, 28, 28, 64)	0
generator_conv_2 (Conv2D)	(None, 28, 28, 64)	102464
batch_normalization_13 (BatchNormalization)	(None, 28, 28, 64)	256
activation_39 (Activation)	(None, 28, 28, 64)	0
generator_conv_3 (Conv2D)	(None, 28, 28, 1)	1601
activation_40 (Activation)	(None, 28, 28, 1)	0
<hr/>		
Total params:	844,161	
Trainable params:	837,377	
Non-trainable params:	6,784	

Figure 4-5. The generator

First though, we need to introduce a new layer type: the *upsampling* layer.

Upsampling

In the decoder of the variational autoencoder that we built in the previous chapter, we doubled the width and height of the tensor at each layer using Conv2DTranspose layers with stride 2. This inserted zero values in between pixels before performing the convolution operations.

In this GAN, we instead use the Keras Upsampling2D layer to double the width and height of the input tensor. This simply repeats each row and column of its input in order to double the size. We then follow this with a normal convolutional layer with stride 1 to perform the convolution operation. It is a similar idea to convolutional

transpose, but instead of filling the gaps between pixels with zeros, upsampling just repeats the existing pixel values.

Both of these methods—`Upsampling + Conv2D` and `Conv2DTranspose`—are acceptable ways to transform back to the original image domain. It really is a case of testing both methods in your own problem setting and seeing which produces better results. It has been shown that the `Conv2DTranspose` method can lead to *artifacts*, or small checkerboard patterns in the output image (see Figure 4-6) that spoil the quality of the output. However, they are still used in many of the most impressive GANs in the literature and have proven to be a powerful tool in the deep learning practitioner's toolbox—again, I suggest you experiment with both methods and see which works best for you.

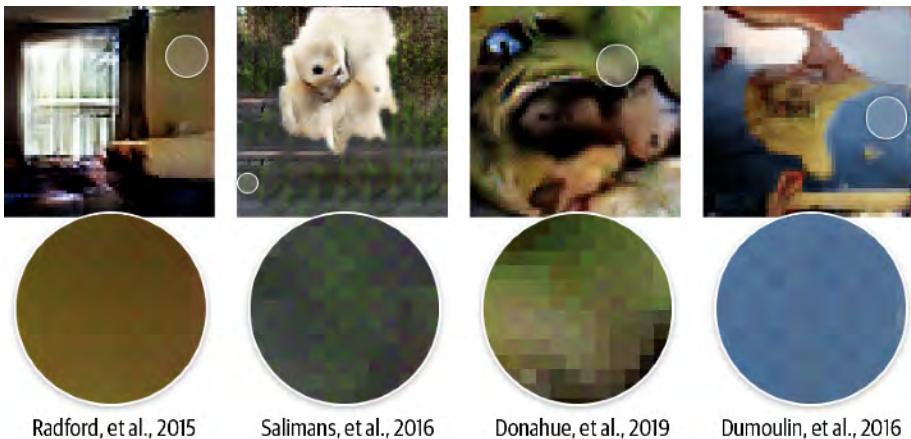


Figure 4-6. Artifacts when using convolutional transpose layers³

The code for building the generator is given in Example 4-3.

Example 4-3. The generator

```
generator_input = Input(shape=(self.z_dim,), name='generator_input') ①
x = generator_input

x = Dense(np.prod(self.generator_initial_dense_layer_size))(x) ②

if self.generator_batch_norm_momentum:
    x = BatchNormalization(momentum = self.generator_batch_norm_momentum)(x)
```

³ Source: Augustus Odena et al., "Deconvolution and Checkerboard Artifacts, 17 October 2016, <http://bit.ly/31MgHUQ>.

```

x = Activation(self.generator_activation)(x)

x = Reshape(self.generator_initial_dense_layer_size)(x) ③

if self.generator_dropout_rate:
    x = Dropout(rate = self.generator_dropout_rate)(x)

for i in range(self.n_layers_generator): ④

    x = UpSampling2D()(x)
    x = Conv2D(
        filters = self.generator_conv_filters[i]
        , kernel_size = self.generator_conv_kernel_size[i]
        , padding = 'same'
        , name = 'generator_conv_' + str(i)
    )(x)

    if i < n_layers_generator - 1: ⑤
        if self.generator_batch_norm_momentum:
            x = BatchNormalization(momentum = self.generator_batch_norm_momentum))(x)
        x = Activation('relu')(x)
    else:
        x = Activation('tanh')(x)

generator_output = x
generator = Model(generator_input, generator_output) ⑥

```

- ① Define the input to the generator—a vector of length 100.
- ② We follow this with a Dense layer consisting of 3,136 units...
- ③ ...which, after applying batch normalization and a ReLU activation function, is reshaped to a $7 \times 7 \times 64$ tensor.
- ④ We pass this through four Conv2D layers, the first two preceded by UpSampling2D layers, to reshape the tensor to 14×14 , then 28×28 (the original image size). In all but the last layer, we use batch normalization and ReLU activation (LeakyReLU could also be used).
- ⑤ After the final Conv2D layer, we use a tanh activation to transform the output to the range $[-1, 1]$, to match the original image domain.
- ⑥ The Keras model that defines the generator—a model that accepts a vector of length 100 and outputs a tensor of shape [28, 28, 1].

Training the GAN

As we have seen, the architecture of the generator and discriminator in a GAN is very simple and not so different from the models that we looked at earlier. The key to understanding GANs is in understanding the training process.

We can train the discriminator by creating a training set where some of the images are randomly selected *real* observations from the training set and some are outputs from the generator. The response would be 1 for the true images and 0 for the generated images. If we treat this as a supervised learning problem, we can train the discriminator to learn how to tell the difference between the original and generated images, outputting values near 1 for the true images and values near 0 for the fake images.

Training the generator is more difficult as there is no training set that tells us the *true* image that a particular point in the latent space should be mapped to. Instead, we only want the image that is generated to fool the discriminator—that is, when the image is fed as input to the discriminator, we want the output to be close to 1.

Therefore, to train the generator, we must first connect it to the discriminator to create a Keras model that we can train. Specifically, we feed the output from the generator (a $28 \times 28 \times 1$ image) into the discriminator so that the output from this combined model is the probability that the generated image is *real*, according to the discriminator. We can train this combined model by creating training batches consisting of randomly generated 100-dimensional latent vectors as input and a response which is set to 1, since we want to train the generator to produce images that the discriminator thinks are real.

The loss function is then just the binary cross-entropy loss between the output from the discriminator and the response vector of 1.

Crucially, we must freeze the weights of the discriminator while we are training the combined model, so that only the generator's weights are updated. If we do not freeze the discriminator's weights, the discriminator will adjust so that it is more likely to predict generated images as real, which is not the desired outcome. We want generated images to be predicted close to 1 (real) because the generator is strong, not because the discriminator is weak.

A diagram of the training process for the discriminator and generator is shown in [Figure 4-7](#).

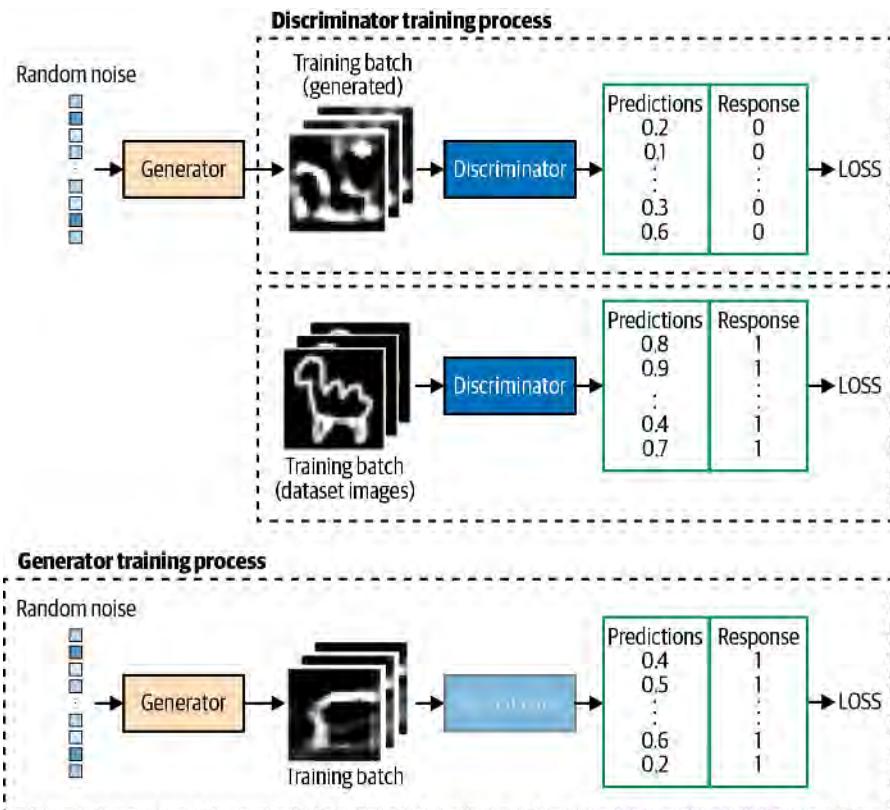


Figure 4-7. Training the GAN

Let's see what this looks like in code. First we need to compile the discriminator model and the model that trains the generator ([Example 4-4](#)).

Example 4-4. Compiling the GAN

```
### COMPILE MODEL THAT TRAINS THE DISCRIMINATOR

self.discriminator.compile(
    optimizer= RMSprop(lr=0.0008)
    , loss = 'binary_crossentropy'
    , metrics = ['accuracy']
) ❶

### COMPILE MODEL THAT TRAINS THE GENERATOR

self.discriminator.trainable = False ❷
model_input = Input(shape=(self.z_dim,), name='model_input')
model_output = discriminator(self.generator(model_input))
```

```

self.model = Model(model_input, model_output) ❸

self.model.compile(
    optimizer=RMSprop(lr=0.0004)
    , loss='binary_crossentropy'
    , metrics=['accuracy']
) ❹

```

- ❶ The discriminator is compiled with binary cross-entropy loss, as the response is binary and we have one output unit with sigmoid activation.
- ❷ Next, we freeze the discriminator weights—this doesn't affect the existing discriminator model that we have already compiled.
- ❸ We define a new model whose input is a 100-dimensional latent vector; this is passed through the generator and frozen discriminator to produce the output probability.
- ❹ Again, we use a binary cross-entropy loss for the combined model—the learning rate is slower than the discriminator as generally we would like the discriminator to be stronger than the generator. The learning rate is a parameter that should be tuned carefully for each GAN problem setting.

Then we train the GAN by alternating training of the discriminator and generator ([Example 4-5](#)).

Example 4-5. Training the GAN

```

def train_discriminator(x_train, batch_size):

    valid = np.ones((batch_size,1))
    fake = np.zeros((batch_size,1))

    # TRAIN ON REAL IMAGES
    idx = np.random.randint(0, x_train.shape[0], batch_size)
    true_imgs = x_train[idx]
    self.discriminator.train_on_batch(true_imgs, valid) ❶

    # TRAIN ON GENERATED IMAGES
    noise = np.random.normal(0, 1, (batch_size, z_dim))
    gen_imgs = generator.predict(noise)
    self.discriminator.train_on_batch(gen_imgs, fake) ❷

def train_generator(batch_size):

    valid = np.ones((batch_size,1))

```

```

noise = np.random.normal(0, 1, (batch_size, z_dim))
self.model.train_on_batch(noise, valid) ❸

epochs = 2000
batch_size = 64

for epoch in range(epochs):

    train_discriminator(x_train, batch_size)
    train_generator(batch_size)

```

- ❶ One batch update of the discriminator involves first training on a batch of true images with the response 1...
- ❷ ...then on a batch of generated images with the response 0.
- ❸ One batch update of the generator involves training on a batch of generated images with the response 1. As the discriminator is frozen, its weights will not be affected; instead, the generator weights will move in the direction that allows it to better generate images that are more likely to fool the discriminator (i.e., make the discriminator predict values close to 1).

After a suitable number of epochs, the discriminator and generator will have found an equilibrium that allows the generator to learn meaningful information from the discriminator and the quality of the images will start to improve ([Figure 4-8](#)).

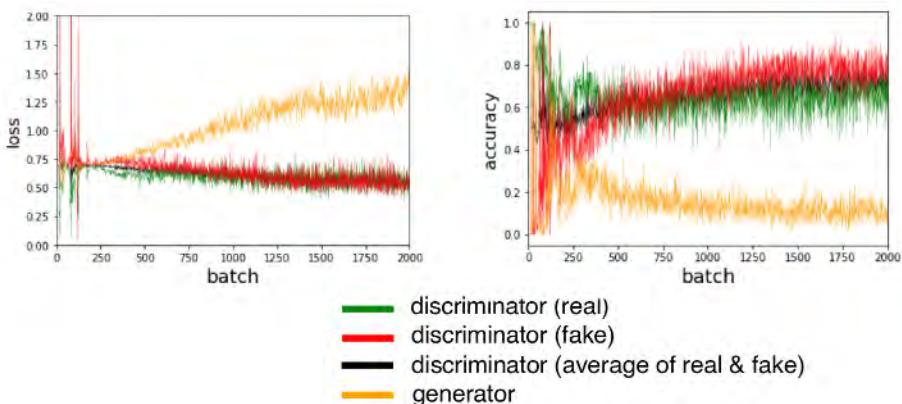


Figure 4-8. Loss and accuracy of the discriminator and generator during training

By observing images produced by the generator at specific epochs during training ([Figure 4-9](#)), it is clear that the generator is becoming increasingly adept at producing images that could have been drawn from the training set.

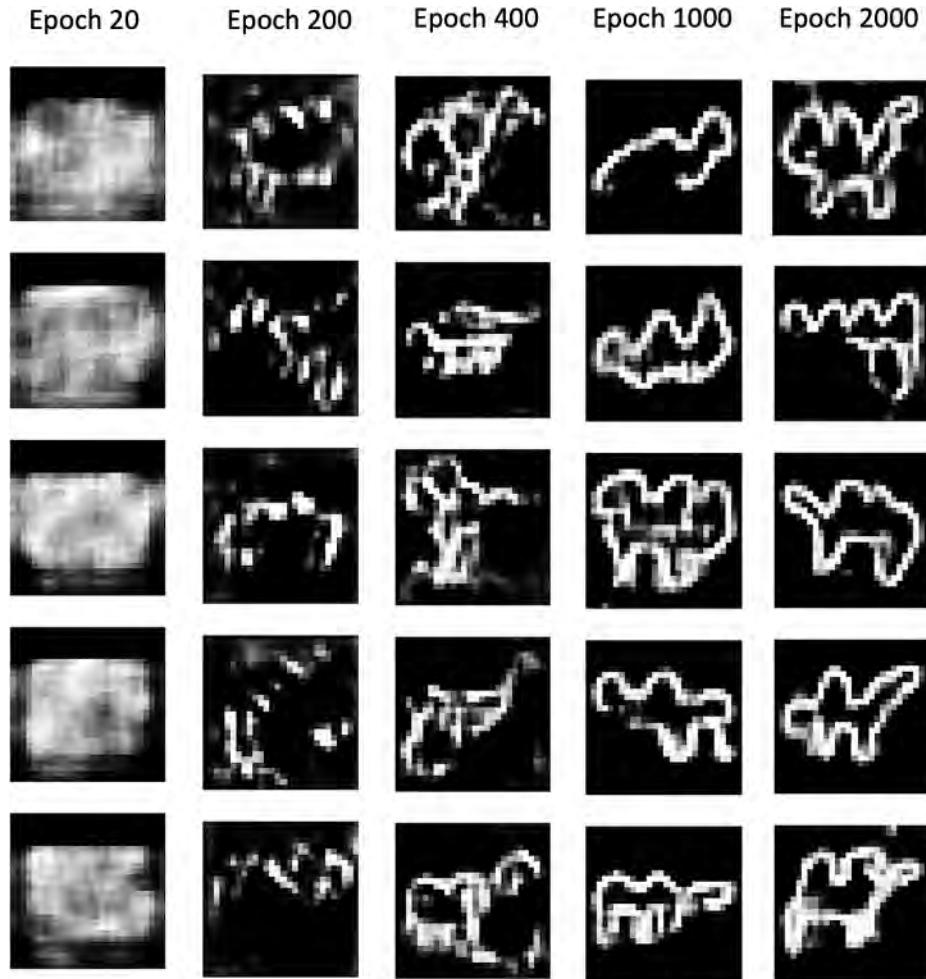


Figure 4-9. Output from the generator at specific epochs during training

It is somewhat miraculous that a neural network is able to convert random noise into something meaningful. It is worth remembering that we haven't provided the model with any additional features beyond the raw pixels, so it has to work out high-level concepts such as how to draw a *hump*, *legs*, or a *head* entirely by itself. The Naive Bayes models that we saw in [Chapter 1](#) wouldn't be able to achieve this level of sophistication since they cannot model the interdependencies between pixels that are crucial to forming these high-level features.

Another requirement of a successful generative model is that it doesn't only reproduce images from the training set. To test this, we can find the image from the

training set that is closest to a particular generated example. A good measure for distance is the L1 distance, defined as:

```
def l1_compare_images(img1, img2):
    return np.mean(np.abs(img1 - img2))
```

Figure 4-10 shows the closest observations in the training set for a selection of generated images. We can see that while there is some degree of similarity between the generated images and the training set, they are not identical and the GAN is also able to complete some of the unfinished drawings by, for example, adding legs or a head. This shows that the generator has understood these high-level features and can generate examples that are distinct from those it has already seen.

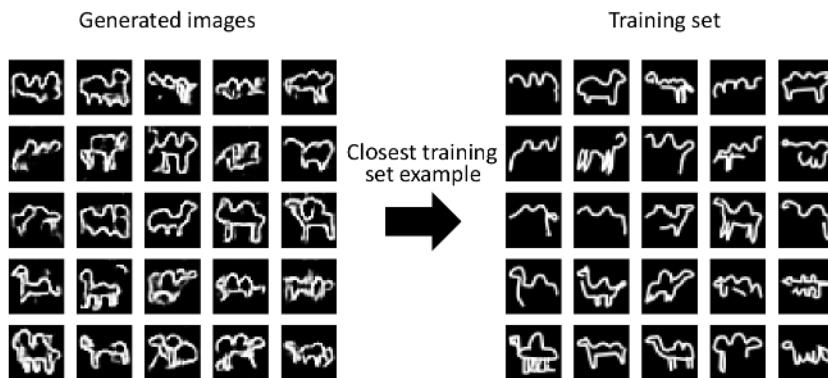


Figure 4-10. Closest matches of generated images from the training set

GAN Challenges

While GANs are a major breakthrough for generative modeling, they are also notoriously difficult to train. We will explore some of the most common problems encountered when training GANs in this section, then we will look at some adjustments to the GAN framework that remedy many of these problems.

Oscillating Loss

The loss of the discriminator and generator can start to oscillate wildly, rather than exhibiting long-term stability. Typically, there is some small oscillation of the loss between batches, but in the long term you should be looking for loss that stabilizes or gradually increases or decreases (see Figure 4-8), rather than erratically fluctuating, to ensure your GAN converges and improves over time. Figure 4-11 shows an example of a GAN where the loss of the discriminator and generator has started to spiral out

of control, at around batch 1,400. It is difficult to establish if or when this might occur as vanilla GANs are prone to this kind of instability.

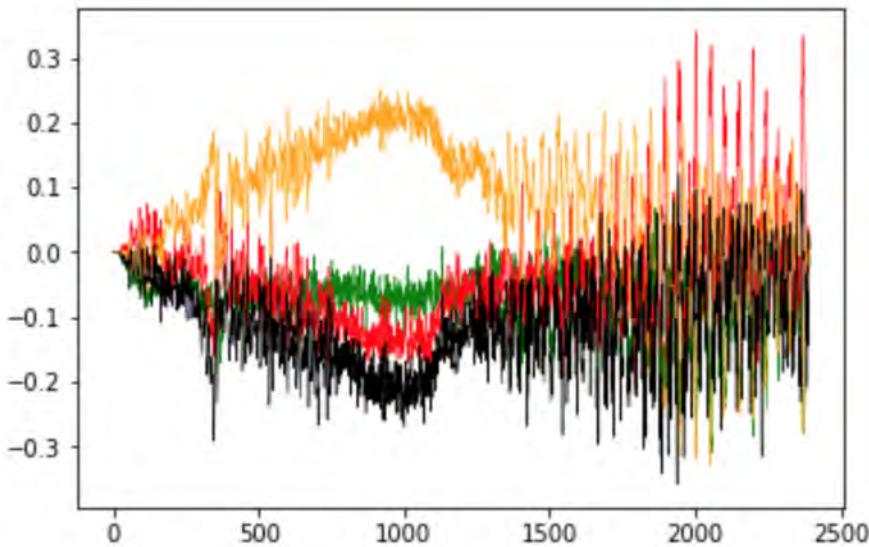


Figure 4-11. Oscillating loss in an unstable GAN

Mode Collapse

Mode collapse occurs when the generator finds a small number of samples that fool the discriminator and therefore isn't able to produce any examples other than this limited set. Let's think about how this might occur. Suppose we train the generator over several batches without updating the discriminator in between. The generator would be inclined to find a single observation (also known as a *mode*) that always fools the discriminator and would start to map every point in the latent input space to this observation. This means that the gradient of the loss function collapses to near 0. Even if we then try to retrain the discriminator to stop it being fooled by this one point, the generator will simply find another mode that fools the discriminator, since it has already become numb to its input and therefore has no incentive to diversify its output. The effect of mode collapse can be seen in [Figure 4-12](#).

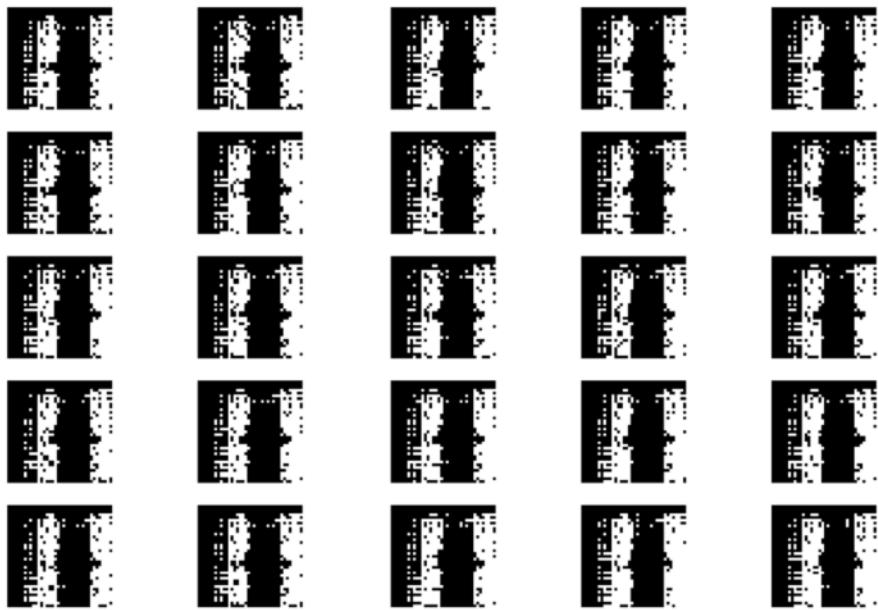


Figure 4-12. Mode collapse

Uninformative Loss

Since the deep learning model is compiled to minimize the loss function, it would be natural to think that the smaller the loss function of the generator, the better the quality of the images produced. However, since the generator is only graded against the current discriminator and the discriminator is constantly improving, we cannot compare the loss function evaluated at different points in the training process. Indeed, in [Figure 4-8](#), the loss function of the generator actually increases over time, even though the quality of the images is clearly improving. This lack of correlation between the generator loss and image quality sometimes makes GAN training difficult to monitor.

Hyperparameters

As we have seen, even with simple GANs, there are a large number of hyperparameters to tune. As well as the overall architecture of both the discriminator and the generator, there are the parameters that govern the batch normalization, dropout, learning rate, activation layers, convolutional filters, kernel size, striding, batch size, and latent space size to consider. GANs are highly sensitive to very slight changes in all of these parameters, and finding a set of parameters that works is often a case of educated trial and error, rather than following an established set of guidelines.

This is why it is important to understand the inner workings of the GAN and know how to interpret the loss function—so that you can identify sensible adjustments to the hyperparameters that might improve the stability of the model.

Tackling the GAN Challenges

In recent years, several key advancements have drastically improved the overall stability of GAN models and diminished the likelihood of some of the problems listed earlier, such as mode collapse.

In the remainder of this chapter we shall explore two such advancements, the Wasserstein GAN (WGAN) and Wasserstein GAN–Gradient Penalty (WGAN-GP). Both are only minor adjustments to the framework we have explored thus far. The latter is now considered best practice for training the most sophisticated GANs available today.

Wasserstein GAN

The Wasserstein GAN was one of the first big steps toward stabilizing GAN training.⁴ With a few changes, the authors were able to show how to train GANs that have the following two properties (quoted from the paper):

- A meaningful loss metric that correlates with the generator’s convergence and sample quality
- Improved stability of the optimization process

Specifically, the paper introduces a new loss function for both the discriminator and the generator. Using this loss function instead of binary cross entropy results in a more stable convergence of the GAN. The mathematical explanation for this is beyond the scope of this book, but there are some excellent resources available online that explain the rationale behind switching to this loss function.

Let’s take a look at the definition of the Wasserstein loss function.

Wasserstein Loss

Let’s first remind ourselves of binary cross-entropy loss - the function that we are currently using to train the discriminator and generator of the GAN:

⁴ Martin Arjovsky et al., “Wasserstein GAN,” 26 January 2017, <https://arxiv.org/pdf/1701.07875.pdf>.

Binary cross-entropy loss

$$-\frac{1}{n} \sum_{i=1}^n (y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

To train the GAN discriminator D , we calculate the loss when comparing predictions for real images $p_i = D(x_i)$ to the response $y_i = 1$ and predictions for generated images $p_i = D(G(z_i))$ to the response $y_i = 0$. Therefore for the GAN discriminator, minimizing the loss function can be written as follows:

GAN discriminator loss minimization

$$\min_D - \left(\mathbb{E}_{x \sim p_X} [\log D(x)] + \mathbb{E}_{z \sim p_Z} [\log (1 - D(G(z)))] \right)$$

To train the GAN generator G , we calculate the loss when comparing predictions for generated images $p_i = D(G(z_i))$ to the response $y_i = 1$. Therefore for the GAN generator, minimizing the loss function can be written as follows:

GAN generator loss minimization

$$\min_G - \left(\mathbb{E}_{z \sim p_Z} [\log D(G(z))] \right)$$

Now let's compare this to the Wasserstein loss function.

First, the *Wasserstein loss* requires that we use $y_i = 1$ and $y_i = -1$ as labels, rather than 1 and 0. We also remove the sigmoid activation from the final layer of the discriminator, so that predictions p_i are no longer constrained to fall in the range $[0, 1]$, but instead can now be any number in the range $[-\infty, \infty]$. For this reason, the discriminator in a WGAN is usually referred to as a *critic*. The Wasserstein loss function is then defined as follows:

Wasserstein loss

$$-\frac{1}{n} \sum_{i=1}^n (y_i p_i)$$

To train the WGAN critic D , we calculate the loss when comparing predictions for a real images $p_i = D(x_i)$ to the response $y_i = 1$ and predictions for generated images $p_i = D(G(z_i))$ to the response $y_i = -1$. Therefore for the WGAN critic, minimizing the loss function can be written as follows:

WGAN critic loss minimization

$$\min_D - \left(\mathbb{E}_{x \sim p_X}[D(x)] - \mathbb{E}_{z \sim p_Z}[D(G(z))] \right)$$

In other words, the WGAN critic tries to maximise the difference between its predictions for real images and generated images, with real images scoring higher.

To train the WGAN generator, we calculate the loss when comparing predictions for generated images $p_i = D(G(z_i))$ to the response $y_i=1$. Therefore for the WGAN generator, minimizing the loss function can be written as follows:

WGAN generator loss minimization

$$\min_G - \left(\mathbb{E}_{z \sim p_Z}[D(G(z))] \right)$$

When we compile the models that train the WGAN critic and generator, we can specify that we want to use the Wasserstein loss instead of the binary cross-entropy, as shown in [Example 4-6](#). We also tend to use smaller learning rates for WGANs.

Example 4-6. Compiling the models that train the critic and generator

```
def wasserstein(y_true, y_pred):
    return -K.mean(y_true * y_pred)

critic.compile(
    optimizer= RMSprop(lr=0.00005)
    , loss = wasserstein
)

model.compile(
    optimizer= RMSprop(lr=0.00005)
    , loss = wasserstein
)
```

The Lipschitz Constraint

It may surprise you that we are now allowing the critic to output any number in the range $[-\infty, \infty]$, rather than applying a sigmoid function to restrict the output to the usual $[0, 1]$ range. The Wasserstein loss can therefore be very large, which is unsettling—usually, large numbers in neural networks are to be avoided!

In fact, the authors of the WGAN paper show that for the Wasserstein loss function to work, we also need to place an additional constraint on the critic. Specifically, it is

required that the critic is a *1-Lipschitz continuous function*. Let's pick this apart to understand what it means in more detail.

The critic is a function D that converts an image into a prediction. We say that this function is 1-Lipschitz if it satisfies the following inequality for any two input images, x_1 and x_2 :

$$\frac{|D(x_1) - D(x_2)|}{|x_1 - x_2|} \leq 1$$

Here, $x_1 - x_2$ is the average pixelwise absolute difference between two images and $|D(x_1) - D(x_2)|$ is the absolute difference between the critic predictions. Essentially, we require a limit on the rate at which the predictions of the critic can change between two images (i.e., the absolute value of the gradient must be at most 1 everywhere). We can see this applied to a Lipschitz continuous 1D function in [Figure 4-13](#) —at no point does the line enter the cone, wherever you place the cone on the line. In other words, there is a limit on the rate at which the line can rise or fall at any point.

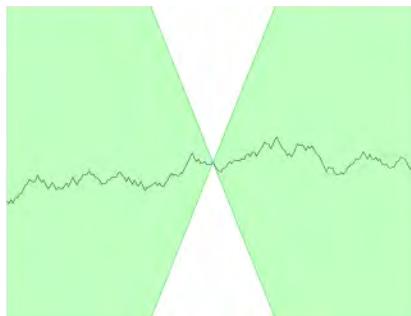


Figure 4-13. A Lipschitz continuous function—there exists a double cone such that wherever it is placed on the line, the function always remains entirely outside the cone⁵

For those who want to delve deeper into the mathematical rationale behind why the Wasserstein loss only works when this constraint is enforced, Jonathan Hui offers an excellent explanation.

Weight Clipping

In the WGAN paper, the authors show how it is possible to enforce the Lipschitz constraint by clipping the weights of the critic to lie within a small range, $[-0.01, 0.01]$, after each training batch.

⁵ Source: Wikipedia, <http://bit.ly/2Xufwd8>.

We can include this clipping process in our WGAN critic training function shown in [Example 4-7](#).

Example 4-7. Training the critic of the WGAN

```
def train_critic(x_train, batch_size, clip_threshold):

    valid = np.ones((batch_size,1))
    fake = -np.ones((batch_size,1))

    # TRAIN ON REAL IMAGES
    idx = np.random.randint(0, x_train.shape[0], batch_size)
    true_imgs = x_train[idx]
    self.critic.train_on_batch(true_imgs, valid)

    # TRAIN ON GENERATED IMAGES
    noise = np.random.normal(0, 1, (batch_size, self.z_dim))
    gen_imgs = self.generator.predict(noise)
    self.critic.train_on_batch(gen_imgs, fake)

    for l in critic.layers:
        weights = l.get_weights()
        weights = [np.clip(w, -clip_threshold, clip_threshold) for w in weights]
        l.set_weights(weights)
```

Training the WGAN

When using the Wasserstein loss function, we should train the critic to convergence to ensure that the gradients for the generator update are accurate. This is in contrast to a standard GAN, where it is important not to let the discriminator get too strong, to avoid vanishing gradients.

Therefore, using the Wasserstein loss removes one of the key difficulties of training GANs—how to balance the training of the discriminator and generator. With WGANs, we can simply train the critic several times between generator updates, to ensure it is close to convergence. A typical ratio used is five critic updates to one generator update.

The training loop of the WGAN is shown in [Example 4-8](#).

Example 4-8. Training the WGAN

```
for epoch in range(epochs):

    for _ in range(5):
        train_critic(x_train, batch_size = 128, clip_threshold = 0.01)

    train_generator(batch_size)
```

We have now covered all of the key differences between a standard GAN and a WGAN. To recap:

- A WGAN uses the Wasserstein loss.
- The WGAN is trained using labels of 1 for real and -1 for fake.
- There is no need for the sigmoid activation in the final layer of the WGAN critic.
- Clip the weights of the critic after each update.
- Train the critic multiple times for each update of the generator.

Analysis of the WGAN

You can train your own WGAN using code from the Jupyter notebook `04_02_wgan_cifar_train.ipynb` in the book repository. This will train a WGAN to generate images of horses from the CIFAR-10 dataset, which we used in [Chapter 2](#).

In [Figure 4-14](#) we show some of the samples generated by the WGAN.



Figure 4-14. Examples from the generator of a WGAN trained on images of horses

Clearly, this is a much more difficult task than our previous ganimal example, but the WGAN has done a good job of establishing the key features of horse images (legs, sky, grass, brownness, shadow, etc.). As well as the images being in color, there are also many varying angles, shapes, and backgrounds for the WGAN to deal with in the training set. Therefore while the image quality isn't yet perfect, we should be encouraged by the fact that our WGAN is clearly learning the high-level features that make up a color photograph of a horse.

One of the main criticisms of the WGAN is that since we are clipping the weights in the critic, its capacity to learn is greatly diminished. In fact, even in the original

WGAN paper the authors write, “Weight clipping is a clearly terrible way to enforce a Lipschitz constraint.”

A strong critic is pivotal to the success of a WGAN, since without accurate gradients, the generator cannot learn how to adapt its weights to produce better samples.

Therefore, other researchers have looked for alternative ways to enforce the Lipschitz constraint and improve the capacity of the WGAN to learn complex features. We shall explore one such breakthrough in the next section.

WGAN-GP

One of the most recent extensions to the WGAN literature is the Wasserstein GAN–Gradient Penalty (WGAN-GP) framework.⁶

The WGAN-GP generator is defined and compiled in exactly the same way as the WGAN generator. It is only the definition and compilation of the critic that we need to change.

In total, there are three changes we need to make to our WGAN critic to convert it to a WGAN-GP critic:

- Include a gradient penalty term in the critic loss function.
- Don’t clip the weights of the critic.
- Don’t use batch normalization layers in the critic.

Let’s start by seeing how we can build the gradient penalty term into our loss function. In the paper introducing this variant, the authors propose an alternative way to enforce the Lipschitz constraint on the critic. Rather than clipping the weights of the critic, they show how the constraint can be enforced directly by including a term in the loss function that penalizes the model if the gradient norm of the critic deviates from 1. This is a much more natural way to achieve the constraint and results in a far more stable training process.

The Gradient Penalty Loss

[Figure 4-15](#) is a diagram of the training process for the critic. If we compare this to the original discriminator training process from [Figure 4-7](#), we can see that the key addition is the gradient penalty loss included as part of the overall loss function, alongside the Wasserstein loss from the real and fake images.

⁶ Ishaan Gulrajani et al., “Improved Training of Wasserstein GANs,” 31 March 2017, <https://arxiv.org/abs/1704.00028>.

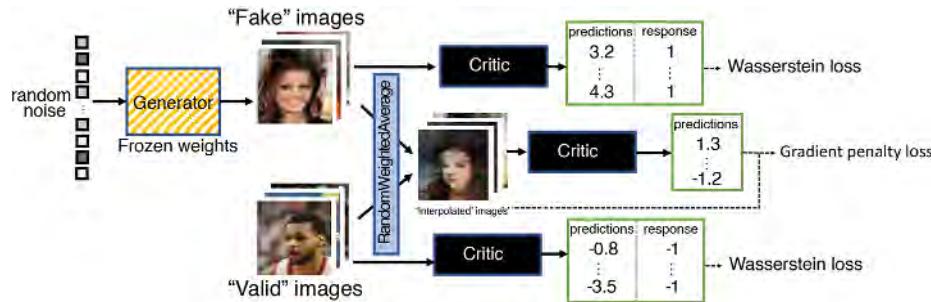


Figure 4-15. The WGAN-GP critic training process

The gradient penalty loss measures the squared difference between the norm of the gradient of the predictions with respect to the input images and 1. The model will naturally be inclined to find weights that ensure the gradient penalty term is minimized, thereby encouraging the model to conform to the Lipschitz constraint.

It is intractable to calculate this gradient everywhere during the training process, so instead the WGAN-GP evaluates the gradient at only a handful of points. To ensure a balanced mix, we use a set of interpolated images that lie at randomly chosen points along lines connecting the batch of real images to the batch of fake images pairwise, as shown in Figure 4-16.

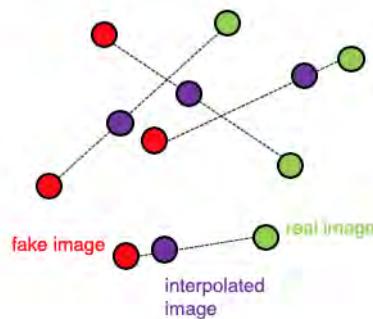


Figure 4-16. Interpolating between images

In Keras, we can create a `RandomWeightedAverage` layer to perform this interpolating operation, by inheriting from the built-in `_Merge` layer:

```
class RandomWeightedAverage(_Merge):
    def __init__(self, batch_size):
        super().__init__()
        self.batch_size = batch_size
```

```
def _merge_function(self, inputs):
    alpha = K.random_uniform((self.batch_size, 1, 1, 1)) ①
    return (alpha * inputs[0]) + ((1 - alpha) * inputs[1]) ②
```

- ➊ Each image in the batch gets a random number, between 0 and 1, stored as the vector `alpha`.
- ➋ The layer returns the set of pixelwise interpolated images that lie along the lines connecting the real images (`inputs[0]`) to the fake images (`inputs[1]`), pairwise, weighted by the `alpha` value for each pair.

The `gradient_penalty_loss` function shown in [Example 4-9](#) returns the squared difference between the gradient calculated at the interpolated points and 1.

Example 4-9. The gradient penalty loss function

```
def gradient_penalty_loss(y_true, y_pred, interpolated_samples):
    gradients = K.gradients(y_pred, interpolated_samples)[0] ①

    gradient_l2_norm = K.sqrt(
        K.sum(
            K.square(gradients),
            axis=[1:len(gradients.shape)])
    )
)
) ②
gradient_penalty = K.square(1 - gradient_l2_norm)
return K.mean(gradient_penalty) ③
```

- ➊ The Keras `gradients` function calculates the gradients of the predictions for the interpolated images (`y_pred`) with respect to the input (`interpolated_samples`).
- ➋ We calculate the L2 norm of this vector (i.e., its Euclidean length).
- ➌ The function returns the squared distance between the L2 norm and 1.

Now that we have the `RandomWeightedAverage` layer that can interpolate between two images and the `gradient_penalty_loss` function that can calculate the gradient loss for the interpolated images, we can use both of these in the model compilation of the critic.

In the WGAN example, we compiled the critic directly, to predict if a given image was real or fake. To compile the WGAN-GP critic, we need to use the interpolated images in the loss function—however, Keras only permits a custom loss function with two parameters, the predictions and the true labels. To get around this issue, we use the Python `partial` function.

Example 4-10 shows the full compilation of the WGAN-GP critic in code.

Example 4-10. Compiling the WGAN-GP critic

```
from functools import partial

### COMPILE CRITIC MODEL

self.generator.trainable = False ①

real_img = Input(shape=self.input_dim) ②
z_disc = Input(shape=(self.z_dim,))
fake_img = self.generator(z_disc)

fake = self.critic(fake_img) ③
valid = self.critic(real_img)

interpolated_img = RandomWeightedAverage(self.batch_size)([real_img, fake_img]) ④
validity_interpolated = self.critic(interpolated_img)

partial_gp_loss = partial(self.gradient_penalty_loss,
                         interpolated_samples = interpolated_img) ⑤
partial_gp_loss.__name__ = 'gradient_penalty' ⑥

self.critic_model = Model(inputs=[real_img, z_disc],
                          outputs=[valid, fake, validity_interpolated]) ⑦

self.critic_model.compile(
    loss=[self.wasserstein, self.wasserstein, partial_gp_loss]
    ,optimizer=Adam(lr=self.critic_learning_rate, beta_1=0.5)
    ,loss_weights=[1, 1, self.grad_weight]
) ⑧
```

- ① Freeze the weights of the generator. The generator forms part of the model that we are using to train the critic, as the interpolated images are now actively involved in the loss function, so this is required.
- ② There are two inputs to the model: a batch of real images and a set of randomly generated numbers that are used to generate a batch of fake images.
- ③ The real and fake images are passed through the critic in order to calculate the Wasserstein loss.
- ④ The `RandomWeightedAverage` layer creates the interpolated images, which are then also passed through the critic.
- ⑤ Keras is expecting a loss function with only two inputs—the predictions and true labels—so we define a custom loss function, `partial_gp_loss`, using the Python

`partial` function to pass the interpolated images through to our `gradient_penalty_loss` function.

- ❶ Keras requires the function to be named.
- ❷ The model that trains the critic is defined to have two inputs: the batch of real images and the random input that will generate the batch of fake images. The model has three outputs: 1 for the real images, -1 for the fake images, and a dummy 0 vector, which isn't actually used but is required by Keras as every loss function must map to an output. Therefore we create the dummy 0 vector to map to the `partial_gp_loss` function.
- ❸ We compile the critic with three loss functions: two Wasserstein losses for the real and fake images, and the gradient penalty loss. The overall loss is the sum of these three losses, with the gradient loss weighted by a factor of 10, in line with the recommendations from the original paper. We use the Adam optimizer, which is generally regarded to be the best optimizer for WGAN-GP models.

Batch Normalization in WGAN-GP

One last consideration we should note before building a WGAN-GP is that batch normalization shouldn't be used in the critic. This is because batch normalization creates correlation between images in the same batch, which makes the gradient penalty loss less effective. Experiments have shown that WGAN-GPs can still produce excellent results even without batch normalization in the critic.

Analysis of WGAN-GP

Running the Jupyter notebook `04_03_wgangp_faces_train.ipynb` in the book repository will train a WGAN-GP model on the CelebA dataset of celebrity faces.

First, let's take a look at some uncurated example outputs from the generator, after 3,000 training batches (Figure 4-17).



Figure 4-17. WGAN-GP CelebA examples

Clearly the model has learned the significant high-level attributes of a face, and there is no sign of mode collapse.

We can also see how the loss functions of the model evolve over time (Figure 4-18)—the loss functions of both the discriminator and generator are highly stable and convergent.

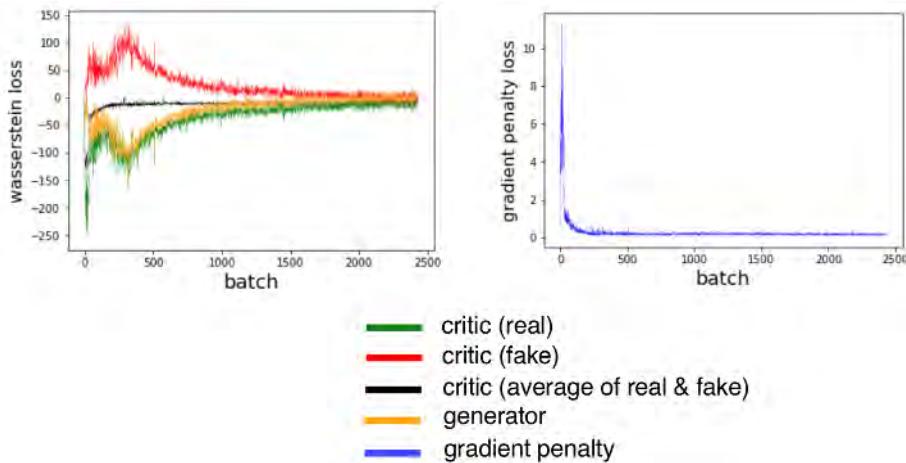


Figure 4-18. WGAN-GP loss

If we compare the WGAN-GP output to the VAE output from the previous chapter, we can see that the GAN images are generally sharper—especially the definition between the hair and the background. This is true in general; VAEs tend to produce

softer images that blur color boundaries, whereas GANs are known to produce sharper, more well-defined images.

It is also true that GANs are generally more difficult to train than VAEs and take longer to reach a satisfactory quality. However, most of the state-of-the-art generative models today are GAN-based, as the rewards for training large-scale GANs on GPUs over a longer period of time are significant.

Summary

In this chapter we have explored three distinct flavors of generative adversarial networks, from the most fundamental vanilla GANs, through the Wasserstein GAN (WGAN), to the current state-of-the-art WGAN-GP models.

All GANs are characterized by a generator versus discriminator (or critic) architecture, with the discriminator trying to “spot the difference” between real and fake images and the generator aiming to fool the discriminator. By balancing how these two adversaries are trained, the GAN generator can gradually learn how to produce similar observations to those in the training set.

We saw how vanilla GANs can suffer from several problems, including mode collapse and unstable training, and how the Wasserstein loss function remedied many of these problems and made GAN training more predictable and reliable. The natural extension of the WGAN is the WGAN-GP, which places the 1-Lipschitz requirement at the heart of the training process by including a term in the loss function to pull the gradient norm toward 1.

Finally, we applied our new technique to the problem of face generation and saw how by simply choosing points from a standard normal distribution, we can generate new faces. This sampling process is very similar to a VAE, though the faces produced by a GAN are quite different—often sharper, with greater distinction between different parts of the image. When trained on a large number of GPUs, this property allows GANs to produce extremely impressive results and has taken the field of generative modeling to ever greater heights.

Overall, we have seen how the GAN framework is extremely flexible and able to be adapted to many interesting problem domains. We’ll look at one such application in the next chapter and explore how we can teach machines to paint.

PART II

Teaching Machines to Paint, Write, Compose, and Play

Part I introduced the field of generative deep learning and analyzed two of the most important advancements in recent years, variational autoencoders and generative adversarial networks. The rest of this book presents a set of case studies showing how generative modeling techniques can be applied to particular tasks. The next three chapters focus on three core pillars of human creativity: painting, writing, and musical composition.

In [Chapter 5](#), we shall examine two techniques relating to machine painting. First we will look at CycleGAN, which as the name suggests is an adaptation of the GAN architecture that allows the model to learn how to convert a photograph into a painting in a particular style (or vice versa). Then we will also explore the neural style transfer technique contained within many photo editing apps that allows you to transfer the style of a painting onto a photograph, to give the impression that it is a painting by the same artist.

In [Chapter 6](#), we shall turn our attention to machine writing, a task that presents different challenges to image generation. This chapter introduces the recurrent neural network (RNN) architecture that allows us to tackle problems involving sequential data. We shall also see how the encoder–decoder architecture works and build a simple question-answer generator.

[Chapter 7](#) looks at music generation, which, while also a sequential generation problem, presents additional challenges such as modeling musical pitch and rhythm. We'll see that many of the techniques that worked for text generation can still be applied in

this domain, but we'll also explore a deep learning architecture known as MuseGAN that applies ideas from [Chapter 4](#) (on GANs) to musical data.

[Chapter 8](#) shows how generative models can be used within other machine learning domains, such as reinforcement learning. This chapter presents one of the most exciting papers published in recent years, in which the authors show how a generative model can be used as the environment in which the agent trains, thus essentially allowing the agent to *dream* of possible future scenarios and imagine what might happen if it were to take certain actions, entirely within its own conceptual model of the environment.

Finally, [Chapter 9](#) summarizes the current landscape of generative modeling and looks back on the techniques that have been presented in this book. We will also look to the future and explore how the most cutting-edge techniques available today might change the way in which we view creativity, and whether we will ever be able to create an artificial entity that can produce content that is creatively indistinguishable from works created by the human pioneers of art, literature, and music.

CHAPTER 5

Paint

So far, we have explored various ways in which we can train a model to generate new samples, given only a training set of data we wish to imitate. We've applied this to several datasets and seen how in each case, VAEs and GANs are able to learn a mapping between an underlying latent space and the original pixel space. By sampling from a distribution in the latent space, we can use the generative model to map this vector to a novel image in the pixel space.

Notice that all of the examples we have seen so far produce novel observations from scratch—that is, there is no input apart from the random latent vector sampled from the latent space that is used to generate the images.

A different application of generative models is in the field of *style transfer*. Here, our aim is to build a model that can transform an input *base image* in order to give the impression that it comes from the same collection as a given set of *style images*. This technique has clear commercial applications and is now being used in computer graphics software, computer game design, and mobile phone applications. Some examples of this are shown in [Figure 5-1](#).

With style transfer, our aim isn't to model the underlying distribution of the style images, but instead to extract only the stylistic components from these images and embed these into the base image. We clearly cannot just merge the style images with the base image through interpolation, as the content of the style images would show through and the colors would become muddy and blurred. Moreover, it may be the style image set as a whole rather than one single image that captures the artist's style, so we need to find a way to allow the model to learn about style across a whole collection of images. We want to give the impression that the artist has used the base image as a guide to produce an original piece of artwork, complete with the same stylistic flair as other works in their collection.



Figure 5-1. Style transfer examples¹

In this chapter you'll learn how to build two different kinds of style transfer model (CycleGAN and Neural Style Transfer) and apply the techniques to your own photos and artwork.

We'll start by visiting a fruit and vegetable shop where all is not as it seems...

Apples and Organges

Granny Smith and Florida own a greengrocers together. To ensure the shop is run as efficiently as possible, they each look after different areas—specifically, Granny Smith takes great pride in her apple selection and Florida spends hours ensuring the oranges are perfectly arranged.

Both are so convinced that they have the better fruit display that they agree to a deal: the profits from the sales of apples will go entirely to Granny Smith and the profits from the sales of oranges will go entirely to Florida.

Unfortunately, neither Granny Smith nor Florida plans on making this a fair contest. When Florida isn't looking, Granny Smith sneaks into the orange section and starts painting the oranges red to look like apples! Florida has exactly the same plan, and tries to make Granny Smith's apples more orange-like with a suitably colored spray when her back is turned.

When customers bring their fruit to the self-checkout tills, they sometimes erroneously select the wrong option on the machine. Customers who took apples sometimes

¹ Jun-Yan Zhu et al., “Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks,” 30 March 2017, <https://arxiv.org/pdf/1703.10593.pdf>.

put them through as oranges due to Florida's fluorescent spray, and customers who took oranges wrongly pay for apples due to their clever disguise, courtesy of Granny Smith.

At the end of the day, the profits for each fruit are summed and split accordingly—Granny Smith loses money every time one of her apples is sold as an orange, and Florida loses every time one of her oranges is sold as an apple.

After closing time, both of the disgruntled greengrocers do their best to clean up their own fruit stocks. However, instead of trying to undo the other's mischievous adjustments, they both simply apply their own tampering process to their own fruit, to try to make it appear as it did before it was sabotaged. It's important that they get this right, because if the fruit doesn't look right, they won't be able to sell it the next day and will again lose profits.

To ensure consistency over time, they also test their techniques out on their own fruit. Florida checks that if she sprays her oranges orange, they'll look exactly as they did originally. Granny Smith tests her apple painting skills on her apples for the same reason. If they find that there are obvious discrepancies, they'll have to spend their hard-earned profits on learning better techniques.

The overall process is shown in [Figure 5-2](#).

At first, customers are inclined to make somewhat random selections at the newly installed self-checkout tills because of their inexperience with the machines. However, over time they become more adept at using the technology, and learn how to identify which fruit has been tampered with.

This forces Granny Smith and Florida to improve at sabotaging each other's fruit, while also always ensuring that they are still able to use the same process to clean up their own fruit after it has been altered. Moreover, they must also make sure that the technique they use doesn't affect the appearance of their own fruit.

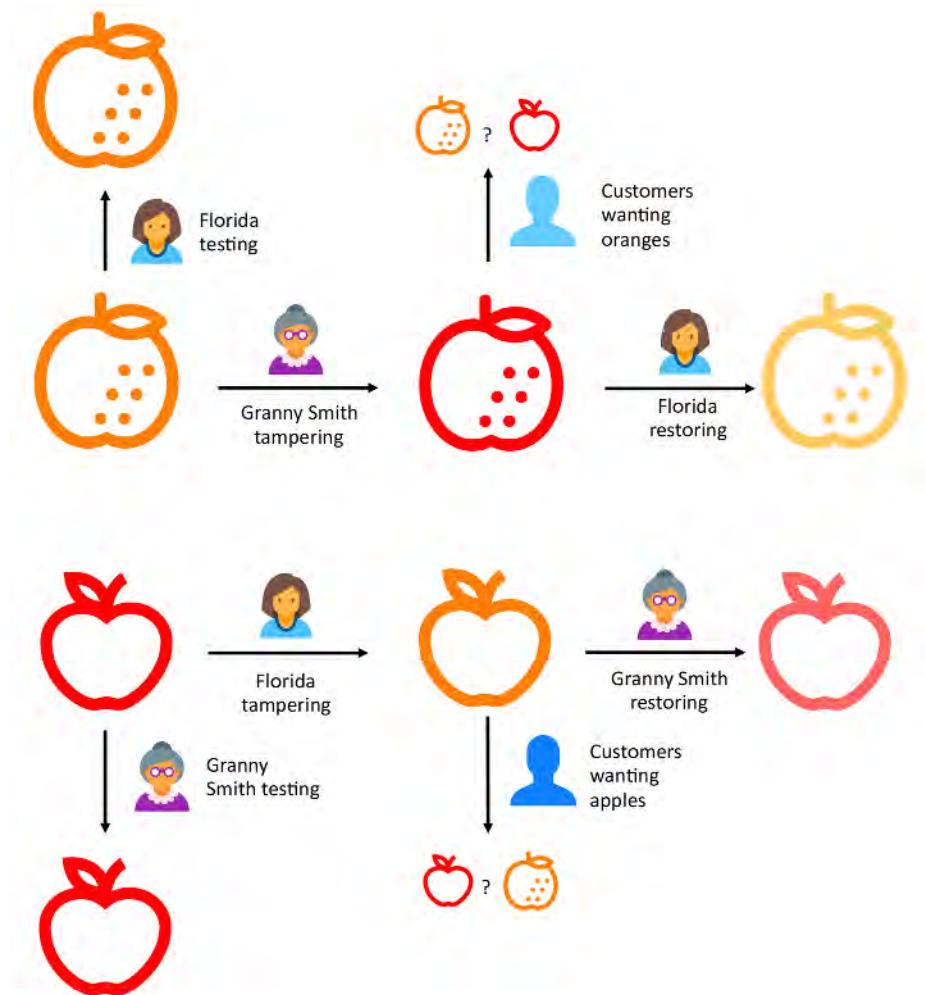


Figure 5-2. Diagram of the greengrocers' tampering, restoration, and testing process

After many days and weeks of this ridiculous game, they realize something amazing has happened. Customers are thoroughly confused and now can't tell the difference between real and fake apples and real and fake oranges. [Figure 5-3](#) shows the state of the fruit after tampering and restoration, as well as after testing.



Figure 5-3. Examples of the oranges and apples in the greengrocers' store

CycleGAN

The preceding story is an allegory for a key development in generative modeling and, in particular, style transfer: the cycle-consistent adversarial network, or *CycleGAN*. The original paper represented a significant step forward in the field of style transfer as it showed how it was possible to train a model that could copy the style from a reference set of images onto a different image, without a training set of paired examples.²

Previous style transfer models, such as *pix2pix*,³ required each image in the training set to exist in both the source and target domain. While it is possible to manufacture this kind of dataset for some style problem settings (e.g., black and white to color photos, maps to satellite images), for others it is impossible. For example, we do not have original photographs of the pond where Monet painted his *Water Lilies* series, nor do we have a Picasso painting of the Empire State Building. It would also take

² Jun-Yan Zhu et al., “Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks,” 30 March 2017, <https://arxiv.org/pdf/1703.10593.pdf>.

³ Phillip Isola et al., “Image-to-Image Translation with Conditional Adversarial Networks,” 2016, <https://arxiv.org/abs/1611.07004>.

enormous effort to arrange photos of horses and zebras standing in identical positions.

The CycleGAN paper was released only a few months after the pix2pix paper and shows how it is possible to train a model to tackle problems where we do not have pairs of images in the source and target domains. [Figure 5-4](#) shows the difference between the paired and unpaired datasets of pix2pix and CycleGAN, respectively.

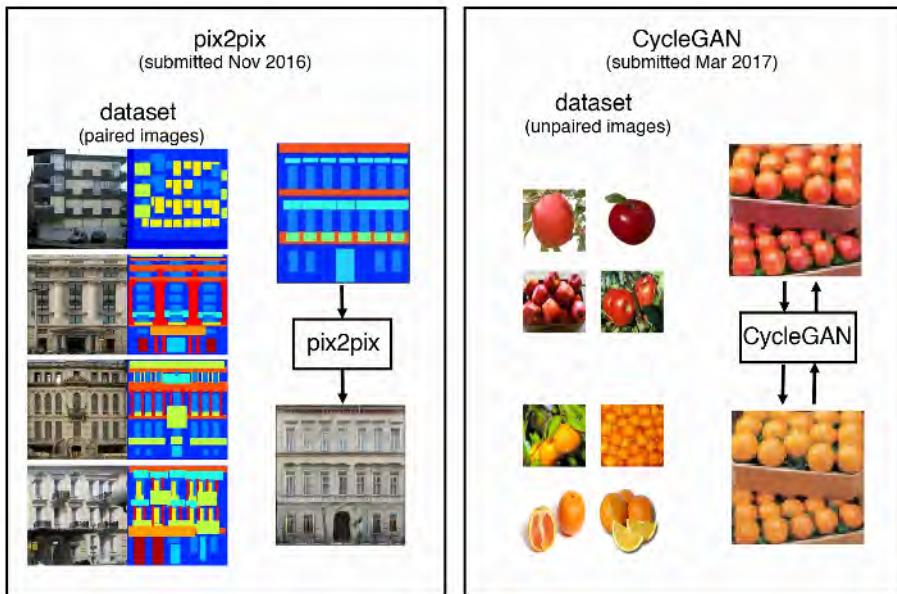


Figure 5-4. pix2pix dataset and domain mapping example

While pix2pix only works in one direction (from source to target), CycleGAN trains the model in both directions simultaneously, so that the model learns to translate images from target to source as well as source to target. This is a consequence of the model architecture, so you get the reverse direction for free.

Let's now see how we can build a CycleGAN model in Keras. To begin with, we shall be using the apples and oranges example from earlier to walk through each part of the CycleGAN and experiment with the architecture. We'll then apply the same technique to build a model that can apply a given artist's style to a photo of your choice.

Your First CycleGAN

Much of the following code has been inspired by and adapted from the amazing [Keras-GAN repository](#) maintained by Erik Linder-Norén. This is an excellent resource for many Keras examples of important GANs from the literature.

To begin, you'll first need to download the data that we'll be using to train the CycleGAN. From inside the folder where you cloned the book's repository, run the following command:

```
bash ./scripts/download_cyclegan_data.sh apple2orange
```

This will download the dataset of images of apples and oranges that we will be using. The data is split into four folders: *trainA* and *testA* contain images of apples and *trainB* and *testB* contain images of oranges. Thus domain A is the space of apple images and domain B is the space of orange images. Our goal is to train a model using the *train* datasets to convert images from domain A into domain B and vice versa. We will test our model using the *test* datasets.

Overview

A CycleGAN is actually composed of four models, two generators and two discriminators. The first generator, G_{AB} , converts images from domain A into domain B. The second generator, G_{BA} , converts images from domain B into domain A.

As we do not have paired images on which to train our generators, we also need to train two discriminators that will determine if the images produced by the generators are convincing. The first discriminator, d_A , is trained to be able to identify the difference between real images from domain A and fake images that have been produced by generator G_{BA} . Conversely, discriminator d_B is trained to be able to identify the difference between real images from domain B and fake images that have been produced by generator G_{AB} . The relationship between the four models is shown in [Figure 5-5](#).

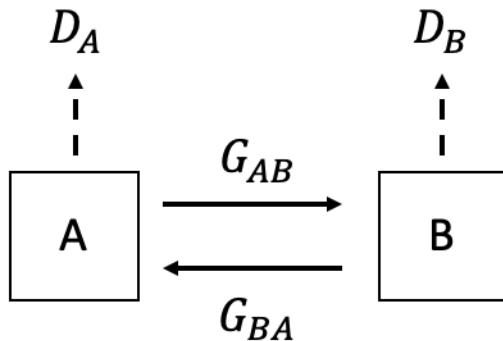


Figure 5-5. Diagram of the four CycleGAN models⁴

Running the notebook `05_01_cyclegan_train.ipynb` in the book repository will start training the CycleGAN. As in previous chapters, you can instantiate a `CycleGAN` object in the notebook, as shown in [Example 5-1](#), and play around with the parameters to see how it affects the model.

Example 5-1. Defining the CycleGAN

```
gan = CycleGAN(
    input_dim = (128,128,3)
    , learning_rate = 0.0002
    , lambda_validation = 1
    , lambda_reconstr = 10
    , lambda_id = 2
    , generator_type = 'u-net'
    , gen_n_filters = 32
    , disc_n_filters = 32
)
```

Let's first take a look at the architecture of the generators. Typically, CycleGAN generators take one of two forms: *U-Net* or *ResNet* (residual network). In their earlier pix2pix paper,⁵ the authors used a U-Net architecture, but they switched to a ResNet architecture for CycleGAN. We'll be building both architectures in this chapter, starting with U-Net.⁶

⁴ Source: Zhu et al., 2017, <https://arxiv.org/pdf/1703.10593.pdf>.

⁵ Isola et al., 2016, <https://arxiv.org/abs/1611.07004>.

⁶ Olaf Ronneberger, Philipp Fischer, and Thomas Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," 18 May 2015, <https://arxiv.org/abs/1505.04597>.

The Generators (U-Net)

Figure 5-6 shows the architecture of the U-Net we will be using—no prizes for guessing why it's called a U-Net!⁷

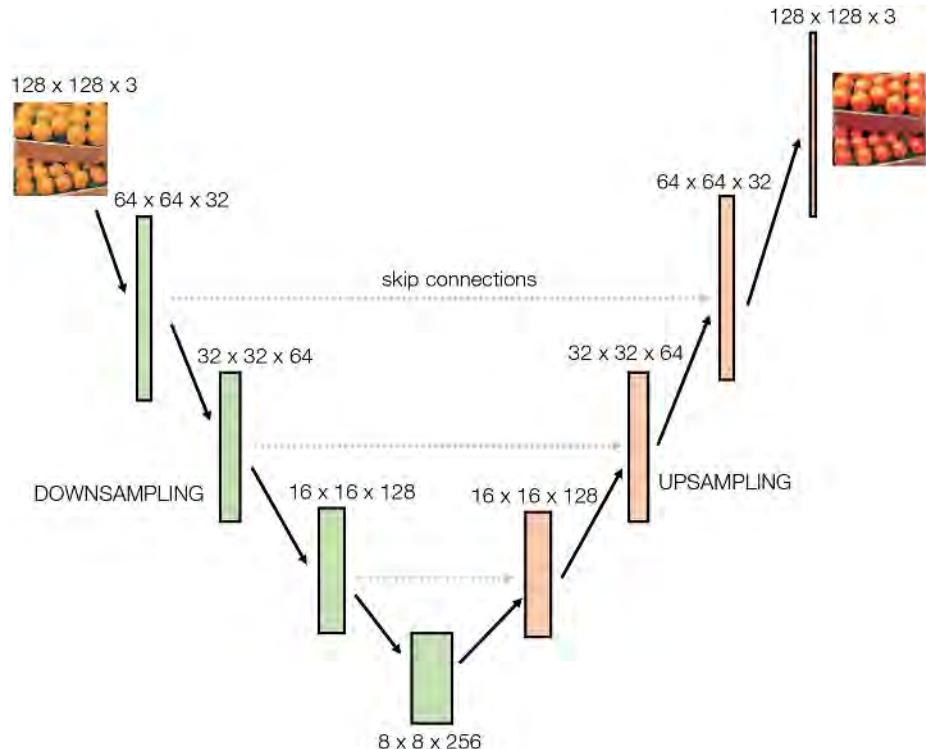


Figure 5-6. The U-Net architecture diagram

In a similar manner to a variational autoencoder, a U-Net consists of two halves: the downsampling half, where input images are compressed spatially but expanded channel-wise, and an upsampling half, where representations are expanded spatially while the number of channels is reduced.

However, unlike in a VAE, there are also *skip connections* between equivalently shaped layers in the upsampling and downsampling parts of the network. A VAE is linear; data flows through the network from input to the output, one layer after another. A U-Net is different, because it contains skip connections that allow information to shortcut parts of the network and flow through to later layers.

⁷ Ronneberger et al., 2015, <https://arxiv.org/abs/1505.04597>.

The intuition here is that with each subsequent layer in the downsampling part of the network, the model increasingly captures the *what* of the images and loses information on the *where*. At the apex of the U, the feature maps will have learned a contextual understanding of what is in the image, with little understanding of where it is located. For predictive classification models, this is all we require, so we could connect this to a final Dense layer to output the probability of a particular class being present in the image. However, for the original U-Net application (image segmentation) and also for style transfer, it is critical that when we upsample back to the original image size, we pass back into each layer the spatial information that was lost during downsampling. This is exactly why we need the skip connections. They allow the network to blend high-level abstract information captured during the downsampling process (i.e., the image *style*) with the specific spatial information that is being fed back in from previous layers in the network (i.e., the image *content*).

To build in the skip connections, we will need to introduce a new type of layer: ‘Concatenate’.

Concatenate Layer

The Concatenate layer simply joins a set of layers together along a particular axis (by default, the last axis). For example, in Keras, we can join two previous layers, x and y together as follows:

```
Concatenate()([x,y])
```

In the U-Net, we use Concatenate layers to connect upsampling layers to the equivalently sized layer in the downsampling part of the network. The layers are joined together along the channels dimension so the number of channels increases from k to $2k$, while the number of spatial dimensions remains the same.

Note that there are no weights to be learned in a Concatenate layer; they are just used to “glue” previous layers together.

The generator also contains another new layer type, `InstanceNormalization`.

Instance Normalization Layer

The generator of this CycleGAN uses `InstanceNormalization` layers rather than `BatchNormalization` layers, which in style transfer problems can lead to more satisfying results.⁸

⁸ Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky, “Instance Normalization: The Missing Ingredient for Fast Stylization,” 27 July 2016, <https://arxiv.org/pdf/1607.08022.pdf>.

An `InstanceNormalization` layer normalizes every single observation individually, rather than as a batch. Unlike a `BatchNormalization` layer, it doesn't require `mu` and `sigma` parameters to be calculated as a running average during training, since at test time the layer can normalize per instance in the same way as it does at train time. The means and standard deviations used to normalize each layer are calculated per channel and per observation.

Also, for the `InstanceNormalization` layers in this network, there are no weights to learn since we do not use scaling (`gamma`) or shift (`beta`) parameters.

Figure 5-7 shows the difference between batch normalization and instance normalization, as well as two other normalization methods (layer and group normalization).

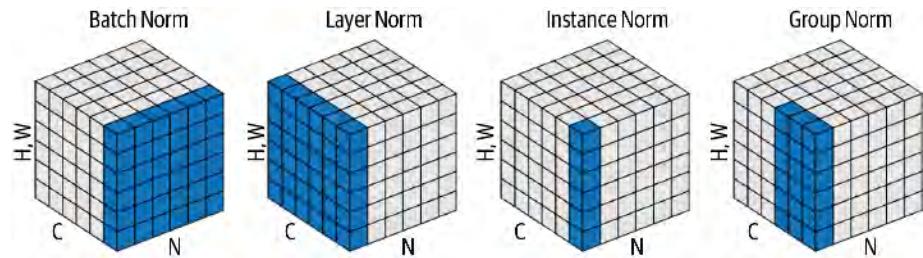


Figure 5-7. Four different normalization methods.⁹

Here, `N` is the batch axis, `C` is the channel axis, and `(H, W)` represent the spatial axes. The cube therefore represents the input tensor to the normalization layer. Pixels colored blue are normalized by the same mean and variance (calculated over the values of these pixels).

We now have everything we need to build a U-Net generator in Keras, as shown in **Example 5-2**.

Example 5-2. Building the U-Net generator

```
def build_generator_unet(self):  
  
    def downsample(layer_input, filters, f_size=4):  
        d = Conv2D(filters, kernel_size=f_size  
                  , strides=2, padding='same')(layer_input)  
        d = InstanceNormalization(axis = -1, center = False, scale = False)(d)  
        d = Activation('relu')(d)  
  
    ...
```

⁹ Source: Yuxin Wu and Kaiming He, "Group Normalization," 22 March 2018, <https://arxiv.org/pdf/1803.08494.pdf>.

```

    return d

def upsample(layer_input, skip_input, filters, f_size=4, dropout_rate=0):
    u = UpSampling2D(size=2)(layer_input)
    u = Conv2D(filters, kernel_size=f_size, strides=1, padding='same')(u)
    u = InstanceNormalization(axis = -1, center = False, scale = False)(u)
    u = Activation('relu')(u)
    if dropout_rate:
        u = Dropout(dropout_rate)(u)

    u = Concatenate()([u, skip_input])
    return u

# Image input
img = Input(shape=self.img_shape)

# Downsampling ①
d1 = downsample(img, self.gen_n_filters)
d2 = downsample(d1, self.gen_n_filters*2)
d3 = downsample(d2, self.gen_n_filters*4)
d4 = downsample(d3, self.gen_n_filters*8)

# Upsampling ②
u1 = upsample(d4, d3, self.gen_n_filters*4)
u2 = upsample(u1, d2, self.gen_n_filters*2)
u3 = upsample(u2, d1, self.gen_n_filters)

u4 = UpSampling2D(size=2)(u3)

output = Conv2D(self.channels, kernel_size=4, strides=1
                , padding='same', activation='tanh')(u4)

return Model(img, output)

```

- ① The generator consists of two halves. First, we downsample the image, using Conv2D layers with stride 2.
- ② Then we upsample, to return the tensor to the same size as the original image. The upsampling blocks contain Concatenate layers, which give the network the U-Net architecture.

The Discriminators

The discriminators that we have seen so far have output a single number: the predicted probability that the input image is “real.” The discriminators in the CycleGAN that we will be building output an 8×8 single-channel tensor rather than a single number.

The reason for this is that the CycleGAN inherits its discriminator architecture from a model known as a *PatchGAN*, where the discriminator divides the image into square overlapping “patches” and guesses if each patch is real or fake, rather than predicting for the image as a whole. Therefore the output of the discriminator is a tensor that contains the predicted probability for each patch, rather than just a single number.

Note that the patches are predicted simultaneously as we pass an image through the network—we do not divide up the image manually and pass each patch through the network one by one. The division of the image into patches arises naturally as a result of the discriminator’s convolutional architecture.

The benefit of using a PatchGAN discriminator is that the loss function can then measure how good the discriminator is at distinguishing images based on their *style* rather than their *content*. Since each individual element of the discriminator prediction is based only on a small square of the image, it must use the style of the patch, rather than its content, to make its decision. This is exactly what we require; we would rather our discriminator is good at identifying when two images differ in style than content.

The Keras code to build the discriminators is provided in [Example 5-3](#).

Example 5-3. Building the discriminators

```
def build_discriminator(self):

    def conv4(layer_input,filters, stride = 2, norm=True):
        y = Conv2D(filters, kernel_size=4, strides=stride
                   , padding='same')(layer_input)

        if norm:
            y = InstanceNormalization(axis = -1, center = False, scale = False)(y)

        y = LeakyReLU(0.2)(y)

        return y

    img = Input(shape=self.img_shape)

    y = conv4(img, self.disc_n_filters, stride = 2, norm = False) ①
    y = conv4(y, self.disc_n_filters*2, stride = 2)
    y = conv4(y, self.disc_n_filters*4, stride = 2)
    y = conv4(y, self.disc_n_filters*8, stride = 1)

    output = Conv2D(1, kernel_size=4, strides=1, padding='same')(y) ②

    return Model(img, output)
```

- ❶ A CycleGAN discriminator is a series of convolutional layers, all with instance normalization (except the first layer).
- ❷ The final layer is a convolutional layer with only one filter and no activation.

Compiling the CycleGAN

To recap, we aim to build a set of models that can convert images that are in domain A (e.g., images of apples) to domain B (e.g., images of oranges) and vice versa. We therefore need to compile four distinct models, two generators and two discriminators, as follows:

`g_AB`

Learns to convert an image from domain A to domain B.

`g_BA`

Learns to convert an image from domain B to domain A.

`d_A`

Learns the difference between real images from domain A and fake images generated by `g_BA`.

`d_B`

Learns the difference between real images from domain B and fake images generated by `g_AB`.

We can compile the two discriminators directly, as we have the inputs (images from each domain) and outputs (binary responses: 1 if the image was from the domain or 0 if it was a generated fake). This is shown in [Example 5-4](#).

Example 5-4. Compiling the discriminator

```
self.d_A = self.build_discriminator()
self.d_B = self.build_discriminator()
self.d_A.compile(loss='mse',
                  optimizer=Adam(self.learning_rate, 0.5),
                  metrics=['accuracy'])
self.d_B.compile(loss='mse',
                  optimizer=Adam(self.learning_rate, 0.5),
                  metrics=['accuracy'])
```

However, we cannot compile the generators directly, as we do not have paired images in our dataset. Instead, we judge the generators simultaneously on three criteria:

1. *Validity*. Do the images produced by each generator fool the relevant discriminator? (For example, does output from g_{BA} fool d_A and does output from g_{AB} fool d_B ?)
2. *Reconstruction*. If we apply the two generators one after the other (in both directions), do we return to the original image? The CycleGAN gets its name from this *cyclic* reconstruction criterion.
3. *Identity*. If we apply each generator to images from its own target domain, does the image remain unchanged?

Example 5-5 shows how we can compile a model to enforce these three criteria (the numeric markers in the code correspond to the preceding list).

Example 5-5. Building the combined model to train the generators

```
self.g_AB = self.build_generator_unet()
self.g_BA = self.build_generator_unet()

self.d_A.trainable = False
self.d_B.trainable = False

img_A = Input(shape=self.img_shape)
img_B = Input(shape=self.img_shape)
fake_A = self.g_BA(img_B)
fake_B = self.g_AB(img_A)

valid_A = self.d_A(fake_A)
valid_B = self.d_B(fake_B) ①

reconstr_A = self.g_BA(fake_B)
reconstr_B = self.g_AB(fake_A) ②

img_A_id = self.g_BA(img_A)
img_B_id = self.g_AB(img_B) ③

self.combined = Model(inputs=[img_A, img_B],
                      outputs=[ valid_A, valid_B,
                                reconstr_A, reconstr_B,
                                img_A_id, img_B_id ])

self.combined.compile(loss=['mse', 'mse',
                           'mae', 'mae',
                           'mae', 'mae'],
                      loss_weights=[self.lambda_validation
                                    , self.lambda_validation
```

```

        , self.lambda_reconstr
        , self.lambda_reconstr
        , self.lambda_id
        , self.lambda_id
    ],
optimizer=optimizer)

```

The combined model accepts a batch of images from each domain as input and provides three outputs (to match the three criteria) for each domain—so, six outputs in total. Notice how we freeze the weights in the discriminator, as is typical with GANs, so that the combined model only trains the generator weights, even though the discriminator is involved in the model.

The overall loss is the weighted sum of the loss for each criterion. Mean squared error is used for the validity criterion—checking the output from the discriminator against the real (1) or fake (0) response—and mean absolute error is used for the image-to-image-based criteria (reconstruction and identity).

Training the CycleGAN

With our discriminators and combined model compiled, we can now train our models. This follows the standard GAN practice of alternating the training of the discriminators with the training of the generators (through the combined model).

In Keras, the code in [Example 5-6](#) describes the training loop.

Example 5-6. Training the CycleGAN

```

batch_size = 1
patch = int(self.img_rows / 2**4)
self.disc_patch = (patch, patch, 1)

valid = np.ones((batch_size,) + self.disc_patch) ❶
fake = np.zeros((batch_size,) + self.disc_patch)

for epoch in range(self.epoch, epochs):
    for batch_i, (imgs_A, imgs_B) in enumerate(data_loader.load_batch(batch_size)):

        fake_B = self.g_AB.predict(imgs_A) ❷
        fake_A = self.g_BA.predict(imgs_B)

        dA_loss_real = self.d_A.train_on_batch(imgs_A, valid)
        dA_loss_fake = self.d_A.train_on_batch(fake_A, fake)
        dA_loss = 0.5 * np.add(dA_loss_real, dA_loss_fake)

        dB_loss_real = self.d_B.train_on_batch(imgs_B, valid)
        dB_loss_fake = self.d_B.train_on_batch(fake_B, fake)
        dB_loss = 0.5 * np.add(dB_loss_real, dB_loss_fake)

```

```
d_loss = 0.5 * np.add(dA_loss, dB_loss)

g_loss = self.combined.train_on_batch([imgs_A, imgs_B],
                                      [valid, valid,
                                       imgs_A, imgs_B,
                                       imgs_A, imgs_B]) ③
```

- ➊ We use a response of 1 for real images and 0 for generated images. Notice how there is one response per patch, as we are using a PatchGAN discriminator.
- ➋ To train the discriminators, we first use the respective generator to create a batch of fake images, then we train each discriminator on this fake set and a batch of real images. Typically, for a CycleGAN the batch size is 1 (a single image).
- ➌ The generators are trained together in one step, through the combined model compiled earlier. See how the six outputs match to the six loss functions defined earlier during compilation.

Analysis of the CycleGAN

Let's see how the CycleGAN performs on our simple dataset of apples and oranges and observe how changing the weighting parameters in the loss function can have dramatic effects on the results.

We have already seen an example of the output from the CycleGAN model in [Figure 5-3](#). Now that you are familiar with the CycleGAN architecture, you might recognize that this image represents the three criteria through which the combined model is judged: validity, reconstruction, and identity.

Let's relabel this image with the appropriate functions from the codebase, so that we can see this more explicitly ([Figure 5-8](#)).

We can see that the training of the network has been successful, because each generator is visibly altering the input picture to look more like a valid image from the opposite domain. Moreover, when the generators are applied one after the other, the difference between the input image and the reconstructed image is minimal. Finally, when each generator is applied to an image from its own input domain, the image doesn't change significantly.

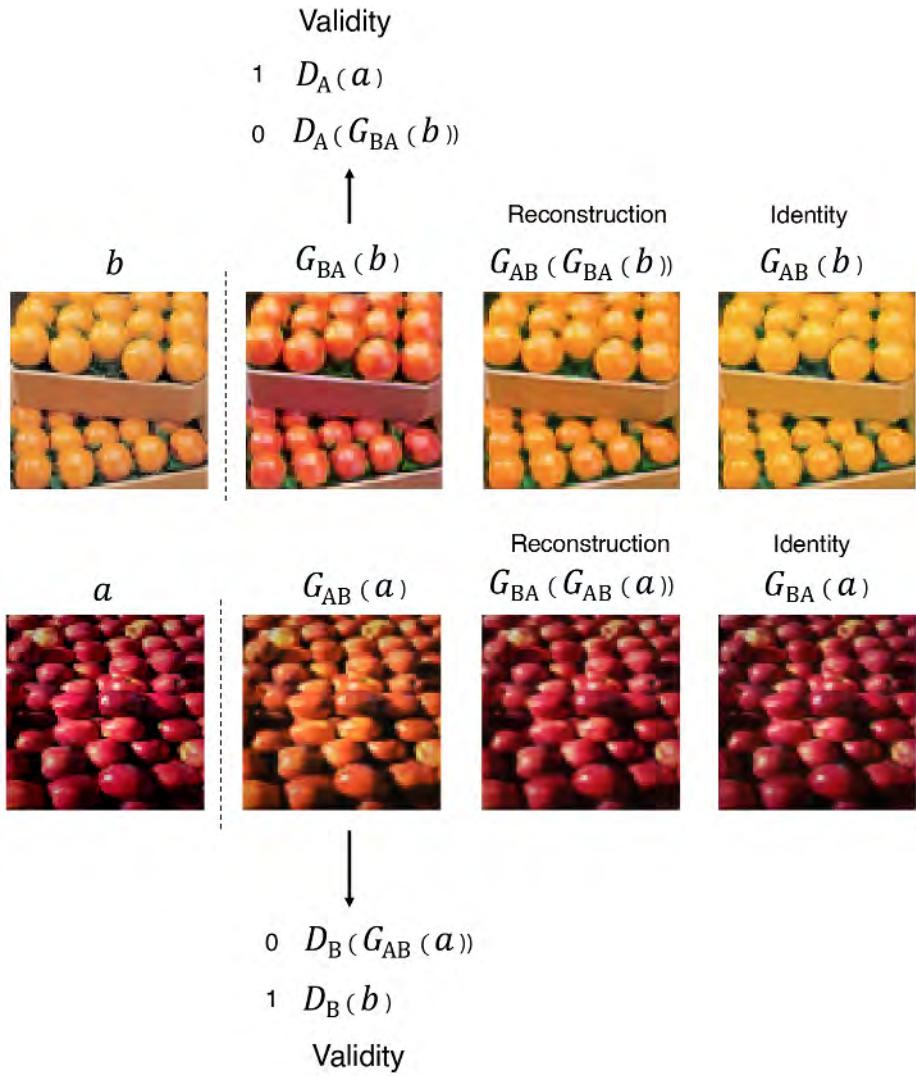


Figure 5-8. Outputs from the combined model used to calculate the overall loss function

In the original CycleGAN paper, the identity loss was included as an optional addition to the necessary reconstruction loss and validity loss. To demonstrate the importance of the identity term in the loss function, let's see what happens if we remove it, by setting the identity loss weighting parameter to zero in the loss function (Figure 5-9).

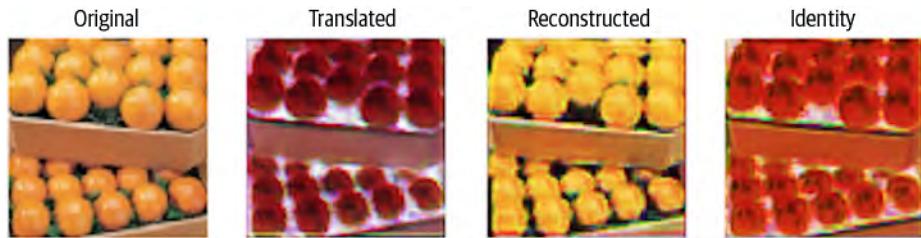


Figure 5-9. Output from the CycleGAN when the identity loss weighting is set to zero

The CycleGAN has still managed to translate the oranges into apples but the color of the tray holding the oranges has flipped from black to white, as there is now no identity loss term to prevent this shift in background colors. The identity term helps regulate the generator to ensure that it only adjust parts of the image that are necessary to complete the transformation and no more.

This highlights the importance of ensuring the weightings of the three loss functions are well balanced—too little identity loss and the color shift problem appears; too much identity loss and the CycleGAN isn't sufficiently incentivized to change the input to look like an image from the opposite domain.

Creating a CycleGAN to Paint Like Monet

Now that we have explored the fundamental structure of a CycleGAN, we can turn our attention to more interesting and impressive applications of the technique.

In the original CycleGAN paper, one of the standout achievements was the ability for the model to learn how to convert a given photo into a painting in the style of a particular artist. As this is a CycleGAN, the model is also able to translate the other way, converting an artist's paintings into realistic-looking photographs.

To download the Monet-to-photo dataset, run the following command from inside the book repository:

```
bash ./scripts/download_cyclegan_data.sh monet2photo
```

This time we will use the parameter set shown in [Example 5-7](#) to build the model:

Example 5-7. Defining the Monet CycleGAN

```
gan = CycleGAN(
    input_dim = (256,256,3)
    , learning_rate = 0.0002
    , lambda_validation = 1
    , lambda_reconstr = 10
    , lambda_id = 5
```

```
, generator_type = 'resnet'  
, gen_n_filters = 32  
, disc_n_filters = 64  
)
```

The Generators (ResNet)

In this example, we shall introduce a new type of generator architecture: a residual network, or ResNet.¹⁰ The ResNet architecture is similar to a U-Net in that it allows information from previous layers in the network to skip ahead one or more layers. However, rather than creating a U shape by connecting layers from the downsampling part of the network to corresponding upsampling layers, a ResNet is built of residual blocks stacked on top of each other, where each block contains a skip connection that sums the input and output of the block, before passing this on to the next layer. A single residual block is shown in Figure 5-10.

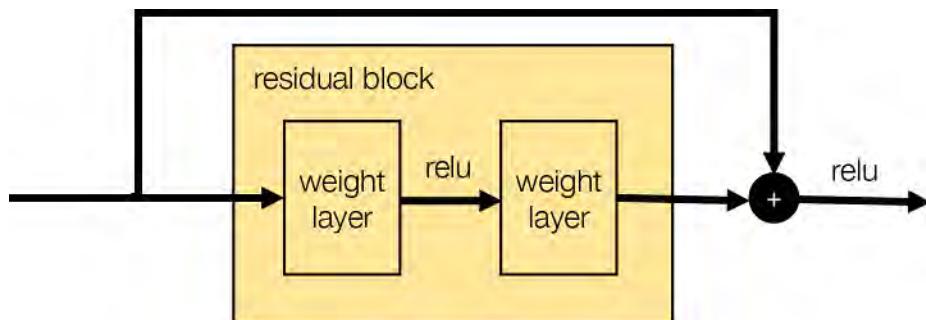


Figure 5-10. A single residual block

In our CycleGAN, the “weight layers” in the diagram are convolutional layers with instance normalization. In Keras, a residual block can be coded as shown in Example 5-8.

Example 5-8. A residual block in Keras

```
from keras.layers.merge import add  
  
def residual(layer_input, filters):  
    shortcut = layer_input  
    y = Conv2D(filters, kernel_size=(3, 3), strides=1, padding='same')(layer_input)  
    y = InstanceNormalization(axis = -1, center = False, scale = False)(y)  
    y = Activation('relu')(y)  
  
    y = add([y, shortcut])  
    y = Activation('relu')(y)  
  
    return y
```

¹⁰ Kaiming He et al., “Deep Residual Learning for Image Recognition,” 10 December 2015, <https://arxiv.org/abs/1512.03385>.

```

y = Conv2D(filters, kernel_size=(3, 3), strides=1, padding='same')(y)
y = InstanceNormalization(axis = -1, center = False, scale = False)(y)

return add([shortcut, y])

```

On either side of the residual blocks, our ResNet generator also contains downsampling and upsampling layers. The overall architecture of the ResNet is shown in [Figure 5-11](#).

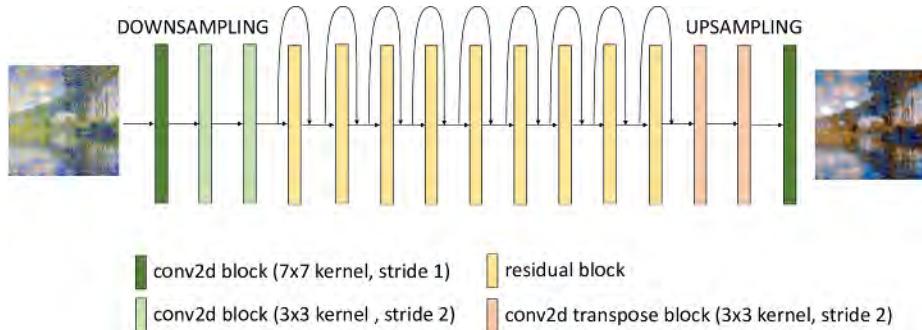


Figure 5-11. A ResNet generator

It has been shown that ResNet architectures can be trained to hundreds and even thousands of layers deep and not suffer from the *vanishing gradient* problem, where the gradients at early layers are tiny and therefore train very slowly. This is due to the fact that the error gradients can backpropagate freely through the network through the skip connections that are part of the residual blocks. Furthermore, it is believed that adding additional layers never results in a drop in model accuracy, as the skip connections ensure that it is always possible to pass through the identity mapping from the previous layer, if no further informative features can be extracted.

Analysis of the CycleGAN

In the original CycleGAN paper, the model was trained for 200 epochs to achieve state-of-the-art results for artist-to-photograph style transfer. In [Figure 5-12](#) we show the output from each generator at various stages of the early training process, to show the progression as the model begins to learn how to convert Monet paintings into photographs and vice versa.

In the top row, we can see that gradually the distinctive colors and brushstrokes used by Monet are transformed into the more natural colors and smooth edges that would be expected in a photograph. Similarly, the reverse is happening in the bottom row, as the generator learns how to convert a photograph into a scene that Monet might have painted himself.

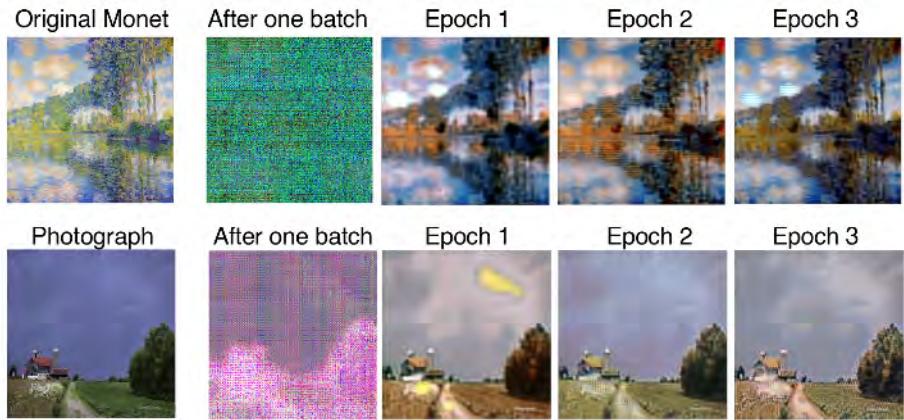


Figure 5-12. Output at various stages of the training process

Figure 5-13 shows some of the results from the original paper achieved by the model after it was trained for 200 epochs.



Figure 5-13. Output after 200 epochs of training¹¹

¹¹ Source: Zhu et al., 2017, <https://junyanz.github.io/CycleGAN>.

Neural Style Transfer

So far, we have seen how a CycleGAN can transpose images between two domains, where the images in the training set are not necessarily paired. Now we shall look at a different application of style transfer, where we do not have a training set at all, but instead wish to transfer the style of one single image onto another, as shown in Figure 5-14. This is known as *neural style transfer*.¹²



Figure 5-14. An example of neural style transfer¹³

The idea works on the premise that we want to minimize a loss function that is a weighted sum of three distinct parts:

Content loss

We would like the combined image to contain the same content as the base image.

Style loss

We would like the combined image to have the same general style as the style image.

Total variance loss

We would like the combined image to appear smooth rather than pixelated.

We minimize this loss via gradient descent—that is, we update each pixel value by an amount proportional to the negative gradient of the loss function, over many

¹² Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge, “A Neural Algorithm of Artistic Style,” 26 August 2015, <https://arxiv.org/abs/1508.06576>.

¹³ Source: Gatys, et al. 2015, <https://arxiv.org/abs/1508.06576>.

iterations. This way, the loss gradually decreases with each iteration and we end up with an image that merges the content of one image with the style of another.

Optimizing the generated output via gradient descent is different to how we have tackled generative modeling problems thus far. Previously we have trained a deep neural network such as a VAE or GAN by backpropagating the error through the entire network to learn from a training set of data and generalize the information learned to generate new images. Here, we cannot take this approach as we only have two images to work with, the base image and the style image. However, as we shall see, we can still make use of a pretrained deep neural network to provide vital information about each image inside the loss functions.

We'll start by defining the three individual loss functions, as they are the core of the neural style transfer engine.

Content Loss

The content loss measures how different two images are in terms of the subject matter and overall placement of their content. Two images that contain similar-looking scenes (e.g., a photo of a row of buildings and another photo of the same buildings taken in different light from a different angle) should have a smaller loss than two images that contain completely different scenes. Simply comparing the pixel values of the two images won't do, because even in two distinct images of the same scene, we wouldn't expect individual pixel values to be similar. We don't really want the content loss to care about the values of individual pixels; we'd rather that it scores images based on the presence and approximate position of higher-level features such as *buildings, sky, or river*.

We've seen this concept before. It's the whole premise behind deep learning—a neural network trained to recognize the content of an image naturally learns higher-level features at deeper layers of the network by combining simpler features from previous layers. Therefore, what we need is a deep neural network that has already been successfully trained to identify the content of an image, so that we can tap into a deep layer of the network to extract the high-level features of a given input image. If we measure the mean squared error between this output for the base image and the current combined image, we have our content loss function!

The pretrained network that we shall be using is called VGG19. This is a 19-layer convolutional neural network that has been trained to classify images into one thousand object categories on more than one million images from the ImageNet dataset. A diagram of the network is shown in [Figure 5-15](#).

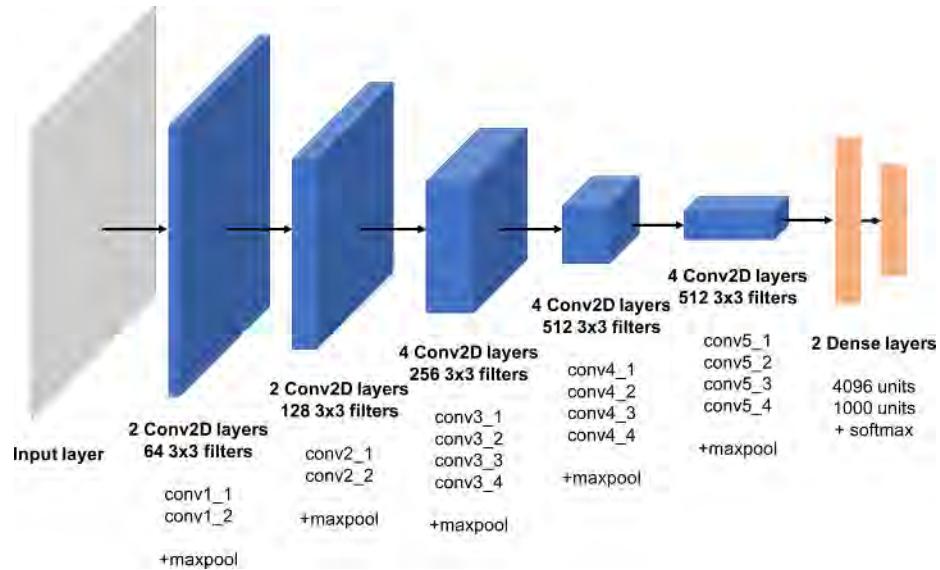


Figure 5-15. The VGG19 model

[Example 5-9](#) is a code snippet that calculates the content loss between two images, adapted from the neural style transfer example in the [official Keras repository](#). If you want to re-create this technique and experiment with the parameters, I suggest working from this repository as a starting point.

Example 5-9. The content loss function

```
from keras.applications import vgg19 ❶
from keras import backend as K

base_image_path = '/path_to_images/base_image.jpg'
style_reference_image_path = '/path_to_images/styled_image.jpg'

content_weight = 0.01

base_image = K.variable(preprocess_image(base_image_path)) ❷
style_reference_image = K.variable(preprocess_image(style_reference_image_path))
combination_image = K.placeholder((1, img_nrows, img_ncols, 3))

input_tensor = K.concatenate([base_image,
                            style_reference_image,
                            combination_image], axis=0) ❸

model = vgg19.VGG19(input_tensor=input_tensor,
                    weights='imagenet', include_top=False) ❹
```

```

outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
layer_features = outputs_dict['block5_conv2'] ⑤

base_image_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :] ⑥

def content_loss(content, gen):
    return K.sum(K.square(gen - content))

content_loss = content_weight * content_loss(base_image_features
                                              , combination_features) ⑦

```

- ➊ The Keras library contains a pretrained VGG19 model that can be imported.
- ➋ We define two Keras variables to hold the base image and style image and a placeholder that will contain the generated combined image.
- ➌ The input tensor to the VGG19 model is a concatenation of the three images.
- ➍ Here we create an instance of the VGG19 model, specifying the input tensor and the weights that we would like to preload. The `include_top = False` parameter specifies that we do not need to load the weights for the final dense layers of the networks that result in the classification of the image. This is because we are only interested in the preceding convolutional layers, which capture the high-level features of an input image, not the actual probabilities that the original model was trained to output.
- ➎ The layer that we use to calculate the content loss is the second convolutional layer of the fifth block. Choosing a layer at a shallower or deeper point in the network affects how the loss function defines “content” and therefore alters the properties of the generated combined image.
- ➏ Here we extract the base image features and combined image features from the input tensor that has been fed through the VGG19 network.
- ➐ The content loss is the sum of squares distance between the outputs of the chosen layer for both images, multiplied by a weighting parameter.

Style Loss

Style loss is slightly more difficult to quantify—how can we measure the similarity in style between two images?

The solution given in the neural style transfer paper is based on the idea that images that are similar in style typically have the same pattern of correlation between feature maps in a given layer. We can see this more clearly with an example.

Suppose in the VGG19 network we have some layer where one channel has learned to identify parts of the image that are colored green, another channel has learned to identify spikiness, and another has learned to identify parts of the image that are brown.

The output from these channels (feature maps) for three inputs is shown in Figure 5-16.

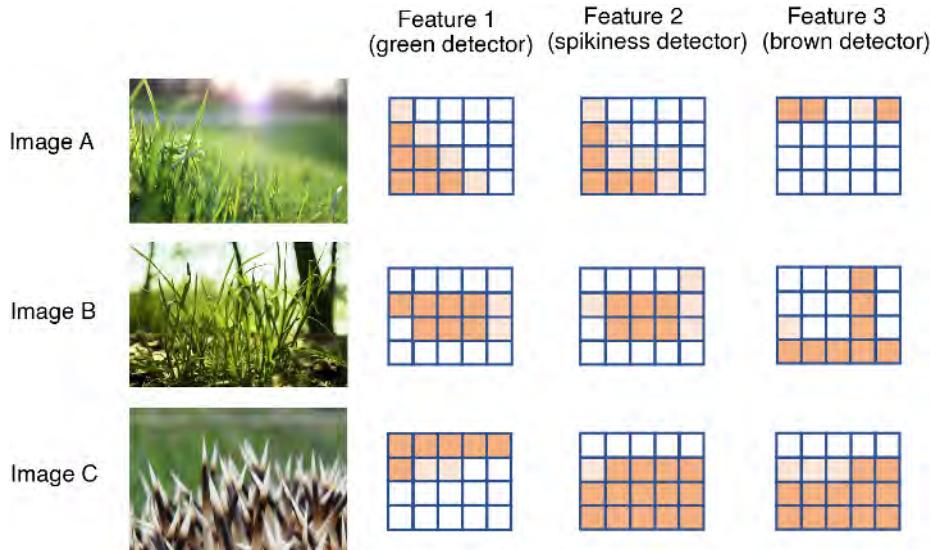


Figure 5-16. The output from three channels (feature maps) for three given input images—the darker orange colors represent larger values

We can see that A and B are similar in style—both are *grassy*. Image C is slightly different in style to images A and B. If we look at the feature maps, we can see that the *green* and *spikiness* channels often fire strongly together at the same spatial point in images A and B, but not in image C. Conversely, the *brown* and *spikiness* channels are often activated together at the same point in image C, but not in images A and B. To numerically measure how much two feature maps are jointly activated together, we can flatten them and calculate the dot product.¹⁴ If the resulting value is high, the feature maps are highly correlated; if the value is low, the feature maps are not correlated.

¹⁴ To calculate the dot product between two vectors, multiply the values of the two vectors in each position and sum the results.

We can define a matrix that contains the dot product between all possible pairs of features in the layer. This is called a *Gram matrix*. Figure 5-17 shows the Gram matrices for the three features, for each of the images.

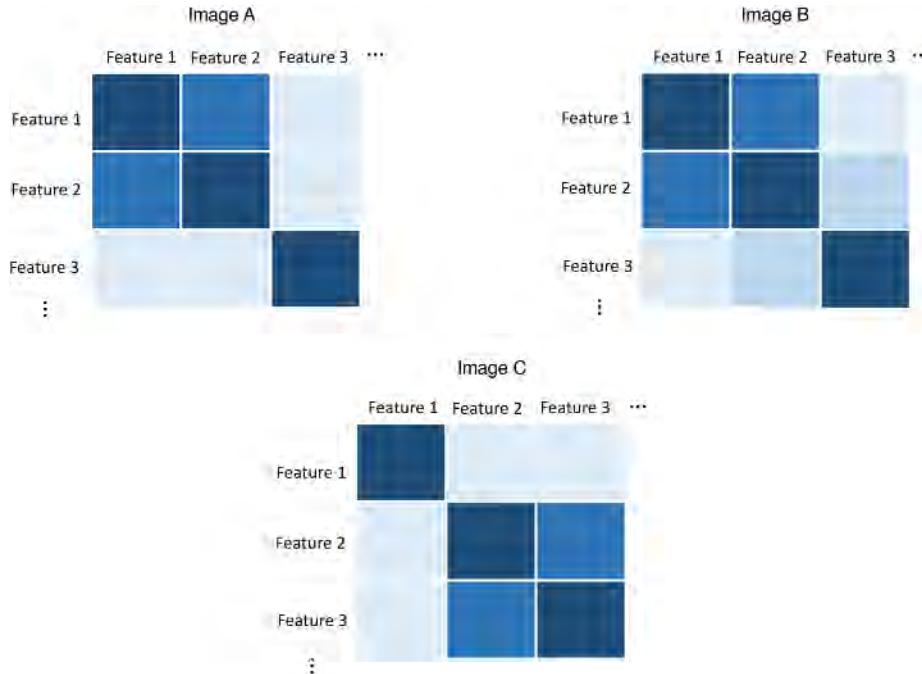


Figure 5-17. Parts of the Gram matrices for the three images—the darker blue colors represent larger values

It is clear that images A and B, which are similar in style, have similar Gram matrices for this layer. Even though their content may be very different, the Gram matrix—a measure of correlation between all pairs of features in the layer—is similar.

Therefore to calculate the style loss, all we need to do is calculate the Gram matrix (GM) for a set of layers throughout the network for both the base image and the combined image and compare their similarity using sum of squared errors. Algebraically, the style loss between the base image (S) and the generated image (G) for a given layer (l) of size M_l (height x width) with N_l channels can be written as follows:

$$L_{GM}(S, G, l) = \frac{1}{4N_l^2 M_l^2} \sum_{ij} (GM[l](S)_{ij} - GM[l](G)_{ij})^2$$

Notice how this is scaled to account for the number of channels (N_l) and size of the layer (M_l). This is because we calculate the overall style loss as a weighted sum across

several layers, all of which have different sizes. The total style loss is then calculated as follows:

$$L_{style}(S, G) = \sum_{l=0}^L w_l L_{GM}(S, G, l)$$

In Keras, the style loss calculations can be coded as shown in [Example 5-10](#).¹⁵

Example 5-10. The style loss function

```
style_loss = 0.0

def gram_matrix(x):
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram

def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_nrows * img_ncols
    return K.sum(K.square(S - C)) / (4.0 * (channels ** 2) * (size ** 2))

feature_layers = ['block1_conv1', 'block2_conv1',
                  'block3_conv1', 'block4_conv1',
                  'block5_conv1'] ①

for layer_name in feature_layers:
    layer_features = outputs_dict[layer_name]
    style_reference_features = layer_features[1, :, :, :] ②
    combination_features = layer_features[2, :, :, :]
    sl = style_loss(style_reference_features, combination_features)
    style_loss += (style_weight / len(feature_layers)) * sl ③
```

- ① The style loss is calculated over five layers—the first convolutional layer in each of the five blocks of the VGG19 model.
- ② Here we extract the style image features and combined image features from the input tensor that has been fed through the VGG19 network.
- ③ The style loss is scaled by a weighting parameter and the number of layers that it is calculated over.

¹⁵ Source: GitHub, <http://bit.ly/2FlVU0P>.

Total Variance Loss

The total variance loss is simply a measure of noise in the combined image. To judge how noisy an image is, we can shift it one pixel to the right and calculate the sum of the squared difference between the translated and original images. For balance, we can also do the same procedure but shift the image one pixel down. The sum of these two terms is the total variance loss.

Example 5-11 shows how we can code this in Keras.¹⁶

Example 5-11. The variance loss function

```
def total_variation_loss(x):
    a = K.square(
        x[:, :img_nrows - 1, :img_ncols - 1, :] - x[:, 1:, :img_ncols - 1, :]) ①
    b = K.square(
        x[:, :img_nrows - 1, :img_ncols - 1, :] - x[:, :img_nrows - 1, 1:, :]) ②
    return K.sum(K.pow(a + b, 1.25))

tv_loss = total_variation_weight * total_variation_loss(combination_image) ③

loss = content_loss + style_loss + tv_loss ④
```

- ① The squared difference between the image and the same image shifted one pixel down.
- ② The squared difference between the image and the same image shifted one pixel to the right.
- ③ The total variance loss is scaled by a weighting parameter.
- ④ The overall loss is the sum of the content, style, and total variance losses.

Running the Neural Style Transfer

The learning process involves running gradient descent to minimize this loss function, with respect to the pixels in the combined image. The full code for this is included in the, *neural_style_transfer.py* script included as part of the official Keras repository. Example 5-12 is a code snippet showing the training loop.

¹⁶ Source: GitHub, <http://bit.ly/2FlVU0P>.

Example 5-12. The training loop for the neural style transfer model

```
from scipy.optimize import fmin_l_bfgs_b

iterations = 1000
x = preprocess_image(base_image_path) ①

for i in range(iterations):
    x, min_val, info = fmin_l_bfgs_b(②
        evaluator.loss ③
        , x.flatten()
        , fprime=evaluator.grads ③
        , maxfun=20
    )
```

- ① The process is initialized with the base image as the starting *combined* image.
- ② At each iteration we pass the current combined image (flattened) into a optimization function, `fmin_l_bfgs_b` from the `scipy.optimize` package, that performs one gradient descent step according to the L-BFGS-B algorithm.
- ③ Here, `evaluator` is an object that contains methods that calculate the overall loss, as described previously, and gradients of the loss with respect to the input image.

Analysis of the Neural Style Transfer Model

Figure 5-18 shows the output of the neural style transfer process at three different stages in the learning process, with the following parameters:

- `content_weight: 1`
- `style_weight: 100`
- `total_variation_weight: 20`



Figure 5-18. Output from the neural style transfer process at 1, 200, and 400 iterations

We can see that with each training step, the algorithm becomes stylistically closer to the style image and loses the detail of the base image, while retaining the overall content structure.

There are many ways to experiment with this architecture. You can try changing the weighting parameters in the loss function or the layer that is used to determine the content similarity, to see how this affects the combined output image and training speed. You can also try decaying the weight given to each layer in the style loss function, to bias the model toward transferring finer or coarser style features.

Summary

In this chapter, we have explored two different ways to generate novel artwork: CycleGAN and neural style transfer.

The CycleGAN methodology allows us to train a model to learn the general style of an artist and transfer this over to a photograph, to generate output that looks as if the artist had painted the scene in the photo. The model also gives us the reverse process for free, converting paintings into realistic photographs. Crucially, paired images from each domain aren't required for a CycleGAN to work, making it an extremely powerful and flexible technique.

The neural style transfer technique allows us to transfer the style of a single image onto a base image, using a cleverly chosen loss function that penalizes the model for straying too far from the content of the base image and artistic style of the style image, while retaining a degree of smoothness to the output. This technique has been commercialized by many high-profile apps to blend a user's photographs with a given set of stylistic paintings.

In the next chapter we shall move away from image-based generative modeling to a domain that presents new challenges: text-based generative modeling.

CHAPTER 6

Write

In this chapter we shall explore methods for building generative models on text data. There are several key differences between text and image data that mean that many of the methods that work well for image data are not so readily applicable to text data. In particular:

- Text data is composed of discrete chunks (either characters or words), whereas pixels in an image are points in a continuous color spectrum. We can easily make a green pixel more blue, but it is not obvious how we should go about making the word *cat* more like the word *dog*, for example. This means we can easily apply backpropagation to image data, as we can calculate the gradient of our loss function with respect to individual pixels to establish the direction in which pixel colors should be changed to minimize the loss. With discrete text data, we can't apply backpropagation in the usual manner, so we need to find a way around this problem.
- Text data has a time dimension but no spatial dimension, whereas image data has two spatial dimensions but no time dimension. The order of words is highly important in text data and words wouldn't make sense in reverse, whereas images can usually be flipped without affecting the content. Furthermore, there are often long-term sequential dependencies between words that need to be captured by the model: for example, the answer to a question or carrying forward the context of a pronoun. With image data, all pixels can be processed simultaneously.
- Text data is highly sensitive to small changes in the individual units (words or characters). Image data is generally less sensitive to changes in individual pixel units—a picture of a house would still be recognizable as a house even if some pixels were altered. However, with text data, changing even a few words can drastically alter the meaning of the passage, or make it nonsensical. This makes it

very difficult to train a model to generate coherent text, as every word is vital to the overall meaning of the passage.

- Text data has a rules-based grammatical structure, whereas image data doesn't follow set rules about how the pixel values should be assigned. For example, it wouldn't make grammatical sense in any context to write "The cat sat on the having." There are also semantic rules that are extremely difficult to model; it wouldn't make sense to say "I am in the beach," even though grammatically, there is nothing wrong with this statement.

Good progress has been made in text modeling, but solutions to the above problems are still ongoing areas of research. We'll start by looking at one of the most utilized and established models for generating sequential data such as text, the recurrent neural network (RNN), and in particular, the long short-term memory (LSTM) layer. In this chapter we will also explore some new techniques that have led to promising results in the field of question-answer pair generation.

First, a trip to the local prison, where the inmates have formed a literary society...

The Literary Society for Troublesome Miscreants

Edward Sopp hated his job as a prison warden. He spent his days watching over the prisoners and had no time to follow his true passion of writing short stories. He was running low on inspiration and needed to find a way to generate new content.

One day, he came up with a brilliant idea that would allow him to produce new works of fiction in his style, while also keeping the inmates occupied—he would get the inmates to collectively write the stories for him! He branded the new society the LSTM (Literary Society for Troublesome Miscreants).

The prison is particularly strange because it only consists of one large cell, containing 256 prisoners. Each prisoner has an opinion on how Edward's current story should continue. Every day, Edward posts the latest word from his novel into the cell, and it is the job of the inmates to individually update their opinions on the current state of the story, based on the new word and the opinions of the inmates from the previous day.

Each prisoner uses a specific thought process to update their own opinion, which involves balancing information from the new incoming word and other prisoners' opinions with their own prior beliefs. First, they decide how much of yesterday's opinion they wish to forget, using the information from the new word and the opinions of other prisoners in the cell. They also use this information to form new thoughts and decide how much of this they want to mix into the old beliefs that they have chosen to carry forward from the previous day. This then forms the prisoner's new opinion for the day.

However, the prisoners are secretive and don't always tell their fellow inmates everything that they believe. They also use the latest chosen word and the opinions of the other inmates to decide how much of their opinion they wish to disclose.

When Edward wants the cell to generate the next word in the sequence, the prisoners tell their disclosable opinions to the somewhat dense guard at the door, who combines this information to ultimately decide on the next word to be appended to the end of the novel. This new word is then fed back into the cell as usual, and the process continues until the full story is completed.

To train the inmates and the guard, Edward feeds short sequences of words that he has written previously into the cell and monitors if the inmates' chosen next word is correct. He updates them on their accuracy, and gradually they begin to learn how to write stories in his own unique style.

After many iterations of this process, Edward finds that the system has become quite accomplished at generating realistic-looking text. While it is somewhat lacking in semantic structure, it certainly displays similar characteristics to his previous stories.

Satisfied with the results, he publishes a collection of the generated tales in his new book, entitled *E. Sopp's Fables*.

Long Short-Term Memory Networks

The story of Mr. Sopp and his crowdsourced fables is an analogy for one of the most utilized and successful deep learning techniques for sequential data such as text: the *long short-term memory* (LSTM) network.

An LSTM network is a particular type of *recurrent neural network* (RNN). RNNs contain a recurrent layer (or *cell*) that is able to handle sequential data by making its own output at a particular timestep form part of the input to the next timestep, so that information from the past can affect the prediction at the current timestep. We say *LSTM network* to mean a neural network with an LSTM recurrent layer.

When RNNs were first introduced, recurrent layers were very simple and consisted solely of a tanh operator that ensured that the information passed between timesteps was scaled between -1 and 1. However, this was shown to suffer from the vanishing gradient problem and didn't scale well to long sequences of data.

LSTM cells were first introduced in 1997 in a paper by Sepp Hochreiter and Jürgen Schmidhuber.¹ In the paper, the authors describe how LSTMs do not suffer from the same vanishing gradient problem experienced by vanilla RNNs and can be trained on

¹ Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory," *Neural Computation* 9 (1997): 1735–1780, <http://bit.ly/2In7NnH>.

sequences that are hundreds of timesteps long. Since then, the LSTM architecture has been adapted and improved, and variations such as *gated recurrent units* (GRUs) are now widely utilized and available as layers in Keras.

Let's begin by taking a look at how to build a very simple LSTM network in Keras that can generate text in the style of Aesop's Fables.

Your First LSTM Network

As usual, you first need to get set up with the data.

You can download a [collection of Aesop's Fables](#) from [Project Gutenberg](#). This is a collection of free ebooks that can be downloaded as plain text files. This is a great resource for sourcing data that can be used to train text-based deep learning models.

To download the data, from inside the book repository run the following command:

```
bash ./scripts/download_gutenberg_data.sh 11339 aesop
```

Let's now take a look at the steps we need to take in order to get the data in the right shape to train an LSTM network. The code is contained in the *06_01_lstm_text_train.ipynb* notebook in the book repository.

Tokenization

The first step is to clean up and tokenize the text. *Tokenization* is the process of splitting the text up into individual units, such as words or characters.

How you tokenize your text will depend on what you are trying to achieve with your text generation model. There are pros and cons to using both word and character tokens, and your choice will affect how you need to clean the text prior to modeling and the output from your model.

If you use word tokens:

- All text can be converted to lowercase, to ensure capitalized words at the start of sentences are tokenized the same way as the same words appearing in the middle of a sentence. In some cases, however, this may not be desirable; for example, some proper nouns, such as names or places, may benefit from remaining capitalized so that they are tokenized independently.
- The text *vocabulary* (the set of distinct words in the training set) may be very large, with some words appearing very sparsely or perhaps only once. It may be wise to replace sparse words with a token for *unknown word*, rather than including them as separate tokens, to reduce the number of weights the neural network needs to learn.

- Words can be *stemmed*, meaning that they are reduced to their simplest form, so that different tenses of a verb remained tokenized together. For example, *browse*, *browsing*, *browses*, and *browsed* would all be stemmed to *brows*.
- You will need to either tokenize the punctuation, or remove it altogether.
- Using word tokenization means that the model will never be able to predict words outside of the training vocabulary.

If you use character tokens:

- The model may generate sequences of characters that form new words outside of the training vocabulary—this may be desirable in some contexts, but not in others.
- Capital letters can either be converted to their lowercase counterparts, or remain as separate tokens.
- The vocabulary is usually much smaller when using character tokenization. This is beneficial for model training speed as there are fewer weights to learn in the final output layer.

For this example, we'll use lowercase word tokenization, without word stemming. We'll also tokenize punctuation marks, as we would like the model to predict when it should end sentences or open/close speech marks, for example. Finally, we'll replace the multiple newlines between stories with a block of *new story* characters, |||||||||. This way, when we generate text using the model, we can seed the model with this block of characters, so that the model knows to start a new story from scratch.

The code in [Example 6-1](#) cleans and tokenizes the text.

Example 6-1. Tokenization

```
import re
from keras.preprocessing.text import Tokenizer

filename = "./data/aesop/data.txt"

with open(filename, encoding='utf-8-sig') as f:
    text = f.read()

seq_length = 20
start_story = '| ' * seq_length

# CLEANUP
text = text.lower()
text = start_story + text
text = text.replace('\n\n\n\n\n', start_story)
```

```

text = text.replace('\n', ' ')
text = re.sub(' +', ' ', text).strip()
text = text.replace('..', '.')

text = re.sub('(!"#$%&()*+,./:;<=>?@[\\]^_`{|}~])', r' \1 ', text)
text = re.sub('\\s{2,}', ' ', text)

# TOKENIZATION
tokenizer = Tokenizer(char_level = False, filters = '')
tokenizer.fit_on_texts([text])
total_words = len(tokenizer.word_index) + 1
token_list = tokenizer.texts_to_sequences([text])[0]

```

An extract of the raw text after cleanup is shown in [Figure 6-1](#).

'||||||| the fox and the grapes . a hungry fox saw some fine bunches of grapes hanging from a vine that was trained along a high trellis , and did his best to reach them by jumping as high as he could into the air . but it was all in vain , for they were just out of reach : so he gave up trying , and walked away with an air of dignity and unconcern , remarking , " i thought those grapes were ripe , but i see now they are quite sour . " ||||||| the goose that laid the golden eggs . a man and his wife had the good fortune to possess a goose which laid a golden egg every day . lucky though they were , they soon began to think they were not getting rich fast enough , and , imagining the bird must be made of gold inside , they decided to kill it in order to secure the whole store of precious metal at once . but when they cut it open they found it was just like any other goose . thus , they neither got rich all at once , as they had hoped , nor enjoyed any longer the daily addition to their wealth . much wants more and loses all . ||||||| the cat and the mice . there was once a house that was overrun with mice . a cat heard of this , and said to herself , " that's the place for me , " and off she went and took up her quarters in the house , and caught the mice one by one and ate them . at last the mice could stand it no longer , and they determined to take to their holes and stay there . " that's awkward , " said the cat to herself : " the only thing to do is to coax them out by a trick . " so she considered a while , and then climbed up the wall and let herself hang down by her hind legs from a peg , and pretended to be dead . by and by

Figure 6-1. The text after cleanup

In [Figure 6-2](#), we can see the dictionary of tokens mapped to their respective indices and also a snippet of tokenized text, with the corresponding words shown in green.

<code>tokenizer.word_index</code>	<code>token_list</code>
' ': 1,	1,
',': 2,	3, the
'the': 3,	56, fox
'and': 4,	4, and
'.': 5,	3, the
'a': 6,	940, grapes
'to': 7,	5, .
'": 8,	6, a
'of': 9,	382, hungry
'he': 10,	56, fox
'his': 11,	94, saw
'was': 12,	

Figure 6-2. The mapping dictionary between words and indices (left) and the text after tokenization (right)

Building the Dataset

Our LSTM network will be trained to predict the next word in a sequence, given a sequence of words preceding this point. For example, we could feed the model the tokens for *the greedy cat and the* and would expect the model to output a suitable next word (e.g., *dog*, rather than *in*).

The sequence length that we use to train the model is a parameter of the training process. In this example we choose to use a sequence length of 20, so we split the text into 20-word chunks. A total of 50,416 such sequences can be constructed, so our training dataset X is an array of shape [50416, 20].

The response variable for each sequence is the subsequent word, one-hot encoded into a vector of length 4,169 (the number of distinct words in the vocabulary). Therefore, our response y is a binary array of shape [50416, 4169].

The dataset generation step can be achieved with the code in [Example 6-2](#).

Example 6-2. Generating the dataset

```
import numpy as np
from keras.utils import np_utils

def generate_sequences(token_list, step):
    X = []
    y = []
```

```

for i in range(0, len(token_list) - seq_length, step):
    X.append(token_list[i:i + seq_length])
    y.append(token_list[i + seq_length])

y = np_utils.to_categorical(y, num_classes = total_words)

num_seq = len(X)
print('Number of sequences:', num_seq, "\n")

return X, y, num_seq

step = 1
seq_length = 20
X, y, num_seq = generate_sequences(token_list, step)

X = np.array(X)
y = np.array(y)

```

The LSTM Architecture

The architecture of the overall model is shown in [Figure 6-3](#). The input to the model is a sequence of integer tokens and the output is the probability of each word in the vocabulary appearing next in the sequence. To understand how this works in detail, we need to introduce two new layer types, `Embedding` and `LSTM`.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, None)	0
embedding_1 (Embedding)	(None, None, 100)	416900
lstm_1 (LSTM)	(None, 256)	365568
dense_1 (Dense)	(None, 4169)	1071433
<hr/>		
Total params:	1,853,901	
Trainable params:	1,853,901	
Non-trainable params:	0	

Figure 6-3. LSTM model architecture

The Embedding Layer

An embedding layer is essentially a lookup table that converts each token into a vector of length `embedding_size` ([Figure 6-4](#)). The number of weights learned by this layer is therefore equal to the size of the vocabulary, multiplied by `embedding_size`.

token	Embedding				
1	-0.13	0.45	...	0.13	-0.04
2	0.22	0.56	...	0.24	-0.63
...
4168	0.16	-0.70	...	-0.35	1.02
4169	-0.98	-0.45	...	-0.15	-0.52

Figure 6-4. An embedding layer is a lookup table for each integer token

The Input layer passes a tensor of integer sequences of shape [batch_size, seq_length] to the Embedding layer, which outputs a tensor of shape [batch_size, seq_length, embedding_size]. This is then passed on to the LSTM layer (Figure 6-5).

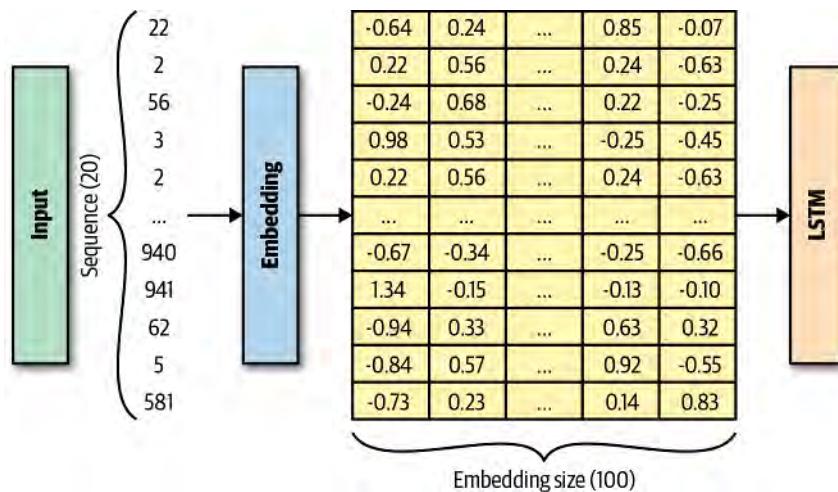


Figure 6-5. A single sequence as it flows through an embedding layer

We embed each integer token into a continuous vector because it enables the model to learn a representation for each word that is able to be updated through backpropagation. We could also just one-hot encode each input token, but using an embedding layer is preferred because it makes the embedding itself trainable, thus giving the model more flexibility in deciding how to embed each token to improve model performance.

The LSTM Layer

To understand the LSTM layer, we must first look at how a general recurrent layer works.

A recurrent layer has the special property of being able to process sequential input data $[x_1, \dots, x_n]$. It consists of a cell that updates its *hidden state*, h , as each element of the sequence x_t is passed through it, one timestep at a time. The hidden state is a vector with length equal to the number of *units* in the cell—it can be thought of as the cell’s current understanding of the sequence. At timestep t , the cell uses the previous value of the hidden state h_{t-1} together with the data from the current timestep x_t to produce an updated hidden state vector h_t . This recurrent process continues until the end of the sequence. Once the sequence is finished, the layer outputs the final hidden state of the cell, h_n , which is then passed on to the next layer of the network. This process is shown in [Figure 6-6](#).

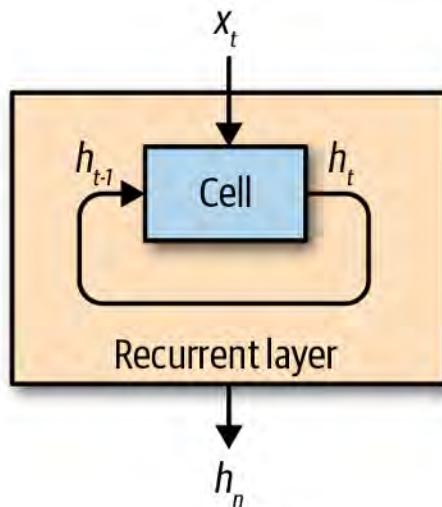


Figure 6-6. A simple diagram of a recurrent layer

To explain this in more detail, let’s unroll the process so that we can see exactly how a single sequence is fed through the layer ([Figure 6-7](#)).

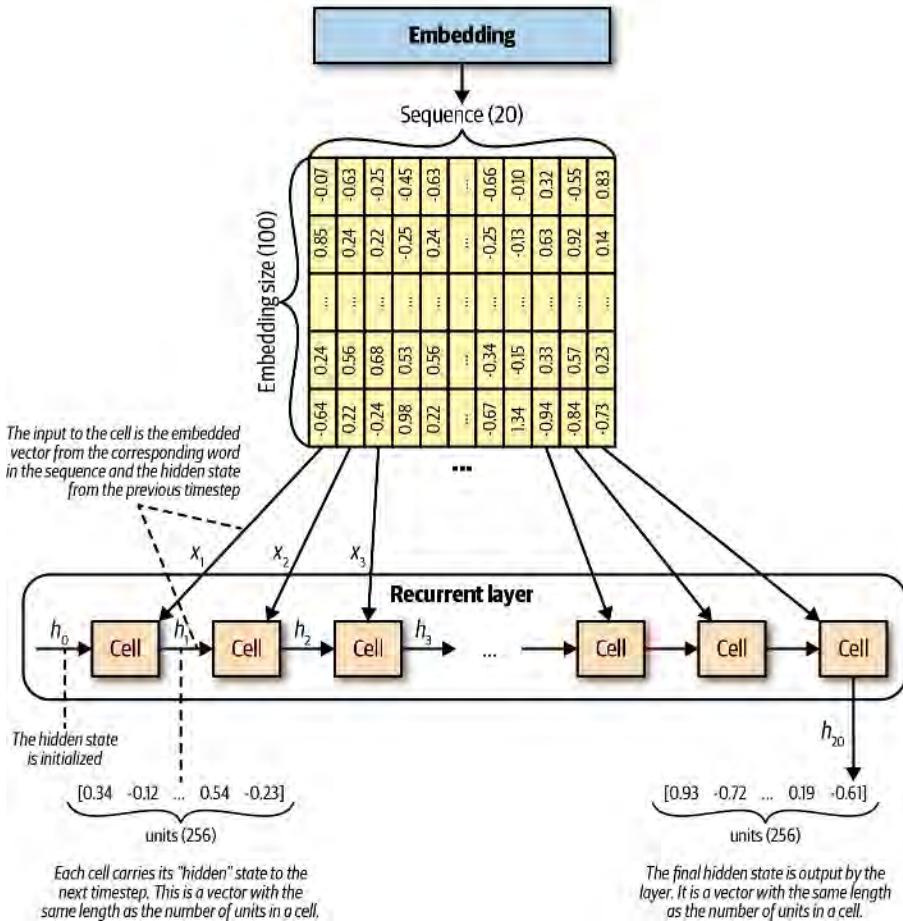


Figure 6-7. How a single sequence flows through a recurrent layer

Here, we represent the recurrent process by drawing a copy of the cell at each timestep and show how the hidden state is constantly being updated as it flows through the cells. We can clearly see how the previous hidden state is blended with the current sequential data point (i.e., the current embedded word vector) to produce the next hidden state. The output from the layer is the final hidden state of the cell, after each word in the input sequence has been processed. It's important to remember that all of the cells in this diagram share the same weights (as they are really the same cell). There is no difference between this diagram and Figure 6-6; it's just a different way of drawing the mechanics of a recurrent layer.



The fact that the output from the cell is called a *hidden* state is an unfortunate naming convention—it's not really hidden, and you shouldn't think of it as such. Indeed, the last hidden state is the overall output from the layer, and we will be making use of the fact that we can access the hidden state at each individual timestep later in this chapter.

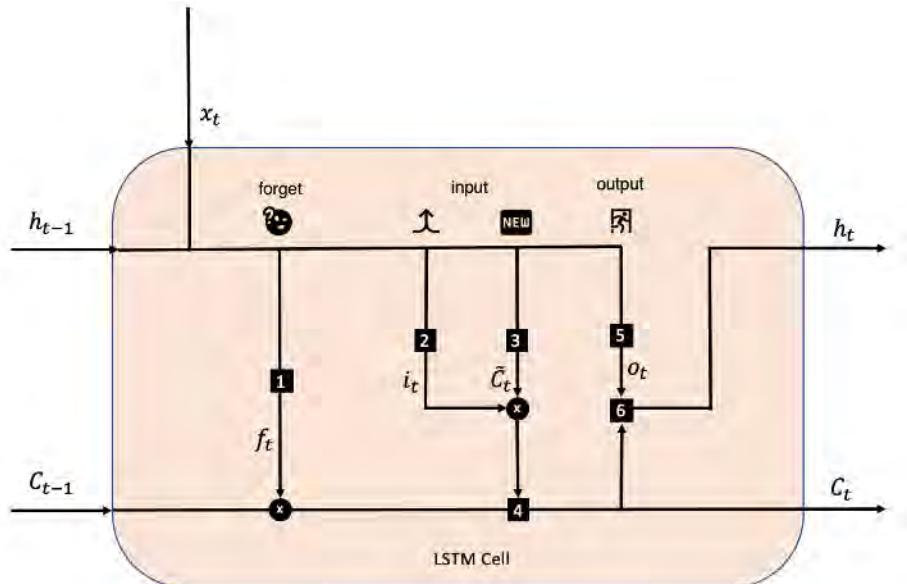
The LSTM Cell

Now that we have seen how a generic recurrent layer works, let's take a look inside an individual LSTM cell.

The job of the LSTM cell is to output a new hidden state, h_t , given its previous hidden state, h_{t-1} , and the current word embedding, x_t . To recap, the length of h_t is equal to the number of units in the LSTM. This is a parameter that is set when you define the layer and has nothing to do with the length of the sequence. Make sure you do not confuse the term *cell* with *unit*. There is one cell in an LSTM layer that is defined by the number of units it contains, in the same way that the prisoner cell from our earlier story contained many prisoners. We often draw a recurrent layer as a chain of cells unrolled, as it helps to visualize how the hidden state is updated at each timestep.

An LSTM cell maintains a cell state, C_t , which can be thought of as the cell's internal beliefs about the current status of the sequence. This is distinct from the hidden state, h_t , which is ultimately output by the cell after the final timestep. The cell state is the same length as the hidden state (the number of units in the cell).

Let's look more closely at a single cell and how the hidden state is updated ([Figure 6-8](#)).



- 1 $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
- 2 $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
- 3 $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$
- 4 $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
- 5 $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
- 6 $h_t = o_t * \tanh(C_t)$

Figure 6-8. An LSTM cell

The hidden state is updated in six steps:

1. The hidden state of the previous timestep, h_{t-1} , and the current word embedding, x_t , are concatenated and passed through the *forget* gate. This gate is simply a dense layer with weights matrix W_f , bias b_f , and a sigmoid activation function. The resulting vector, f_t , has a length equal to the number of units in the cell and contains values between 0 and 1 that determine how much of the previous cell state, C_{t-1} , should be retained.
2. The concatenated vector is also passed through an *input* gate which, like the forget gate, is a dense layer with weights matrix W_i , bias b_i , and a sigmoid activation function. The output from this gate, i_t , has length equal to the number of units in

the cell and contains values between 0 and 1 that determine how much new information will be added to the previous cell state, C_{t-1} .

3. The concatenated vector is passed through a dense layer with weights matrix W_C , bias b_C and a tanh activation function to generate a vector \tilde{C}_t that contains the new information that the cell wants to consider keeping. It also has length equal to the number of units in the cell and contains values between -1 and 1.
4. f_t and C_{t-1} are multiplied element-wise and added to the element-wise multiplication of i_t and \tilde{C}_t . This represents forgetting parts of the previous cell state and then adding new relevant information to produce the updated cell state, C_t .
5. The original concatenated vector is also passed through an *output* gate: a dense layer with weights matrix W_o , bias b_o , and a sigmoid activation. The resulting vector, o_t , has a length equal to the number of units in the cell and stores values between 0 and 1 that determine how much of the updated cell state, C_t , to output from the cell.
6. o_t is multiplied element-wise with the updated cell state C_t after a tanh activation has been applied to produce the new hidden state, h_t .

The code to build the LSTM network is given in [Example 6-3](#).

Example 6-3. Building the LSTM network

```
from keras.layers import Dense, LSTM, Input, Embedding, Dropout
from keras.models import Model
from keras.optimizers import RMSprop

n_units = 256
embedding_size = 100

text_in = Input(shape = (None,))
x = Embedding(total_words, embedding_size)(text_in)
x = LSTM(n_units)(x)
x = Dropout(0.2)(x)
text_out = Dense(total_words, activation = 'softmax')(x)

model = Model(text_in, text_out)

opti = RMSprop(lr = 0.001)
model.compile(loss='categorical_crossentropy', optimizer=opti)

epochs = 100
batch_size = 32
model.fit(X, y, epochs=epochs, batch_size=batch_size, shuffle = True)
```

Generating New Text

Now that we have compiled and trained the LSTM network, we can start to use it to generate long strings of text by applying the following process:

1. Feed the network with an existing sequence of words and ask it to predict the following word.
2. Append this word to the existing sequence and repeat.

The network will output a set of probabilities for each word that we can sample from. Therefore, we can make the text generation stochastic, rather than deterministic. Moreover, we can introduce a `temperature` parameter to the sampling process to indicate how deterministic we would like the process to be.

This is achieved with the code in [Example 6-4](#).

Example 6-4. Generating text with an LSTM network

```
def sample_with_temp(preds, temperature=1.0): ❶
    # helper function to sample an index from a probability array
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probs = np.random.multinomial(1, preds, 1)
    return np.argmax(probs)

def generate_text(seed_text, next_words, model, max_sequence_len, temp):
    output_text = seed_text
    seed_text = start_story + seed_text ❷

    for _ in range(next_words):
        token_list = tokenizer.texts_to_sequences([seed_text])[0] ❸
        token_list = token_list[-max_sequence_len:] ❹
        token_list = np.reshape(token_list, (1, max_sequence_len))

        probs = model.predict(token_list, verbose=0)[0] ❺
        y_class = sample_with_temp(probs, temperature = temp) ❻

        output_word = tokenizer.index_word[y_class] if y_class > 0 else ''
        if output_word == "|": ❼
            break

        seed_text += output_word + ' ' ❽
        output_text += output_word + ' ' ❽

    return output_text
```

- ❶ This function weights the logits with a `temperature` scaling factor before reapplying the softmax function. A temperature close to zero makes the sampling more deterministic (i.e., the word with highest probability is very likely to be chosen), whereas a temperature of 1 means each word is chosen with the probability output by the model.
- ❷ The seed text is a string of words that you would like to give the model to start the generation process (it can be blank). This is prepended with the block of characters we use to indicate the start of a story (|||||||).
- ❸ The words are converted to a list of tokens.
- ❹ Only the last `max_sequence_len` tokens are kept. The LSTM layer can accept any length of sequence as input, but the longer the sequence is the more time it will take to generate the next word, so the sequence length should be capped.
- ❺ The model outputs the probabilities of each word being next in the sequence.
- ❻ The probabilities are passed through the sampler to output the next word, parameterized by `temperature`.
- ❼ If the output word is the start story token, we stop generating any more words as this is the model telling us it wants to end this story and start the next one!
- ❽ Otherwise, we append the new word to the seed text, ready for the next iteration of the generative process.

Let's take a look at this in action, at two different `temperature` values ([Figure 6-9](#)).

Temperature 0.2

the ass and his lion . a certain man who had an ass and a stag at them in a good . " do , " said the other saying , " you but the ass to be so . to come and you are in the way of ever see how you are , in the man of a lion . " the lion replied , " you not very much to me , and i will be a eyes upon you , and , as that you were for the good of a on tha t men will be had all the dog to be a lion . as the cock came , and the ground for it in a very much to wolf , " you take my oh , if you will do not get me in any time tha t a very way of some day . " well again , which had very well on ly a man , he turned up and said to the , " but i see what you see how i are day : but you not one of them to be not well ? " . he saw what be a of master in the ass was said to him , " the other , you are a man of me , and i were so . " the wolf was so much to the man and said to his master , " you , in the friend of you are ; and then he should be running up they found for you said he had a way to make them . " as the very way , for it was that he could not see a t the eagle and said to him , " i will be a lion and i have was one day , and you know that you go again , but come a nd see what you are of the better . "

Temperature 1.0

the great man and the time . a sheep master fell on a head and had been in a other to get them . so the ass , caught a young horse and was by me , and so he got himself into a horse than the lion of their put out ; and why , he found a lion in a ass . you must go , he they we not only could not a bull by one day the tortoise in gratitude in a place live of the very do wish you . so , as he could not the fox no one make the ass of the day , where in the well have w ould all she had been in a good first , but the other dog and then cock in a moment . i'm not a lion , who had " co me off , that do you see that you find that the lion , who they fell on to be very much more good good strength , and have to a mouse go . when the other who is same , who had no town mouse , and found a cat up out of a great much . " the better feed in the water , that she came to a bird . the eagle never do so good a frog or and drink ; so my maste r , saw that the master is was well and the ass at my a very good time . the ox turned too use for a way to have to h ertext ; but if the will very fox came to let him one as one as it , and man as he should there well on by the place and stood there . for the time the lamb ass in the ass , for his the lost these all , should found turned down . a us e of the hard went to make it , " and the never soon can be of out of a ass . " the master , so not much be a eagle n ever should be oh ! " presently , not to be a ass , who do not how were heavy : i'm not a do you , you get off in the water , and not come to we are very take the just so ever even in my master .

Figure 6-9. Example of LSTM-generated passages, at two different temperature values

There are a few things to note about these two passages. First, both are stylistically similar to a fable from the original training set. They both open with the familiar statement of the characters in the story, and generally the text within speech marks is more dialogue-like, using personal pronouns and prepared by the occurrence of the word *said*.

Second, the text generated at `temperature = 0.2` is less adventurous but more coherent in its choice of words than the text generated at `temperature = 1.0`, as lower temperature values result in more deterministic sampling.

Last, it is clear that neither flows particularly well as a story across multiple sentences, because the LSTM network cannot grasp the semantic meaning of the words that it is generating. In order to generate passages that have greater chance of being semantically reasonable, we can build a human-assisted text generator, where the model outputs the top 10 words with the highest probabilities and it is then ultimately up to a human to choose the next word from among this list. This is similar to predictive text on your mobile phone, where you are given the choice of a few words that might follow on from what you have already typed.

To demonstrate this, [Figure 6-10](#) shows the top 10 words with the highest probabilities to follow various sequences (not from the training set).

Noun	Verb
the fox and the stag . there was a 21.3% : time	the fox and the snake . one day a fox 49.9% : came
19.1% : lion	19.8% : saw
15.7% : man	8.0% : went
11.6% : ass	6.0% : ,
8.6% : good	5.5% : was
4.3% : fox	4.2% : were
2.0% : a	1.2% : said
1.8% : once	1.1% : go
1.7% : old	0.4% : had
1.3% : dogs	0.3% : put
Preposition	Adjective / Verb
the dog and the hare . a dog was lying 68.6% : in	the farmer and his sheep . a farmer was 17.2% : unable
21.6% : on	8.3% : afraid
6.7% : at	6.1% : so
0.6% : by	5.4% : good
0.5% : into	5.0% : lying
0.4% : from	4.6% : sitting
0.3% : to	4.2% : who
0.3% : with	2.4% : by
0.2% : about	2.3% : going
0.2% : for	2.3% : caught
Article	Punctuation
the eagle and the sea .. 89.4% : an	the lion said , 98.0% : "
4.1% : the	1.1% : and
1.5% : just	0.4% : that
1.2% : thus	0.1% : the
0.7% : two	0.1% : much
0.7% : presently	0.0% : you
0.6% : a	0.0% : do
0.5% : there	0.0% : time
0.4% : jupiter	0.0% : as
0.1% : at	0.0% : a
	the lion said , and 84.4% : the 5.2% : you 3.4% : that 3.0% : if 0.3% : well 0.2% : " 0.2% : i 0.2% : my 0.2% : we 0.2% : very

Figure 6-10. Distribution of word probabilities following various sequences

The model is able to generate a suitable distribution for the next most likely word across a range of contexts. For example, even though the model was never told about parts of speech such as nouns, verbs, adjectives, and prepositions, it is generally able to separate words into these classes and use them in a way that is grammatically correct. It can also guess that the article that begins a story about an eagle is more likely to be *an*, rather than *a*.

The punctuation example from Figure 6-10 shows how the model is also sensitive to subtle changes in the input sequence. In the first passage (*the lion said ,*), the model guesses that speech marks follow with 98% likelihood, so that the clause precedes the spoken dialogue. However, if we instead input the next word as *and*, it is able to understand that speech marks are now unlikely, as the clause is more likely to have superseded the dialogue and the sentence will more likely continue as descriptive prose.

RNN Extensions

The network in the preceding section is a simple example of how an LSTM network can be trained to learn how to generate text in a given style. In this section we will explore several extensions to this idea.

Stacked Recurrent Networks

The network we just looked at contained a single LSTM layer, but we can also train networks with stacked LSTM layers, so that deeper features can be learned from the text.

To achieve this, we set the `return_sequences` parameter within the first LSTM layer to `True`. This makes the layer output the hidden state from every timestep, rather than just the final timestep. The second LSTM layer can then use the hidden states from the first layer as its input data. This is shown in [Figure 6-11](#), and the overall model architecture is shown in [Figure 6-12](#).

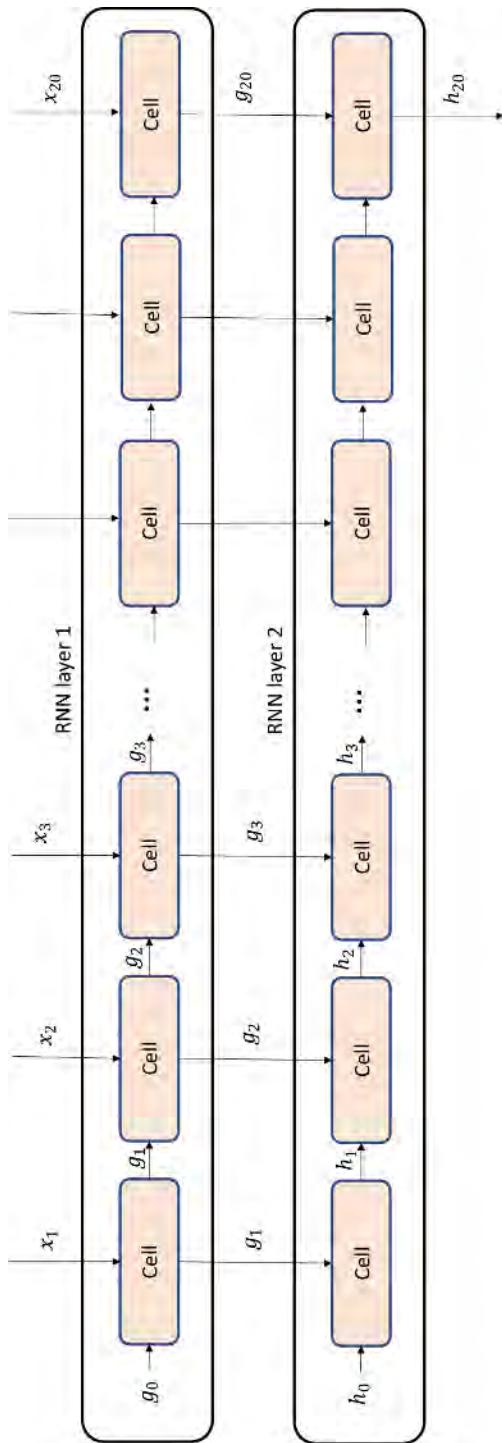


Figure 6-11. Diagram of a multilayer RNN: g_t denotes hidden states of the first layer and h_t denotes hidden states of the second layer

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, None)	0
embedding_1 (Embedding)	(None, None, 100)	416900
lstm_1 (LSTM)	(None, None, 256)	365568
lstm_2 (LSTM)	(None, 256)	525312
dropout_1 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 4169)	1071433
<hr/>		
Total params: 2,379,213		
Trainable params: 2,379,213		
Non-trainable params: 0		

Figure 6-12. A stacked LSTM network

The code to build the stacked LSTM network is given in [Example 6-5](#).

Example 6-5. Building a stacked LSTM network

```
text_in = Input(shape = (None,))
embedding = Embedding(total_words, embedding_size)
x = embedding(text_in)
x = LSTM(n_units, return_sequences = True)(x)
x = LSTM(n_units)(x)
x = Dropout(0.2)(x)
text_out = Dense(total_words, activation = 'softmax')(x)

model = Model(text_in, text_out)
```

Gated Recurrent Units

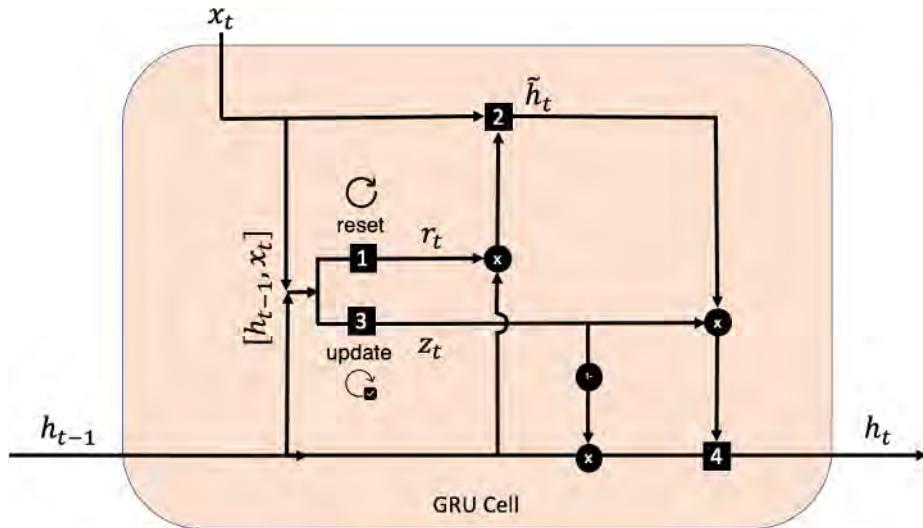
Another type of commonly used RNN layer is the *gated recurrent unit* (GRU).² The key differences from the LSTM unit are as follows:

1. The *forget* and *input* gates are replaced by *reset* and *update* gates.

² Kyunghyun Cho et al., “Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation,” 3 June 2014, <https://arxiv.org/abs/1406.1078>.

2. There is no *cell state* or *output gate*, only a *hidden state* that is output from the cell.

The hidden state is updated in four steps, as illustrated in [Figure 6-13](#).



- 1 $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$
- 2 $\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$
- 3 $z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$
- 4 $h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$

Figure 6-13. A single GRU cell

The process is as follows:

1. The hidden state of the previous timestep, h_{t-1} , and the current word embedding, x_t , are concatenated and used to create the *reset* gate. This gate is a dense layer, with weights matrix W_r and a sigmoid activation function. The resulting vector, r_t , has a length equal to the number of units in the cell and stores values between 0 and 1 that determine how much of the previous hidden state, h_{t-1} , should be carried forward into the calculation for the new beliefs of the cell.
2. The reset gate is applied to the hidden state, h_{t-1} , and concatenated with the current word embedding, x_t . This vector is then fed to a dense layer with weights matrix W and a tanh activation function to generate a vector, \tilde{h}_t , that stores the

- new beliefs of the cell. It has length equal to the number of units in the cell and stores values between -1 and 1 .
3. The concatenation of the hidden state of the previous timestep, h_{t-1} , and the current word embedding, x_t , are also used to create the *update* gate. This gate is a dense layer with weights matrix W_z and a sigmoid activation. The resulting vector, z_t , has length equal to the number of units in the cell and stores values between 0 and 1 , which are used to determine how much of the new beliefs, \tilde{h}_t , to blend into the current hidden state, h_{t-1} .
 4. The new beliefs of the cell \tilde{h}_t and the current hidden state, h_{t-1} , are blended in a proportion determined by the update gate, z_t , to produce the updated hidden state, h_t , that is output from the cell.

Bidirectional Cells

For prediction problems where the entire text is available to the model at inference time, there is no reason to process the sequence only in the forward direction—it could just as well be processed backwards. A *bidirectional* layer takes advantage of this by storing two sets of hidden states: one that is produced as a result of the sequence being processed in the usual forward direction and another that is produced when the sequence is processed backwards. This way, the layer can learn from information both preceding and succeeding the given timestep.

In Keras, this is implemented as a wrapper around a recurrent layer, as shown here:

```
layer = Bidirectional(GRU(100))
```

The hidden states in the resulting layer are vectors of length equal to double the number of units in the wrapped cell (a concatenation of the forward and backward hidden states). Thus, in this example the hidden states of the layer are vectors of length 200.

Encoder–Decoder Models

So far, we have looked at using LSTM networks for generating the continuation of an existing text sequence. We have seen how a single LSTM layer can process the data sequentially to update a hidden state that represents the layer’s current understanding of the sequence. By connecting the final hidden state to a dense layer, the network can output a probability distribution for the next word.

For some tasks, the goal isn’t to predict the single next word in the existing sequence; instead we wish to predict a completely different sequence of words that is in some way related to the input sequence. Some examples of this style of task are:

Language translation

The network is fed a text string in the source language and the goal is to output the text translated into a target language.

Question generation

The network is fed a passage of text and the goal is to generate a viable question that could be asked about the text.

Text summarization

The network is fed a long passage of text and the goal is to generate a short summary of the passage.

For these kinds of problems, we can use a type of network known as an *encoder-decoder*. We have already seen one type of encoder–decoder network in the context of image generation: the variational autoencoder. For sequential data, the encoder–decoder process works as follows:

1. The original input sequence is summarized into a single vector by the encoder RNN.
2. This vector is used to initialize the decoder RNN.
3. The hidden state of the decoder RNN at each timestep is connected to a dense layer that outputs a probability distribution over the vocabulary of words. This way, the decoder can generate a novel sequence of text, having been initialized with a representation of the input data produced by the encoder.

This process is shown in [Figure 6-14](#), in the context of translation between English and German.

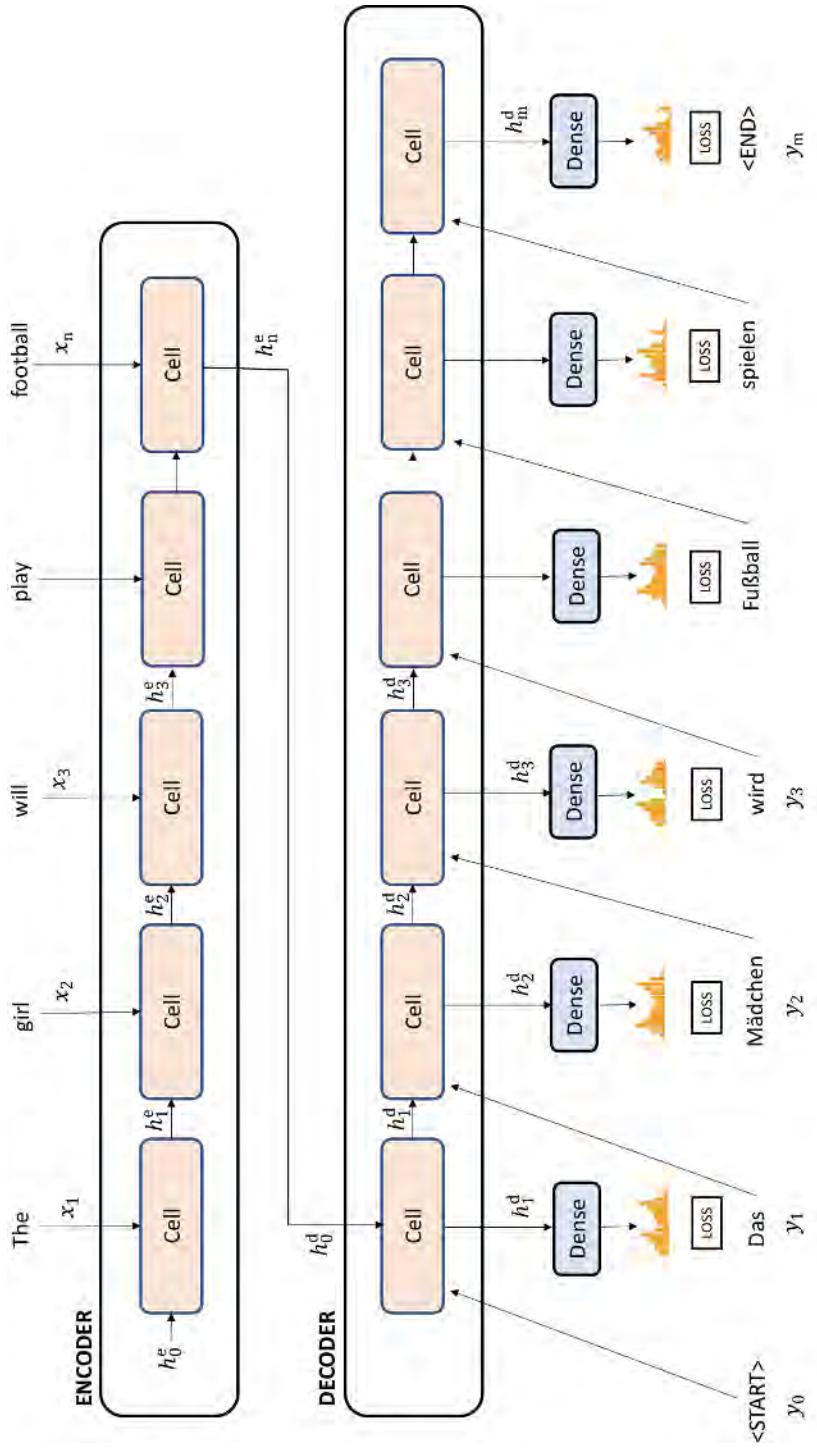


Figure 6-14. An encoder-decoder network

The final hidden state of the encoder can be thought of as a representation of the entire input document. The decoder then transforms this representation into sequential output, such as the translation of the text into another language, or a question relating to the document.

During training, the output distribution produced by the decoder at each timestep is compared against the true next word, to calculate the loss. The decoder doesn't need to sample from these distributions to generate words during the training process, as the subsequent cell is fed with the ground-truth next word, rather than a word sampled from the previous output distribution. This way of training encoder-decoder networks is known as *teacher forcing*. We can imagine that the network is a student sometimes making erroneous distribution predictions, but no matter what the network outputs at each timestep, the teacher provides the correct response as input to the network for the attempt at the next word.

A Question and Answer Generator

We're now going to put everything together and build a model that can generate question and answer pairs from a block of text. This project is inspired by the [qgen-workshop TensorFlow codebase](#) and the model proposed by Tong Wang, Xingdi Yuan, and Adam Trischler.³

The model consists of two parts:

- An RNN that identifies candidate answers from the block of text
- An encoder-decoder network that generates a suitable question, given one of the candidate answers highlighted by the RNN

For example, consider the following opening to a passage of text about a football match:

```
The winning goal was scored by 23-year-old striker Joe Bloggs during the match  
between Arsenal and Barcelona .  
Arsenal recently signed the striker for 50 million pounds . The next match is in  
two weeks time, on July 31st 2005 . "
```

We would like our first network to be able to identify potential answers such as:

```
"Joe Bloggs"  
"Arsenal"  
"Barcelona"  
"50 million pounds"  
"July 31st 2005"
```

³ Tong Wang, Xingdi Yuan, and Adam Trischler, "A Joint Model for Question Answering and Question Generation," 5 July 2017, <https://arxiv.org/abs/1706.01450>.

And our second network should be able to generate a question, given each of the answers, such as:

```
"Who scored the winning goal?"  
"Who won the match?"  
"Who were Arsenal playing?"  
"How much did the striker cost?"  
"When is the next match?"
```

Let's first take a look at the dataset we shall be using in more detail.

A Question-Answer Dataset

We'll be using the Maluuba NewsQA dataset, which you can download by following the set of instructions [on GitHub](#).

The resulting *train.csv*, *test.csv*, and *dev.csv* files should be placed in the *./data/qa/* folder inside the book repository. These files all have the same column structure, as follows:

`story_id`

A unique identifier for the story.

`story_text`

The text of the story (e.g., “The winning goal was scored by 23-year-old striker Joe Bloggs during the match...”).

`question`

A question that could be asked about the story text (e.g., “How much did the striker cost?”).

`answer_token_ranges`

The token positions of the answer in the story text (e.g., 24:27). There might be multiple ranges (comma separated) if the answer appears multiple times in the story.

This raw data is processed and tokenized so that it is able to be used as input to our model. After this transformation, each observation in the training set consists of the following five features:

`document_tokens`

The tokenized story text (e.g., [1, 4633, 7, 66, 11, ...]), clipped/padded with zeros to be of length `max_document_length` (a parameter).

`question_input_tokens`

The tokenized question (e.g., [2, 39, 1, 52, ...]), padded with zeros to be of length `max_question_length` (another parameter).

`question_output_tokens`

The tokenized question, offset by one timestep (e.g., [39, 1, 52, 1866, ...], padded with zeros to be of length `max_question_length`.

`answer_masks`

A binary mask matrix having shape `[max_answer_length, max_document_length]`. The $[i, j]$ value of the matrix is 1 if the i th word of the answer to the question is located at the j th word of the document and 0 otherwise.

`answer_labels`

A binary vector of length `max_document_length` (e.g., [0, 1, 1, 0, ...]). The i th element of the vector is 1 if the i th word of the document could be considered part of an answer and 0 otherwise.

Let's now take a look at the model architecture that is able to generate question-answer pairs from a given block of text.

Model Architecture

Figure 6-15 shows the overall model architecture that we will be building. Don't worry if this looks intimidating! It's only built from elements that we have seen already and we will be walking through the architecture step by step in this section.

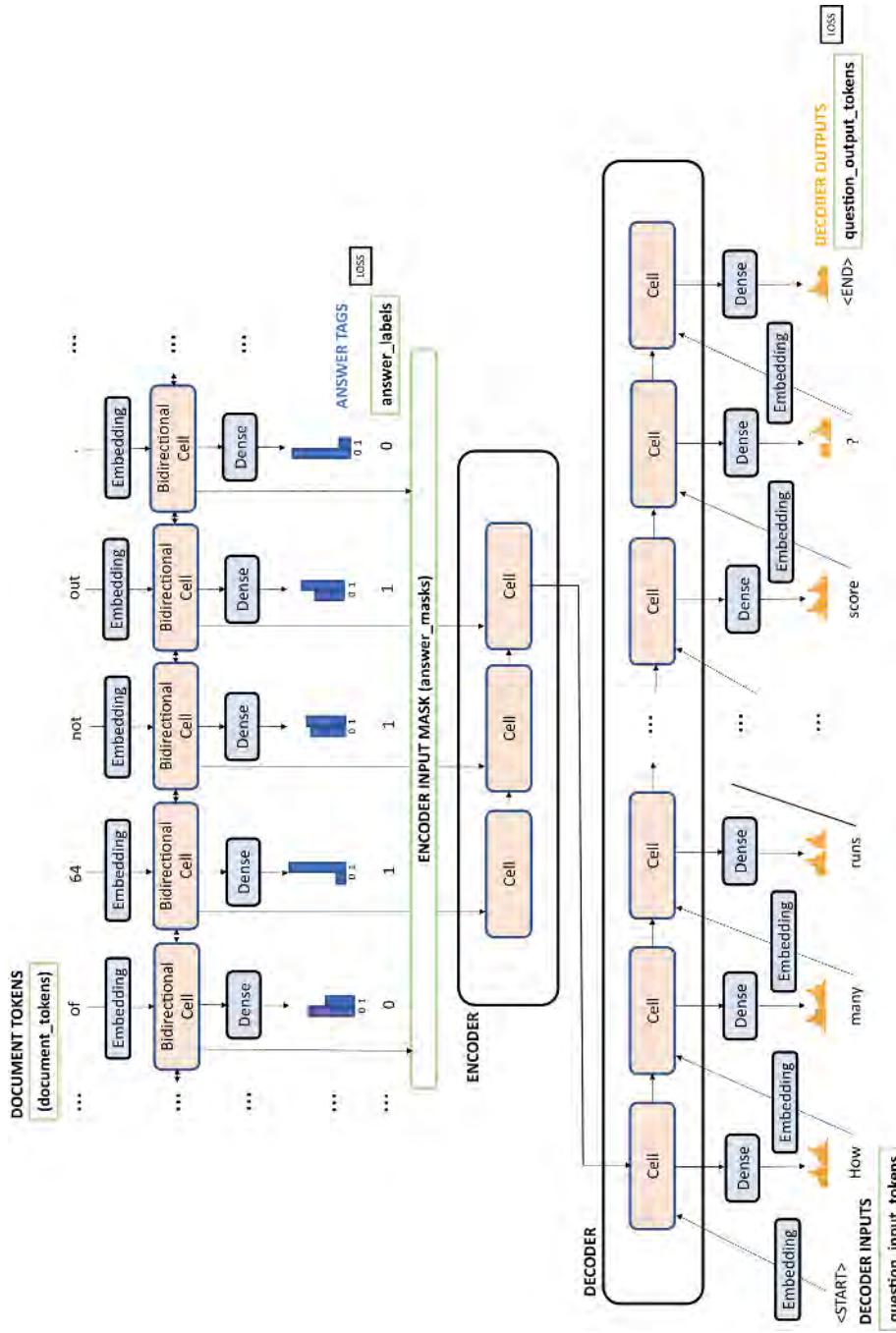


Figure 6-15. The architecture for generating question-answer pairs; input data is shown in green bordered boxes

Let's start by taking a look at the Keras code that builds the part of the model at the top of the diagram, which predicts if each word in the document is part of an answer or not. This code is shown in [Example 6-6](#). You can also follow along with the accompanying notebook in the book repository, `06_02_qa_train.ipynb`.

Example 6-6. Model architecture for generating question–answer pairs

```
from keras.layers import Input, Embedding, GRU, Bidirectional, Dense, Lambda
from keras.models import Model, load_model
import keras.backend as K
from qgen.embedding import glove

#### PARAMETERS ####

VOCAB_SIZE = glove.shape[0] # 9984
EMBEDDING_DIMENS = glove.shape[1] # 100

GRU_UNITS = 100
DOC_SIZE = None
ANSWER_SIZE = None
Q_SIZE = None

document_tokens = Input(shape=(DOC_SIZE,), name="document_tokens") ❶

embedding = Embedding(input_dim = VOCAB_SIZE, output_dim = EMBEDDING_DIMENS
                      , weights=[glove], mask_zero = True, name = 'embedding') ❷
document_emb = embedding(document_tokens)

answer_outputs = Bidirectional(GRU(GRU_UNITS, return_sequences=True)
                               , name = 'answer_outputs')(document_emb) ❸
answer_tags = Dense(2, activation = 'softmax'
                     , name = 'answer_tags')(answer_outputs) ❹
```

- ❶ The document tokens are provided as input to the model. Here, we use the variable `DOC_SIZE` to describe the size of this input, but the variable is actually set to `None`. This is because the architecture of the model isn't dependent on the length of the input sequence—the number of cells in the layer will adapt to equal the length of the input sequence, so we don't need to specify it explicitly.
- ❷ The embedding layer is initialized with *GloVe* word vectors (explained in the following sidebar).
- ❸ The recurrent layer is a bidirectional GRU that returns the hidden state at each timestep.

- ④ The output Dense layer is connected to the hidden state at each timestep and has only two units, with a softmax activation, representing the probability that each word is part of an answer (1) or is not part of an answer (0).

GloVe Word Vectors

The embedding layer is initialized with a set of pretrained word embeddings, rather than random vectors as we have seen previously. These word vectors have been created as part of the Stanford **GloVe (“Global Vectors”)** project, which uses unsupervised learning to obtain representative vectors for a large set of words.

These vectors have many beneficial properties, such as the similarity of vectors between connected words. For example, the vector between embeddings for the words *man* and *woman* is approximately the same as the vector between the words *king* and *queen*. It is as if the gender of the word is encoded into the latent space in which the word vectors exist. Initializing an embedding layer with GloVe is often better than training from scratch because a lot of the hard work of capturing the representation of a word has already been achieved by the GloVe training process. Your algorithm can then tweak the word embeddings to suit the particular context of your machine learning problem.

To work with the GloVe word vectors within this project, download the file *glove.6B.100d.txt* (6 billion words each with an embedding of length 100) from the GloVe project website and then run the following Python script from the book repository to trim this file to only include words that are present in the training corpus:

```
python ./utils/write.py
```

The second part of the model is the encoder-decoder network that takes a given answer and tries to formulate a matching question (the bottom part of [Figure 6-15](#)).

The Keras code for this part of the network is given in [Example 6-7](#).

Example 6-7. Model architecture for the encoder-decoder network that formulates a question given an answer

```
encoder_input_mask = Input(shape=(ANSWER_SIZE, DOC_SIZE)
                           , name="encoder_input_mask")①
encoder_inputs = Lambda(lambda x: K.batch_dot(x[0], x[1])
                        , name="encoder_inputs")([encoder_input_mask, answer_outputs])
encoder_cell = GRU(2 * GRU_UNITS, name = 'encoder_cell')(encoder_inputs) ②

decoder_inputs = Input(shape=(Q_SIZE,), name="decoder_inputs") ③
decoder_emb = embedding(decoder_inputs) ④
decoder_emb.trainable = False
decoder_cell = GRU(2 * GRU_UNITS, return_sequences = True, name = 'decoder_cell')
```

```

decoder_states = decoder_cell(decoder_emb, initial_state = [encoder_cell]) ⑤

decoder_projection = Dense(VOCAB_SIZE, name = 'decoder_projection'
    , activation = 'softmax', use_bias = False)
decoder_outputs = decoder_projection(decoder_states) ⑥

total_model = Model([document_tokens, decoder_inputs, encoder_input_mask]
    , [answer_tags, decoder_outputs])
answer_model = Model(document_tokens, [answer_tags])
decoder_initial_state_model = Model([document_tokens, encoder_input_mask]
    , [encoder_cell])

```

- ① The answer mask is passed as an input to the model—this allows us to pass the hidden states from a single answer range through to the encoder–decoder. This is achieved using a Lambda layer.
- ② The encoder is a GRU layer that is fed the hidden states for the given answer range as input data.
- ③ The input data to the decoder is the question matching the given answer range.
- ④ The question word tokens are passed through the same embedding layer used in the answer identification model.
- ⑤ The decoder is a GRU layer and is initialized with the final hidden state from the encoder.
- ⑥ The hidden states of the decoder are passed through a Dense layer to generate a distribution over the entire vocabulary for the next word in the sequence.

This completes our network for question–answer pair generation. To train the network, we pass the document text, question text, and answer masks as input data in batches and minimize the cross-entropy loss on both the answer position prediction and question word generation, weighted equally.

Inference

To test the model on an input document sequence that it has never seen before, we need to run the following process:

1. Feed the document string to the answer generator to produce sample positions for answers in the document.
2. Choose one of these answer blocks to carry forward to the encoder–decoder question generator (i.e., create the appropriate answer mask).

3. Feed the document and answer mask to the encoder to generate the initial state for the decoder.
4. Initialize the decoder with this initial state and feed in the <START> token to generate the first word of the question. Continue this process, feeding in generated words one by one until the <END> token is predicted by the model.

As discussed previously, during training the model uses teacher forcing to feed the ground-truth words (rather than the predicted next words) back into the decoder cell. However during inference the model must generate a question by itself, so we want to be able to feed the predicted words back into the decoder cell while retaining its hidden state.

One way we can achieve this is by defining an additional Keras model (`question_model`) that accepts the current word token and current decoder hidden state as input, and outputs the predicted next word distribution and updated decoder hidden state. This is shown in [Example 6-8](#).

Example 6-8. Inference models

```
decoder_inputs_dynamic = Input(shape=(1,), name="decoder_inputs_dynamic")
decoder_emb_dynamic = embedding(decoder_inputs_dynamic)
decoder_init_state_dynamic = Input(shape=(2 * GRU_UNITS,))
    , name = 'decoder_init_state_dynamic')
decoder_states_dynamic = decoder_cell(decoder_emb_dynamic
    , initial_state = [decoder_init_state_dynamic])
decoder_outputs_dynamic = decoder_projection(decoder_states_dynamic)

question_model = Model([decoder_inputs_dynamic, decoder_init_state_dynamic]
    , [decoder_outputs_dynamic, decoder_states_dynamic])
```

We can then use this model in a loop to generate the output question word by word, as shown in [Example 6-9](#).

Example 6-9. Generating question–answer pairs from a given document

```
test_data_gen = test_data()
batch = next(test_data_gen)
answer_preds = answer_model.predict(batch["document_tokens"])

idx = 0
start_answer = 37
end_answer = 39

answers = [[0] * len(answer_preds[idx])]
for i in range(start_answer, end_answer + 1):
    answers[idx][i] = 1
```

```

answer_batch = expand_answers(batch, answers)

next_decoder_init_state = decoder_initial_state_model.predict(
    [answer_batch['document_tokens'][[idx]], answer_batch['answer_masks'][[idx]]])

word_tokens = [START_TOKEN]
questions = [look_up_token(START_TOKEN)]

ended = False

while not ended:

    word_preds, next_decoder_init_state = question_model.predict(
        [word_tokens, next_decoder_init_state])

    next_decoder_init_state = np.squeeze(next_decoder_init_state, axis = 1)
    word_tokens = np.argmax(word_preds, 2)[0]

    questions.append(look_up_token(word_tokens[0]))

    if word_tokens[0] == END_TOKEN:
        ended = True

questions = ' '.join(questions)

```

Model Results

Sample results from the model are shown in [Figure 6-16](#) (see also the accompanying notebook in the book repository, [06_03_qa_analysis.ipynb](#)). The chart on the right shows the probability of each word in the document forming part of an answer, according to the model. These answer phrases are then fed to the question generator and the output of this model is shown on the lefthand side of the diagram (“Predicted Question”).

First, notice how the answer generator is able to accurately identify which words in the document are most likely to be contained in an answer. This is already quite impressive given that it has never seen this text before and also may not have seen certain words from the document that are included in the answer, such as *Bloggs*. It is able to understand from the context that this is likely to be the surname of a person and therefore likely to form part of an answer.

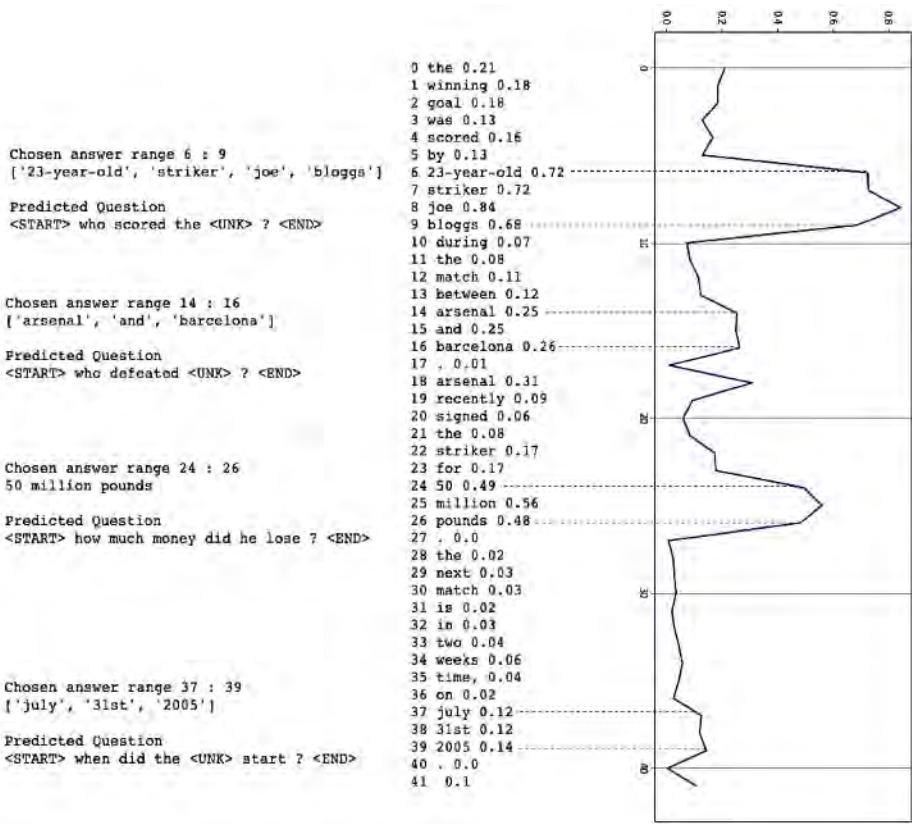


Figure 6-16. Sample results from the model

The encoder extracts the context from each of these possible answers, so that the decoder is able to generate suitable questions. It is remarkable that the encoder is able to capture that the person mentioned in the first answer, *23-year-old striker Joe Bloggs*, would probably have a matching question relating to his goal-scoring abilities, and is able to pass this context on to the decoder so that it can generate the question “who scored the <UNK> ?” rather than, for example, “who is the president ?”

The decoder has finished this question with the tag <UNK>, but not because it doesn’t know what to do next—it is predicting that the word that follows is likely to be from outside the core vocabulary. We shouldn’t be surprised that the model resorts to using the tag <UNK> in this context, as many of the niche words in the original corpus would be tokenized this way.

We can see that in each case, the decoder chooses the correct “type” of question—who, how much, or when—depending on the type of answer. There are still some problems though, such as asking *how much money did he lose ?* rather than *how much*

money was paid for the striker ?. This is understandable, as the decoder only has the final encoder state to work with and cannot reference the original document for extra information.

There are several extensions to encoder–decoder networks that improve the accuracy and generative power of the model. Two of the most widely used are *pointer networks*⁴ and *attention mechanisms*.⁵ Pointer networks give the model the ability to “point” at specific words in the input text to include in the generated question, rather than only relying on the words in the known vocabulary. This helps to solve the <UNK> problem mentioned earlier. We shall explore attention mechanisms in detail in the next chapter.

Summary

In this chapter we have seen how recurrent neural networks can be applied to generate text sequences that mimic a particular style of writing, and also generate plausible question–answer pairs from a given document.

We explored two different types of recurrent layer, long short-term memory and GRU, and saw how these cells can be stacked or made bidirectional to form more complex network architectures. The encoder–decoder architecture introduced in this chapter is an important generative tool as it allows sequence data to be condensed into a single vector that can then be decoded into another sequence. This is applicable to a range of problems aside from question–answer pair generation, such as translation and text summarization.

In both cases we have seen how it is important to understand how to transform unstructured text data to a structured format that can be used with recurrent neural network layers. A good understanding of how the shape of the tensor changes as data flows through the network is also pivotal to building successful networks, and recurrent layers require particular care in this regard as the time dimension of sequential data adds additional complexity to the transformation process.

In the next chapter we will see how many of the same ideas around RNNs can be applied to another type of sequential data: music.

⁴ Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly, “Pointer Networks,” 9 July 2015, <https://arxiv.org/abs/1506.03134>.

⁵ Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” 1 September 2014, <https://arxiv.org/abs/1409.0473>.

CHAPTER 7

Compose

Alongside visual art and creative writing, musical composition is another core act of creativity that we consider to be uniquely human.

For a machine to compose music that is pleasing to our ear, it must master many of the same technical challenges that we saw in the previous chapter in relation to text. In particular, our model must be able to learn from and re-create the sequential structure of music and must also be able to choose from a discrete set of possibilities for subsequent notes.

However, musical generation presents additional challenges that are not required for text generation, namely pitch and rhythm. Music is often polyphonic—that is, there are several streams of notes played simultaneously on different instruments, which combine to create harmonies that are either dissonant (clashing) or consonant (harmonious). Text generation only requires us to handle a single stream of text, rather than the parallel streams of chords that are present in music.

Also, text generation can be handled one word at a time. We must consider carefully if this is an appropriate way to process musical data, as much of the interest that stems from listening to music is in the interplay between different rhythms across the ensemble. A guitarist might play a flurry of quicker notes while the pianist holds a longer sustained chord, for example. Therefore, generating music note by note is complex, because we often do not want all the instruments to change note simultaneously.

We will start this chapter by simplifying the problem to focus on music generation for a single (monophonic) line of music. We will see that many of the RNN techniques from the previous chapter on text generation can also be used for music generation as the two tasks share many common themes. This chapter will also introduce the attention mechanism that will allow us to build RNNs that are able to choose which

previous notes to focus on in order to predict which notes will appear next. Lastly, we'll tackle the task of polyphonic music generation and explore how we can deploy an architecture based around GANs to create music for multiple voices.

Preliminaries

Anyone tackling the task of music generation must first have a basic understanding of musical theory. In this section we'll go through the essential notation required to read music and how we can represent this numerically, in order to transform music into the input data required to train a generative model.

We'll work through the notebook *07_01_notation_compose.ipynb* in the book repository. Another excellent resource for getting started with music generation using Python is Sigurður Skúli's [blog post](#) and accompanying [GitHub repository](#).

The raw dataset that we shall be using is a set of MIDI files for the Cello Suites by J.S. Bach. You can use any dataset you wish, but if you want to work with this dataset, you can find instructions for downloading the MIDI files in the notebook.

To view and listen to the music generated by the model, you'll need some software that can produce musical notation. [MuseScore](#) is a great tool for this purpose and can be downloaded for free.

Musical Notation

We'll be using the Python library `music21` to load the MIDI files into Python for processing. [Example 7-1](#) shows how to load a MIDI file and visualize it ([Figure 7-1](#)), both as a score and as structured data.

Example 7-1. Importing a MIDI file

```
from music21 import converter

dataset_name = 'cello'
filename = 'cs1-2all'
file = "./data/{}/{}.mid".format(dataset_name, filename)

original_score = converter.parse(file).chordify()
```

`original_score.show()`

beats 0 1 2 3 4 5 6 7 8

$J=250$ $J=77$

A barline

`original_score.show('text')`

```

{0.0} <music21.instrument.Violoncello Violoncello>
{0.0} <music21.tempo.MetronomeMark Quarter=250.0>
{0.0} <music21.key.Key of G major>
{0.0} <music21.meter.TimeSignature 4/4>
{0.0} <music21.note.Rest rest>
{3.5} <music21.tempo.MetronomeMark Quarter=77.0>
{3.75} <music21.chord.Chord B3>
{4.0} <music21.chord.Chord G2 D3 B3>
{5.0} <music21.chord.Chord B3>
{5.25} <music21.chord.Chord A3>
{5.5} <music21.chord.Chord Chord G3>
{5.75} <music21.chord.Chord Chord F#3>
{6.0} <music21.chord.Chord Chord G3>
{6.25} <music21.chord.Chord Chord D3>
{6.5} <music21.chord.Chord Chord B3>
{6.75} <music21.chord.Chord Chord F#3>
{7.0} <music21.chord.Chord Chord G3>
{7.25} <music21.chord.Chord Chord A3>
{7.5} <music21.chord.Chord Chord B3>
{7.75} <music21.chord.Chord Chord C4>
{8.0} <music21.chord.Chord D4>

```

The MIDI file starts with some metadata around the instrumentation, tempo, and key of the piece— we won't be using this information

This note starts on beat 4 of the piece [zero indexed]. It has duration of 1 beat (as the next note starts on beat 5) and consists of a chord of low G, D, and B.

This note starts on beat 6 of the piece. It has duration of a quarter of a beat (as the next note starts on beat 6.25) and consists of a single note —G.

This note starts a quarter of a beat before beat 8 of the piece, it has duration of a quarter of a beat and consists of a single note — high C.

Figure 7-1. Musical notation

We use the `chordify` method to squash all simultaneously played notes into chords within a single part, rather than them being split between many parts. Since this piece is performed by one instrument (a cello), we are justified in doing this, though sometimes we may wish to keep the parts separate to generate music that is polyphonic in nature. This presents additional challenges that we shall tackle later on in this chapter.

The code in [Example 7-2](#) loops through the score and extracts the pitch and duration for each note (and rest) in the piece into two lists. Individual notes in chords are separated by a dot, so that the whole chord can be stored as a single string. The number after each note name indicates the *octave* that the note is in—since the note names (A to G) repeat, this is needed to uniquely identify the pitch of the note. For example, G2 is an octave below G3.

Example 7-2. Extracting the data

```
notes = []
durations = []

for element in original_score.flat:

    if isinstance(element, chord.Chord):
        notes.append('.'.join(n.nameWithOctave for n in element.pitches))
        durations.append(element.duration.quarterLength)

    if isinstance(element, note.Note):
        if element.isRest:
            notes.append(str(element.name))
            durations.append(element.duration.quarterLength)
        else:
            notes.append(str(element.nameWithOctave))
            durations.append(element.duration.quarterLength)
```

The output from this process is shown in [Table 7-1](#).

The resulting dataset now looks a lot more like the text data that we have dealt with previously. The words are the *pitches*, and we should try to build a model that predicts the next pitch, given a sequence of previous pitches. The same idea can also be applied to the list of durations. Keras gives us the flexibility to be able to build a model that can handle the pitch and duration prediction simultaneously.

Table 7-1. The pitch and duration of each note, stored as lists

Duration	Pitch
0.25	B3
1.0	G2.D3.B3
0.25	B3
0.25	A3
0.25	G3
0.25	F#3
0.25	G3
0.25	D3
0.25	E3
0.25	F#3
0.25	G3
0.25	A3

Your First Music-Generating RNN

To create the dataset that will train the model, we first need to give each pitch and duration an integer value (Figure 7-2), exactly as we have done previously for each word in a text corpus. It doesn't matter what these values are as we shall be using an embedding layer to transform the integer lookup values into vectors.

```
note_to_int           duration_to_int
{'A2': 0,             {0: 0,
'A2.A3': 1,          Fraction(1, 12): 1,
'A2.B2': 2,          Fraction(1, 6): 2,
'A2.C3': 3,          0.25: 3,
'A2.D3': 4,          Fraction(1, 3): 4,
'A2.E-3': 5,          0.5: 5,
'A2.E3': 6,          Fraction(2, 3): 6,
'A2.E3.A3': 7,        0.75: 7,
'A2.E3.C#4': 8,      1.0: 8,
'A2.E3.C#4.A4': 9,    1.25: 9,
'A2.E3.C#4.E4': 10,   Fraction(4, 3): 10,
```

Figure 7-2. The integer lookup dictionaries for pitch and duration

We then create the training set by splitting the data into small chunks of 32 notes, with a response variable of the next note in the sequence (one-hot encoded), for both pitch and duration. One example of this is shown in Figure 7-3.

Figure 7-3. The inputs and outputs for the musical generative model

The model we will be building is a stacked LSTM network with an attention mechanism. In the previous chapter, we saw how we are able to stack LSTM layers by passing the hidden states of the previous layer as input to the next LSTM layer. Stacking layers in this way gives the model freedom to learn more sophisticated features from the data. In this section we will introduce the attention mechanism¹ that now forms an integral part of most sophisticated sequential generative models. It has ultimately given rise to the *transformer*, a type of model based entirely on attention that doesn't even require recurrent or convolutional layers. We will introduce the transformer architecture in more detail in [Chapter 9](#).

For now, let's focus on incorporating attention into a stacked LSTM network to try to predict the next note, given a sequence of previous notes.

Attention

The attention mechanism was originally applied to text translation problems—in particular, translating English sentences into French.

In the previous chapter, we saw how encoder–decoder networks can solve this kind of problem, by first passing the input sequence through an encoder to generate a

¹ Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," 1 September 2014, <https://arxiv.org/abs/1409.0473>.

context vector, then passing this vector through the decoder network to output the translated text. One observed problem with this approach is that the context vector can become a bottleneck. Information from the start of the source sentence can become diluted by the time it reaches the context vector, especially for long sentences. Therefore these kinds of encoder–decoder networks sometimes struggle to retain all the required information for the decoder to accurately translate the source.

As an example, suppose we want the model to translate the following sentence into German: *I scored a penalty in the football match against England*.

Clearly, the meaning of the entire sentence would be changed by replacing the word *scored* with *missed*. However, the final hidden state of the encoder may not be able to sufficiently retain this information, as the word *scored* appears early in the sentence.

The correct translation of the sentence is: *Ich habe im Fußballspiel gegen England einen Elfmeter erzielt*.

If we look at the correct German translation, we can see that the word for *scored* (*erzielt*) actually appears right at the end of the sentence! So not only would the model have to retain the fact that the penalty was scored rather than missed through the encoder, but also all the way through the decoder as well.

Exactly the same principle is true in music. To understand what note or sequence of notes is likely to follow from a particular given passage of music, it may be crucial to use information from far back in the sequence, not just the most recent information. For example, take the opening passage of the Prelude to Bach’s Cello Suite No. 1 (Figure 7-4).



Figure 7-4. The opening of Bach’s Cello Suite No. 1 (Prelude)

What note do you think comes next? Even if you have no musical training you may still be able to guess. If you said G (the same as the very first note of the piece), then you’d be correct. How did you know this? You may have been able to see that every bar and half bar starts with the same note and used this information to inform your decision. We want our model to be able to perform the same trick—in particular, we want it to not only care about the hidden state of the network *now*, but also to pay

particular attention to the hidden state of the network eight notes ago, when the previous low G was registered.

The attention mechanism was proposed to solve this problem. Rather than only using the final hidden state of the encoder RNN as the context vector, the attention mechanism allows the model to create the context vector as a weighted sum of the hidden states of the encoder RNN at each previous timestep. The attention mechanism is just a set of layers that converts the previous encoder hidden states and current decoder hidden state into the summation weights that generate the context vector.

If this sounds confusing, don't worry! We'll start by seeing how to apply an attention mechanism after a simple recurrent layer (i.e., to solve the problem of predicting the next note of Bach's Cello Suite No. 1), before we see how this extends to encoder-decoder networks, where we want to predict a whole sequence of subsequent notes, rather than just one.

Building an Attention Mechanism in Keras

First, let's remind ourselves of how a standard recurrent layer can be used to predict the next note given a sequence of previous notes. [Figure 7-5](#) shows how the input sequence (x_1, \dots, x_n) is fed to the layer one step at a time, continually updating the hidden state of the layer. The input sequence could be the note embeddings, or the hidden state sequence from a previous recurrent layer. The output from the recurrent layer is the final hidden state, a vector with the same length as the number of units. This can then be fed to a Dense layer with softmax output to predict a distribution for the next note in the sequence.

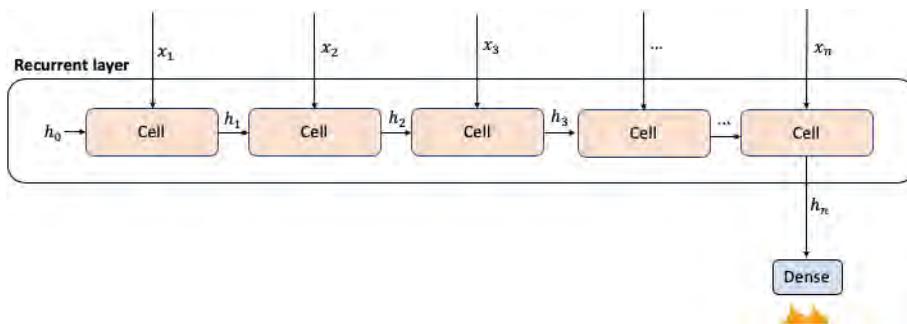
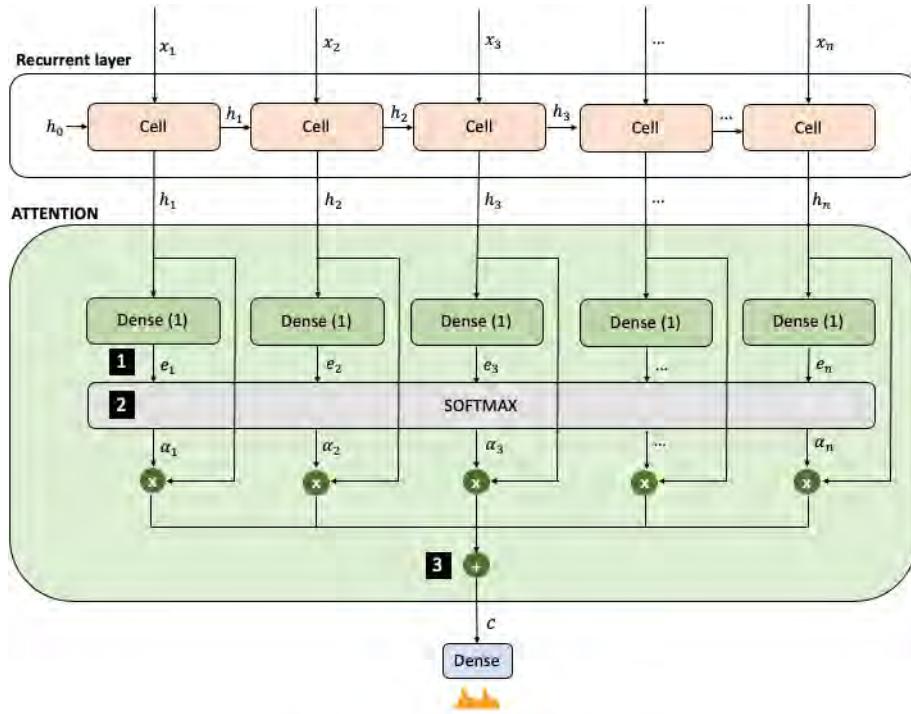


Figure 7-5. A recurrent layer for predicting the next note in a sequence, without attention

[Figure 7-6](#) shows the same network, but this time with an attention mechanism applied to the hidden states of the recurrent layer.



$$1 \quad e_j = a(\mathbf{h}_j) = \tanh(W \cdot \mathbf{h}_j)$$

$$2 \quad \alpha_j = [\text{softmax}(\mathbf{e})]_j = \frac{\exp(e_j)}{\sum_k \exp(e_k)}$$

$$3 \quad \mathbf{c} = \sum_j \alpha_j \mathbf{h}_j$$

Figure 7-6. A recurrent layer for predicting the next note in a sequence, with attention

Let's walk through this process step by step:

1. First, each hidden state h_j (a vector of length equal to the number of units in the recurrent layer) is passed through an *alignment* function a to generate a scalar, e_j . In this example, this function is simply a densely connected layer with one output unit and a tanh activation function.
2. Next, the softmax function is applied to the vector e_1, \dots, e_n to produce the vector of weights $\alpha_1, \dots, \alpha_n$.

3. Lastly, each hidden state vector h_j is multiplied by its respective weight α_j , and the results are then summed to give the context vector c (thus c has the same length as a hidden state vector).

The context vector can then be passed to a Dense layer with softmax output as usual, to output a distribution for the potential next note.

This network can be built in Keras as shown in [Example 7-3](#).

Example 7-3. Building the RNN with attention

```

notes_in = Input(shape = (None,)) ①
durations_in = Input(shape = (None,))

x1 = Embedding(n_notes, embed_size)(notes_in) ②
x2 = Embedding(n_durations, embed_size)(durations_in)

x = Concatenate()([x1,x2]) ③

x = LSTM(rnn_units, return_sequences=True)(x) ④
x = LSTM(rnn_units, return_sequences=True)(x)

e = Dense(1, activation='tanh')(x) ⑤
e = Reshape([-1])(e)

alpha = Activation('softmax')(e) ⑥

c = Permute([2, 1])(RepeatVector(rnn_units)(alpha)) ⑦
c = Multiply()([x, c])
c = Lambda(lambda xin: K.sum(xin, axis=1), output_shape=(rnn_units,))(c)

notes_out = Dense(n_notes, activation = 'softmax', name = 'pitch')(c) ⑧
durations_out = Dense(n_durations, activation = 'softmax', name = 'duration')(c)

model = Model([notes_in, durations_in], [notes_out, durations_out]) ⑨

att_model = Model([notes_in, durations_in], alpha) ⑩

opti = RMSprop(lr = 0.001)
model.compile(loss=['categorical_crossentropy', 'categorical_crossentropy'],
              optimizer=opti) ⑪

```

- ➊ There are two inputs to the network: the sequence of previous note names and duration values. Notice how the sequence length isn't specified—the attention mechanism does not require a fixed-length input, so we can leave this as variable.
- ➋ The Embedding layers convert the integer values of the note names and durations into vectors.

- ③ The vectors are concatenated to form one long vector that will be used as input into the recurrent layers.
- ④ Two stacked LSTM layers are used as the recurrent part of the network. Notice how we set `return_sequences` to `True` to make each layer pass the full sequence of hidden states to the next layer, rather than just the final hidden state.
- ⑤ The alignment function is just a `Dense` layer with one output unit and `tanh` activation. We can use a `Reshape` layer to squash the output to a single vector, of length equal to the length of the input sequence (`seq_length`).
- ⑥ The weights are calculated through applying a softmax activation to the alignment values.
- ⑦ To get the weighted sum of the hidden states, we need to use a `RepeatVector` layer to copy the weights `rnn_units` times to form a matrix of shape `[rnn_units, seq_length]`, then transpose this matrix using a `Permute` layer to get a matrix of shape `[seq_length, rnn_units]`. We can then multiply this matrix pointwise with the hidden states from the final LSTM layer, which also has shape `[seq_length, rnn_units]`. Finally, we use a `Lambda` layer to perform the summation along the `seq_length` axis, to give the context vector of length `rnn_units`.
- ⑧ The network has a double-headed output, one for the next note name and one for the next note length.
- ⑨ The final model accepts the previous note names and note durations as input and outputs a distribution for the next note name and next note duration.
- ⑩ We also create a model that outputs the `alpha` layer vector, so that we will be able to understand how the network is attributing weights to previous hidden states.
- ⑪ The model is compiled using `categorical_crossentropy` for both the note name and note duration output heads, as this is a multiclass classification problem.

A diagram of the full model built in Keras is shown in [Figure 7-7](#).

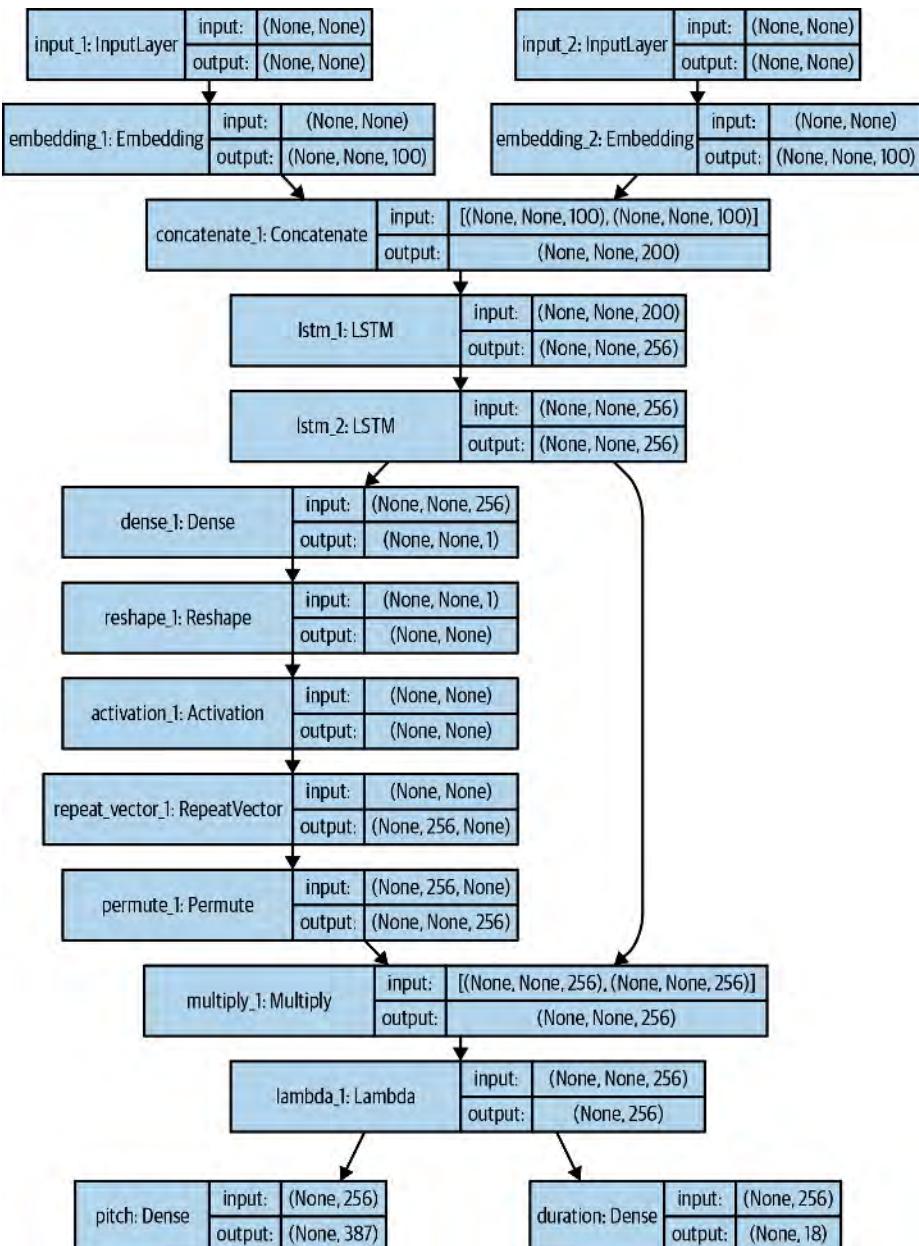


Figure 7-7. The LSTM model with attention for predicting the next note in a sequence

You can train this LSTM with attention by running the notebook called `07_02_lstm_compose_train.ipynb` in the book repository.

Analysis of the RNN with Attention

The following analysis can be produced by running the notebook *07_03_lstm_compose_analysis.ipynb* from the book repository, once you have trained your network.

We'll start by generating some music from scratch, by seeding the network with only a sequence of <START> tokens (i.e., we are telling the model to assume it is starting from the beginning of the piece). Then we can generate a musical passage using the same iterative technique we used in [Chapter 6](#) for generating text sequences, as follows:

1. Given the current sequence (of note names and note durations), the model predicts two distributions, for the next note name and duration.
2. We sample from both of these distributions, using a `temperature` parameter to control how much variation we would like in the sampling process.
3. The chosen note is stored and its name and duration are appended to the respective sequences.
4. If the length of the sequence is now greater than the sequence length that the model was trained on, we remove one element from the start of the sequence.
5. The process repeats with the new sequence, and so on, for as many notes as we wish to generate.

[Figure 7-8](#) shows examples of music generated from scratch by the model at various epochs of the training process.

Most of our analysis in this section will focus on the note pitch predictions, rather than rhythms, as for Bach's Cello Suites the harmonic intricacies are more difficult to capture and therefore more worthy of investigation. However, you can also apply the same analysis to the rhythmic predictions of the model, which may be particularly relevant for other styles of music that you could use to train this model (such as a drum track).

There are several points to note about the generated passages in [Figure 7-8](#). First, see how the music is becoming more sophisticated as training progresses. To begin with, the model plays it safe by sticking to the same group of notes and rhythms. By epoch 10, the model has begun to generate small runs of notes, and by epoch 20 it is producing interesting rhythms and is firmly established in a set key (E-flat major).

The figure consists of three vertically stacked rectangular boxes, each containing a musical staff. The top box is labeled 'EPOCH 2' on the right. It contains two staves: the top staff has a treble clef, a key signature of one sharp (F#), and a time signature of common time; the bottom staff has a bass clef, a key signature of one sharp (F#), and a time signature of common time. The middle box is labeled 'EPOCH 10' on the right. It also contains two staves: the top staff has a bass clef, a key signature of one flat (B-flat), and a time signature of common time; the bottom staff has a bass clef, a key signature of one flat (B-flat), and a time signature of common time. The bottom box is labeled 'EPOCH 20' on the right. It contains two staves: the top staff has a bass clef, a key signature of one flat (B-flat), and a time signature of common time; the bottom staff has a bass clef, a key signature of one flat (B-flat), and a time signature of common time.

Figure 7-8. Some examples of passages generated by the model when seeded only with a sequence of <START> tokens; here we use a temperature of 0.5 for the note names and durations

Second, we can analyze the distribution of note pitches over time by plotting the predicted distribution at each timestep as a heatmap. [Figure 7-9](#) shows this heatmap for the example from epoch 20 in [Figure 7-8](#).

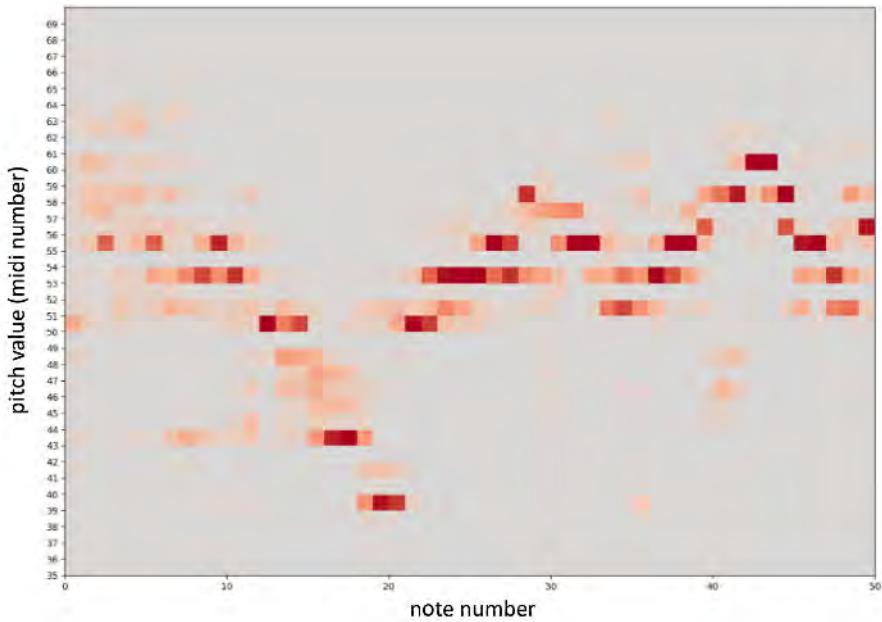


Figure 7-9. The distribution of possible next notes over time (at epoch 20): the darker the square, the more certain the model is that the next note is at this pitch

An interesting point to note here is that the model has clearly learned which notes belong to particular *keys*, as there are gaps in the distribution at notes that do not belong to the key. For example, there is a gray gap along the row for note 54 (corresponding to Gb/F#). This note is highly unlikely to appear in a piece of music in the key of E-flat major. Early on in the generation process (the lefthand side of the diagram) the key is not yet firmly established and therefore there is more uncertainty in how to choose the next note. As the piece progresses, the model settles on a key and certain notes become almost certain not to appear. What is remarkable is that the model hasn't explicitly decided to set the music in a certain key at the beginning, but instead is literally making it up as it goes along, trying to choose the note that best fits with those it has chosen previously.

It is also worth pointing out that the model has learned Bach's characteristic style of dropping to a low note on the cello to end a phrase and bouncing back up again to start the next. See how around note 20, the phrase ends on a low E-flat—it is common in the Bach Cello Suites to then return to a higher, more sonorous range of the instrument for the start of next phrase, which is exactly what the model predicts. There is a large gray gap between the low E-flat (pitch number 39) and the next note, which is predicted to be around pitch number 50, rather than continuing to rumble around the depths of the instrument.

Lastly, we should check to see if our attention mechanism is working as expected. **Figure 7-10** shows the values of the alpha vector elements calculated by the network at each point in the generated sequence. The horizontal axis shows the generated sequence of notes; the vertical axis shows where the attention of the network was aimed when predicting each note along the horizontal axis (i.e., the alpha vector). The darker the square, the greater the attention placed on the hidden state corresponding to this point in the sequence.

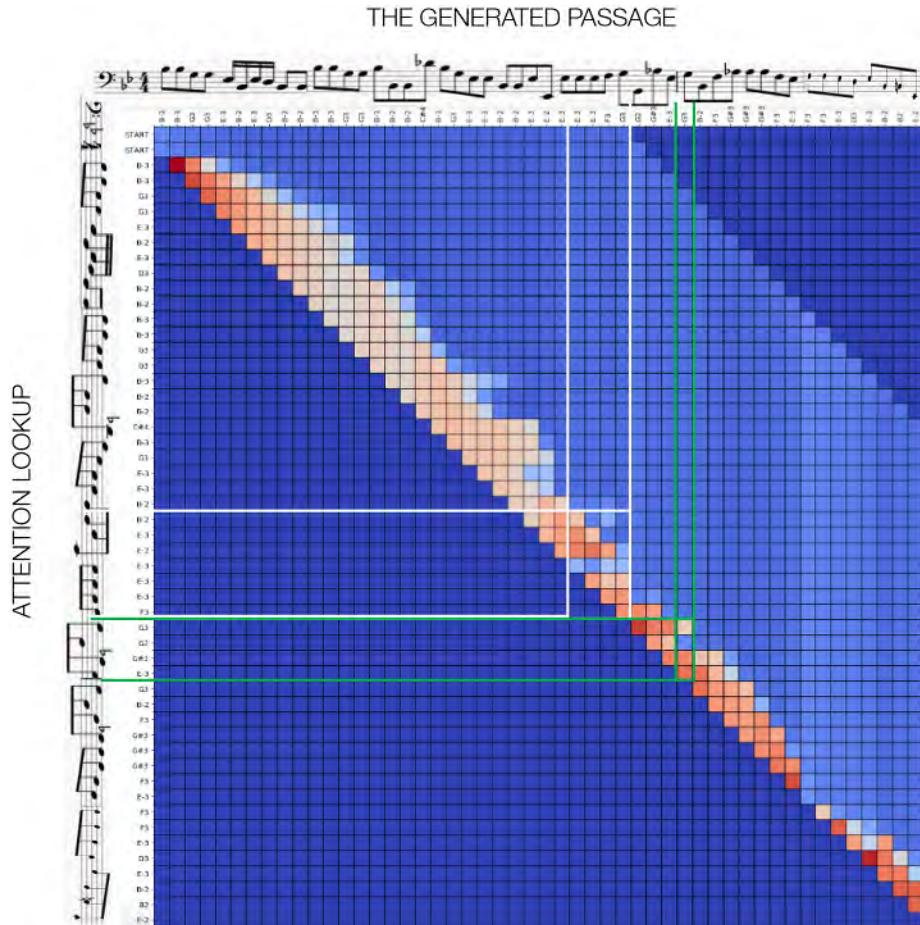


Figure 7-10. Each square in the matrix indicates the amount of attention given to the hidden state of the network corresponding to the note on the vertical axis, at the point of predicting the note on the horizontal axis; the more red the square, the more attention was given

We can see that for the second note of the piece ($B-3 = B$ -flat), the network chose to place almost all of its attention on the fact that the first note of the piece was also $B-3$. This makes sense; if you know that the first note is a B -flat, you will probably use this information to inform your decision about the next note.

As we move through the next few notes, the network spreads its attention roughly equally among previous notes—however, it rarely places any weight on notes more than six notes ago. Again, this makes sense; there is probably enough information contained in the previous six hidden states to understand how the phrase should continue.

There are also examples of where the network has chosen to ignore a certain note nearby, as it doesn't add any additional information to its understanding of the phrase. For example, take a look inside the white box marked in the center of the diagram, and note how there is a strip of boxes in the middle that cuts through the usual pattern of looking back at the previous four to six notes. Why would the network willingly choose to ignore this note when deciding how to continue the phrase?

If you look across to see which note this corresponds to, you can see that it is the first of three $E-3$ (E -flat) notes. The model has chosen to ignore this because the note prior to this is also an E -flat, an octave lower ($E-2$). The hidden state of the network at this point will provide ample information for the model to understand that E -flat is an important note in this passage, and therefore the model does not need to pay attention to the subsequent higher E -flat, as it doesn't add any extra information.

Additional evidence that the model has started to understand the concept of an octave can be seen inside the green box below and to the right. Here the model has chosen to ignore the low G ($G2$) because the note prior to this was also a G ($G3$), an octave higher. Remember we haven't told the model anything about which notes are related through octaves—it has worked this out for itself just by studying the music of J.S. Bach, which is remarkable.

Attention in Encoder–Decoder Networks

The attention mechanism is a powerful tool that helps the network decide which previous states of the recurrent layer are important for predicting the continuation of a sequence. So far, we have seen this for one-note-ahead predictions. However, we may also wish to build attention into encoder–decoder networks, where we predict a sequence of future notes by using an RNN decoder, rather than building up sequences one note at a time.

To recap, [Figure 7-11](#) shows how a standard encoder–decoder model for music generation might look, without attention—the kind that we introduced in [Chapter 6](#).

[Figure 7-12](#) shows the same network, but with an attention mechanism between the encoder and the decoder.

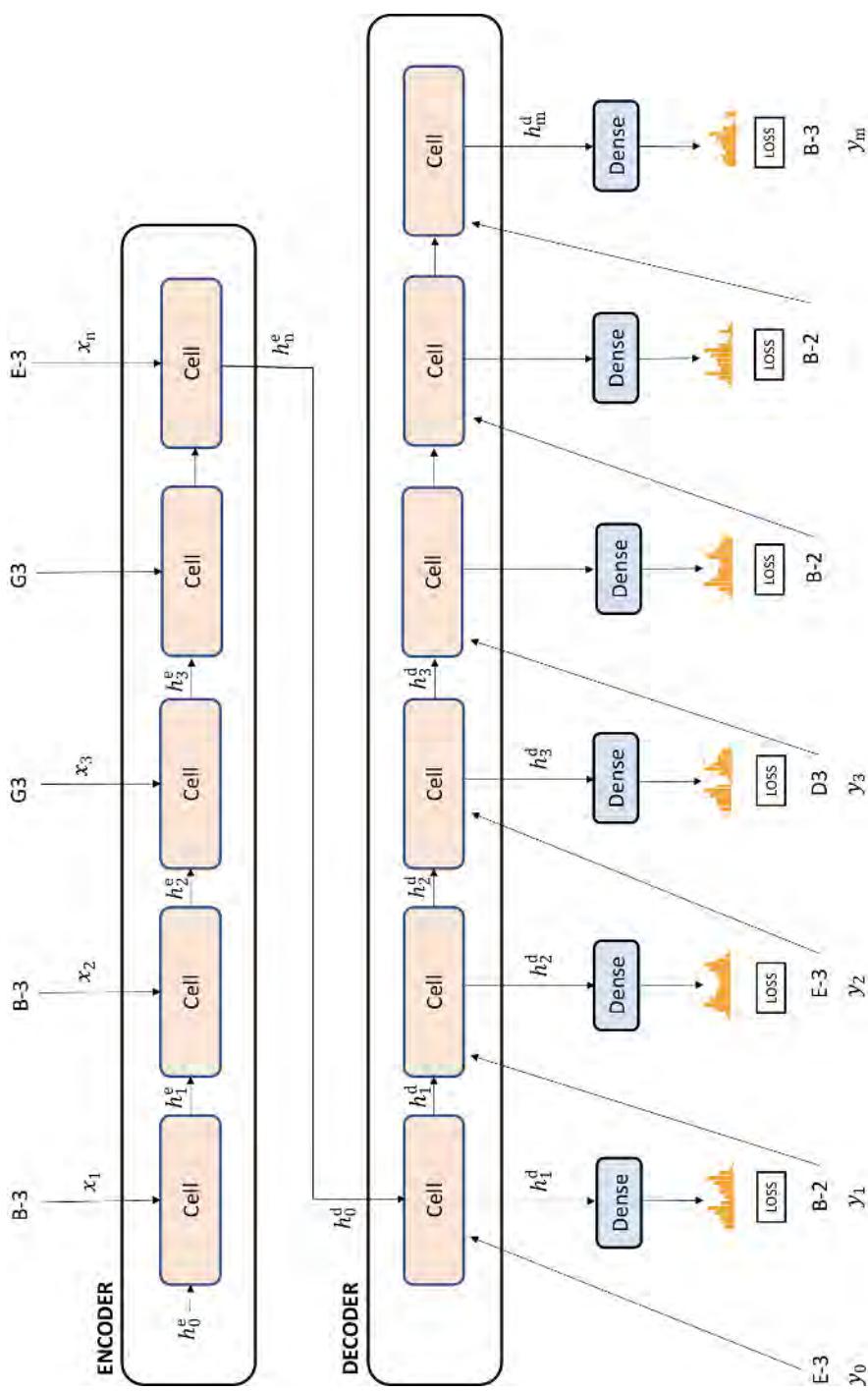


Figure 7-11. The standard encoder-decoder model

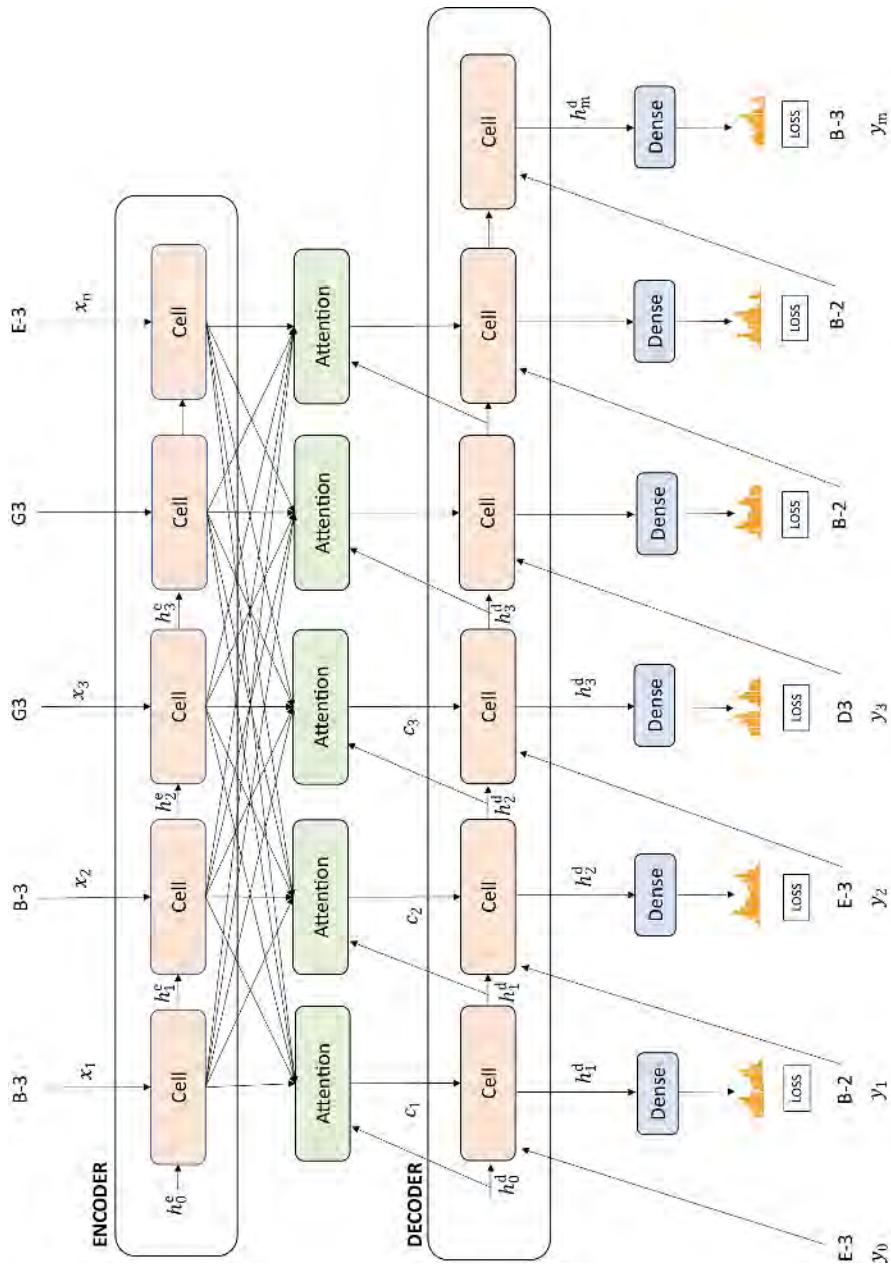


Figure 7-12. An encoder-decoder model with attention

The attention mechanism works in exactly the same way as we have seen previously, with one alteration: the hidden state of the decoder is also rolled into the mechanism so that the model is able to decide where to focus its attention not only through the previous encoder hidden states, but also from the current decoder hidden state. [Figure 7-13](#) shows the inner workings of an attention module within an encoder-decoder framework.

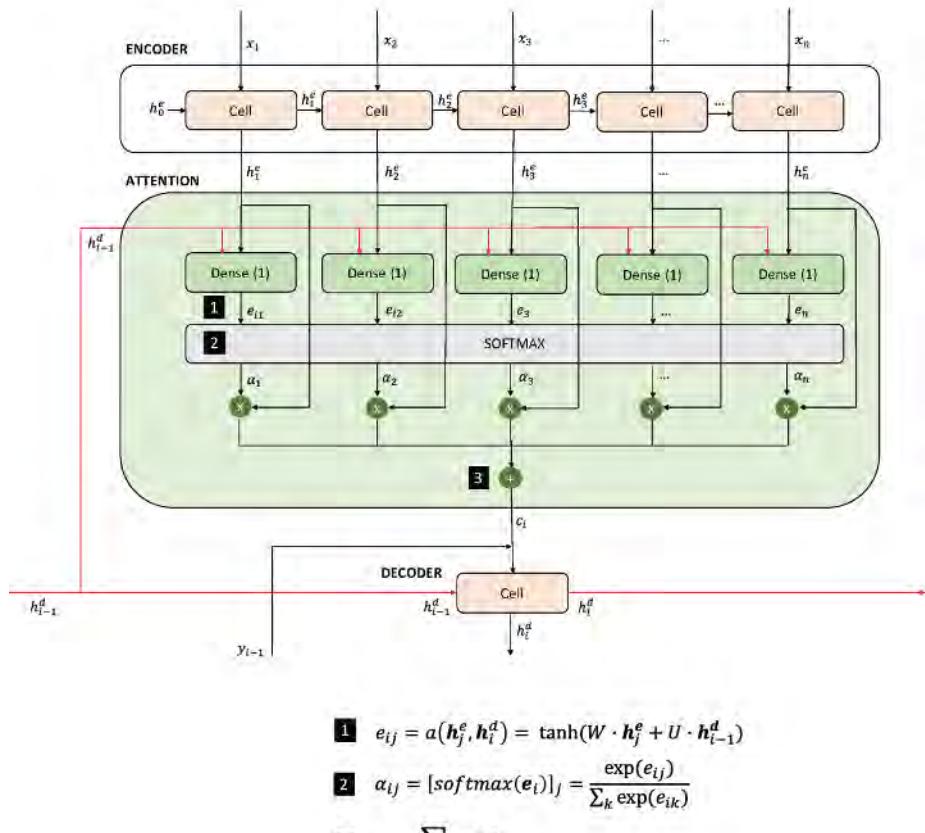


Figure 7-13. An attention mechanism within the context of an encoder-decoder network, connected to decoder cell i

While there are many copies of the attention mechanism within the encoder-decoder network, they all share the same weights, so there is no extra overhead in the number of parameters to be learned. The only change is that now, the decoder hidden state is rolled into the attention calculations (the red lines in the diagram). This slightly

changes the equations to incorporate an extra index (i) to specify the step of the decoder.

Also notice how in [Figure 7-11](#) we use the final state of the encoder to initialize the hidden state of the decoder. In an encoder–decoder with attention, we instead initialize the decoder using the built-in standard initializers for a recurrent layer. The context vector c_i is concatenated with the incoming data y_{i-1} to form an extended vector of data into each cell of the decoder. Thus, we treat the context vectors as additional data to be fed into the decoder.

Generating Polyphonic Music

The RNN with attention mechanism framework that we have explored in this section works well for single-line (monophonic) music, but could it be adapted to multiline (polyphonic) music?

The RNN framework is certainly flexible enough to conceive of an architecture whereby multiple lines of music are generated simultaneously, through a recurrent mechanism. But as it stands, our current dataset isn't well set up for this, as we are storing chords as single entities rather than parts that consist of multiple individual notes. There is no way for our current RNN to know, for example, that a C-major chord (C, E, and G) is actually very close to an A-minor chord (A, C, and E)—only one note would need to change, the G to an A. Instead, it treats both as two distinct elements to be predicted independently.

Ideally, we would like to design a network that can accept multiple channels of music as individual streams and learn how these streams should interact with each other to generate beautiful-sounding music, rather than disharmonious noise.

Doesn't this sound a bit like generating images? For image generation we have three channels (red, green, and blue), and we want the network to learn how to combine these channels to generate beautiful-looking images, rather than random pixelated noise.

In fact, as we shall see in the next section, we can treat music generation directly as an image generation problem. This means that instead of using recurrent networks we can apply the same convolutional-based techniques that worked so well for image generation problems to music—in particular, GANs.

Before we explore this new architecture, there is just enough time to visit the concert hall, where a performance is about to begin...

The Musical Organ

The conductor taps his baton twice on the podium. The performance is about to begin. In front of him sits an orchestra. However, this orchestra isn't about to launch

into a Beethoven symphony or a Tchaikovsky overture. This orchestra composes original music live during the performance and is powered entirely by a set of players giving instructions to a huge Musical Organ (MuseGAN for short) in the middle of the stage, which converts these instructions into beautiful music for the pleasure of the audience. The orchestra can be trained to generate music in a particular style, and no two performances are ever the same.

The 128 players in the orchestra are divided into 4 equal sections of 32 players. Each section gives instructions to the MuseGAN and has a distinct responsibility within the orchestra.

The *style* section is in charge of producing the overall musical stylistic flair of the performance. In many ways, it has the easiest job of all the sections as each player simply has to generate a single instruction at the start of the concert that is then continually fed to the MuseGAN throughout the performance.

The *groove* section has a similar job, but each player produces several instructions: one for each of the distinct musical *tracks* that are output by the MuseGAN. For example, in one concert, each member of the groove section produced five instructions, one for each of the vocal, piano, string, bass, and drum tracks. Thus, their job is to provide the *groove* for each individual instrumental sound that is then constant throughout the performance.

The style and groove sections do not change their instructions throughout the piece. The dynamic element of the performance is provided by the final two sections, which ensure that the music is constantly changing with each bar that goes by. A *bar* (or *measure*) is a small unit of music that contains a fixed, small number of beats. For example, if you can count 1, 2, 1, 2 along to a piece of music, then there are two beats in each bar and you're probably listening to a march. If you can count 1, 2, 3, 1, 2, 3, then there are three beats to each bar and you may be listening to a waltz.

The players in the *chords* section change their instructions at the start of each bar. This has the effect of giving each bar a distinct musical character, for example, through a change of chord. The players in the chords section only produce one instruction per bar that then applies to every instrumental track.

The players in the *melody* section have the most exhausting job, because they give different instructions to each instrumental track at the start of every bar throughout the piece. These players have the most fine-grained control over the music, and this can therefore be thought of as the section that provides the melodic interest.

This completes the description of the orchestra. We can summarize the responsibilities of each section as shown in [Table 7-2](#).

Table 7-2. Sections of the MuseGAN orchestra

	Instructions change with each bar?	Different instruction per track?
Style	X	X
Groove	X	✓
Chords	✓	X
Melody	✓	✓

It is up to the MuseGAN to generate the next bar of music, given the current set of 128 instructions (one from each player). Training the MuseGAN to do this isn't easy. Initially the instrument only produces horrendous noise, as it has no way to understand how it should interpret the instructions to produce bars that are indistinguishable from genuine music.

This is where the conductor comes in. The conductor tells the MuseGAN when the music it is producing is clearly distinguishable from real music, and the MuseGAN then adapts its internal wiring to be more likely to fool the conductor the next time around. The conductor and the MuseGAN use exactly the same process as we saw in [Chapter 4](#), when Di and Gene worked together to continuously improve the photos of ganimals taken by Gene.

The MuseGAN players tour the world giving concerts in any style where there is sufficient existing music to train the MuseGAN. In the next section we'll see how we can build a MuseGAN using Keras, to learn how to generate realistic polyphonic music.

Your First MuseGAN

The MuseGAN was introduced in the 2017 paper "MuseGAN: Multi-Track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment."² The authors show how it is possible to train a model to generate polyphonic, multitrack, multibar music through a novel GAN framework. Moreover, they show how, by dividing up the responsibilities of the noise vectors that feed the generator, they are able to maintain fine-grained control over the high-level temporal and track-based features of the music.

To begin this project, you'll first need to download the MIDI files that we'll be using to train the MuseGAN. We'll use a dataset of 229 J.S. Bach chorales for four voices, available on [GitHub](#). Download this dataset and place it inside the *data* folder of the book repository, in a folder called *chorales*. The dataset consists of an array of four numbers for each timestep: the MIDI note pitches of each of the four voices. A timestep in this dataset is equal to a 16th note (a semiquaver). So, for example, in a single

² Hao-Wen Dong et al., "MuseGAN: Multi-Track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment," 19 September 2017, <https://arxiv.org/abs/1709.06298>.

bar of 4 quarter (crotchet) beats, there would be 16 timesteps. Also, the dataset is automatically split into *train*, *validation*, and *test* sets. We will be using the *train* dataset to train the MuseGAN.

We first need to get the data into the correct shape to feed the GAN. In this example, we'll generate two bars of music, so we'll first extract only the first two bars of each chorale. [Figure 7-14](#) shows how two bars of raw data are converted into the transformed dataset that will feed the GAN with the corresponding musical notation.

Each bar consists of 16 timesteps and there are a potential 84 pitches across the 4 tracks. Therefore, a suitable shape for the transformed data is:

```
[batch_size, n_bars, n_steps_per_bar, n_pitches, n_tracks]
```

where

```
n_bars = 2  
n_steps_per_bar = 16  
n_pitches = 84  
n_tracks = 4
```

To get the data into this shape, we one-hot encode the pitch numbers into a vector of length 84 and split each sequence of notes into two groups of 16, to replicate 2 bars.³

Now that we have transformed our dataset, let's take a look at the overall structure of the MuseGAN, starting with the generator.

³ We are making the assumption here that each chorale in the dataset has four beats in each bar, which is reasonable, and even if this were not the case it would not adversely affect the training of the model.

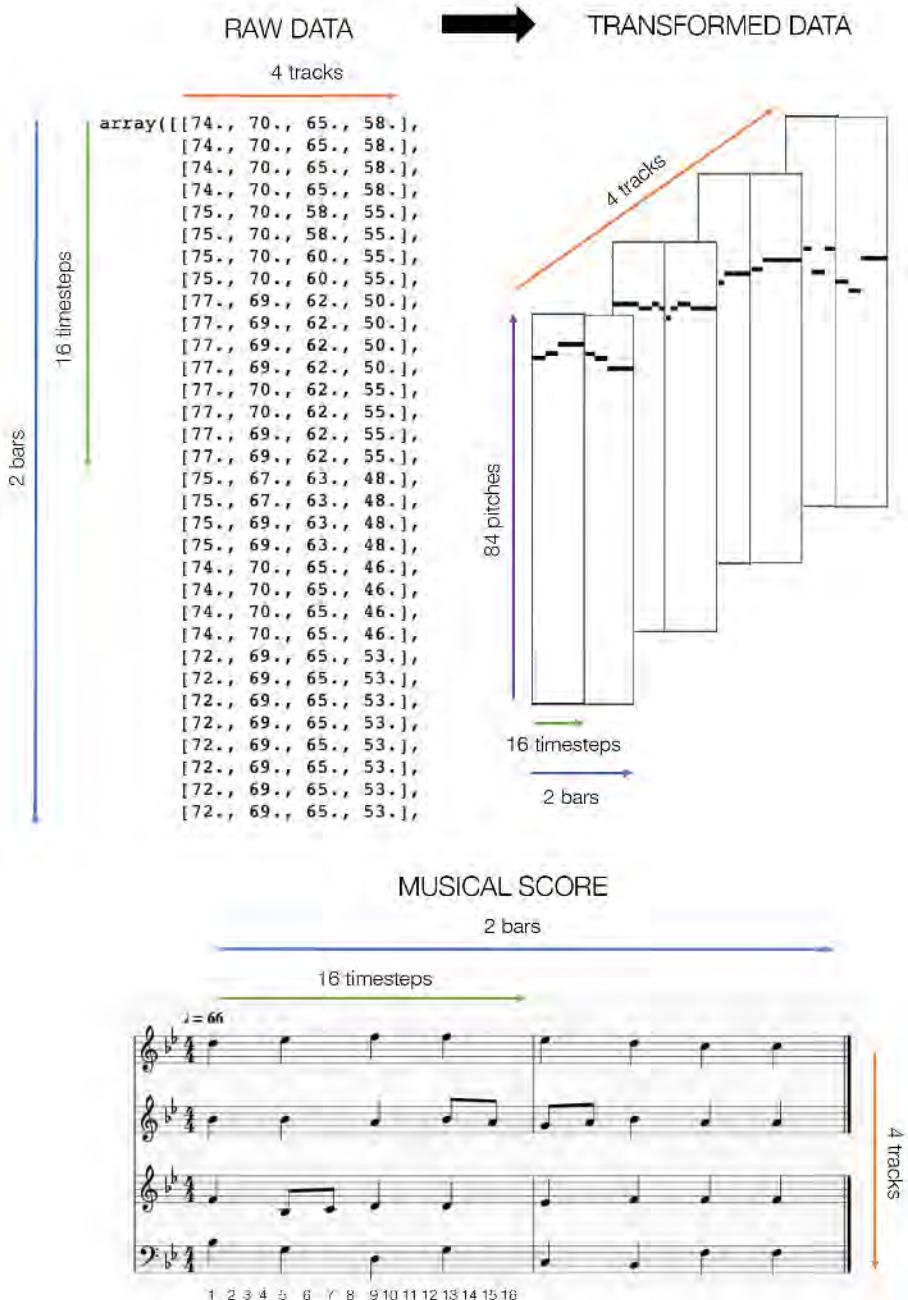


Figure 7-14. Example of MuseGAN raw data

The MuseGAN Generator

Like all GANs, the MuseGAN consists of a generator and a critic. The generator tries to fool the critic with its musical creations, and the critic tries to prevent this from happening by ensuring it is able to tell the difference between the generator's forged Bach chorales and the real thing.

Where the MuseGAN is different is the fact that the generator doesn't just accept a single noise vector as input, but instead has four separate inputs, which correspond to the four sections of the orchestra in the story—chords, style, melody, and groove. By manipulating each of these inputs independently we can change high-level properties of the generated music.

A high-level view of the generator is shown in [Figure 7-15](#).

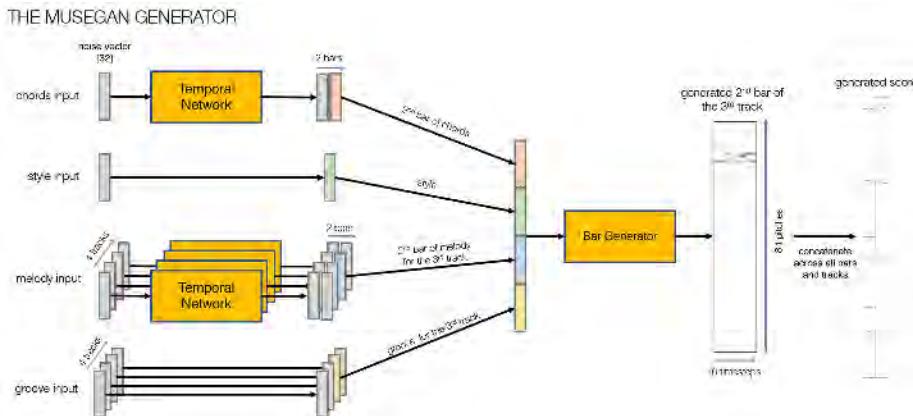


Figure 7-15. High-level diagram of the MuseGAN generator

The diagram shows how the chords and melody inputs are first passed through a temporal network that outputs a tensor with one of the dimensions equal to the number of bars to be generated. The style and groove inputs are not stretched temporally in this way, as they remain constant through the piece.

Then, to generate a particular bar for a particular track, the relevant vectors from the chords, style, melody, and groove parts of the network are concatenated to form a longer vector. This is then passed to a bar generator, which ultimately outputs the specified bar for the specified track.

By concatenating the generated bars for all tracks, we create a score that can be compared with real scores by the critic. You can start training the MuseGAN using the notebook `07_04_musegan_train.ipynb` in the book repository. The parameters to the model are given in [Example 7-4](#).

Example 7-4. Defining the MuseGAN

```
BATCH_SIZE = 64
n_bars = 2
n_steps_per_bar = 16
n_pitches = 84
n_tracks = 4
z_dim = 32

gan = MuseGAN(input_dim = data_binary.shape[1:]
, critic_learning_rate = 0.001
, generator_learning_rate = 0.001
, optimiser = 'adam'
, grad_weight = 10
, z_dim = 32
, batch_size = 64
, n_tracks = 4
, n_bars = 2
, n_steps_per_bar = 16
, n_pitches = 84
)
```

Chords, Style, Melody, and Groove

Let's now take a closer look at the four different inputs that feed the generator.

Chords

The chords input is a vector of length 32 (`z_dim`). We need to output a different vector for every bar, as its job is to control the general dynamic nature of the music over time. Note that while this is labeled `chords_input`, it really could control anything about the music that changes per bar, such as general rhythmic style, without being specific to any particular track.

The way this is achieved is with a neural network consisting of convolutional transpose layers that we call the *temporal network*. The Keras code to build this is shown in [Example 7-5](#).

Example 7-5. Building the temporal network

```
def conv_t(self, x, f, k, s, a, p, bn):
    x = Conv2DTranspose(
        filters = f
        , kernel_size = k
        , padding = p
        , strides = s
        , kernel_initializer = self.weight_init
    )(x)

    if bn:
```

```

x = BatchNormalization(momentum = 0.9)(x)

if a == 'relu':
    x = Activation(a)(x)
elif a == 'lrelu':
    x = LeakyReLU()(x)

return x

def TemporalNetwork(self):

    input_layer = Input(shape=(self.z_dim,), name='temporal_input') ①

    x = Reshape([1,1,self.z_dim])(input_layer) ②
    x = self.conv_t(x, f=1024, k=(2,1), s=(1,1), a= 'relu',
                      p = 'valid', bn = True)
    x = self.conv_t(x, f=self.z_dim, k=(self.n_bars - 1,1),
                   s=(1,1), a = 'relu', p = 'valid', bn = True) ③

    output_layer = Reshape([self.n_bars, self.z_dim])(x) ④

    return Model(input_layer, output_layer)

```

- ① The input to the temporal network is a vector of length 32 (`z_dim`).
- ② We reshape this vector to a 1×1 tensor with 32 channels, so that we can apply convolutional transpose operations to it.
- ③ We apply Conv2DTranspose layers to expand the size of the tensor along one axis, so that it is the same length as `n_bars`.
- ④ We remove the unnecessary extra dimension with a Reshape layer.

The reason we use convolutional operations rather than requiring two independent chord vectors into the network is because we would like the network to learn how one bar should follow on from another in a consistent way. Using a neural network to expand the input vector along the time axis means the model has a chance to learn how music flows across bars, rather than treating each bar as completely independent of the last.

Style

The style input is also a vector of length `z_dim`. This is carried across to the bar generator without any change, as it is independent of the track and bar. In other words, the bar generator should use this vector to establish consistency between bars and tracks.

Melody

The melody input is an array of shape [n_tracks, z_dim]—that is, we provide the model with a random noise vector of length z_dim for each track.

Each of these vectors is passed through its own copy of the temporal network specified previously. Note that the weights of these copies are not shared. The output is therefore a vector of length z_dim for every track of every bar. This way, the bar generator will be able to use this vector to fine-tune the content of every single bar and track independently.

Groove

The groove input is also an array of shape [n_tracks, z_dim]—a random noise vector of length z_dim for each track. Unlike the melody input, these are not passed through the temporal network but instead are fed straight through to the bar generator unchanged, just like the style vector. However, unlike in the style vector there is a distinct groove input for every track, meaning that we can use these vectors to adjust the overall output for each track independently.

The Bar Generator

The bar generator converts a vector of length 4 * z_dim to a single bar for a single track—i.e., a tensor of shape [1, n_steps_per_bar, n_pitches, 1]. The input vector is created through the concatenation of the four relevant chord, style, melody, and groove vectors, each of length z_dim.

The bar generator is a neural network that uses convolutional transpose layers to expand the time and pitch dimensions. We will be creating one bar generator for every track, and weights are not shared. The Keras code to build a bar generator is given in [Example 7-6](#).

Example 7-6. Building the bar generator

```
def BarGenerator(self):  
  
    input_layer = Input(shape=(self.z_dim * 4,), name='bar_generator_input') ❶  
  
    x = Dense(1024)(input_layer)  
    x = BatchNormalization(momentum = 0.9)(x)  
    x = Activation('relu')(x)  
  
    x = Reshape([2,1,512])(x) ❷  
    x = self.conv_t(x, f=512, k=(2,1), s=(2,1), a= 'relu', p = 'same', bn = True) ❸  
    x = self.conv_t(x, f=256, k=(2,1), s=(2,1), a= 'relu', p = 'same', bn = True)  
    x = self.conv_t(x, f=256, k=(2,1), s=(2,1), a= 'relu', p = 'same', bn = True)  
    x = self.conv_t(x, f=256, k=(1,7), s=(1,7), a= 'relu', p = 'same',bn = True) ❹
```

```

x = self.conv_t(x, f=1, k=(1,12), s=(1,12), a= 'tanh', p = 'same', bn = False) ⑤

output_layer = Reshape([1, self.n_steps_per_bar , self.n_pitches ,1])(x) ⑥

return Model(input_layer, output_layer)

```

- ➊ The input to the bar generator is a vector of length `4 * z_dim`.
- ➋ After passing through a `Dense` layer, we reshape the tensor to prepare it for the convolutional transpose operations.
- ➌ First we expand the tensor along the timestep axis...
- ➍ ...then along the pitch axis.
- ➎ The final layer has a tanh activation applied, as we will be using a WGAN-GP (which requires tanh output activation) to train the network.
- ➏ The tensor is reshaped to add two extra dimensions of size 1, to prepare it for concatenation with other bars and tracks.

Putting It All Together

Ultimately the MuseGAN has one single generator that incorporates all of the temporal networks and bar generators. This network takes the four input tensors and converts them into a multitrack, multibar score. The Keras code to build the overall generator is provided in [Example 7-7](#).

Example 7-7. Building the MuseGAN generator

```

chords_input = Input(shape=(self.z_dim,), name='chords_input') ➊
style_input = Input(shape=(self.z_dim,), name='style_input')
melody_input = Input(shape=(self.n_tracks, self.z_dim), name='melody_input')
groove_input = Input(shape=(self.n_tracks, self.z_dim), name='groove_input')

# CHORDS -> TEMPORAL NETWORK ➋
self.chords_tempNetwork = self.TemporalNetwork()
self.chords_tempNetwork.name = 'temporal_network'
chords_over_time = self.chords_tempNetwork(chords_input) # [n_bars, z_dim]

# MELODY -> TEMPORAL NETWORK ➌
melody_over_time = [None] * self.n_tracks # list of n_tracks [n_bars, z_dim] tensors
self.melody_tempNetwork = [None] * self.n_tracks
for track in range(self.n_tracks):
    self.melody_tempNetwork[track] = self.TemporalNetwork()
    melody_track = Lambda(lambda x: x[:,track,:])(melody_input)
    melody_over_time[track] = self.melody_tempNetwork[track](melody_track)

```

```

# CREATE BAR GENERATOR FOR EACH TRACK ④
self.barGen = [None] * self.n_tracks
for track in range(self.n_tracks):
    self.barGen[track] = self.BarGenerator()

# CREATE OUTPUT FOR EVERY TRACK AND BAR ⑤
bars_output = [None] * self.n_bars
for bar in range(self.n_bars):
    track_output = [None] * self.n_tracks

    c = Lambda(lambda x: x[:,bar,:]
              , name = 'chords_input_bar_' + str(bar))(chords_over_time)
    s = style_input

    for track in range(self.n_tracks):

        m = Lambda(lambda x: x[:,bar,:])(melody_over_time[track])
        g = Lambda(lambda x: x[:,track,:])(groove_input)

        z_input = Concatenate(axis = 1
                              , name = 'total_input_bar_{}_track_{}'.format(bar, track))
        ([c,s,m,g])

        track_output[track] = self.barGen[track](z_input)

    bars_output[bar] = Concatenate(axis = -1)(track_output)

generator_output = Concatenate(axis = 1, name = 'concat_bars')(bars_output) ⑥

self.generator = Model([chords_input, style_input, melody_input, groove_input]
                      , generator_output) ⑦

```

- ① The inputs to the generator are defined.
- ② Pass the chords input through the temporal network.
- ③ Pass the melody input through the temporal network.
- ④ Create an independent bar generator network for every track.
- ⑤ Loop over the tracks and bars, creating a generated bar for each combination.
- ⑥ Concatenate everything together to form a single output tensor.
- ⑦ The MuseGAN model takes four distinct noise tensors as input and outputs a generated multitrack, multibar score.

The Critic

In comparison to the generator, the critic architecture is much more straightforward (as is often the case with GANs).

The critic tries to distinguish full multitrack, multibar scores created by the generator from real excepts from the Bach chorales. It is a convolutional neural network, consisting mostly of Conv3D layers that collapse the score into a single output prediction. So far, we have only worked with Conv2D layers, applicable to three-dimensional input images (width, height, channels). Here we have to use Conv3D layers, which are analogous to Conv2D layers but accept four-dimensional input tensors (`n_bars`, `n_steps_per_bar`, `n_pitches`, `n_tracks`).

Also, we do not use batch normalization layers in the critic as we will be using the WGAN-GP framework for training the GAN, which forbids this.

The Keras code to build the critic is given in [Example 7-8](#).

Example 7-8. Building the MuseGAN critic

```
def conv(self, x, f, k, s, a, p):
    x = Conv3D(
        filters = f
        , kernel_size = k
        , padding = p
        , strides = s
        , kernel_initializer = self.weight_init
    )(x)

    if a == 'relu':
        x = Activation(a)(x)
    elif a == 'lrelu':
        x = LeakyReLU()(x)

    return x

critic_input = Input(shape=self.input_dim, name='critic_input') ❶

x = critic_input
x = self.conv(x, f=128, k = (2,1,1), s = (1,1,1), a = 'lrelu', p = 'valid') ❷
x = self.conv(x, f=128, k = (self.n_bars - 1,1,1)
            , s = (1,1,1), a = 'lrelu', p = 'valid')

x = self.conv(x, f=128, k = (1,1,12), s = (1,1,12), a = 'lrelu', p = 'same') ❸
x = self.conv(x, f=128, k = (1,1,7), s = (1,1,7), a = 'lrelu', p = 'same')
x = self.conv(x, f=128, k = (1,2,1), s = (1,2,1), a = 'lrelu', p = 'same') ❹
x = self.conv(x, f=128, k = (1,2,1), s = (1,2,1), a = 'lrelu', p = 'same')
x = self.conv(x, f=256, k = (1,4,1), s = (1,2,1), a = 'lrelu', p = 'same')
```

```

x = self.conv(x, f=512, k = (1,3,1), s = (1,2,1), a = 'lrelu', p = 'same')

x = Flatten()(x)

x = Dense(1024, kernel_initializer = self.weight_init)(x)
x = LeakyReLU()(x)
critic_output = Dense(1, activation=None
    , kernel_initializer = self.weight_init)(x) ⑤

self.critic = Model(critic_input, critic_output)

```

- ➊ The input to the critic is an array of multitrack, multibar scores, each of shape [n_bars, n_steps_per_bar, n_pitches, n_tracks].
- ➋ First, we collapse the tensor along the bar axis. We apply Conv3D layers throughout the critic as we are working with 4D tensors.
- ➌ Next, we collapse the tensor along the pitch axis.
- ➍ Finally, we collapse the tensor along the timesteps axis.
- ➎ The output is a Dense layer with a single unit and no activation function, as required by the WGAN-GP framework.

Analysis of the MuseGAN

We can perform some experiments with our MuseGAN by generating a score, then tweaking some of the input noise parameters to see the effect on the output.

The output from the generator is an array of values in the range $[-1, 1]$ (due to the tanh activation function of the final layer). To convert this to a single note for each track, we choose the note with the maximum value over all 84 pitches for each timestep. In the original MuseGAN paper the authors use a threshold of 0, as each track can contain multiple notes; however, in this setting we can simply take the maximum, to guarantee exactly one note per timestep per track, as is the case for the Bach chorales.

[Figure 7-16](#) shows a score that has been generated by the model from random normally distributed noise vectors (top left). We can find the closest score in the dataset (by Euclidean distance) and check that our generated score isn't a copy of a piece of music that already exists in the dataset—the closest score is shown just below it, and we can see that it does not resemble our generated score.

The figure consists of five musical score snippets arranged in a grid.
 - Top-left: 'GENERATED SCORE' shows a standard generated musical score with four staves of notes.
 - Top-right: 'CHANGING THE STYLE NOISE' shows the same score but with a different style, where the notes are more rhythmic and dynamic.
 - Middle-left: 'CLOSEST REAL SCORE' shows a real musical score from training data, which looks very similar to the generated score.
 - Middle-right: 'CHANGING THE MELODY NOISE FOR THE TOP LINE ONLY' shows the top line of the score with changed melody noise, resulting in faster and more dynamic notes.
 - Bottom-left: 'CHANGING THE CHORD NOISE' shows the score with changed chord noise, where the harmonic structure is altered.
 - Bottom-right: 'CHANGING THE GROOVE NOISE FOR THE BOTTOM LINE ONLY' shows the bottom line of the score with changed groove noise, resulting in a more rhythmic and dynamic bass line.

Figure 7-16. Example of a MuseGAN predicted score, showing the closest real score in the training data and how the generated score is affected by changing the input noise

Let's now play around with the input noise to tweak our generated score. First, we can try changing the noise vector—the bottom-left score in [Figure 7-16](#) shows the result. We can see that every track has changed, as expected, and also that the two bars exhibit different properties. In the second bar, the baseline is more dynamic and the top line is higher in pitch than in the first bar.

When we change the style vector (top right), both bars change in a similar way. There is no great difference in style between the two bars, but the whole passage has changed from the original generated score.

We can also alter tracks individually, through the melody and groove inputs. In [Figure 7-16](#) we can see the effect of changing just the melody noise input for the top line. All other parts remain unaffected, but the top-line notes change significantly. Also, we can see a rhythmic change between the two bars in the top line: the second bar is more dynamic, containing faster notes than the first bar.

Lastly, the bottom-right score in the diagram shows the predicted score when we alter the groove input parameter for only the baseline. Again, all other parts remain unaffected, but the baseline is different. Moreover, the overall pattern of the baseline remains similar between bars, as we would expect.

This shows how each of the input parameters can be used to directly influence high-level features of the generated musical sequence, in much the same way as we were able to adjust the latent vectors of VAEs and GANs in previous chapters to alter the appearance of a generated image. One drawback to the model is that the number of bars to generate must be specified up front. To tackle this, the authors show an extension to the model that allows previous bars to be fed in as input, therefore allowing the model to generate long-form scores by continually feeding the most recent predicted bars back into the model as additional input.

Summary

In this chapter we have explored two different kinds of model for music generation: a stacked LSTM with attention and a MuseGAN.

The stacked LSTM is similar in design to the networks we saw in [Chapter 6](#) for text generation. Music and text generation share a lot of features in common, and often similar techniques can be used for both. We enhanced the recurrent network with an attention mechanism that allows the model to focus on specific previous timesteps in order to predict the next note and saw how the model was able to learn about concepts such as octaves and keys, simply by learning to accurately generate the music of Bach.

Then we saw that generating sequential data does not always require a recurrent model—the MuseGAN uses convolutions to generate polyphonic musical scores with multiple tracks, by treating the score as a kind of image where the tracks are individual channels of the image. The novelty of the MuseGAN lies in the way the four input noise vectors (chords, style, melody, and groove) are organized so that it is possible to maintain full control over high-level features of the music. While the underlying harmonization is still not as perfect or varied as Bach, it is a good attempt at what is an extremely difficult problem to master and highlights the power of GANs to tackle a wide variety of problems.

In the next chapter we shall introduce one of the most remarkable models developed in recent years, the *world model*. In their groundbreaking paper describing it, the authors show how it is possible to build a model that enables a car to drive around a simulated racetrack by first testing out strategies in its own generated “dream” of the environment. This allows the car to excel at driving around the track without ever having attempted the task, as it has already imagined how to do this successfully in its own imagined world model.

CHAPTER 8

Play

In March 2018, David Ha and Jürgen Schmidhuber published their “World Models” paper.¹ The paper showed how it is possible to train a model that can learn how to perform a particular task through experimentation within its own generative hallucinated dreams, rather than inside the environment itself. It is an excellent example of how generative modeling can be used to solve practical problems, when applied alongside other machine learning techniques such as reinforcement learning.

A key component of the architecture is a generative model that can construct a probability distribution for the next possible state, given the current state and action. Having built up an understanding of the underlying physics of the environment through random movements, the model is then able to train itself from scratch on a new task, entirely within its own internal representation of the environment. This approach led to world-best scores for both of the tasks on which it was tested.

In this chapter, we will explore the model in detail and show how it is possible to create your own version of this amazing cutting-edge technology.

Based on the original paper, we will be building a reinforcement learning algorithm that learns how to drive a car around a racetrack as fast as possible. While we will be using a 2D computer simulation as our environment, the same technique could also be applied to real-world scenarios where testing strategies in the live environment is expensive or infeasible.

Before we start building the model, however, we need to take a closer look at the concept of reinforcement learning and the OpenAI Gym platform.

¹ David Ha and Jürgen Schmidhuber, “World Models,” 27 March 2018, <https://arxiv.org/abs/1803.10122>.

Reinforcement Learning

Reinforcement learning can be defined as follows:

Reinforcement learning (RL) is a field of machine learning that aims to train an agent to perform optimally within a given environment, with respect to a particular goal.

While both discriminative modeling and generative modeling aim to minimize a loss function over a dataset of observations, reinforcement learning aims to maximize the long-term reward of an agent in a given environment. It is often described as one of the three major branches of machine learning, alongside *supervised learning* (predicting using labeled data) and *unsupervised learning* (learning structure from unlabeled data).

Let's first introduce some key terminology relating to reinforcement learning:

Environment

The world in which the agent operates. It defines the set of rules that govern the game state update process and reward allocation, given the agent's previous action and current game state. For example, if we were teaching a reinforcement learning algorithm to play chess, the environment would consist of the rules that govern how a given action (e.g., the move e4) affects the next game state (the new positions of the pieces on the board) and would also specify how to assess if a given position is checkmate and allocate the winning player a reward of 1 after the winning move.

Agent

The entity that takes actions in the environment.

Game state

The data that represents a particular situation that the agent may encounter (also just called a *state*), for example, a particular chessboard configuration with accompanying game information such as which player will make the next move.

Action

A feasible move that an agent can make.

Reward

The value given back to the agent by the environment after an action has been taken. The agent aims to maximize the long-term sum of its rewards. For example, in a game of chess, checkmating the opponent's king has a reward of 1 and every other move has a reward of 0. Other games have rewards constantly awarded throughout the episode (e.g., points in a game of *Space Invaders*).

Episode

One run of an agent in the environment; this is also called a *rollout*.

Timestep

For a discrete event environment, all states, actions, and rewards are subscripted to show their value at timestep t .

The relationship between these definitions is shown in [Figure 8-1](#).

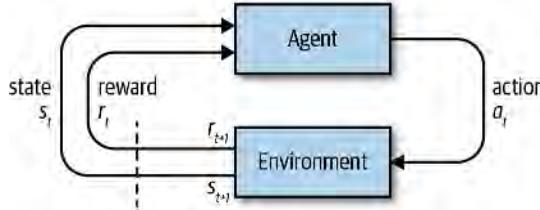


Figure 8-1. Reinforcement learning diagram

The environment is first initialized with a current game state, s_0 . At timestep t , the agent receives the current game state s_t and uses this to decide its next best action a_t , which it then performs. Given this action, the environment then calculates the next state s_{t+1} and reward r_{t+1} and passes these back to the agent, for the cycle to begin again. The cycle continues until the end criterion of the episode is met (e.g., a given number of timesteps elapse or the agent wins/loses).

How can we design an agent to maximize the sum of rewards in a given environment? We could build an agent that contains a set of rules for how to respond to any given game state. However, this quickly becomes infeasible as the environment becomes more complex and doesn't ever allow us to build an agent that has superhuman ability in a particular task, as we are hardcoding the rules. Reinforcement learning involves creating an agent that can learn optimal strategies by itself in complex environments through repeated play—this is what we will be using in this chapter to build our agent.

I'll now introduce OpenAI Gym, home of the `CarRacing` environment that we will use to simulate a car driving around a track.

OpenAI Gym

[OpenAI Gym](#) is a toolkit for developing reinforcement learning algorithms that is available as a Python library.

Contained within the library are several classic reinforcement learning environments, such as `CartPole` and `Pong`, as well as environments that present more complex challenges, such as training an agent to walk on uneven terrain or win an Atari game. All of the environments provide a `step` method through which you can submit a given action; the environment will return the next state and the reward. By repeatedly

calling the `step` method with the actions chosen by the agent, you can play out an episode in the environment.

In addition to the abstract mechanics of each environment, OpenAI Gym also provides graphics that allow you to watch your agent perform in a given environment. This is useful for debugging and finding areas where your agent could improve.

We will make use of the `CarRacing` environment within OpenAI Gym. Let's see how the game state, action, reward, and episode are defined for this environment:

Game state

A 64×64 -pixel RGB image depicting an overhead view of the track and car.

Action

A set of three values: the steering direction (-1 to 1), acceleration (0 to 1), and braking (0 to 1). The agent must set all three values at each timestep.

Reward

A negative penalty of -0.1 for each timestep taken and a positive reward of $1000/N$ if a new track tile is visited, where N is the total number of tiles that make up the track.

Episode

The episode ends when either the car completes the track, drives off the edge of the environment, or 3,000 timesteps have elapsed.

These concepts are shown on a graphical representation of a game state in [Figure 8-2](#). Note that the car doesn't see the track from its point of view, but instead we should imagine an agent floating above the track controlling the car from a bird's-eye view.

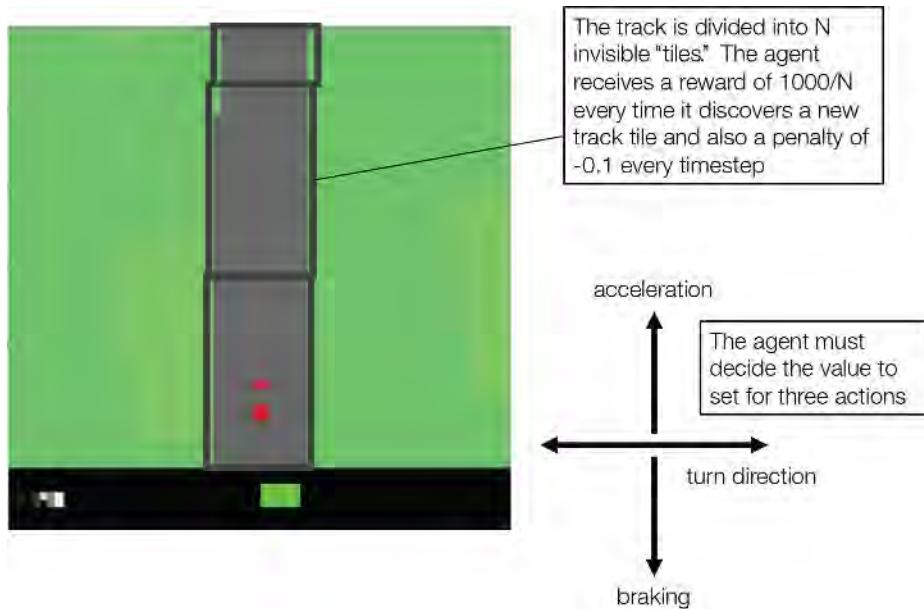


Figure 8-2. A graphical representation of one game state in the CarRacing environment

World Model Architecture

We'll now cover a high-level overview of the entire architecture that we will be using to build the agent that learns through reinforcement learning, before we explore the detailed steps required to build each component.

The solution consists of three distinct parts, as shown in [Figure 8-3](#), that are trained separately:

V

A variational autoencoder.

M

A recurrent neural network with a mixture density network (MDN-RNN).

C

A controller.

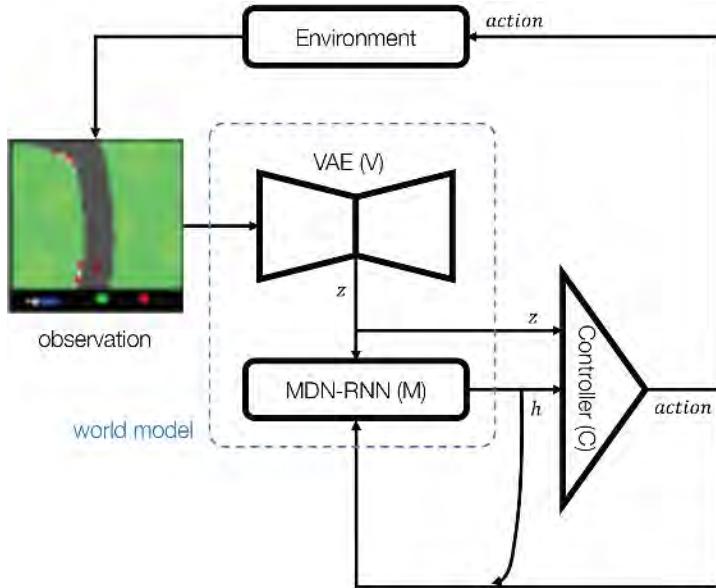


Figure 8-3. World model architecture diagram

The Variational Autoencoder

When you make decisions while driving, you don't actively analyze every single pixel in your view—instead, you condense the visual information into a smaller number of latent entities, such as the straightness of the road, upcoming bends, and your position relative to the road, to inform your next action.

We saw in [Chapter 3](#) how a VAE can take a high-dimensional input image and condense it into a latent random variable that approximately follows a standard multivariate normal distribution, through minimization of the reconstruction error and KL divergence. This ensures that the latent space is continuous and that we are able to easily sample from it to generate meaningful new observations.

In the car racing example, the VAE condenses the $64 \times 64 \times 3$ (RGB) input image into a 32-dimensional normally distributed random variable, parameterized by two variables, `mu` and `log_var`. Here, `log_var` is the logarithm of the variance of the distribution.

We can sample from this distribution to produce a latent vector z that represents the current state. This is passed on to the next part of the network, the MDN-RNN.

The MDN-RNN

As you drive, each subsequent observation isn't a complete surprise to you. If the current observation suggests a left turn in the road ahead and you turn the wheel to the left, you expect the next observation to show that you are still in line with the road.

If you didn't have this ability, your driving would probably snake all over the road as you wouldn't be able see that a slight deviation from the center is going to be worse in the next timestep unless you do something about it now.

This forward thinking is the job of the MDN-RNN, a network that tries to predict the distribution of the next latent state based on the previous latent state and the previous action.

Specifically, the MDN-RNN is an LSTM layer with 256 hidden units followed by a mixture density network (MDN) output layer that allows for the fact that the next latent state could actually be drawn from any one of several normal distributions.

The same technique was applied by one of the authors of the “World Models” paper, David Ha, to a [handwriting generation](#) task, as shown in [Figure 8-4](#), to describe the fact that the next pen point could land in any one of the distinct red areas.

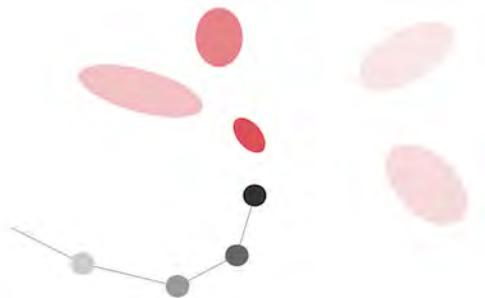


Figure 8-4. MDN for handwriting generation

In the car racing example, we allow for each element of the next observed latent state to be drawn from any one of five normal distributions.

The Controller

Until this point, we haven't mentioned anything about choosing an action. That responsibility lies with the controller.

The controller is a densely connected neural network, where the input is a concatenation of z (the current latent state sampled from the distribution encoded by the VAE) and the hidden state of the RNN. The three output neurons correspond to the three actions (turn, accelerate, brake) and are scaled to fall in the appropriate ranges.

We will need to train the controller using reinforcement learning as there is no training dataset that will tell us that a certain action is *good* and another is *bad*. Instead, the agent will need to discover this for itself through repeated experimentation.

As we shall see later in the chapter, the crux of the “World Models” paper is that it demonstrates how this reinforcement learning can take place within the agent’s own generative model of the environment, rather than the OpenAI Gym environment. In other words, it takes place in the agent’s *hallucinated* version of how the environment behaves, rather than the real thing.

To understand the different roles of the three components and how they work together, we can imagine a dialogue between them:

VAE (looking at latest $64 \times 64 \times 3$ observation): This looks like a straight road, with a slight left bend approaching, with the car facing in the direction of the road (z).

RNN: Based on that description (z) and the fact that the controller chose to accelerate hard at the last timestep (*action*), I will update my hidden state so that the next observation is predicted to still be a straight road, but with slightly more left turn in view.

Controller: Based on the description from the VAE (z) and the current hidden state from the RNN (h), my neural network outputs $[0.34, 0.8, 0]$ as the next action.

The action from the controller is then passed to the environment, which returns an updated observation, and the cycle begins again.

For further information on the model, there is also an excellent interactive explanation available [online](#).

Setup

We are now ready to start exploring how to build and train this model in Keras. If you’ve got a high-spec laptop, you can run the solution locally, but I’d recommend using cloud resources such as [Google Cloud Compute Engine](#) for access to powerful machines that you can use in short bursts.



The following code has been tested on Ubuntu 16.04, so it is specific to a Linux terminal.

First install the following libraries:

```
sudo apt-get install cmake swig python3-dev \
    zlib1g-dev python-opengl mpich xvfb \
    xserver-xephyr vnc4server
```

Then clone the following repository:

```
git clone https://github.com/AppliedDataSciencePartners/WorldModels.git
```

As the codebase for this project is stored separately from the book repository, I suggest creating a separate virtual environment to work in:

```
mkvirtualenv worldmodels  
cd WorldModels  
pip install -r requirements.txt
```

Now you're good to go!

Training Process Overview

Here's an overview of the five-step training process:

1. Collect random rollout data Here, the agent does not care about the given task, but instead simply explores the environment at random. This will be conducted using OpenAI Gym to simulate multiple episodes and store the observed state, action, and reward at each timestep. The idea here is to build up a dataset of how the physics of the environment works, which the VAE can then learn from to capture the states efficiently as latent vectors. The MDN-RNN can then subsequently learn how the latent vectors evolve over time.
2. Train the VAE Using the randomly collected data, we train a VAE on the observation images.
3. Collect data to train the MDN-RNN Once we have a trained VAE, we use it to encode each of the collected observations into `mu` and `log_var` vectors, which are saved alongside the current action and reward.
4. Train the MDN-RNN We take batches of 100 episodes and load the corresponding `mu`, `log_var`, `action`, and `reward` variables at each timestep that were generated in step 3. We then sample a `z` vector from the `mu` and `log_var` vectors. Given the current `z` vector, `action`, and `reward`, the MDN-RNN is then trained to predict the subsequent `z` vector and `reward`.
5. Train the controller With a trained VAE and RNN, we can now train the controller to output an action given the current `z` and hidden state, `h`, of the RNN. The controller uses an evolutionary algorithm, CMA-ES (Covariance Matrix Adaptation Evolution Strategy), as its optimizer. The algorithm rewards matrix weightings that generate actions that lead to overall high scores on the task, so that future generations are also likely to inherit this desired behavior.

Let's now take a closer look at each of these steps in more detail.

Collecting Random Rollout Data

To start collecting data, run the following command from your terminal:

```
bash 01_generate_data.sh <env_name> <parallel_process> <episodes_per_process> \  
<render> <action_refresh_rate>
```

where the parameters are as follows:

<env_name>

The name of the environment used by the `make_env` function (e.g., `car_racing`).

<parallel_process>

The number of processes to run (e.g., 8 for an 8-core machine).

<episodes_per_process>

How many episodes each process should run (e.g., 125, so 8 processes would create 1,000 episodes overall).

<max_timesteps>

The maximum number of timesteps per episode (e.g., 300).

<render>

1 to render the rollout process in a window (otherwise 0).

<action_refresh_rate>

The number of timesteps to freeze the current action for before changing. This prevents the action from changing too rapidly for the car to make progress.

For example, on an 8-core machine, you could run:

```
bash 01_generate_data.sh car_racing 8 125 300 0 5
```

This would start 8 processes running in parallel, each simulating 125 episodes, with a maximum of 300 timesteps each and an action refresh rate of 5 timesteps.

Each process calls the Python file `01_generate_data.py`. The key part of the script is outlined in [Example 8-1](#).

Example 8-1. 01_generate_data.py excerpt

```
# ...  
  
DIR_NAME = './data/rollout/'  
  
env = make_env(current_env_name) ❶  
s = 0  
while s < total_episodes:  
    episode_id = random.randint(0, 2**31-1)  
    filename = DIR_NAME + str(episode_id)+".npz"  
    observation = env.reset()  
    env.render()  
    t = 0  
    obs_sequence = []
```

```

action_sequence = []
reward_sequence = []
done_sequence = []
reward = -0.1
done = False

while t < time_steps:
    if t % action_refresh_rate == 0:
        action = config.generate_data_action(t, env) ②
        observation = config.adjust_obs(observation) ③
        obs_sequence.append(observation)
        action_sequence.append(action)
        reward_sequence.append(reward)
        done_sequence.append(done)
        observation, reward, done, info = env.step(action) ④
        t = t + 1

print("Episode {} finished after {} timesteps".format(s, t))
np.savez_compressed(filename
    , obs=obs_sequence
    , action=action_sequence
    , reward = reward_sequence
    , done = done_sequence) ⑤
s = s + 1
env.close()

```

- ➊ `make_env` is a custom function in the repository that creates the appropriate OpenAI Gym environment. In this case, we are creating the `CarRacing` environment, with a few tweaks. The `environment` file is stored in the `custom_envs` folder.
- ➋ `generate_data_action` is a custom function that stores the rules for generating random actions.
- ➌ The observations that are returned by the environment are scaled between 0 and 255. We want observations that are scaled between 0 and 1, so this function is simply a division by 255.
- ➍ Every OpenAI Gym environment includes a `step` method. This returns the next observation, reward, and done flag, given an action.
- ➎ We save each episode as an individual file inside the `./data/rollout/` directory.

Figure 8-5 shows an excerpt from frames 40 to 59 of one episode, as the car approaches a corner, alongside the randomly chosen action and reward. Note how the reward changes to 3.22 as the car rolls over new track tiles but is otherwise -0.1. Also, the action changes every five frames as the `action_refresh_rate` is 5.



Figure 8-5. Frames 40 to 59 of one episode

Training the VAE

We can now build a generative model (a VAE) on this collected data.

Remember, the aim of the VAE is to allow us to collapse one $64 \times 64 \times 3$ image into a normally distributed random variable, whose distribution is parameterized by two vectors, μ and \log_{var} . Each of these vectors is of length 32.

To start training the VAE, run the following command from your terminal:

```
python 02_train_vae.py --new_model [--N] [--epochs]
```

where the parameters are as follows:

--new_model

Whether the model should be trained from scratch. Set this flag initially; if it's not set, the code will look for a `./vae/vae.json` file and continue training a previous model.

--N (optional)

The number of episodes to use when training the VAE (e.g., 1000—the VAE does not need to use all the episodes to achieve good results, so to speed up training you can use only a sample of the episodes).

--epochs (optional)

The number of training epochs (e.g., 3).

The output of the training process should be as shown in [Figure 8-6](#). A file storing the weights of the trained network is saved to `./vae/vae.json` every epoch.

```
Imported 850 / 1000 :: Current data size = 255000 observations
Imported 900 / 1000 :: Current data size = 270000 observations
Imported 950 / 1000 :: Current data size = 285000 observations
Imported 1000 / 1000 :: Current data size = 300000 observations
Imported 1000 / 1000 :: Current data size = 300000 observations
DATA SHAPE = (300000, 64, 64, 3)
EPOCH 0
Epoch 1/1
300000/300000 [=====] - 2632s 9ms/step - loss: 86.7616 - vae_r_loss: 69.8272 - vae_kl_loss: 16.9344
EPOCH 1
Epoch 1/1
300000/300000 [=====] - 5270s 18ms/step - loss: 48.5049 - vae_r_loss: 32.0335 - vae_kl_loss: 16.4714
EPOCH 2
Epoch 1/1
300000/300000 [=====] - 5151s 17ms/step - loss: 43.7458 - vae_r_loss: 27.3376 - vae_kl_loss: 16.4081
EPOCH 3
Epoch 1/1
300000/300000 [=====] - 3423s 11ms/step - loss: 41.6400 - vae_r_loss: 25.2663 - vae_kl_loss: 16.3737
```

Figure 8-6. Training the VAE

The VAE Architecture

As we have seen previously, the Keras functional API allows us to not only define the full VAE model that will be trained, but also additional models that reference the encoder and decoder parts of the trained network separately. These will be useful when we want to encode a specific image, or decode a given z vector, for example.

In this example, we define four different models on the VAE:

full_model

This is the full end-to-end model that is trained.

encoder

This accepts a $64 \times 64 \times 3$ observation as input and outputs a sampled z vector. If you run the `predict` method of this model for the same input multiple times, you

will get different output, since even though the `mu` and `log_var` values are constant, the randomly sampled `z` vector will be different each time.

`encoder_mu_log_var`

This accepts a $64 \times 64 \times 3$ observation as input and outputs the `mu` and `log_var` vectors corresponding to this input. Unlike with the `vae_encoder` model, if you run the `predict` method of this model multiple times, you will always get the same output: a `mu` vector of length 32 and a `log_var` vector of length 32.

`decoder`

This accepts a `z` vector as input and returns the reconstructed $64 \times 64 \times 3$ observation.

A diagram of the VAE is given in [Figure 8-7](#). You can play around with the VAE architecture by editing the `./vae/arch.py` file. This is where the VAE class and parameters of the neural network are defined.

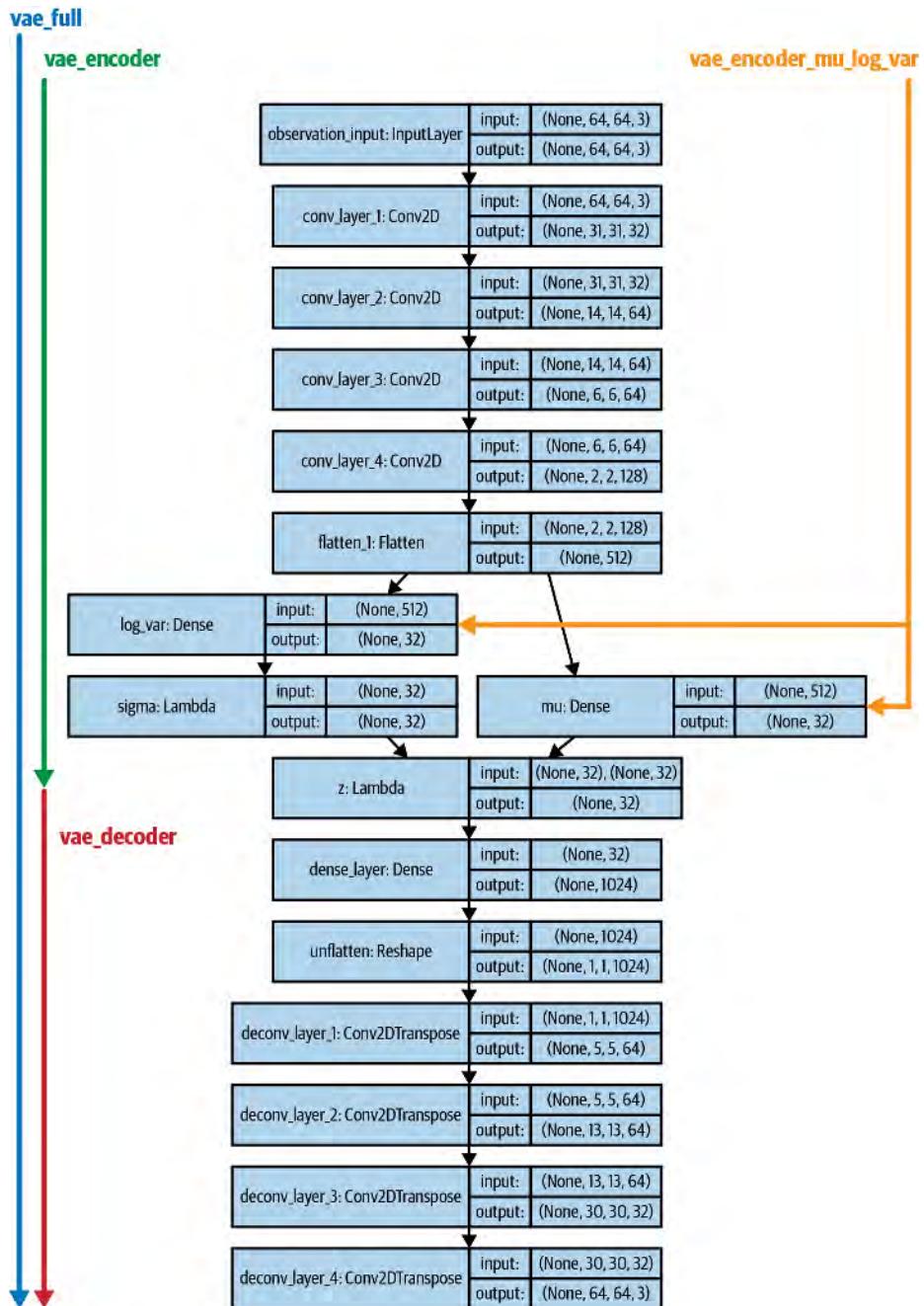


Figure 8-7. The VAE architecture for the “World Models” paper

Exploring the VAE

We'll now take a look at the output from the `predict` methods of the different models built on the VAE to see how they differ, and then see how the VAE can be used to generate completely new track observations.

The full model

If we feed the `full_model` with an observation, it is able to reconstruct an accurate representation of the image, as shown in [Figure 8-8](#). This is useful to visually check that the VAE is working correctly.

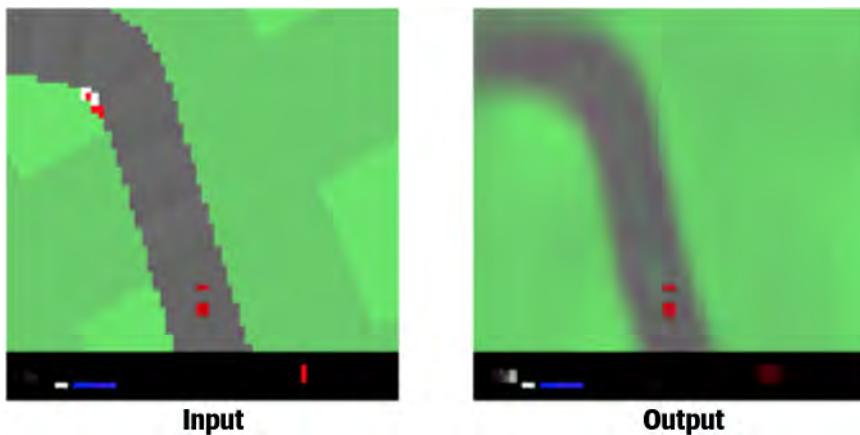


Figure 8-8. The input and output from the full VAE model

The encoder models

If we feed the `encoder_mu_log_var` model with an observation, the output is the generated `mu` and `log_var` vectors describing a multivariate normal distribution.

The `encoder` model goes one step further by sampling a particular `z` vector from this distribution.

The output from the two encoder models is shown in [Figure 8-9](#).

```

encoded_mu_log_var = vae.encoder_mu_log_var.predict(np.array([obs]))
mu = encoded_mu_log_var[0][0]
log_var = encoded_mu_log_var[1][0]
print("mu = " + str(mu))
print("log_var = " + str(log_var))

encoded_z = vae.encoder.predict(np.array([obs]))[0]
print("z = " + str(encoded_z))

mu = [ 0.461  0.2408  0.9884  0.0459  0.9623  1.0147  0.6555 -0.586 -0.4304
-0.0892 -1.5475  0.0319  0.426  0.1936  0.0838 -0.9127 -0.0823 -0.0321
0.0829 -0.2372 -0.1084 -0.0069  0.011 -0.0201 -1.735 -0.8933  0.0082
0.5342  0.1982  0.056 -0.0389  0.0389]
log_var = [-1.8887 -0.1232 -2.1576  0.0093 -5.2961 -2.2058 -3.0662 -3.1458  0.0018
-0.0412 -3.1433  0.0278 -0.0169  0.0186 -0.0313 -0.8884 -0.0468 -0.0565
0.0075  0.0149 -0.0285 -0.0013 -0.006 -0.0468 -3.2442 -3.9082  0.0348
-2.0272 -0.0264 -0.0202 -0.0585 -0.0103]
z = [ 0.4681  0.3364  0.969 -0.3118  1.0631  1.1805  1.1039 -0.6603 -1.5317
-0.0679 -1.4819 -0.537  1.2687 -1.2981 -0.0219 -0.7657  0.5345 -1.3642
-0.1958 -1.2382  1.6123 -1.9301  0.1032 -0.1125 -1.8621 -1.2199  0.3788
0.6171  0.6412 -0.6444 -0.0741 -1.4384]

```

Figure 8-9. The output from the encoder models

It is interesting to plot the value of `mu` and `log_var` for each of the 32 dimensions (Figure 8-10), for a particular observation. Notice how only 12 of the 32 dimensions differ significantly from the standard normal distribution (`mu = 0, log_var = 0`). This is because the VAE is trying to minimize the KL divergence, so it tries to differ from the standard normal distribution in as few dimensions as possible. It has decided that 12 dimensions are enough to capture sufficient information about the observations for accurate reconstruction.

```
plt.plot(mu);
plt.plot(log_var);
hot_zs = np.where(abs(log_var) > 0.1)[0]
hot_zs
array([ 0,  1,  2,  4,  5,  6,  7, 10, 15, 24, 25, 27])
```

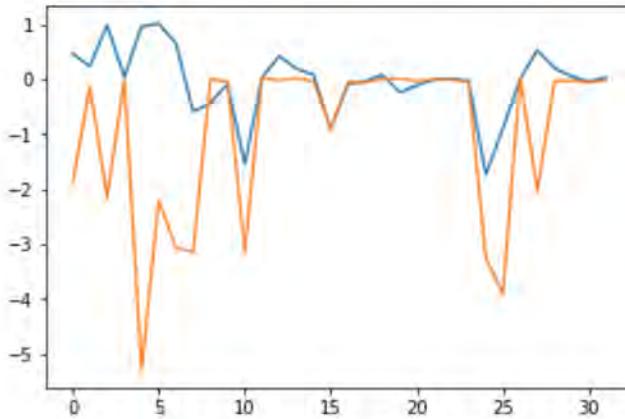


Figure 8-10. A plot of μ (blue line) and \log_{var} (orange line) for each of the 32 dimensions of a particular observation

The decoder model

The decoder model accepts a z vector as input and reconstructs the original image. In Figure 8-11 we linearly interpolate two of the dimensions of z to show how each dimension appears to encode a particular aspect of the track—for example, $z[4]$ controls the immediate left/right direction of the track nearest the car and $z[7]$ controls the sharpness of the approaching left turn.

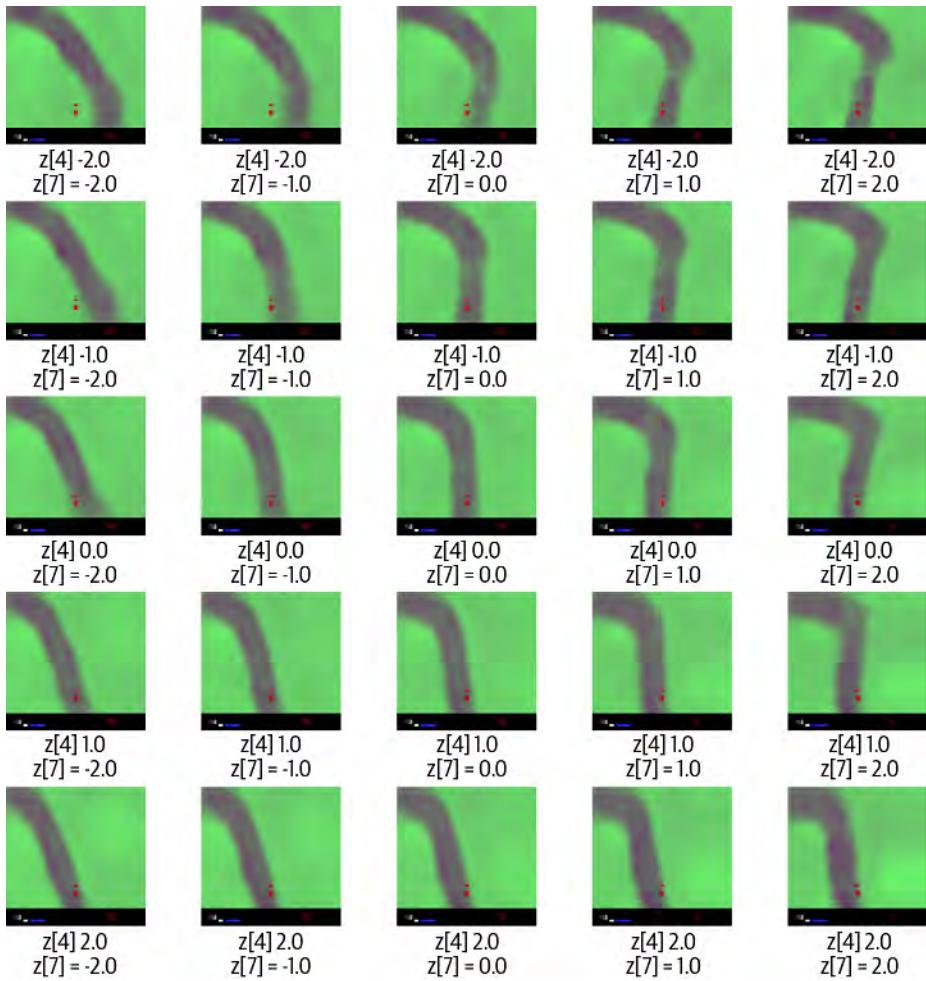


Figure 8-11. A linear interpolation of two dimensions of z

This shows that the latent space that the VAE has learned is continuous and can be used to generate new track segments that have never before been observed by the agent.

Collecting Data to Train the RNN

Now that we have a trained VAE, we can use this to generate training data for our RNN.

In this step, we pass all of the random rollout data through the `encoder_mu_log_var` model and store the `mu` and `log_var` vectors corresponding to each observation. This

encoded data, along with the already collected actions and rewards, will be used to train the MDN-RNN.

To start collecting data, run the following command from your terminal:

```
python 03_generate_rnn_data.py
```

Example 8-2 contains an excerpt from the *03_generate_data.py* file that shows how the MDN-RNN training data is generated.

Example 8-2. Excerpt from 03_generate_data.py

```
def encode_episode(vae, episode):

    obs = episode['obs']
    action = episode['action']
    reward = episode['reward']
    done = episode['done']

    mu, log_var = vae.encoder_mu_log_var.predict(obs) ❶

    done = done.astype(int)
    reward = np.where(reward>0, 1, 0) * np.where(done==0, 1, 0) ❷

    initial_mu = mu[0, :]
    initial_log_var = log_var[0, :]

    return (mu, log_var, action, reward, done, initial_mu, initial_log_var)

vae = VAE()
vae.set_weights('./vae/weights.h5')

for file in filelist:
    rollout_data = np.load(ROLLOUT_DIR_NAME + file)
    mu, log_var, action, reward, done, initial_mu
        , initial_log_var = encode_episode(vae, rollout_data)

    np.savez_compressed(SERIES_DIR_NAME + file, mu=mu, log_var=log_var
        , action = action, reward = reward, done = done)
    initial_mus.append(initial_mu)
    initial_log_vars.append(initial_log_var)

np.savez_compressed(ROOT_DIR_NAME + 'initial_z.npz', initial_mu=initial_mus
    , initial_log_var=initial_log_vars) ❸
```

- ❶ Here, we're using the `encoder_mu_log_var` model of the VAE to get the `mu` and `log_var` vectors for a particular observation.
- ❷ The reward value is transformed to be either 0 or 1, so that it can be used as input into the MDN-RNN.

- ③ We also save the initial `mu` and `log_var` vectors into a separate file—this will be useful later, for initializing the dream environment.

Training the MDN-RNN

We can now train the MDN-RNN to predict the distribution of the next z vector and reward, given the current z value, current action, and previous reward.

The aim of the MDN-RNN is to predict one timestep ahead into the future—we can then use the internal hidden state of the LSTM as part of the input into the controller.

To start training the MDN-RNN, run the following command from your terminal:

```
python 04_train_rnn.py (--new_model) (--batch_size) (--steps)
```

where the parameters are as follows:

`new_model`

Whether the model should be trained from scratch. Set this flag initially; if it's not set, the code will look for a `./rnn/rnn.json` file and continue training a previous model.

`batch_size`

The number of episodes fed to the MDN-RNN in each training iteration.

`steps`

The total number of training iterations.

The output of the training process is shown in [Figure 8-12](#). A file storing the weights of the trained network is saved to `./rnn/rnn.json` every 10 steps.

```
STEP 0
Epoch 1/1
100/100 [=====] - 1s 20ms/step - loss: 2.0871 - rnn_z_loss: 1.3943 - rnn_rew_loss: 0.6928
STEP 1
Epoch 1/1
100/100 [=====] - 1s 12ms/step - loss: 2.0435 - rnn_z_loss: 1.3928 - rnn_rew_loss: 0.6507
STEP 2
Epoch 1/1
100/100 [=====] - 1s 12ms/step - loss: 2.0018 - rnn_z_loss: 1.3881 - rnn_rew_loss: 0.6138
STEP 3
Epoch 1/1
100/100 [=====] - 1s 12ms/step - loss: 1.9670 - rnn_z_loss: 1.3881 - rnn_rew_loss: 0.5789
STEP 4
Epoch 1/1
100/100 [=====] - 1s 12ms/step - loss: 1.9073 - rnn_z_loss: 1.3794 - rnn_rew_loss: 0.5278
STEP 5
Epoch 1/1
100/100 [=====] - 1s 11ms/step - loss: 1.8410 - rnn_z_loss: 1.3711 - rnn_rew_loss: 0.4700
STEP 6
Epoch 1/1
100/100 [=====] - 1s 11ms/step - loss: 1.7674 - rnn_z_loss: 1.3606 - rnn_rew_loss: 0.4068
```

Figure 8-12. Training the MDN-RNN

The MDN-RNN Architecture

The architecture of the MDN-RNN is shown in [Figure 8-13](#).

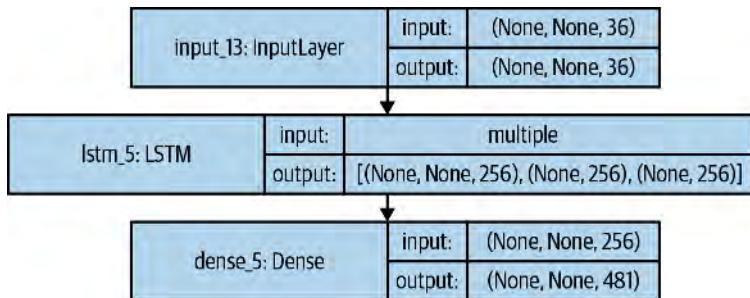


Figure 8-13. The MDN-RNN architecture

It consists of an LSTM layer (the RNN), followed by a densely connected layer (the MDN) that transforms the hidden state of the LSTM into the parameters of mixture distribution. Let's walk through the network step by step.

The input to the LSTM layer is a vector of length 36—a concatenation of the encoded z vector (length 32) from the VAE, the current action (length 3), and the previous reward (length 1).

The output from the LSTM layer is a vector of length 256—one value for each LSTM cell in the layer. This is passed to the MDN, which is just a densely connected layer that transforms the vector of length 256 into a vector of length 481.

Why 481? [Figure 8-14](#) explains the composition of the output from the MDN-RNN. Remember, the aim of a mixture density network is to model the fact that our next z could be drawn from one of several possible distributions with a certain probability. In the car racing example, we choose five normal distributions. How many parameters do we need to define these distributions? For each of the five mixtures, we need a μ and a \log_σ (to define the distribution) and a probability of this mixture being chosen (\log_π), for each of the 32 dimensions of z . This makes $5 \times 3 \times 32 = 480$ parameters. The one extra parameter is for the reward prediction—more specifically, the log odds of reward at the next timestep.

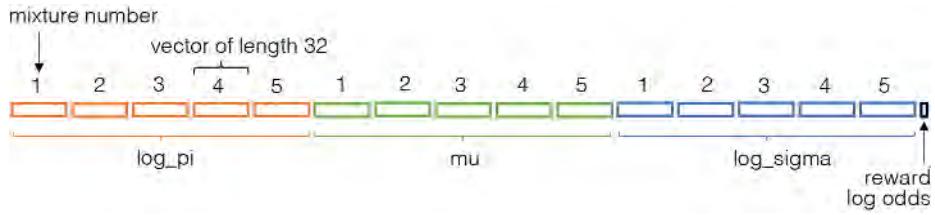


Figure 8-14. The output from the mixture density network

Sampling the Next z and Reward from the MDN-RNN

We can sample from the MDN output to generate a prediction for the next z and reward at the following timestep, through the following process:

1. Split the 481-dimensional output vector into the 3 variables (`log_pi`, `mu`, `log_sigma`) and the reward value.
2. Exponentiate and scale `log_pi` so that it can be interpreted as 32 probability distributions over the 5 mixture indices.
3. For each of the 32 dimensions of z , sample from the distributions created from `log_pi` (i.e., choose which of the 5 distributions should be used for each dimension of z).
4. Fetch the corresponding values of `mu` and `log_sigma` for this distribution.
5. Sample a value for each dimension of z from the normal distribution parameterized by the chosen parameters of `mu` and `log_sigma` for this dimension.
6. If the reward log odds value is greater than 0, predict 1 for the reward; otherwise, predict 0.

The MDN-RNN Loss Function

The loss function for the MDN-RNN is the sum of the z vector reconstruction loss and the reward loss.

The excerpt from the `rnn/arch.py` file for the MDN-RNN in Example 8-3 shows how we construct the custom loss function in Keras.

Example 8-3. Excerpt from rnn/arch.py

```
def get_responses(self, y_true):
    z_true = y_true[:, :, :Z_DIM]
    rew_true = y_true[:, :, -1]
```

```

    return z_true, rew_true

def get_mixture_coef(self, z_pred):
    log_pi, mu, log_sigma = tf.split(z_pred, 3, 1)
    log_pi = log_pi - K.log(K.sum(K.exp(log_pi), axis = 1, keepdims = True))
    return log_pi, mu, log_sigma

def tf_lognormal(self, z_true, mu, log_sigma):
    logSqrtTwoPI = np.log(np.sqrt(2.0 * np.pi))
    return -0.5 * ((z_true - mu) / K.exp(log_sigma)) ** 2 - log_sigma - logSqrtTwoPI

def rnn_z_loss(y_true, y_pred):
    z_true, rew_true = self.get_responses(y_true)

    d = normal_distribution_MIXTURES * Z_DIM
    z_pred = y_pred[:, :, : (3 * d)]
    z_pred = K.reshape(z_pred, [-1, normal_distribution_MIXTURES * 3])

    log_pi, mu, log_sigma = self.get_mixture_coef(z_pred) ❶

    flat_z_true = K.reshape(z_true, [-1, 1])

    z_loss = log_pi + self.tf_lognormal(flat_z_true, mu, log_sigma) ❷
    z_loss = -K.log(K.sum(K.exp(z_loss), 1, keepdims=True))

    z_loss = K.mean(z_loss)

    return z_loss

def rnn_rew_loss(y_true, y_pred):
    z_true, rew_true = self.get_responses(y_true) #, done_true

    d = normal_distribution_MIXTURES * Z_DIM
    reward_pred = y_pred[:, :, -1]

    rew_loss = K.binary_crossentropy(rew_true, reward_pred, from_logits = True) ❸
    rew_loss = K.mean(rew_loss)

    return rew_loss

def rnn_loss(y_true, y_pred):
    z_loss = rnn_z_loss(y_true, y_pred)
    rew_loss = rnn_rew_loss(y_true, y_pred)

```

```
    return Z_FACTOR * z_loss + REWARD_FACTOR * rew_loss ④

opti = Adam(lr=LEARNING_RATE)
model.compile(loss=rnn_loss, optimizer=opti, metrics = [rnn_z_loss, rnn_rew_loss])
```

- ❶ Split the 481-dimensional output vector into the 3 variables (`log_pi`, `mu`, `log_sigma`) and the reward value.
- ❷ This is the calculation of the `z` vector reconstruction loss: the negative log-likelihood of observing the true `z`, under the mixture distribution parameterized by the output from the MDN-RNN. We want this value to be as large as possible, or equivalently, we seek to minimize the negative log likelihood.
- ❸ For the reward loss, we simply use the binary cross entropy between the true reward and the predicted log odds from the network.
- ❹ The loss is the sum of the `z` reconstruction loss and the reward loss—we set the weighting parameters `Z_FACTOR` and `REWARD_FACTOR` both to 1, though these can be adjusted to prioritize reconstruction loss or reward loss.

Notice that to train the MDN-RNN, we do not need to sample specific `z` vectors from the MDN output, but instead calculate the loss directly using the 481-dimensional output vector.

Training the Controller

The final step is to train the controller (the network that outputs the chosen action) using an evolutionary algorithm called CMA-ES (Covariance Matrix Adaptation Evolution Strategy).

To start training the controller, run the following command from your terminal (all on one line):

```
xvfb-run -a -s "-screen 0 1400x900x24" python 05_train_controller.py car_racing  
-n 16 -t 2 -e 4 --max_length 1000
```

where the parameters are as follows:

n

The number of workers that will test solutions in parallel (this should be no greater than the number of cores on your machine)

t

The number of solutions that each worker will be given to test at each generation

e

The number of episodes that each solution will be tested against to calculate the average reward

max_length

The maximum number of timeframes in each episode

eval_steps

The number of generations between evaluations of the current best parameter set

The above command uses a virtual frame buffer (xvfb) to render the frames, so that the code can run on a Linux machine without a physical screen. The population size, `pop_size = n * t`.

The Controller Architecture

The architecture of the controller is very simple. It is a densely connected neural network with no hidden layer; it connects the input vector directly to the action vector.

The input vector is a concatenation of the current `z` vector (length 32) and the current hidden state of the LSTM (length 256), giving a vector of length 288. Since we are connecting each input unit directly to the 3 output action units, the total number of weights to tune is $288 \times 3 = 864$, plus 3 bias weights, giving 867 in total.

How should we train this network? Notice that this is not a supervised learning problem—we are not trying to *predict* the correct action. There is no training set of *correct* actions, as we do not know what the optimal action is for a given state of the environment. This is what distinguishes this as a reinforcement learning problem. We need the agent to discover the optimal values for the weights itself by experimenting within the environment and updating its weights based on received feedback.

Evolutionary strategies are becoming a popular choice for solving reinforcement learning problems, due to their simplicity, efficiency, and scalability. We shall use one particular strategy, known as CMA-ES.

CMA-ES

Evolutionary strategies generally adhere to the following process:

1. Create a *population* of agents and randomly initialize the parameters to be optimized for each agent.
2. Loop over the following:
 - a. Evaluate each agent in the environment, returning the average reward over multiple episodes.

- b. Breed the agents with the best scores to create new members of the population.
- c. Add randomness to the parameters of the new members.
- d. Update the population pool by adding the newly created agents and removing poorly performing agents.

This is similar to the process through which animals evolve in nature—hence the name *evolutionary* strategies. “Breeding” in this context simply means combining the existing best-scoring agents such that the next generation are more likely to produce high-quality results, similar to their parents. As with all reinforcement learning solutions, there is a balance to be found between greedily searching for locally optimal solutions and exploring unknown areas of the parameter space for potentially better solutions. This is why it is important to add randomness to the population, to ensure we are not too narrow in our search field.

CMA-ES is just one form of evolutionary strategy. In short, it works by maintaining a normal distribution from which it can sample the parameters of new agents. At each generation, it updates the mean of the distribution to maximize the likelihood of sampling the high-scoring agents from the previous timestep. At the same time, it updates the covariance matrix of the distribution to maximize the likelihood of sampling the high-scoring agents, given the previous mean. It can be thought of as a form of naturally arising gradient descent, but with the added benefit that it is derivative-free, meaning that we do not need to calculate or estimate costly gradients.

One generation of the algorithm demonstrated on a toy example is shown in [Figure 8-15](#). Here we are trying to find the minimum point of a highly nonlinear function in two dimensions—the value of the function in the red/black areas of the image is greater than the value of the function in the white/yellow parts of the image.

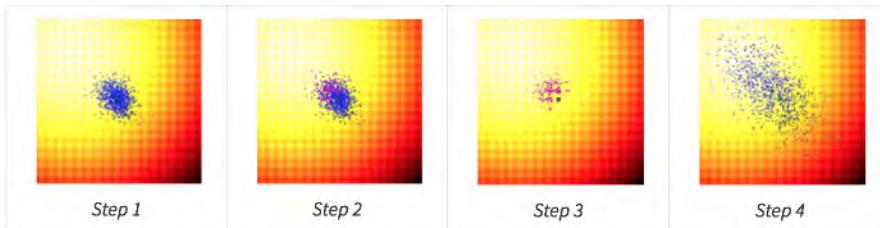


Figure 8-15. One update step from the CMA-ES algorithm²

² Reproduced with permission from David Ha, 2017, <http://bit.ly/2XufRwq>.

The steps are as follows:

1. We start with a randomly generated 2D normal distribution and sample a population of candidates, shown in blue.
2. We then calculate the value of the function for each candidate and isolate the best 25%, shown in purple—we'll call this set of points P .
3. We set the mean of the new normal distribution to be the mean of the points in P . This can be thought of as the breeding stage, wherein we only use the best candidates to generate a new mean for the distribution. We also set the covariance matrix of the new normal distribution to be the covariance matrix of the points in P , but use the existing mean in the covariance calculation rather than the current mean of the points in P . The larger the difference between the existing mean and the mean of the points in P , the wider the variance of the next normal distribution. This has the effect of naturally creating *momentum* in the search for the optimal parameters.
4. We can then sample a new population of candidates from our new normal distribution with an updated mean and covariance matrix.

Figure 8-16 shows several generations of the process. See how the covariance widens as the mean moves in large steps toward the minimum, but narrows as the mean settles into the true minimum.

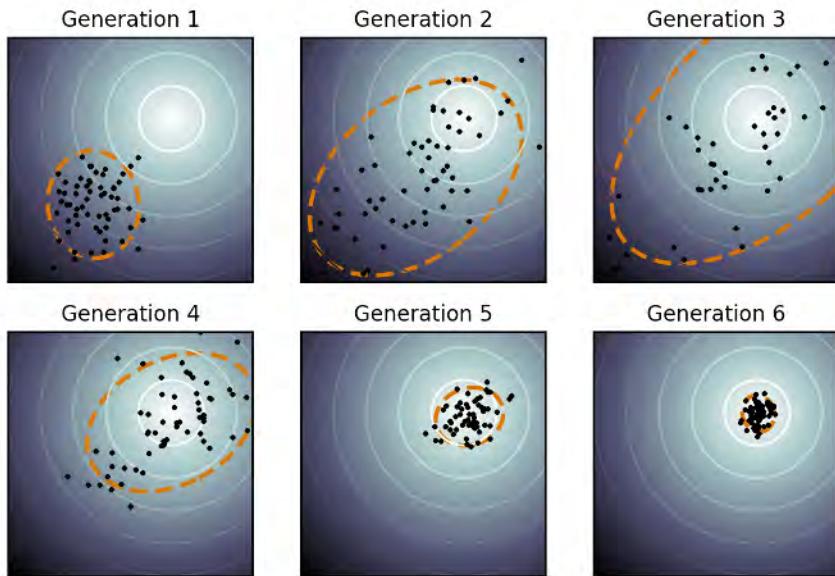


Figure 8-16. CMA-ES³

For the car racing task, we do not have a well-defined function to maximize, but instead an environment where the 867 parameters to be optimized determine how well the agent scores. Initially, some sets of parameters will, by random chance, generate scores that are higher than others and the algorithm will gradually move the normal distribution in the direction of those parameters that score highest in the environment.

Parallelizing CMA-ES

One of the great benefits of CMA-ES is that it can be easily parallelized using a Python library created by David Ha called `es.py`. The most time-consuming part of the algorithm is calculating the score for a given set of parameters, since it needs to simulate an agent with these parameters in the environment. However, this process can be parallelized, since there are no dependencies between individual simulations. In the codebase, we use a master/slave setup, where there is a master process that sends out parameter sets to be tested to many slave processes in parallel. The slave nodes return the results to the master, which accumulates the results and then passes the overall result of the generation to the CMA-ES object. This object updates the

³ Source: <https://en.wikipedia.org/wiki/CMA-ES>.

mean and covariance matrix of the normal distribution as per [Figure 8-15](#) and provides the master with a new population to test. The loop then starts again. [Figure 8-17](#) explains this in a diagram.

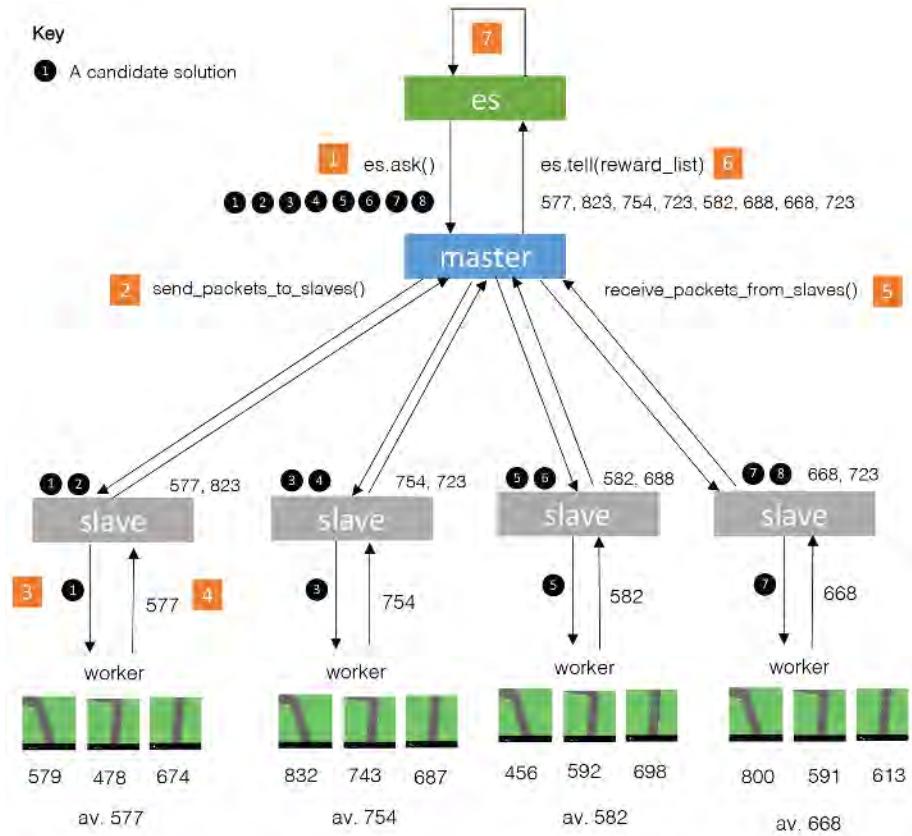


Figure 8-17. Parallelizing CMA-ES—here there is a population size of 8 and 4 slave nodes (so $t = 2$, the number of trials that each slave is responsible for)

- ① The master asks the CMA-ES object (es) for a set of parameters to trial.
- ② The master divides the parameters into the number of slave nodes available. Here, each of the four slave processes gets two parameter sets to trial.
- ③ The slave nodes run a worker process that loops over each set of parameters and runs several episodes for each. Here we run three episodes for each set of parameters.

- ④ The rewards from each episode are averaged to give a single score for each set of parameters.
- ⑤ The slave node returns the list of scores to the master.
- ⑥ The master groups all the scores together and sends this list to the es object.
- ⑦ The es object uses this list of rewards to calculate the new normal distribution as per [Figure 8-15](#).

Output from the Controller Training

The output of the training process is shown in [Figure 8-18](#). A file storing the weights of the trained network is saved every eval_steps generations.

```
('car_racing', (1, 151, -77.57, -93.47, -57.68, 13.43, 0.49663, 1000.0, 1000))  
('car_racing', (2, 298, -64.16, -82.57, -26.64, 14.17, 0.49346, 1000.0, 1000))  
('car_racing', (3, 449, -65.13, -76.36, -53.9, 7.41, 0.49046, 1000.0, 1000))  
('car_racing', (4, 595, -63.1, -92.23, -28.59, 16.03, 0.48763, 1000.0, 1000))  
('car_racing', (5, 743, -52.26, -72.98, 1.18, 20.67, 0.48492, 1000.0, 1000))  
('car_racing', (6, 892, -47.26, -74.36, -8.07, 21.21, 0.48233, 1000.0, 1000))  
('car_racing', (7, 1039, -36.81, -57.4, -19.15, 13.37, 0.47984, 1000.0, 1000))  
('car_racing', (8, 1189, -35.75, -76.86, 13.82, 26.54, 0.47744, 1000.0, 1000))  
('car_racing', (9, 1338, -33.7, -79.83, 28.03, 38.95, 0.47511, 1000.0, 1000))  
('car_racing', (10, 1485, -17.43, -63.65, 69.14, 36.05, 0.47284, 1000.0, 1000))  
('car_racing', (11, 1632, -10.99, -63.11, 186.43, 43.87, 0.47063, 1000.0, 1000))  
('car_racing', (12, 1781, 7.1, -70.57, 65.42, 38.67, 0.46848, 1000.0, 1000))  
('car_racing', (13, 1933, 38.88, -61.85, 197.08, 76.47, 0.46638, 1000.0, 1000))  
('car_racing', (14, 2085, 72.03, -31.57, 147.58, 43.74, 0.46433, 1000.0, 1000))  
('car_racing', (15, 2238, 95.18, -15.08, 268.42, 75.69, 0.46234, 1000.0, 1000))  
('car_racing', (16, 2395, 136.41, 33.28, 246.12, 62.78, 0.4604, 1000.0, 1000))  
('car_racing', (17, 2549, 133.55, -1.84, 248.34, 62.98, 0.45854, 1000.0, 1000))  
('car_racing', (18, 2702, 172.56, 64.58, 237.96, 55.28, 0.45672, 1000.0, 1000))  
('car_racing', (19, 2847, 171.07, 51.86, 308.79, 88.94, 0.45497, 1000.0, 1000))
```

Figure 8-18. Training the controller

Each line of the output represents one generation of training. The reported statistics for each generation are as follows:

1. Environment name (e.g., car_racing)
2. Generation number (e.g., 16)
3. Current elapsed time in seconds (e.g., 2395)
4. Average reward of the generation (e.g., 136.44)
5. Minimum reward of the generation (e.g., 33.28)
6. Maximum reward of the generation (e.g., 246.12)
7. Standard deviation of the rewards (e.g., 62.78)
8. Current standard deviation factor of the ES process (initialized at 0.5 and decays each timestep; e.g., 0.4604)

9. Minimum timesteps taken before termination (e.g., 1000.0)
10. Maximum timesteps taken before termination (e.g., 1000)

After `eval_steps` timesteps, each slave node evaluates the current best-scoring parameter set and returns the average rewards across several episodes. These rewards are again averaged to return the overall score for the parameter set.

After around 200 timesteps, the training process achieves an average reward score of 840 for the car racing task.

In-Dream Training

So far, the controller training has been conducted using the OpenAI Gym `CarRacing` environment to implement the `step` method that moves the simulation from one state to the next. This function calculates the next state and reward, given the current state of the environment and chosen action.

Notice how the `step` method performs a very similar function to the MDN-RNN in our model. Sampling from the MDN-RNN outputs a prediction for the next z and reward, given the current z and chosen action.

In fact, the MDN-RNN can be thought of as an environment in its own right, but operating in z -space rather than in the original image space. Incredibly, this means that we can actually substitute the real environment with a copy of the MDN-RNN and train the controller entirely within an MDN-RNN-inspired *dream* of how the environment should behave.

In other words, the MDN-RNN has learned enough about the general physics of the real environment from the original random movement dataset that it can be used as a proxy for the real environment when training the controller. This is quite remarkable—it means that the agent can train itself to learn a new task by *thinking* about how it can maximize reward in its dream environment, without ever having to test out strategies in the real world. It can then perform well at the task first time, having never attempted the task in reality.

This is one reason why the “World Models” paper is highly important and why generative modeling will almost certainly form a key component of artificial intelligence in the future.

A comparison of the architectures for training in the real environment and the dream environment follows: the real-world architecture is shown in [Figure 8-19](#) and the in-dream training setup is illustrated in [Figure 8-20](#).

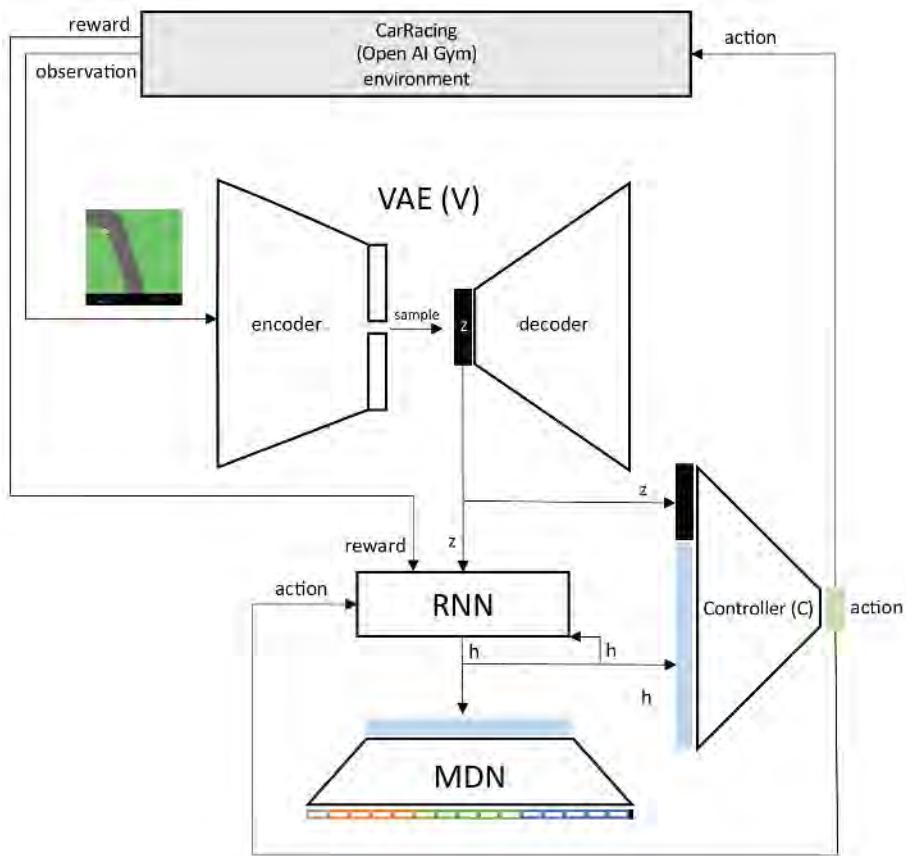


Figure 8-19. Training the controller in the OpenAI Gym environment

Notice how in the dream architecture, the training of the controller is performed entirely in z -space without the need to ever decode the z vectors back into recognizable track images. We can of course do so, in order to visually inspect the performance of the agent, but it is not required for training.

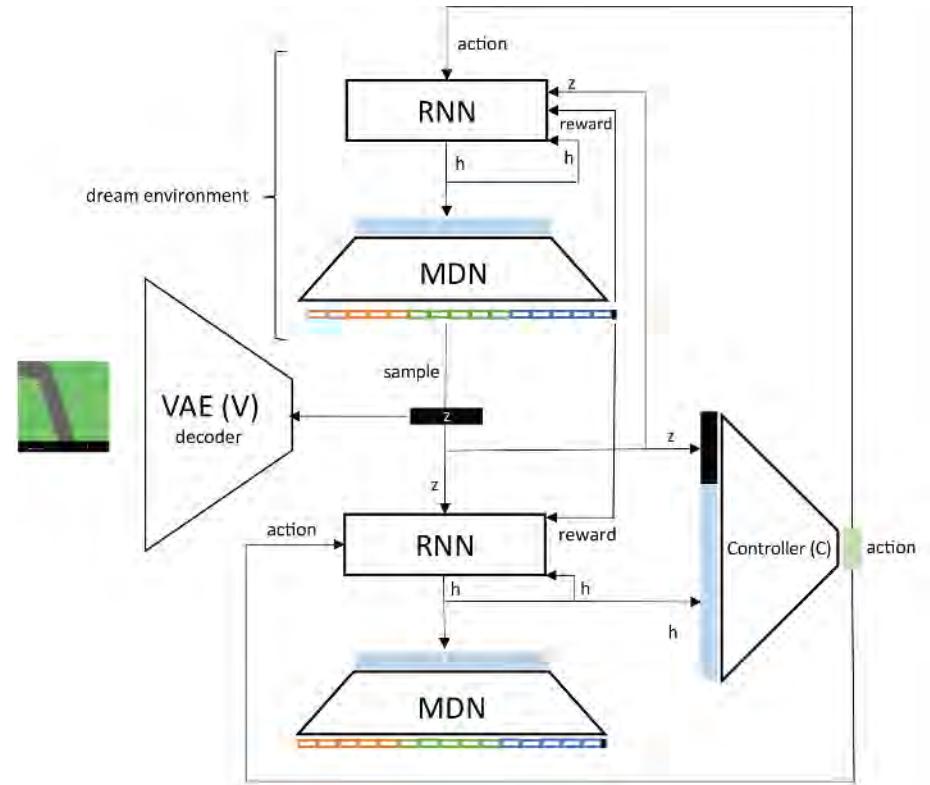


Figure 8-20. Training the controller in the MDN-RNN dream environment

In-Dream Training the Controller

To train the controller using the dream environment, run the following command from your terminal (on one line):

```
xvfb-run -a -s "-screen 0 1400x900x24" python 05_train_controller.py car_racing
-n 16 -t 2 -e 4 --max_length 1000 --dream_mode 1
```

This is the same command used to train the controller in the real environment, but with the added flag `--dream_mode 1`.

The output of the training process is shown in [Figure 8-21](#).

```

(['car_racing', (1, 69, 2.24, 1.0, 10.25, 2.14, 0.09895, 1000.0, 1000))
('car_racing', (2, 133, 5.52, 1.0, 21.25, 4.28, 0.098, 1000.0, 1000))
('car_racing', (3, 196, 16.81, 1.0, 71.75, 16.87, 0.09713, 1000.0, 1000))
('car_racing', (4, 261, 17.18, 1.5, 71.5, 16.79, 0.0963, 1000.0, 1000))
('car_racing', (5, 324, 29.7, 1.25, 118.0, 28.6, 0.09553, 1000.0, 1000))
('car_racing', (6, 388, 46.91, 5.75, 136.75, 35.33, 0.09481, 1000.0, 1000))
('car_racing', (7, 453, 65.85, 5.5, 152.0, 40.97, 0.09412, 1000.0, 1000))
('car_racing', (8, 518, 105.47, 15.75, 203.5, 41.4, 0.09348, 1000.0, 1000))
('car_racing', (9, 584, 123.28, 40.75, 206.5, 36.28, 0.09287, 1000.0, 1000))
('car_racing', (10, 650, 139.37, 70.5, 218.5, 38.2, 0.09228, 1000.0, 1000))
[349.541 489.1644 644.3677 708.0521 495.2128 647.6338 530.0317 515.0535
661.6697 527.1665 720.0582 665.7434 646.2141 637.3213 572.7351 629.6684
676.5256 529.0831 626.0878 575.0685 626.9549 664.0143 632.5556 599.2186
326.8668 599.7362 694.7311 652.5841 603.1318 659.2366 447.6374 418.6205]
[586.61520625]
('improvement', 10, 584.37520625, 'curr', 586.61520625, 'prev', 2.24, 'best', 586.61520625)
('car_racing', (11, 975, 168.35, 99.25, 213.5, 24.7, 0.0917, 1000.0, 1000))
('car_racing', (12, 1039, 163.49, 70.25, 253.5, 38.67, 0.09116, 1000.0, 1000))
('car_racing', (13, 1104, 184.42, 80.75, 268.0, 41.62, 0.09065, 1000.0, 1000))
('car_racing', (14, 1168, 200.78, 122.75, 265.5, 29.68, 0.09016, 1000.0, 1000))
('car_racing', (15, 1232, 205.96, 80.75, 261.5, 44.21, 0.08969, 1000.0, 1000))
('car_racing', (16, 1297, 201.39, 96.0, 260.25, 44.5, 0.08923, 1000.0, 1000))
('car_racing', (17, 1360, 221.05, 101.25, 293.75, 35.19, 0.08878, 1000.0, 1000))
('car_racing', (18, 1426, 236.72, 159.25, 281.25, 28.93, 0.08835, 1000.0, 1000))
('car_racing', (19, 1492, 226.21, 75.5, 283.5, 42.45, 0.08793, 1000.0, 1000))
('car_racing', (20, 1558, 233.88, 119.75, 300.75, 42.47, 0.08752, 1000.0, 1000))
[387.4787 111.7849 417.2107 533.8831 448.2589 536.753 204.1228 188.0443
333.7994 335.6777 486.4987 306.7614 252.5269 247.3025 417.719 212.4605
511.8903 353.9119 398.2054 516.7442 374.3491 624.1784 413.6094 372.7289
262.7903 449.3083 322.7305 324.8756 301.511 299.5016 388.2149 302.5429]
[363.667975]
('improvement', 20, -222.94723125000002, 'curr', 363.667975, 'prev', 586.61520625, 'best', 586.61520625)

```

Figure 8-21. Output from in-dream training

When training in the dream environment, the scores of each generation are given in terms of the average sum of the dream rewards (i.e., 0 or 1 at each timestep). However, the evaluation performed after every 10 generations is still conducted in the real environment and is therefore scored based on the sum of rewards from the OpenAI Gym environment, so that we can compare training methods.

After just 10 generations of training in the dream environment, the agent scores an average of 586.6 in the real environment. The car is able to drive accurately around the track and can handle most corners, except those that are especially sharp.

This is an amazing achievement—remember, when the controller was evaluated after 10 generations it had *never* attempted the task of driving fast around the track in the real environment. It had only ever driven around the environment randomly (to train the VAE and MDN-RNN) and then in its own dream environment to train the controller.

As a comparison, after 10 generations the agent trained in the real environment is barely able to move off the start line. Moreover, each generation of training in the dream environment is around 3–4 times faster than training in the real environment, since z and reward prediction by the MDN-RNN is faster than z and reward calculation by the OpenAI Gym environment.

Challenges of In-Dream Training

One of the challenges of training agents entirely within the MDN-RNN dream environment is overfitting. This occurs when the agent finds a strategy that is rewarding in the dream environment, but does not generalize well to the real environment, due to the MDN-RNN not fully capturing how the true environment behaves under certain conditions.

We can see this happening in [Figure 8-21](#): after 20 generations, even though the in-dream scores continue to rise, the agent only scores 363.7 in the real environment, which is worse than its score after 10 generations.

The authors of the original “World Models” paper highlight this challenge and show how including a `temperature` parameter to control model uncertainty can help alleviate the problem. Increasing this parameter magnifies the variance when sampling z through the MDN-RNN, leading to more volatile rollouts when training in the dream environment. The controller receives higher rewards for safer strategies that encounter well-understood states and therefore tend to generalize better to the real environment. Increased `temperature`, however, needs to be balanced against not making the environment so volatile that the controller cannot learn any strategy, as there is not enough consistency in how the dream environment evolves over time.

In the original paper, the authors show this technique successfully applied to a different environment: `DoomTakeCover`, based around the computer game *Doom*. [Figure 8-22](#) shows how changing the `temperature` parameter affects both the virtual (dream) score and the actual score in the real environment.

TEMPERATURE τ	VIRTUAL SCORE	ACTUAL SCORE
0.10	2086 ± 140	193 ± 58
0.50	2060 ± 277	196 ± 50
1.00	1145 ± 690	868 ± 511
1.15	918 ± 546	1092 ± 556
1.30	732 ± 269	753 ± 139
RANDOM POLICY	N/A	210 ± 108
GYM LEADER	N/A	820 ± 58

Figure 8-22. Using temperature to control dream environment volatility⁴

⁴ Source: Ha and Schmidhuber, 2018.

Summary

In this chapter we have seen how a generative model (a VAE) can be utilized within a reinforcement learning setting to enable an agent to learn an effective strategy by testing policies within its own generated dreams, rather than within the real environment.

The VAE is trained to learn a latent representation of the environment, which is then used as input to a recurrent neural network that forecasts future trajectories within the latent space.

Amazingly, the agent can then use this generative model as a pseudoenvironment to iteratively test policies, using an evolutionary methodology, which generalize well to the real environment.

The Future of Generative Modeling

I started writing this book in May 2018, shortly after the “World Models” paper discussed in [Chapter 8](#) was published. I knew at the time that I wanted this paper to be the focus of the final core chapter of the book, as it is the first practical example of how generative models can facilitate a deeper form of learning that takes place inside the agent’s own world model of the environment. To this day, I still find this example completely astonishing. It is a glimpse into a future where agents learn not only through maximizing a single reward in an environment of our choice, but by generating their own internal representation of an environment and therefore having the capability to create their own reward functions to optimize. In this chapter, we will run with this idea and see where it takes us.

First, we must place ourselves at the very edge of the generative modeling landscape, among the most radical, innovative, and leading ideas in the field. Since the inception of this book, significant advancements in GAN and attention-based methodologies have taken us to the point where we can now generate images, text, and music that is practically indistinguishable from human-generated content. We shall start by framing these advancements alongside examples that we have already explored and walking through the most cutting-edge architectures available today.

Five Years of Progress

The history of generative modeling in its current form is short in comparison to the more widely studied discriminative modeling—the invention of the GAN in 2014 can perhaps be thought of as the spark that lit the touchpaper. [Figure 9-1](#) shows a summary of the key developments in generative modeling, many of which we have already explored together in this book.

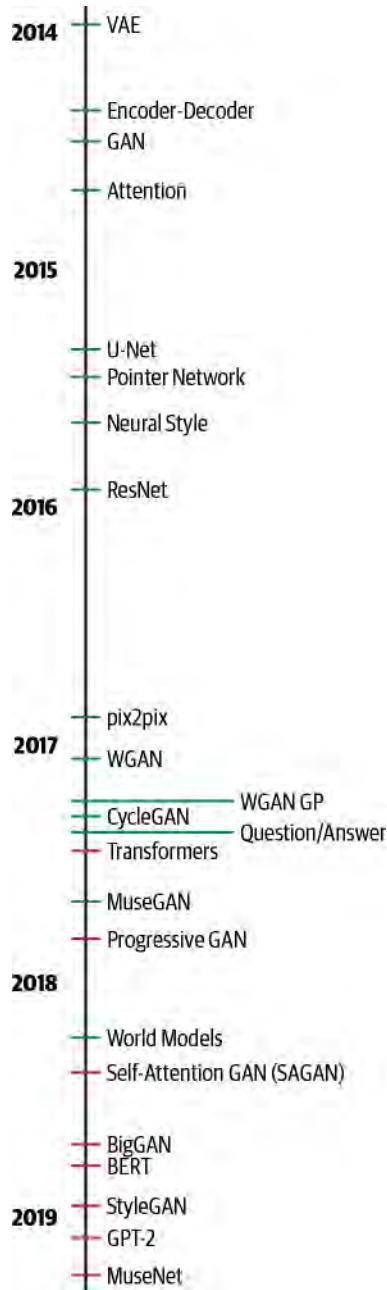


Figure 9-1. A brief history of generative modeling: green marks represent ideas that are covered in this book and red marks are ideas that we shall explore in this chapter

This is by no means an exhaustive list; there are dozens of GAN flavors that are groundbreaking in their own fields (e.g., video generation or text-to-image generation). Here, I show a selection of the most recent developments that have pushed the boundaries of generative modeling in general.

Since mid-2018 there has been a flurry of remarkable developments in both sequence and image-based generative modeling. Sequence modeling has primarily been driven by the invention of the Transformer, an attention-based module that removes the need for recurrent or convolutional neural networks entirely and now powers most state-of-the-art sequential models, such as BERT, GPT-2, and MuseNet. Image generation has reached new heights through the development of new GAN-based techniques such as ProGAN, SAGAN, BigGAN, and StyleGAN.

Explaining these developments and their repercussions in detail could easily fill another book. In this chapter, we will simply explore each in enough detail to understand the fundamental ideas behind the current state of the art in generative modeling. Armed with this knowledge, we will then hypothesize how the field will continue to develop in the near future, providing a tantalizing view of what might be possible in the years to come.

The Transformer

The Transformer was first introduced in the 2017 paper “Attention is All You Need,”¹ where the authors show how it is possible to create powerful neural networks for sequential modeling that do not require complex recurrent or convolutional architectures but instead only rely on attention mechanisms. The architecture now powers some of the most impressive practical examples of generative modeling, such as Google’s BERT and GPT-2 for language tasks and MuseNet for music generation.

The overall architecture of the Transformer is shown in [Figure 9-2](#).

¹ Ashish Vaswani et al., “Attention Is All You Need,” 12 June 2017, <https://arxiv.org/abs/1706.03762>.

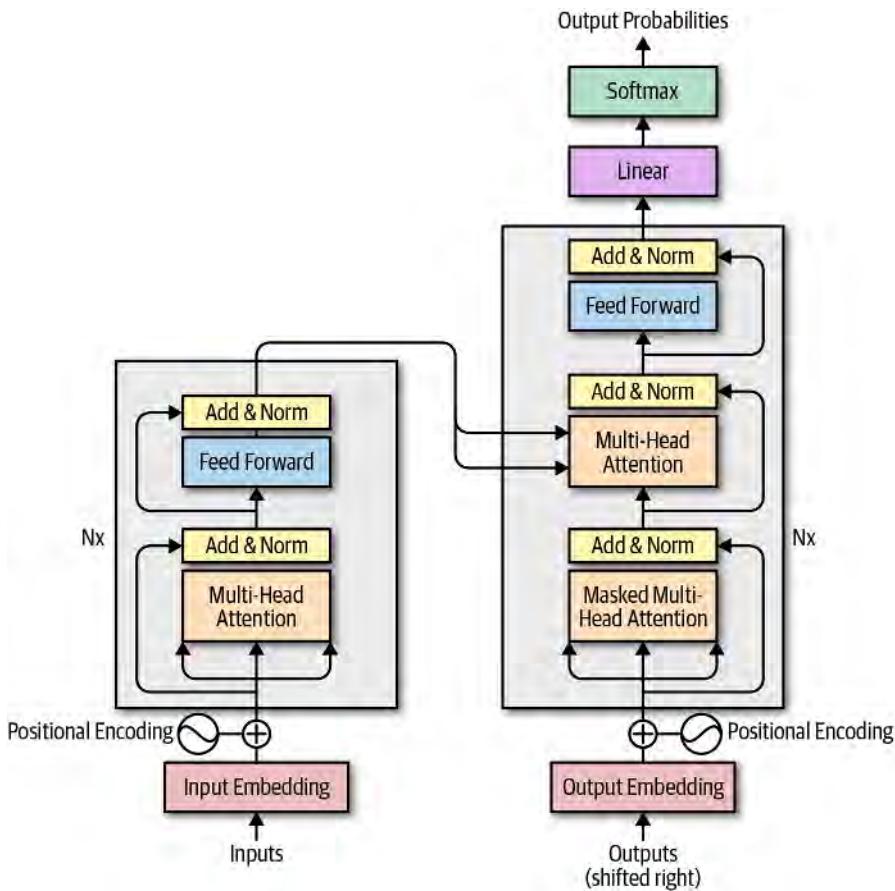


Figure 9-2. The Transformer model architecture²

The authors apply the Transformer to English–German and English–French translation datasets. As is common for translation models, the Transformer has an encoder–decoder architecture (described in [Chapter 6](#)). The difference here is that instead of using a recurrent layer such as an LSTM inside the encoder and decoder, the Transformer uses stacked attention layers.

In the lefthand half of [Figure 9-2](#), a set of $N = 6$ stacked attention layers encodes the input sentence $x = (x_1, \dots, x_n)$ to a sequence of representations. The decoder in the righthand half of the diagram then uses this encoding to generate output words one at a time, using previous words as additional input into the model.

² Source: Vaswani et al., 2017.

To understand how this works in practice, let's follow a sample input sequence through the model, step by step.

Positional Encoding

The words are first passed through an embedding layer to convert each into a vector of length $d_{model} = 512$. Now that we are not using a recurrent layer, we also need to encode the position of each word in the sentence. To achieve this, we use the following positional encoding function that converts the position pos of the word in the sentence into a vector of length d_{model} :

$$PE_{pos, 2i} = \sin \left(\frac{pos}{10000^{2i/d_{model}}} \right)$$
$$PE_{pos, 2i+1} = \cos \left(\frac{pos}{10000^{(2i+1)/d_{model}}} \right)$$

For small i , the wavelength of this function is short and therefore the function value changes rapidly along the position axis. Larger values of i create a longer wavelength, and therefore nearby words are given approximately the same value. Each position thus has its own unique encoding, and since the function can be applied to any value of pos it can be used to encode any position, no matter what the sequence length of the input is.

To construct the input into the first encoder layer, the matrix of positional encodings is added to the word embedding matrix, as shown in [Figure 9-3](#). This way, both the meaning and position for each word in the sequence are captured in a single vector, of length d_{model} .

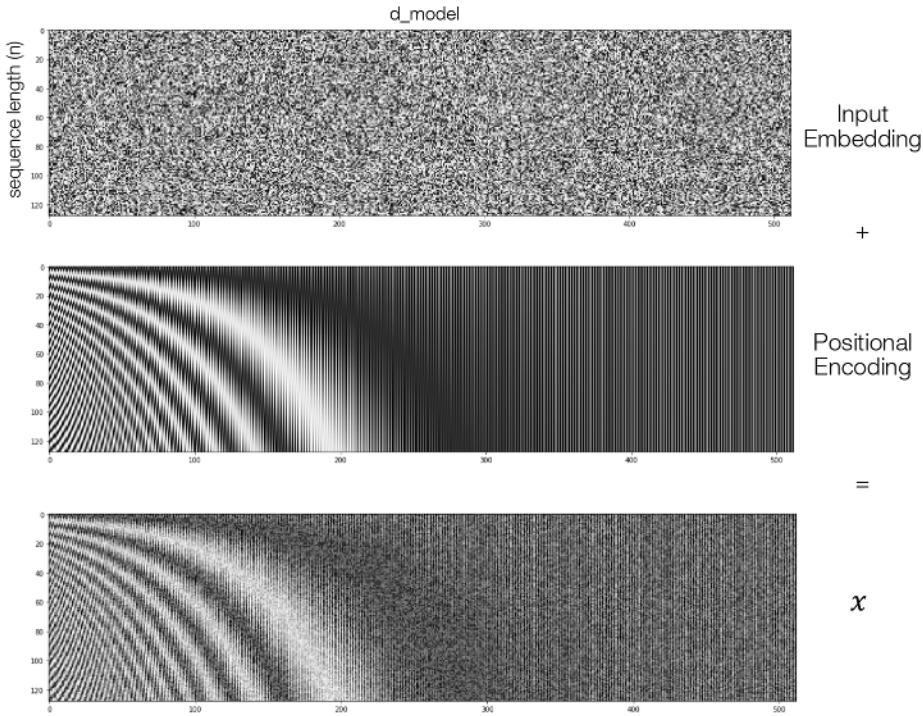


Figure 9-3. The input embedding matrix is added to the positional encoding matrix to give the input into the first encoder layer

Multihead Attention

This tensor then flows through to the first of six encoder layers. Each encoder layer consists of several sublayers, starting with the *multihead attention layer*.

The same multihead attention architecture is used in both the encoder and decoder, with a few small changes. The general architecture is shown in [Figure 9-4](#).

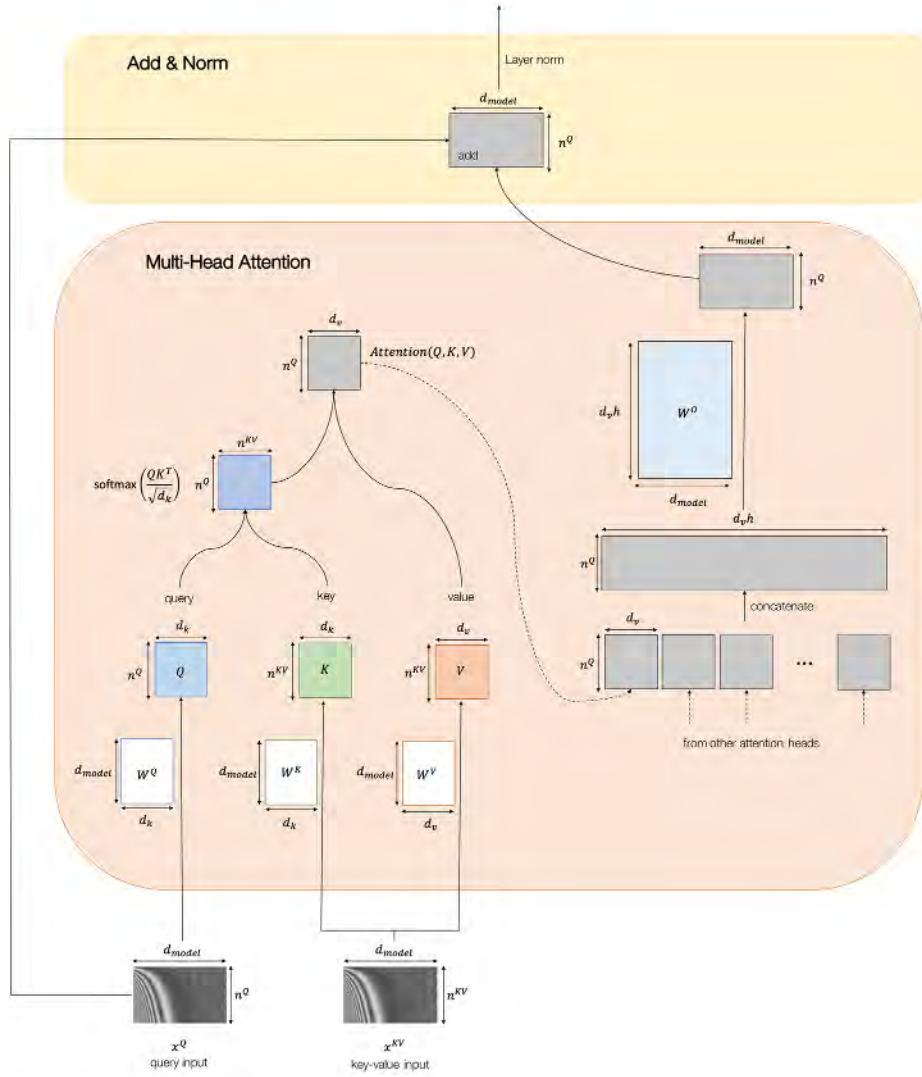


Figure 9-4. Diagram of a multihead attention module, followed by the add & norm layer

The multihead attention layer requires two inputs: the query input, x^Q , and the key-value input, x^{KV} . The job of the layer is to learn which positions in the key-value input it should attend to, for every position of the query input. None of the layer's weight matrices are dependent on the sequence length of the query input (n^Q) or the key-value input (n^{KV}), so the layer can handle sequences of arbitrary length.

The encoder uses *self-attention*—that is, the query input and key-value input are the same (the output from the previous layer in the encoder). For example, in the first

encoder layer, both inputs are the positionally encoded embedding of the input sequence. In the decoder, the query input comes from the previous layer in the decoder and the key–value input comes from the final output from the encoder.

The first step of the layer is to create three matrices, the query Q , key K , and value V , through multiplication of the input with three weight matrices, W^Q , W^K , and W^V , as follows:

$$Q = x^Q W^Q$$

$$K = x^{KV} W^K$$

$$V = x^{KV} W^V$$

Q and K are representations of the query input and key–value input, respectively. We want to measure the similarity of these representations across each position in the query input and key–value input.

We can achieve this by performing a matrix multiplication of Q with K^T and scaling by a factor $\sqrt{d_k}$. This is known as *scaled dot-product attention*. Scaling is important, to ensure that the dot product between vectors in Q and K does not grow too large.

We then apply a softmax function to ensure all rows sum to 1. This matrix is of shape $n_Q \times n_{KV}$ and is the equivalent of the attention matrix in [Figure 7-10](#).

The final step to complete the single attention head is to matrix multiply the attention matrix with the value matrix V . In other words, the head outputs a weighted sum of the value representations V for each position in the query, where the weights are determined by the attention matrix.

There's no reason to only stop at one attention head! In the paper, the authors choose eight heads that are trained in parallel, each outputting an $n_Q \times d_v$ matrix. Incorporating multiple heads allows each to learn a distinct attention and value mechanism, therefore enriching the output from the multihead attention layer.

The output matrices from the multiple heads are concatenated and passed through one final matrix multiplication with a weights matrix W^O . This is then added pointwise to the original query input through a skip connection, and layer normalization (see [Figure 5-7](#)) is applied to the result.

The final part of the encoder consists of a feed-forward (densely connected) layer applied to each position separately. The weights are shared across positions, but not between layers of the encoder–decoder. The encoder concludes with one final skip connection and normalization layer. Notice that the output from the layer is the same shape as the query input ($n_Q \times d_{model}$). This allows us to stack several encoder layers on top of each other, allowing the model to learn deeper features.

The Decoder

The decoder layers are very similar to the encoder layers, with two key differences:

1. The initial self-attention layer is masked, so that information from subsequent timesteps cannot be attended to during training. This is achieved by setting the appropriate elements of the input to the softmax to $-\infty$.
2. The output from the encoder layer is also incorporated into each layer of the decoder, after the initial self-attention mechanism. Here, the query input comes from the previous layer of the decoder and the key-value input comes from the encoder.

Each position in the output from the final decoder layer is fed through one final dense layer with a softmax activation function to give next word probabilities.

Analysis of the Transformer

The Tensorflow GitHub repository contains a [Colab notebook](#) where you can play around with a trained Transformer model and see how the attention mechanisms of the encoder and decoder impact the translation of a given sentence into German.

For example, [Figure 9-5](#) shows how two attention heads of the decoder layer are able to work together to provide the correct German translation for the word *the*, when used in the context of *the street*. In German, there are three definite articles (*der*, *die*, *das*) depending on the gender of the noun, but the Transformer knows to choose *die* because one attention head is able to attend to the word *street* (a feminine word in German), while another attends to the word to translate (*the*).

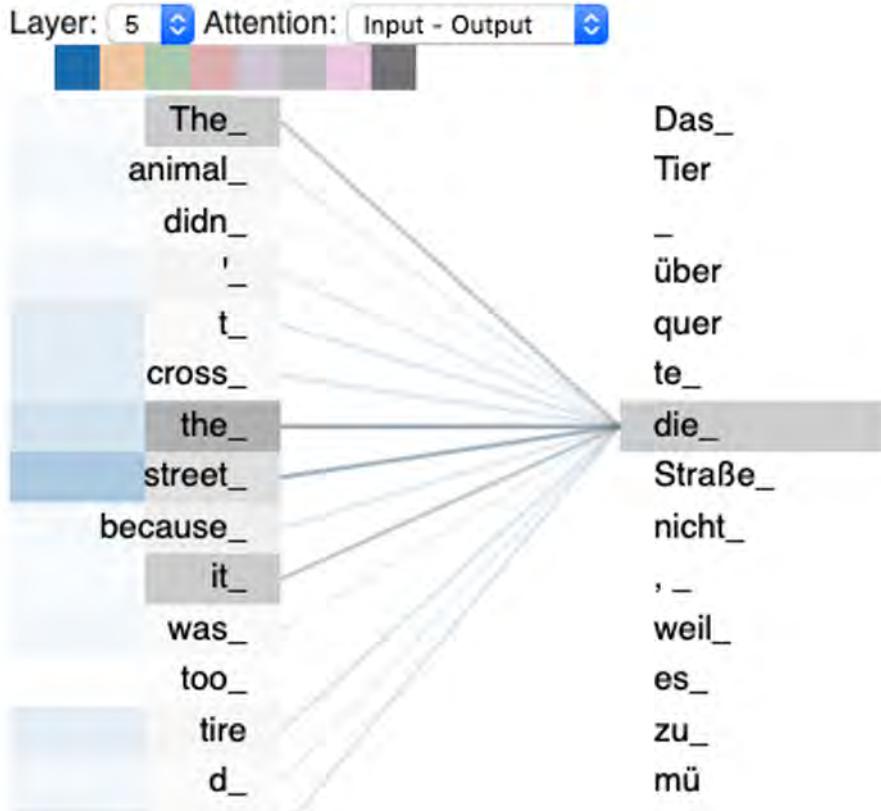


Figure 9-5. An example of how one attention head attends to the word “the” and another attends to the word “street” in order to correctly translate the word “the” to the German word “die” as the feminine definite article to “Straße”

This gives the Transformer the ability to translate extremely complex and long sentences, as it can incorporate information from several places across the input sentence and current translation to form its decision about the next word.

The Transformer architecture has inspired several subsequent models that make use of the multihead attention mechanism. We'll look at some of these briefly next.

BERT

BERT (Bidirectional Encoder Representations from Transformers)³ is a model developed by Google that predicts missing words from a sentence, given context from both before and after the missing word in all layers. It achieves this through a *masked language model*: during training, 15% of words are randomly masked out and the model must try to re-create the original sentence, given the masked input. Crucially, 10% of the tokens marked for masking are actually swapped with another word, rather than the <MASK> token, so not only must the model learn how to replace the <MASK> tokens with actual words, but also it should be looking out for words in the input sentence that do not seem to fit, as they could be words that have been switched.

The word representations learned by BERT are superior to counterparts such as GloVe because they change depending on the context of the word. For example, the word *water* can be used either as a verb (*I need to water the plant*) or as a noun (*The ocean is full of water*). GloVe vectors allocate exactly the same representation to the word *water* regardless of the context, whereas BERT incorporates surrounding information to create a bespoke representation for the word in context.

BERT can be built upon by appending output layers that are specific to a given downstream task. For example, classification tasks such as sentiment analysis can be constructed by adding a classification layer on top of the Transformer output, and question answering tasks can be tackled by marking the answer in the input sequence using a pointer network as the output layer to BERT. By starting from a pretrained BERT model and fine-tuning the appended output layers, it is therefore possible to quickly train extremely sophisticated language models for a variety of modeling tasks.

GPT-2

GPT-2 is a model developed by OpenAI that is trained to predict the next word in a passage of text. Whereas BERT was Google's response to OpenAI's earlier GPT model, GPT-2 is a direct response to BERT. The key difference between the models is that while BERT is bidirectional, GPT-2 is unidirectional. This means that GPT-2 does not use information from subsequent words to form representations of the current word and therefore is set up for sentence generation tasks, such as the Aesop's Fables task that we explored in [Chapter 6](#). An example of the output from GPT-2, given a system prompt sentence, is shown in [Figure 9-6](#).

³ Jacob Devlin et al., "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding," 11 October 2018, <https://arxiv.org/abs/1810.04805v1>.

SYSTEM PROMPT (HUMAN-WRITTEN)	<i>A train carriage containing controlled nuclear materials was stolen in Cincinnati today. Its whereabouts are unknown.</i>
MODEL COMPLETION (MACHINE-WRITTEN, FIRST TRY)	<p>The incident occurred on the downtown train line, which runs from Covington and Ashland stations.</p> <p>In an email to Ohio news outlets, the U.S. Department of Energy said it is working with the Federal Railroad Administration to find the thief.</p> <p>"The theft of this nuclear material will have significant negative consequences on public and environmental health, our workforce and the economy of our nation," said Tom Hicks, the U.S. Energy Secretary, in a statement. "Our top priority is to secure the theft and ensure it doesn't happen again."</p> <p>The stolen material was taken from the University of Cincinnati's Research Triangle Park nuclear research site, according to a news release from Department officials.</p> <p>The Nuclear Regulatory Commission did not immediately release any information.</p> <p>According to the release, the U.S. Department of Energy's Office of Nuclear Material Safety and Security is leading that team's investigation.</p> <p>"The safety of people, the environment and the nation's nuclear stockpile is our highest priority," Hicks said. "We will get to the bottom of this and make no excuses."</p>

Figure 9-6. An example of how GPT-2 can extend a given system prompt⁴

If you are slightly terrified at how realistic this appears, you are not alone. Due to concerns about how this model could be abused by malicious parties, for example to generate fake news, forged essays, fake accounts on social media, or impersonations of people online, OpenAI has decided to not release the dataset, code, or GPT-2 model weights. Instead, **only small (117M parameter) and medium (345M parameter) versions of GPT-2** have been released officially.⁵

MuseNet

MuseNet is a model also released by OpenAI that applies the Transformer architecture to music generation. Like GPT-2, it is unidirectional, trained to predict the next note given a sequence of previous notes.

In music generation tasks, the length of the sequence N grows large as the music progresses, and this means that the $N \times N$ attention matrix for each head becomes expen-

⁴ Source: "Better Language Models and Their Implications", 2019, <https://openai.com/blog/better-language-models>.

⁵ As of May 2019, a 1.5B parameter version of GPT-2 has also been released to trusted partners who are researching the potential impact of such sophisticated models and how to counteract misuse of GPT-2.

sive to store and compute. We cannot just clip the input sequence, as we would like the model to construct the piece around a long-term structure and recapitulate motifs and phrases from several minutes ago, as would be the case with a human composer.

To tackle this problem, MuseNet utilizes a form of Transformer known as a *Sparse Transformer*. Each output position in the attention matrix only computes weights for a subset of input positions, thereby reducing the computational complexity and memory required to train the model. MuseNet can therefore operate with full attention over 4,096 tokens and can learn long-term structure and melodic structure across a range of styles. (See, for example, OpenAI’s Chopin and Mozart recordings on SoundCloud.)

Advances in Image Generation

In recent years, image-based generative modeling has been revolutionized by several significant advancements in the architecture and training of GAN-based models, in the same way that the Transformer has been pivotal to the progression of sequential generative modeling. In this section we will introduce four such developments—ProGAN, SAGAN, BigGAN, and StyleGAN.

ProGAN

ProGAN is a new technique developed by NVIDIA Labs to improve both the speed and stability of GAN training.⁶ Instead of immediately training a GAN on full-resolution images, the paper suggests first training the generator and discriminator on low-resolution images of, say, 4×4 pixels and then incrementally adding layers throughout the training process to increase the resolution. This process is shown in Figure 9-7.

⁶ Tero Karras et al., “Progressive Growing of GANs for Improved Quality, Stability, and Variation,” 27 October 2017, <https://arxiv.org/abs/1710.10196>

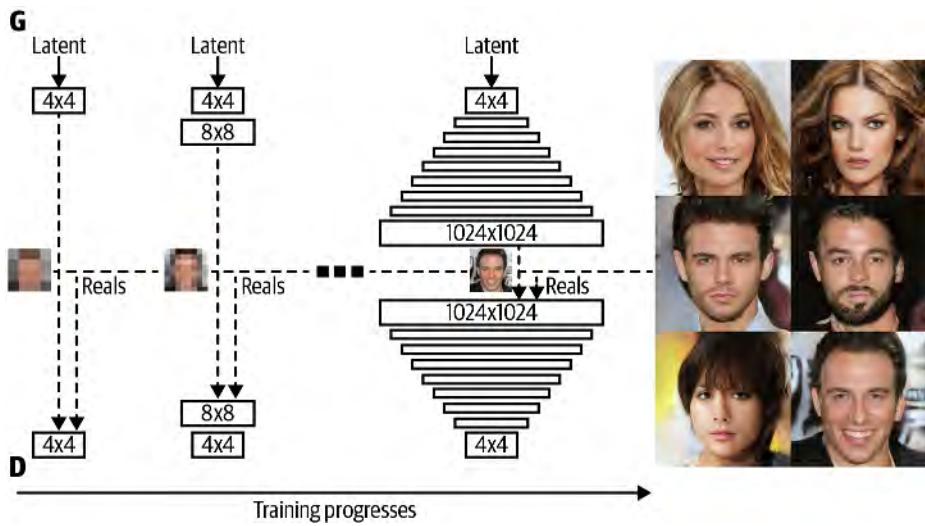


Figure 9-7. The Progressive GAN training mechanism, and some example generated faces⁷

The earlier layers are not frozen as training progresses, but remain fully trainable. The new training mechanism was also applied to images from the LSUN dataset with excellent results, as shown in [Figure 9-8](#).

⁷ Source: Karras et al., 2017.

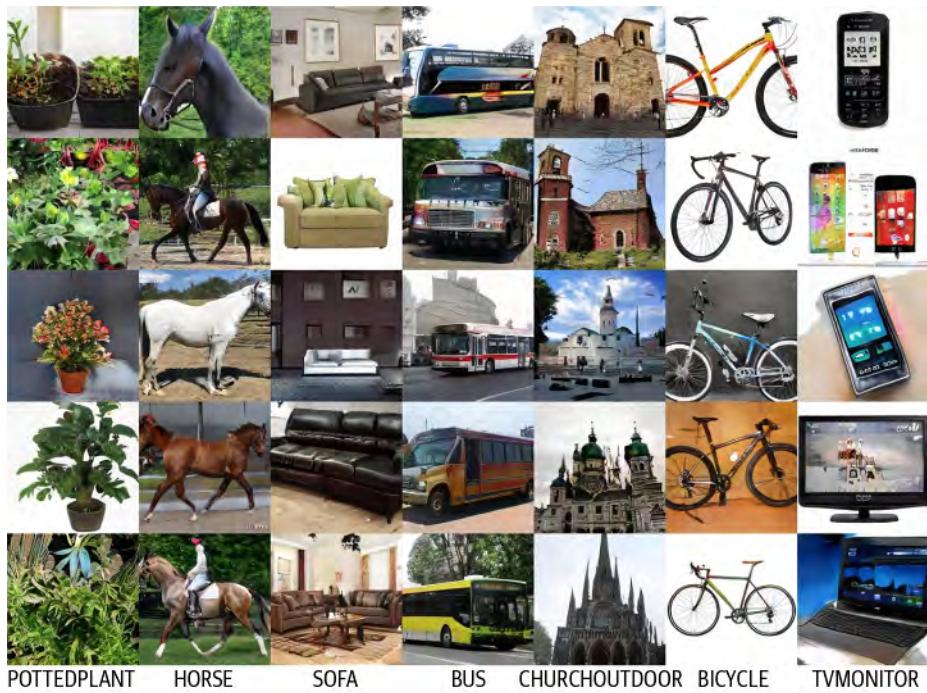


Figure 9-8. Generated examples from a GAN trained progressively on the LSUN dataset at 256×256 resolution⁸

Self-Attention GAN (SAGAN)

The Self-Attention GAN (SAGAN)⁹ is a key development for GANs as it shows how the attention mechanism that powers sequential models such as the Transformer can also be incorporated into GAN-based models for image generation. [Figure 9-9](#) shows the self-attention mechanism from the paper. Note the similarity with the Transformer attention head architecture in [Figure 9-2](#).

⁸ Source: Karras et al., 2017.

⁹ Han Zhang et al., “Self-Attention Generative Adversarial Networks,” 21 May 2018, <https://arxiv.org/abs/1805.08318>.

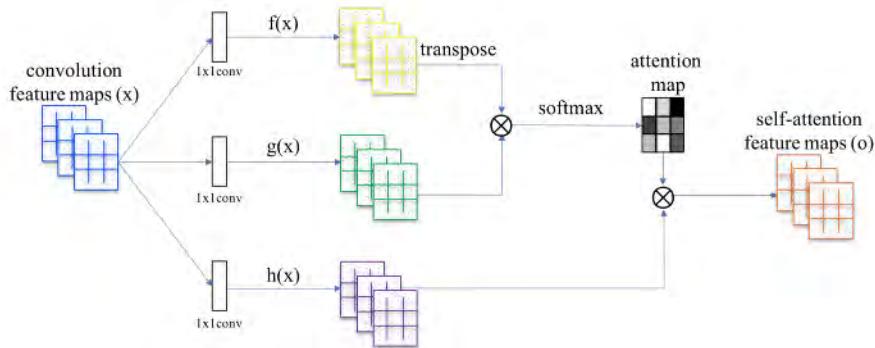


Figure 9-9. The self-attention mechanism within the SAGAN model¹⁰

The problem with GAN-based models that do not incorporate attention is that convolutional feature maps are only able to process information locally. Connecting pixel information from one side of an image to the other requires multiple convolutional layers that reduce the spatial dimension of the image, while increasing the number of channels. Precise positional information is reduced throughout this process in favor of capturing higher-level features, making it computationally inefficient for the model to learn long-range dependencies between distantly connected pixels. SAGAN solves this problem by incorporating the attention mechanism that we explored earlier in this chapter into the GAN. The effect of this inclusion is shown in Figure 9-10.



Figure 9-10. A SAGAN-generated image of a bird (leftmost cell) and the attention maps of the final attention-based generator layer for the pixels covered by the three colored dots (rightmost cells)¹¹

The red dot is a pixel that is part of the bird's body, and so attention naturally falls on surrounding body cells. The green dot is part of the background, and here the atten-

¹⁰ Source: Zhang et al., 2018.

¹¹ Source: Zhang et al., 2018.

tion actually falls on the other side of the bird’s head, on other background pixels. The blue dot is part of the bird’s long tail and so attention falls on other tail pixels, some of which are distant from the blue dot. It would be difficult to maintain this long-range dependency for pixels without attention, especially for long, thin structures in the image (such as the tail in this case).

BigGAN

BigGAN,¹² developed at DeepMind, extends the ideas from the SAGAN paper with extraordinary results. [Figure 9-11](#) shows some of the images generated by BigGAN, trained on the ImageNet dataset.



Figure 9-11. Examples of images generated by BigGAN, trained on the ImageNet dataset at 128×128 resolution¹³

BigGAN is currently the state-of-the-art model for image generation on the ImageNet dataset. As well as some incremental changes to the base SAGAN model, there are also several innovations outlined in the paper that take the model to the next level of sophistication.

One such innovation is the so-called *truncation trick*. This is where the latent distribution used for sampling is different from the $z \sim N(0, 1)$ distribution used during training. Specifically, the distribution used during sampling is a *truncated normal distribution* (resampling z that have magnitude greater than a certain threshold). The smaller the truncation threshold, the greater the believability of generated samples, at the expense of reduced variability. This concept is shown in [Figure 9-12](#).

¹² Andrew Brock, Jeff Donahue, and Karen Simonyan, “Large Scale GAN Training for High Fidelity Natural Image Synthesis,” 28 September 2018, <https://arxiv.org/abs/1809.11096>.

¹³ Source: Brock et al., 2018.



Figure 9-12. The truncation trick: from left to right, the threshold is set to 2, 1, 0.5, and 0.04^{14}

Also, as the name suggests, BigGAN is an improvement over SAGAN in part simply by being *bigger*. BigGAN uses a batch size of 2,048—8 times larger than the batch size of 256 used in SAGAN—and a channel size that is increased by 50% in each layer. However, BigGAN additionally shows that SAGAN can be improved structurally by the inclusion of a shared embedding, by orthogonal regularization, and by incorporating the latent vector z into each layer of the generator, rather than just the initial layer.

For a full description of the innovations introduced by BigGAN, I recommend reading the original paper and [accompanying presentation material](#).

StyleGAN

One of the most recent additions to the GAN literature is StyleGAN,¹⁵ from NVIDIA Labs. This builds upon two techniques that we have already explored in this book, ProGAN and neural style transfer (from [Chapter 5](#)). Often when training GANs it is difficult to separate out vectors in the latent space corresponding to high-level attributes—they are frequently *entangled*, meaning that adjusting an image in the latent space to give a face more freckles, for example, might also inadvertently change the background color. While ProGAN generates fantastically realistic images, it is no exception to this general rule. We would ideally like to have full control of the style of the image, and this requires a disentangled separation of high-level features in the latent space.

The overall architecture of the StyleGAN generator is shown in [Figure 9-13](#).

¹⁴ Source: Brock, Donahue, and Simonyan, 2018.

¹⁵ Tero Karras, Samuli Laine, and Timo Aila, “A Style-Based Generator Architecture for Generative Adversarial Networks,” 12 December 2018, <https://arxiv.org/abs/1812.04948>.

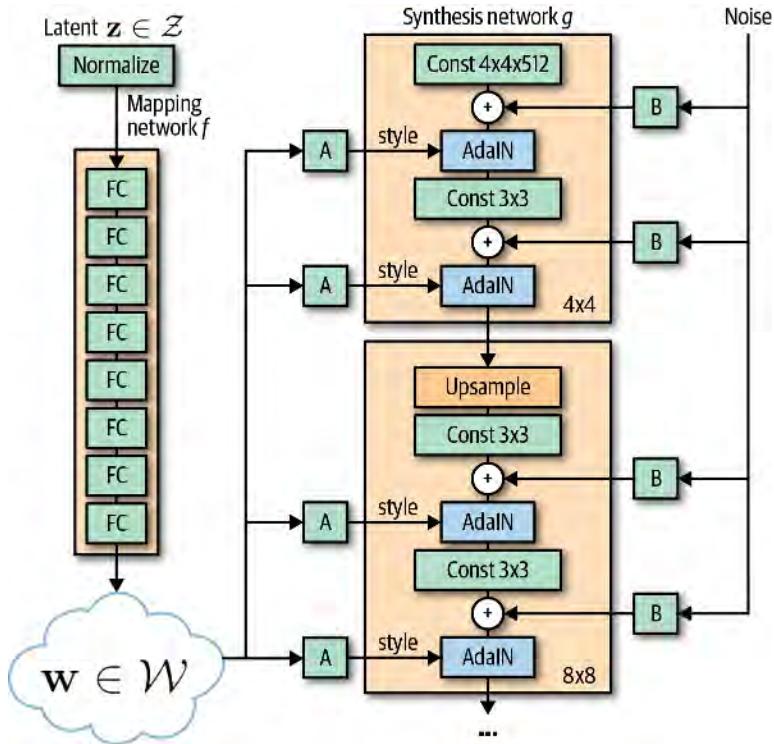


Figure 9-13. The StyleGAN generator architecture¹⁶

StyleGAN solves the entanglement problem by borrowing ideas from the style transfer literature. In particular, StyleGAN utilizes a method called *adaptive instance normalization*.¹⁷ This is a type of neural network layer that adjusts the mean and variance of each feature map \mathbf{x}_i output from a given layer in the synthesis network with a reference style bias $\mathbf{y}_{b,i}$ and scale $\mathbf{y}_{s,i}$, respectively. The equation for adaptive instance normalization is as follows:

$$\text{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i}$$

¹⁶ Source: Karras, Laine, and Aila, 2018.

¹⁷ Xun Huang and Serge Belongie, “Arbitrary Style Transfer in Real-Time with Adaptive Instance Normalization,” 20 March 2017, <https://arxiv.org/abs/1703.06868>.

The style parameters are calculated by first passing a latent vector z through a mapping network f to produce an intermediate vector w . This is then transformed through a densely connected layer (A) to generate the $y_{b,i}$ and $y_{s,i}$ vectors, both of length n (the number of channels output from the convolutional layer in the synthesis network). The point of doing this is to separate out the process of choosing a style for the image (the mapping network) from the generation of an image with a given style (the synthesis network). The adaptive instance normalization layers ensure that the style vectors that are injected into each layer only affect features at that layer, by preventing any style information from leaking through between layers. The authors show that this results in the latent vectors w being significantly more disentangled than the original z vectors.

Since the synthesis network is based on the ProGAN architecture, the style vectors at earlier layers in the synthesis network (when the resolution of the image is lowest— 4×4 , 8×8) will affect coarser features than those later in the network (64×64 to $1,024 \times 1,024$ resolution). This means that not only do we have complete control over the generated image through the latent vector w , but we can also switch the w vector at different points in the synthesis network to change the style at a variety of levels of detail.

Figure 9-14 shows this in action. Here, two images, source A and source B, are generated from two different w vectors. To generate a merged image, the source A w vector is passed through the synthesis network but, at some point, switched for the source B w vector. If this switch happens early on (4×4 or 8×8 resolution), coarse styles such as pose, face shape, and glasses from source B are carried across onto source A. However, if the switch happens later, only fine-grained detail is carried across from source B, such as colors and microstructure of the face, while the coarse features from source A are preserved.

Finally, the StyleGAN architecture adds noise after each convolution to account for stochastic details such as the placement of individual hairs, or the background behind the face. Again, the depth at which the noise is injected affects the coarseness of the impact on the image.

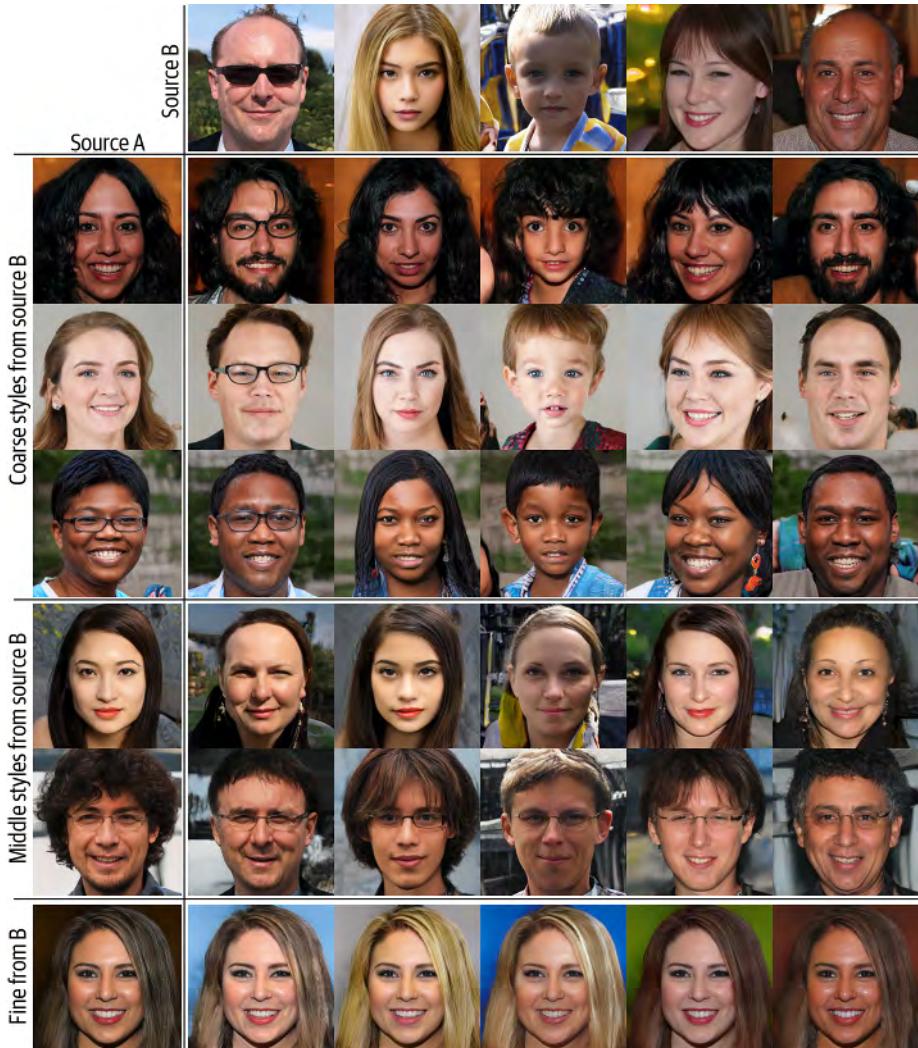


Figure 9-14. Merging styles between two generated images at different levels of detail¹⁸

18 Source: Karras, Laine, and Aila, 2018.

Applications of Generative Modeling

As is clear from the preceding examples, generative modeling has come a long way in the last five years. The field has developed to the point where it is not unreasonable to suggest that the next generation will be just as comfortable marveling at computer-generated art as human art, reading computer-generated novels, and listening to computer-generated music in their favorite style. This movement has already started to gather momentum, particularly among artists and musicians.

AI Art

I recently attended a meetup in London entitled “Form, Figures and BigGAN,” featuring presentations by artist Scott Eaton and BigGAN creator Andrew Brock, organized by AI art curator Luba Elliott. One facet of Scott’s work centers around using pix2pix models to create art of the human form. The model is trained on color photographs of dancers and matching grayscale edge-highlighted images. He is able to create new forms by producing a line drawing (i.e., in the edge-highlighted space) and allowing the model to convert back to the color photograph domain. These line drawings need not be realistic human forms; the model will find a way to make the drawing look as human as possible, as this is what it has been trained to do. Two examples of his work are shown in [Figure 9-15](#). You can find more about his artistic process on [YouTube](#).



Figure 9-15. Two examples of Scott Eaton’s work, generated through a pix2pix model trained on photographs of dancers¹⁹

AI Music

As well as producing aesthetically interesting and evocative images, generative modeling has practical application in the field of music generation, especially for video games and films. We have already seen how MuseNet is able to generate endless amounts of music in a given style and therefore could be adapted to provide the background mood music for a film or video game.

In fact, on April 25, 2019, OpenAI live-streamed an experimental concert in which MuseNet generated music across a range of styles that no human had ever heard before. Could it be that before long, we will be able to tune into a radio station that plays music in our favorite style nonstop, so that we never hear the same thing twice? Perhaps we could have the option of saving passages that we particularly like to listen to again, or exploring new music generated on the fly by the model. We are not yet at the stage where text and music can be convincingly combined to produce pop songs with long-term structure, but given the impressive progress in both text and music generation in recent years it surely won’t be long before this is a reality.

¹⁹ Source: Scott Eaton, 2018, <http://www.scott-eaton.com>.

CHAPTER 10

Conclusion

In this book, we have taken a journey through the last half-decade of generative modeling research, starting out with the basic ideas behind variational autoencoders, GANs, and recurrent neural networks and building upon these foundations to understand how state-of-the-art models such as the Transformer, advanced GAN architectures, and world models are now pushing the boundaries of what generative models are capable of achieving, across a variety of tasks.

I believe that in the future, generative modeling may be the key to a deeper form of artificial intelligence that transcends any one particular task and instead allows machines to organically formulate their own rewards, strategies, and ultimately awareness within their environment.

As babies, we are constantly exploring our surroundings, building up a mental model of possible futures with no apparent aim other than to develop a deeper understanding of the world. There are no labels on the data that we receive—a seemingly random stream of light and sound waves that bombard our senses from the moment we are born. Even when our mother or father points to an apple and says *apple*, there is no reason for our young brains to associate the two and learn that the way in which light entered our eye at that particular moment is in any way related to the way the sound waves entered our ear. There is no training set of sounds and images, no training set of smells and tastes, and no training set of actions and rewards. Just an endless stream of extremely noisy data.

And yet here you are now, reading this sentence, perhaps enjoying the taste of a cup of coffee in a noisy cafe. You pay no attention to the background noise as you concentrate on converting the absence of light on a tiny proportion of your retina into a sequence of abstract concepts that convey almost no meaning individually but, when combined, trigger a wave of parallel representations in your mind’s eye—images,

emotions, ideas, beliefs, and potential actions all flood your consciousness, awaiting your recognition.

The same noisy stream of data that was essentially meaningless to your infant brain is not so noisy any more. Everything makes sense to you. You see structure everywhere. You are never surprised by the physics of everyday life. The world is the way that it is, because your brain decided it should be that way.

In this sense, your brain is an extremely sophisticated generative model, equipped with the ability to attend to particular parts of the input data, form representations of concepts within a latent space of neural pathways, and process sequential data over time. But what exactly is it generating?

At this point, I must switch into pure speculation mode as we are close to the edge of what we currently understand about the human brain (and certainly at the very edge of what *I* understand about the human brain). However, we can conduct a thought experiment to understand the links between generative modeling and the brain.

Suppose that the brain is a near-perfect generative model of the input stream of data that it is subjected to. In other words, it can generate the likely sequence of input data that would follow from receiving the cue of an egg-shaped region of light falling through the visual field to the sound of a *splat* as the egg-shaped region stops moving abruptly. It does this by creating representations of the key aspects of the visual and auditory fields and modeling how these latent representations will evolve over time. There is one fallacy in this view, however: the brain is not a passive observer of events. It's attached to a neck and a set of legs that can put its core input sensors in any myriad of positions relative to the source of the input data. The generated sequence of possible futures is not only dependent on its understanding of the physics of the environment, but also on its understanding of *itself* and how it acts.

This is the core idea that I believe will propel generative modeling into the spotlight in the next decade, as one of the keys to unlocking artificial general intelligence. Imagine if we could build a generative model that doesn't model possible futures of the environment given an action, as per the world models example, but instead includes its own action-generating process as part of the environment to be modeled.

If actions are random to begin with, why would the model learn anything except to predict random actions from the body in which it resides? The answer is simple: because nonrandom actions make the stream of environmental data easier to generate. If the sole goal of a brain is to minimize the amount of *surprise* between the actual input stream of data and the model of the future input stream, then the brain must find a way to make its actions create the future that it expects.

This may seem backward—wouldn't it make more sense for the brain to act according to some policy that tries to maximize a reward? The problem with this is that nature does not provide us with *rewards*; it just provides data. The only true reward is

staying alive, and this can hardly be used to explain every action of an intelligent being. Instead, if we flip this on its head and require that the action is part of the environment to be generated and that the sole goal of intelligence is to generate actions and futures that match the reality of the input data, then perhaps we avoid the need for any external reward function from the environment. However, whether this setup would generate actions that could be classed as intelligent remains to be seen.

As I stated, this is purely a speculative view, but it is fun to speculate, so I will continue doing so. I encourage you to do the same and to continue learning more about generative models from all the great material that is available online and in other books. Thank you for taking the time to read to the end of this book—I hope you have enjoyed reading it as much as I have enjoyed generating it. <END>

Index

Symbols

1-Lipschitz continuous function, 117

A

activation functions, 38

Adam optimizer, 42

AI art, 296

(see also style transfer)

AI music, 297

(see also music generation)

Anaconda, 27

artifacts, 105

artificial neural networks (ANNs), 33

arXiv, xi

attention mechanisms

analysis of, 213-217

building in Keras, 208-212

in encoder-decoder networks, 217-221

examples in language translation, 206

generating polyphonic music, 221

multihead attention module, 280

autoencoders

analysis of, 72-75

building, 66

decoder architecture, 68-71

encoder architecture, 66

joining encoder to decoder, 71

parts of, 64

process used by, 65

representation vectors in, 65

uses for, 65

B

backpropagation, 34

batch normalization, 51-53, 55, 125

BERT (Bidirectional Encoder Representations from Transformers), 285

BigGAN, 291

binary cross-entropy loss, 42, 71, 107

C

categorical cross-entropy loss, 42

CelebFaces Attributes (CelebA) dataset, 86

character tokens, 169

CIFAR-10 dataset, 35, 120

CMA-ES (covariance matrix adaptation evolution strategy), 261-268

CNTK, 34

code examples, obtaining and using, x, xii, 27

comments and questions, xiii

composition (see music generation)

concatenate layer, 140

content loss, 154-156

convolutional layers in neural networks, 46-51, 59

convolutional transpose layers, 68-69

covariate shift, 52

CycleGAN (cycle-consistent adversarial network)

analysis of, 147

benefits of, 135

compiling, 144-146

CycleGAN versus pix2pix, 135

discriminators, 142

generators (ResNet), 150-151

generators (U-Net), 139-142

Keras-GAN code repository, 137

Monet-style transfer example, 149-153

overview of, 137
published paper on, 135
training, 146
training data, 137

D

DCGAN (deep convolutional generative adversarial network), 101
deep learning
 deep neural networks, 33, 59
 defined, 31
 Keras and TensorFlow for, 34
 model creation, 35-46
 model improvement, 46-58
 premise behind, 154
 structured versus unstructured data, 31
dense layers, 33
density function, 11
discriminative modeling, 2
dropout layers, 54

E

encoder-decoder models, 187-190, 217-221
environment setup, 27-29
evolutionary strategies, 262
exploding gradient problem, 51

F

facial image generation
 dataset used, 86
 encoder and decoder architecture, 88
 generating new faces, 92
 latent space arithmetic, 93
 morphing between faces, 94
 progress in, 5
 VAE analysis, 91
 VAE training, 87
features, 2
fit_generator method, 88
Functional API (Keras), 37-41

G

gated recurrent units (GRUs), 168, 185
generative adversarial networks (GANs)
 challenges of, 112-115
 defining, 100-106
 discriminators, 101, 142
 "ganimal" example, 97-99

published paper on, 97
theory underlying, 99
training, 107-112
Wasserstein GAN, 115-121
WGAN-GP, 121-127
generative deep learning
 additional resources, xi
 advances in generative modeling, 5-7, 277-297
 challenges of, 22-27
 future of, 299-301
 Generative modeling framework, 7-10
 history of generative modeling, 275-277
 introduction to, 1-5
 learning objectives and approach, x
 learning prerequisites, x
 probabilistic generative models, 10-21

generators

 attention-based, 290
 bar generator, 229
 in GANs, 103-106
 MuseGAN generator, 226
 question-answer generators, 190-200
 ResNet generators, 150-151
 StyleGAN generator, 292
 U-Net generators, 139
 GloVe ("Global Vectors"), 195, 285
 Goodfellow, Ian, 97
 Google Colaboratory, xi
 GPT-2 language model, 5, 285
gradient descent, 153
Gram matrices, 158

H

Ha, David, 237, 243, 265
Hello Wrold! example, 13-21
hidden layers, 34
hidden state, 174, 176
Hinton, Geoffrey, 4, 54
Hochreiter, Sepp, 167
Hou, Xianxu, 91
Hull, Jonathan, 118
hyperparameters, 114

I

identity, 145
image generation (see also facial image generation; neural style transfer technique)
 BigGAN, 291

CIFAR-10 dataset for, 35
generative modeling process, 1
generative versus discriminative modeling,
 2
key breakthrough in, 4
ProGAN, 287
progress in facial image generation, 5
representation learning for, 23
rise of generative modeling for, 5
Self-Attention GAN (SAGAN), 289
StyleGAN, 292
ImageNet dataset, 154
ImageNet Large Scale Visual Recognition Chal-
lenge (ILSVRC), 4
in-dream training, 268-272
inference, 196
instance normalization layers, 140

K

Keras
 attention mechanisms in, 208-212
 autoencoder creation in, 67
 backends for, 34
 benefits of, 34
 content loss calculation in, 155
 Conv2DTranspose layer, 70, 105
 custom loss function creation, 259
 CycleGAN creation and training, 137-149
 data loading, 35
 decoder creation in, 70
 documentation, 43
 fit_generator method, 88
 GAN discriminator creation in, 102
 importing, 28
 inference model in, 197
 LSTM in, 168
 model building, 37-41
 model compilation, 41
 model evaluation, 44
 model improvement, 46-58
 model training, 43
 MuseGAN generator in, 230
 PatchGAN discriminators in, 143
 residual blocks in, 150
 U-Net generators in, 141
 VAE creation in, 81
Keras layers
 Activation, 56
 Batch Normalization, 51

Bidirectional, 187
Concatenate, 140
Conv2D, 47
Conv2DTranspose, 70
Conv3D, 232
Dense, 38
Dropout, 54
Embedding, 172
Flatten, 38
GRU, 168
Input, 38
Lambda, 82
LeakyReLU, 56
LSTM, 174
Reshape, 211
Upsampling2D layer, 104
Kingma, Diederik, 61
Kullback–Leibler (KL) divergence, 84

L

L-BFGS-B algorithm, 161
labels, 3
language translation, 188, 206
layers, 33
LeakyReLU, 38
likelihood, 12
Lipschitz constraint, 117, 121
loss functions, 41
LSTM (long short-term memory) networks
 with attention mechanism, 206
dataset used, 168
embedding layer, 172
generating datasets, 171
generating new text, 179-182
history of, 167
LSTM architecture, 172
LSTM cell, 176-178
LSTM layer, 174-176
published paper on, 167
tokenizing the text, 168-170

M

machine learning
 advances in, 4
 major branches of, 238
machine painting (see style transfer)
machine writing (see text data generation)
Machine-Learning-as-a-Service (MLaaS), 5
Maluuba NewsQA dataset, 191

masked language model, 285
maximum likelihood estimation, 13
MDN (mixture density network), 243, 255-261
mean squared error loss, 42
MIDI files, 202
mode collapse, 113
models
 CycleGAN, 135-152
 deep neural networks, 35-58
 encoder-decoder models, 187-190
 generative adversarial networks (GANs), 97-115
 generative modeling, 1-10
 generative versus discriminative modeling, 2
 improving models, 46-58
 LSTM (long short-term memory) networks, 168-178
 neural style transfer, 153-162
 parametric modeling, 11
 probabilistic generative models, 10-21
 probabilistic versus deterministic, 2
 question-answer generators, 190-200
 RNNs (recurrent neural networks), 205-221
 variational autoencoders (VAEs), 61-96
 Wasserstein GAN, 115-121
 WGAN-GP, 121-127
 World Model architecture, 241-244
Monet-to-photo dataset, 149
multihead attention module, 280
multilayer RNNs, 183
MuseGAN, 223-235
 analysis of, 233
 creation, 223-231
MuseNet, 286, 297
MuseScore, 202
music generation
 challenges of, 201
 data extraction, 204
 dataset used, 202
 generating polyphonic music, 221
 importing MIDI files, 202
 MuseGAN analysis, 233
 MuseGAN creation, 223-231
 MuseGAN critic, 232
 MuseGAN example, 221
 music versus text generation, 201
 musical notation, 204
 prerequisites to, 202

RNN (recurrent neural network) for, 205-221
music21 library, 202

N

Naive Bayes parametric model, 17-20
neural style transfer technique
 analysis of, 161
 content loss, 154-156
 definition of, 153
 premise of, 153
 running, 160
 style loss, 156-159
 total variance loss, 160
nontrainable parameters, 53
normal distribution, 79

O

observations, 1
OpenAI Gym, 239
optimizers, 41
oscillating loss, 112
overfitting, 54

P

padding, 48
painting (see style transfer)
Papers with Code, xi
parameters, trainable and nontrainable, 53
parametric modeling, 11
PatchGAN discriminators, 143
pix2pix, 135
positional encoding, 279
probabilistic generative models
 challenges of, 22
 Hello Wrod! example, 13, 20
 model construction, 14-17
 Naive Bayes parametric model, 17-20
 probabilistic theory behind, 10-13
probability density function, 11, 79
ProGAN, 287
Project Gutenberg, 168
Python, 27

Q

qgen-workshop TensorFlow codebase, 190
question-answer generators
 dataset used, 191

encoder-decoder models, 188
inference, 196
model architecture, 192-196
model parts, 190
model results, 198-200
questions and comments, xiii

R

reconstruction loss, 84
regularization techniques, 54
reinforcement learning (RL)
defined, 238
key terminology, 238
OpenAI Gym toolkit for, 239
process of, 239
ReLU (rectified linear unit), 38
representation learning, 23-27
representation vectors, 65
residual networks (ResNets), 150-151
RMSProp optimizer, 43
RNNs (recurrent neural networks)
bidirectional cells, 187
gated recurrent units (GRUs), 185
history of, 167, 167
LSTM (long short-term memory) networks,
167-182
MDN-RNN World Model architecture, 243
for music generation, 205-221
stacked recurrent networks, 183, 206
root mean squared error (RMSE), 71

S

sample space, 11
scaled dot-product attention, 282
Schmidhuber, Jürgen, 167, 237
self-attention, 281
Self-Attention GAN (SAGAN), 289
sequence modeling, 277
Sequential models (Keras), 37-41
sigmoid activation, 39
skip connections, 139
softmax activation, 39
stacked recurrent networks, 183
standard deviation, 79
standard normal curves, 79
stemming, 169
stochastic (random) elements, 2
strides parameter (Keras), 48
structured data, 31

style loss, 156-159
style transfer
aim of, 131
apples and oranges example, 132-134
CycleGAN analysis, 147
CycleGAN creation and training, 137-147
CycleGAN introduction, 135
CycleGAN Monet example, 149-152
neural style transfer technique, 153-162
uses for, 131
StyleGAN, 5, 292
supervised learning, 3

T

TensorFlow, 34
text data generation
encoder-decoder models, 187-190
LSTM (long short-term memory) networks,
167-182
question-answer generators, 190-200
RNN (recurrent neural network) exten-
sions, 183-187
short story generation example, 166
text versus image data, 165
text versus music generation, 201
text summarization, 188
Theano, 34
tokenization, 168-170
total variance loss, 160
trainable parameters, 53
training data, 1
training process, 34
Transformer
analysis of, 283
BERT model, 285
decoder layers, 283
GPT-2 language model, 285
history of, 206
models architecture, 277
multihed attention layer, 280
MuseNet model, 286
positional encoding function, 279
published paper on, 277
truncation trick, 291

U

U-Net, 139-142
uninformative loss, 114
units, 33, 174

unstructured data, 31
upsampling, 104

V

validity, 145
vanishing gradient problem, 151, 167
variance, 79
variational autoencoders (VAEs)
 autoencoder analysis, 72-75
 autoencoder example, 66-72
 autoencoder parts, 64
 decoders, 68-71
 encoders, 66-68
 facial image generation using, 86-95
 generative art example, 61-64, 75-77
 published paper on, 61, 91
 VAE analysis, 85
 VAE build in Keras, 81
 VAE diagram, 82
 VAE loss function, 84
 VAE parts, 78-85
 World Model architecture, 242
 World Model training, 248-255
 VGG19 network, 154
 virtual environments, 27

analysis of, 120
benefits of, 115
Lipschitz constraint, 117
training, 119
Wasserstein loss, 115-117
weight clipping, 118
Wasserstein GAN–Gradient Penalty (WGAN-GP)
 analysis of, 125
 converting WGAN to WGAN-GP, 121
 gradient penalty loss, 121-125
 weight clipping, 118, 121
weights, 33
Welling, Max, 61
World Models
 collecting random rollout data, 245
 collecting RNN training data, 255
 model setup, 244
 published paper on, 237
 training in-dream, 268-272
 training overview, 245
 training the controller, 261-268
 training the MDN-RNN, 257-261
 training the VAE, 248-255
 World Model architecture, 241-244
World Models paper, 237, 243, 275
Wrodll!, 13

W

Wasserstein GANs (WGANS)

About the Author

David Foster is the cofounder of Applied Data Science, a data science consultancy delivering bespoke solutions for clients. He holds an MA in mathematics from Trinity College, Cambridge, UK, and an MSc in operational research from the University of Warwick.

David has won several international machine learning competitions, including the InnoCente Predicting Product Purchase challenge, and was awarded first prize for a visualization that enables a pharmaceutical company in the US to optimize site selection for clinical trials.

He is an active participant in the online data science community and has authored several successful blog posts on deep reinforcement learning including “[How To Build Your Own AlphaZero AI Using Python and Keras](#)”.

Colophon

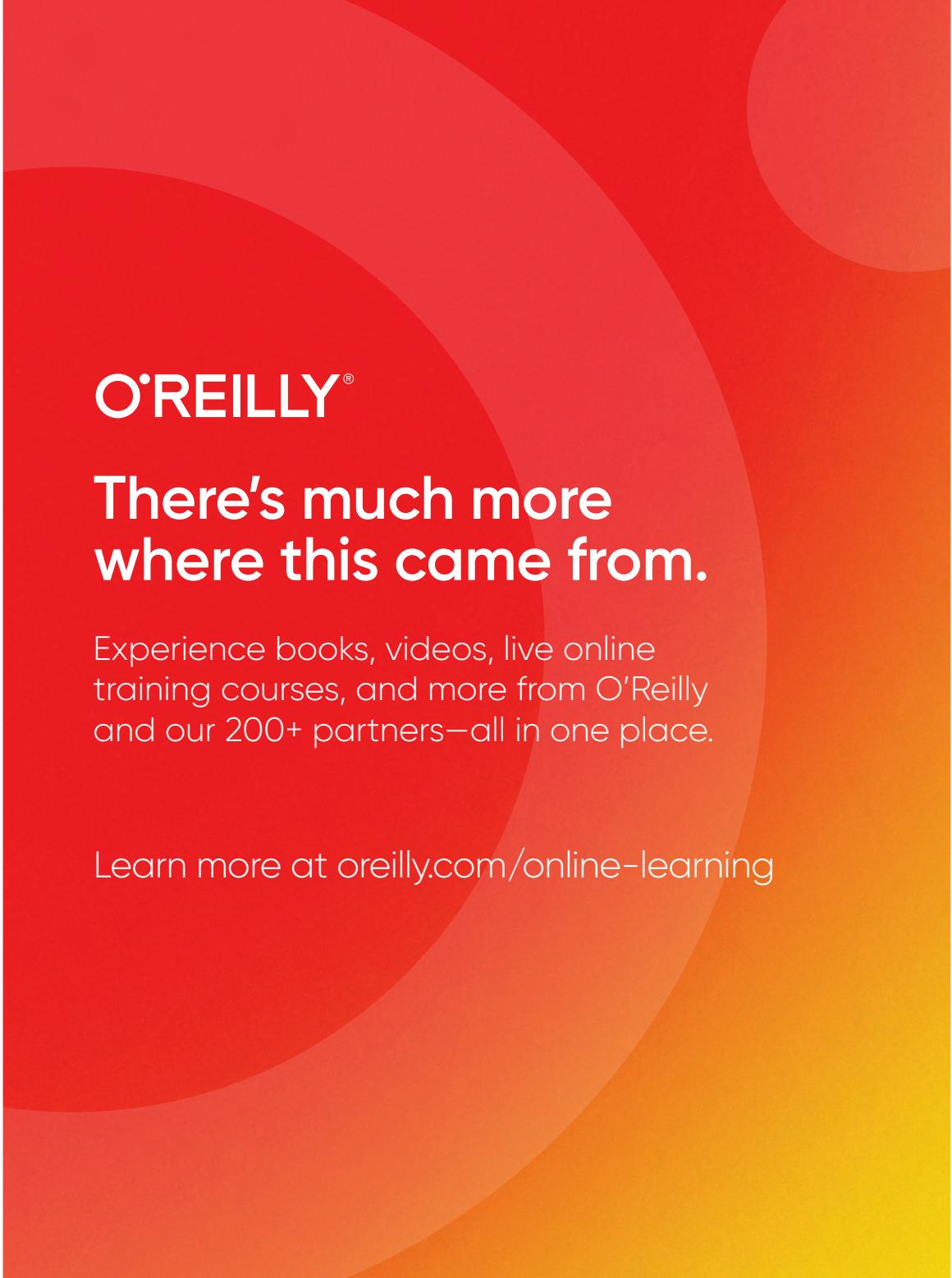
The animal on the cover of *Generative Deep Learning* is a painted parakeet (*Pyrrhura picta*). The *Pyrrhura* genus falls under the family *Psittacidae*, one of three families of parrots. Within its subfamily *Arinae* are several macaw and parakeet species of the Western Hemisphere. The painted parakeet inhabits the coastal forest and mountains of northeastern South America.

Bright green feathers cover most of a painted parakeet, but they are blue above the beak, brown in the face, and reddish in the breast and tail. Most strikingly, the feathers on the painted parakeet’s neck look like scales; the brown center is outlined in off-white. This combination of colors camouflages the birds in the rainforest.

Painted parakeets tend to feed in the forest canopy, where their green plumage masks them best. They forage in flocks of 5 to 12 birds for a wide variety of fruits, seeds, and flowers. Occasionally, when feeding below the canopy, painted parakeets will eat algae from forest pools. They grow to about 9 inches in length and live for 13 to 15 years. A clutch of painted parakeet chicks is usually around five eggs, which are less than one inch wide at hatching.

Many of the animals on O’Reilly’s covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Shaw’s Zoology*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.



O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning