

TensorFlow Machine Learning

Cookbook

Second Edition

Over 60 recipes to build intelligent machine learning systems
with the power of Python



Packt

www.packt.com

By Nick McClure

TensorFlow Machine Learning Cookbook
Second Edition

Over 60 recipes to build intelligent machine learning systems with the power of Python

Nick McClure

Packt

BIRMINGHAM - MUMBAI

TensorFlow Machine Learning Cookbook Second Edition

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Veena Pagare

Acquisition Editor: Divya Poojari

Content Development Editor: Rhea Henriques

Technical Editor: Kushal Shingote

Copy Editor: Safis Editing

Project Coordinator: Manthan Patel

Proofreader: Safis Editing

Indexer: Mariammal Chettiar

Graphics: Jisha Chirayil

Production Coordinator: Aparna Bhagat

First published: February 2017

Second edition: August 2018

Production reference: 1300818

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78913-168-0

www.packtpub.com

*This book is dedicated to my wife and partner; without your support,
none of this would be possible.*

*Special thanks goes out to the TensorFlow team at Google. Their great product and skill speaks volumes,
and is accompanied by great documentation, tutorials, and examples.*



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Nick McClure is currently a senior data scientist at PayScale, Inc. in Seattle, WA. Prior to this, he has worked at Zillow Group and Caesar's Entertainment Corporation. He got his degrees in Applied Mathematics from The University of Montana and the College of Saint Benedict and Saint John's University.

He has a passion for learning and advocating for analytics, machine learning, and artificial intelligence. Nick occasionally puts his thoughts and musings on his blog, fromdata.org, or through his Twitter account, @nfmcclure.

I am grateful to my parents, who have always encouraged me to pursue knowledge. I also want to thank my friends and family, especially my wife, who have endured my long monologues about the subjects in this book and always have been encouraging and have listened to me. Writing this book was made easier by the amazing efforts of the open source community and the great documentation of many projects out there related to TensorFlow.

About the reviewer

Sujit Pal is a technology research Director at Elsevier Labs, an advanced technology group within the Reed-Elsevier Group of companies. His areas of interests include semantic search, natural language processing, machine learning and deep learning. At Elsevier, he has worked on several initiatives involving search quality measurement and improvement, image classification, duplicate detection, annotation and ontology development for medical and scientific corpora. He has co-authored a book on deep learning with Antonio Gulli and writes about technology on his blog Salmon Run. He has also worked as a technical reviewer on the book *Reinforcement Learning in Python*.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Title Page
Copyright and Credits
TensorFlow Machine Learning Cookbook Second Edition
Dedication
Packt Upsell
Why subscribe?
PacktPub.com
Contributors
About the author
About the reviewer
Packt is searching for authors like you
Preface
Who this book is for
What this book covers
To get the most out of this book
Download the example code files
Conventions used
Sections
Getting ready
How to do it...
How it works...
There's more...
See also
Get in touch
Reviews
1. Getting Started with TensorFlow
Introduction
How TensorFlow works
Getting ready
How to do it...
How it works...
See also
Declaring variables and tensors
Getting ready
How to do it...
How it works...
There's more...
Using placeholders and variables

Getting ready
How to do it...
How it works...
There's more...
Working with matrices
Getting ready
How to do it...
How it works...
Declaring operations
Getting ready
How to do it...
How it works...
There's more...
Implementing activation functions
Getting ready
How to do it...
How it works...
There's more...
Working with data sources
Getting ready
How to do it...
How it works...
See also
** Additional resources**
Getting ready
How to do it...

2. The TensorFlow Way

Introduction
Operations in a computational graph
Getting ready
How to do it...
How it works...
Layering nested operations
Getting ready
How to do it...
How it works...
There's more...
Working with multiple layers
Getting ready
How to do it...
How it works...
Implementing loss functions

Getting ready
How to do it...
How it works...
There's more...
Implementing backpropagation
Getting ready
How to do it...
How it works...
There's more...
See also
Working with batch and stochastic training
Getting ready
How to do it...
How it works...
There's more...
Combining everything together
Getting ready
How to do it...
How it works...
There's more...
See also
Evaluating models
Getting ready
How to do it...
How it works...

3. [Linear Regression](#)

Introduction
Using the matrix inverse method
Getting ready
How to do it...
How it works...
Implementing a decomposition method
Getting ready
How to do it...
How it works...
Learning the TensorFlow way of linear regression
Getting ready
How to do it...
How it works...
Understanding loss functions in linear regression
Getting ready
How to do it...

How it works...

There's more...

Implementing deming regression

- Getting ready
- How to do it...
- How it works...

Implementing lasso and ridge regression

- Getting ready
- How to do it...
- How it works...

There's more...

Implementing elastic net regression

- Getting ready
- How to do it...
- How it works...

Implementing logistic regression

- Getting ready
- How to do it...
- How it works...

4. Support Vector Machines

Introduction

Working with a linear SVM

- Getting ready
- How to do it...
- How it works...

Reduction to linear regression

- Getting ready
- How to do it...
- How it works...

Working with kernels in TensorFlow

- Getting ready
- How to do it...
- How it works...

There's more...

Implementing a non-linear SVM

- Getting ready
- How to do it...
- How it works...

Implementing a multi-class SVM

- Getting ready
- How to do it...
- How it works...

5. Nearest-Neighbor Methods

Introduction

Working with nearest-neighbors

Getting ready

How to do it...

How it works...

There's more...

Working with text based distances

Getting ready

How to do it...

How it works...

There's more...

Computing with mixed distance functions

Getting ready

How to do it...

How it works...

There's more...

Using an address matching example

Getting ready

How to do it...

How it works...

Using nearest-neighbors for image recognition

Getting ready

How to do it...

How it works...

There's more...

6. Neural Networks

Introduction

Implementing operational gates

Getting ready

How to do it...

How it works...

Working with gates and activation functions

Getting ready

How to do it...

How it works...

There's more...

Implementing a one-layer neural network

Getting ready

How to do it...

How it works...

There's more...

Implementing different layers
 Getting ready
 How to do it...
 How it works...
Using a multilayer neural network
 Getting ready
 How to do it...
 How it works...
Improving the predictions of linear models
 Getting ready
 How to do it
 How it works...
Learning to play Tic Tac Toe
 Getting ready
 How to do it...
 How it works...

7. Natural Language Processing

Introduction
Working with bag-of-words embeddings
 Getting ready
 How to do it...
 How it works...
 There's more...
Implementing TF-IDF
 Getting ready
 How to do it...
 How it works...
 There's more...
Working with Skip-Gram embeddings
 Getting ready
 How to do it...
 How it works...
 There's more...
Working with CBOW embeddings
 Getting ready
 How to do it...
 How it works...
 There's more...
Making predictions with word2vec
 Getting ready
 How to do it...
 How it works...

There's more...
Using doc2vec for sentiment analysis

Getting ready
How to do it...
How it works...

8. Convolutional Neural Networks

Introduction
Implementing a simple CNN

Getting ready
How to do it...
How it works...
There's more...

See also

Implementing an advanced CNN

Getting ready
How to do it...
How it works...

See also

Retraining existing CNN models

Getting ready
How to do it...
How it works...
See also

Applying stylenet and the neural-style project

Getting ready
How to do it...
How it works...
See also

Implementing DeepDream

Getting ready
How to do it...
There's more...
See also

9. Recurrent Neural Networks

Introduction
Implementing RNN for spam prediction

Getting ready
How to do it...
How it works...
There's more...

Implementing an LSTM model

Getting ready

- How to do it...
- How it works...
- There's more...

Stacking multiple LSTM layers

- Getting ready
- How to do it...
- How it works...

Creating sequence-to-sequence models

- Getting ready
- How to do it...
- How it works...
- There's more...

Training a Siamese similarity measure

- Getting ready
- How to do it...
- There's more...

10. Taking TensorFlow to Production

- Introduction
- Implementing unit tests
 - Getting ready
 - How it works...
- Using multiple executors
 - Getting ready
 - How to do it...
 - How it works...
 - There's more...
- Parallelizing TensorFlow
 - Getting ready
 - How to do it...
 - How it works...
- Taking TensorFlow to production
 - Getting ready
 - How to do it...
 - How it works...
- An example of productionalizing TensorFlow
 - Getting ready
 - How to do it...
 - How it works...
- Using TensorFlow Serving
 - Getting ready
 - How to do it...
 - How it works...

There's more...

11. More with TensorFlow

Introduction

Visualizing graphs in TensorBoard

Getting ready

How to do it...

There's more...

Working with a genetic algorithm

Getting ready

How to do it...

How it works...

There's more...

Clustering using k-means

Getting ready

How to do it...

There's more...

Solving a system of ordinary differential equations

Getting ready

How to do it...

How it works...

See also

Using a random forest

Getting ready

How to do it...

How it works...

See also

Using TensorFlow with Keras

Getting ready

How to do it...

How it works...

See also

Other Books You May Enjoy

Leave a review - let other readers know what you think

Preface

TensorFlow was open source in November 2015 by Google, and since then it has become the most starred machine learning repository on GitHub. TensorFlow's popularity is due to its approach to creating computational graphs, automatic differentiation, and customizability. Because of these features, TensorFlow is a very powerful and adaptable tool that can be used to solve many different machine learning algorithms.

This book addresses many machine learning algorithms, applies them to real situations and data, and shows how to interpret the results.

Who this book is for

This book is for the hobbyists, programmers, or enthusiasts who want both an introduction to TensorFlow and curated recipe examples for major machine learning algorithms. This book relies on basic knowledge of mathematics and Python programming. The major goal of this book is to introduce TensorFlow and provide well-documented examples of various machine learning algorithms. It is not within the scope of this book to delve into the specifics of mathematics, machine learning, or even programming. The best description of the book's target area is to give a gentle introduction to all three. Because of this, a reader with expertise in one area may find some parts of the book too slow and some parts too fast. Users with an extensive machine learning background may find the TensorFlow code enlightening, and users with an extensive Python programming background may find the explanations helpful. For the curious reader that would like more information on specific areas, there will be a *There's more...* section at the end of most chapters, which provides references and resources on the subject.

What this book covers

[Chapter 1](#), *Getting Started with TensorFlow*, covers the main objects and concepts in TensorFlow. We introduce tensors, variables, and placeholders. We also show how to work with matrices and various mathematical operations in TensorFlow. At the end of the chapter, we show how to access the data sources used in the rest of the book.

[Chapter 2](#), *The TensorFlow Way*, establishes how to connect all the algorithm components from [Chapter 1](#), *Getting Started with TensorFlow*, into a computational graph in multiple ways to create a simple classifier. Along the way, we cover computational graphs, loss functions, back propagation, and training with data.

[Chapter 3](#), *Linear Regression*, focuses on using TensorFlow for exploring various linear regression techniques, such as deming, lasso and ridge, elastic net, and logistic regression. We show how to implement each in a TensorFlow computational graph.

[Chapter 4](#), *Support Vector Machines*, introduces support vector machines (SVMs) and shows how to use TensorFlow to implement linear SVMs, non-linear SVMs, and multi-class SVMs.

[Chapter 5](#), *Nearest-Neighbor Methods*, shows how to implement nearest neighbor techniques using numerical metrics, textual metrics, and scaled distance functions. We use nearest neighbor techniques to perform record matching of addresses and to classify hand-written digits from the MNIST database.

[Chapter 6](#), *Neural Networks*, covers how to implement neural networks in TensorFlow, starting with the operational gates and activation function concepts. We then show a shallow neural network and how to build up various different types of layers. We end the chapter by teaching a TensorFlow neural network to play tic tac toe.

[Chapter 7](#), *Natural Language Processing*, illustrates various text processing techniques with TensorFlow. We show how to implement the bag-of-words

technique and TF-IDF (text frequency - inverse document frequency) for text. We then introduce text representations (CBOW, continuous bag-of-words, and skip-gram) and use these techniques for Word2Vec and Doc2Vec to make real-world predictions such as predicting whether a text message is spam.

[Chapter 8](#), *Convolutional Neural Networks*, expands our knowledge of neural networks by illustrating how to use images with convolutional layers (and other image layers and functions). We show how to build a shortened CNN for MNIST digit recognition and extend it to color images in the CIFAR-10 task. We also illustrate how to extend prior trained image recognition models for custom tasks. We end the chapter by explaining and demonstrating the stylenet/neural style and deep-dream algorithms in TensorFlow.

[Chapter 9](#), *Recurrent Neural Networks*, explains how to implement recurrent neural networks in TensorFlow. We show how to do text-spam prediction, and expand the RNN model to perform text generation based on the works of Shakespeare. We also train a sequence-to-sequence model for German-English translation. We finish the chapter by showing the usage of Siamese RNNs for record matching on addresses.

[Chapter 10](#), *Taking TensorFlow to Production*, gives tips and examples on moving TensorFlow to a production environment and how to take advantage of multiple processing devices (for example, GPUs) and setting up TensorFlow distributed on multiple machines. We end the chapter by showing an example of setting up an RNN model on TensorFlow serving an API.

[Chapter 11](#), *More with TensorFlow*, demonstrates the versatility of TensorFlow by illustrating how to use the k-means and genetic algorithms, and how to solve a system of ordinary differential equations (ODEs). We also show the various uses of TensorBoard, and how to view computational graph metrics and charts.

To get the most out of this book

The mathematical concepts in this book should be accessible to anyone with basic knowledge of matrices and statistics. The programming knowledge required for this book is an intermediate level of Python programming. This book is biased toward the use of functions over classes, though not always. The recipes in this book use TensorFlow, which is available at <https://www.tensorflow.org/>, and are based on Python 3, available at <https://www.python.org/downloads/>. Most of the recipes will initially require the use of an internet connection to download the necessary data. The reader should be aware that as TensorFlow progresses and is developed by the open source community, the code may become obsolete or may not even work in some cases. Updated code and examples can be found on the author's GitHub site at https://github.com/nfmccclure/tensorflow_cookbook, or alternatively, on the Packt code repository at <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.



If you run into any programming or mathematical issues in the book, feel free to raise an issue on the preceding GitHub site. The GitHub site may even already contain a fix for the problem.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, and user input. Here is an example: "Here, we define our loss function, `our_loss_fun()`, which will return the loss we need."

A block of code is set as follows:

```
| embedding_matrix = tf.Variable(tf.random_uniform([n, m], -1.0, 1.0))
| embedding_output = tf.nn.embedding_lookup(embedding_matrix, x_data_placeholder)
```

Any command-line input or output is written as follows:

```
| $ mkdir css
| $ cd css
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select System info from the Administration panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we, have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

If you are interested in contributing to the source code: If you wish to make code contributions or fixes, please clone the GitHub repository: https://github.com/nfmcclure/tensorflow_cookbook and make your changes locally, then push the changes with a pull request to the master branch. All contributions are welcome, this includes adding comments, bug fixes, adding documentation, or even adding completely new recipes.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you.

For more information about Packt, please visit packtpub.com.

Getting Started with TensorFlow

In this chapter, we will cover some basic recipes in order to understand how TensorFlow works and how to access data for this book and additional resources.

By the end of this chapter, you should have knowledge of the following:

- How TensorFlow works
- Declaring variables and tensors
- Using placeholders and variables
- Working with matrices
- Declaring operations
- Implementing activation functions
- Working with data sources
- Additional resources

Introduction

Google's TensorFlow engine has a unique way of solving problems. This unique way allows us to solve machine learning problems very efficiently. Machine learning is used in almost all areas of life and work, but some of the more famous areas are computer vision, speech recognition, language translations, healthcare, and many more. We will cover the basic steps to understand how TensorFlow operates and eventually build up to production code techniques later in this book. These fundamentals are important to understanding recipes for the rest of this book.

How TensorFlow works

At first, computation in TensorFlow may seem needlessly complicated. But there is a reason for it: because of how TensorFlow treats computation, developing more complicated algorithms is relatively easy. This recipe will guide us through the pseudocode of a TensorFlow algorithm.

Getting ready

Currently, TensorFlow is supported on Linux, macOS, and Windows. The code for this book has been created and run on a Linux system, but should run on any other system as well. The code for the book is available on GitHub at https://github.com/nfmccclure/tensorflow_cookbook as well as the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

Throughout this book, we will only concern ourselves with the Python library wrapper of TensorFlow, although most of the original core code for TensorFlow is written in C++. This book will use Python 3.6+ (<https://www.python.org>) and TensorFlow 1.10.0+ (<https://www.tensorflow.org>). While TensorFlow can run on the CPU, most algorithms run faster if processed on the GPU, and it is supported on graphics cards with Nvidia Compute Capability v4.0+ (v5.1 recommended).

Popular GPUs for TensorFlow are Nvidia Tesla architectures and Pascal architectures with at least 4 GB of video RAM. To run on a GPU, you will also need to download and install the Nvidia CUDA toolkit as well as version 5.x+ (<https://developer.nvidia.com/cuda-downloads>).

Some of the recipes in this chapter will rely on a current installation of the SciPy, NumPy, and scikit-learn Python packages. These accompanying packages are also included in the Anaconda package (<https://www.continuum.io/downloads>).

How to do it...

Here, we will introduce the general flow of TensorFlow algorithms. Most recipes will follow this outline:

1. **Import or generate dataset:** All of our machine learning algorithms will depend on datasets. In this book, we will either generate data or use an outside source of datasets. Sometimes, it is better to rely on generated data because we just want to know the expected outcome. Most of the time, we will access public datasets for the given recipe. The details on accessing these datasets are in additional resources, section 8 of this chapter.
2. **Transform and normalize data:** Normally, input datasets do not come into the picture. TensorFlow expects that we need to transform TensorFlow so that they get the accepted shape. The data is usually not in the correct dimension or type that our algorithms expect. We will have to transform our data before we can use it. Most algorithms also expect normalized data and we will look at this here as well. TensorFlow has built-in functions that can normalize data for you, as follows:

```
| import tensorflow as tf  
| data = tf.nn.batch_norm_with_global_normalization(...)
```

3. **Partition the dataset into train, test, and validation sets:** We generally want to test our algorithms on different sets that we have trained on. Also, many algorithms require hyperparameter tuning, so we set aside a validation set for determining the best set of hyperparameters.
4. **Set algorithm parameters (hyperparameters):** Our algorithms usually have a set of parameters that we hold constant throughout the procedure. For example, this can be the number of iterations, the learning rate, or other fixed parameters of our choosing. It is considered good practice to initialize these together so that the reader or user can easily find them, as follows:

```
| learning_rate = 0.01  
| batch_size = 100  
| iterations = 1000
```

5. **Initialize variables and placeholders:** TensorFlow depends on knowing what it can and cannot modify. TensorFlow will modify/adjust the variables (model weights/biases) during optimization to minimize a loss function. To

accomplish this, we feed in data through placeholders. We need to initialize both variables and placeholders with size and type so that TensorFlow knows what to expect. TensorFlow also needs to know the type of data to expect. For most of this book, we will use `float32`. TensorFlow also provides `float64` and `float16`. Note that more bytes used for precision results in slower algorithms, but less used results in less precision. See the following code:

```
| a_var = tf.constant(42)
| x_input = tf.placeholder(tf.float32, [None, input_size])
| y_input = tf.placeholder(tf.float32, [None, num_classes])
```

6. **Define the model structure:** After we have the data, and have initialized our variables and placeholders, we have to define the model. This is done by building a computational graph. We will talk more in depth about computational graphs in the *Operations in a Computational Graph* TensorFlow recipe in [chapter 2](#), *The TensorFlow Way*. The model for this example will be a linear model ($y = mx + b$):

```
| y_pred = tf.add(tf.mul(x_input, weight_matrix), b_matrix)
```

7. **Declare the loss functions:** After defining the model, we must be able to evaluate the output. This is where we declare the loss function. The loss function is very important as it tells us how far off our predictions are from the actual values. The different types of loss functions are explored in greater detail in the *Implementing Back Propagation* recipe in [chapter 2](#), *The TensorFlow Way*. Here, we implement the mean squared error for n-points, that is, $\text{loss} = (1/n) \sum (y_{\text{actual}} - y_{\text{predicted}})^2$:

```
| loss = tf.reduce_mean(tf.square(y_actual - y_pred))
```

8. **Initialize and train the model:** Now that we have everything in place, we need to create an instance of our graph, feed in the data through the placeholders, and let TensorFlow change the variables to better predict our training data. Here is one way to initialize the computational graph:

```
| with tf.Session(graph=graph) as session:
| ...
|     session.run(...)
| ...
| Note that we can also initiate our graph with:
| session = tf.Session(graph=graph)
| session.run(...)
```

9. **Evaluate the model:** Once we have built and trained the model, we should evaluate the model by looking at how well it does with new data through some specified criteria. We evaluate on the train and test set and these evaluations will allow us to see if the model is under overfitting. We will address this in later recipes.
10. **Tune hyperparameters:** Most of the time, we will want to go back and change some of the hyperparameters based on the model's performance. We then repeat the previous steps with different hyperparameters and evaluate the model on the validation set.
11. **Deploy/predict new outcomes:** It is also important to know how to make predictions on new and unseen data. We can do this with all of our models once we have them trained.

How it works...

In TensorFlow, we have to set up the data, variables, placeholders, and model before we can tell the program to train and change the variables to improve predictions. TensorFlow accomplishes this through computational graphs. These computational graphs are directed graphs with no recursion, which allows for computational parallelism. To do this, we need to create a loss function for TensorFlow to minimize. TensorFlow accomplishes this by modifying the variables in the computational graph. TensorFlow knows how to modify the variables because it keeps track of the computations in the model and automatically computes the variable gradients (how to change each variable) to minimize the loss. Because of this, we can see how easy it can be to make changes and try different data sources.

See also

For more TensorFlow introduction and resources, see the official documentation and tutorials as well:

- **A better place to start is the official Python API documentation:** https://www.tensorflow.org/api_docs/python/
- **There are also tutorials available:** <https://www.tensorflow.org/tutorials/>
- **An unofficial collection of TensorFlow tutorials, projects, presentations, and code repositories:** <https://github.com/jtoy/awesome-tensorflow>

Declaring variables and tensors

Tensors are the primary data structure that TensorFlow uses to operate on the computational graph. We can declare these tensors as variables and/or feed them in as placeholders. To do this, first, we must learn how to create tensors.



A tensor is a mathematical term that refers to generalized vectors or matrices. If vectors are one-dimensional and matrices are two-dimensional, a tensor is n-dimensional (where n could be 1, 2, or even larger).

Getting ready

When we create a tensor and declare it as a variable, TensorFlow creates several graph structures in our computation graph. It is also important to point out that just by creating a tensor, TensorFlow is not adding anything to the computational graph. TensorFlow does this only after running an operation to initialize the variables. See the next section, on variables and placeholders, for more information.

How to do it...

Here, we will cover the main ways that we can create tensors in TensorFlow:

1. Fixed tensors:

- In the following code, we are creating a zero-filled tensor:

```
|     zero_tsr = tf.zeros([row_dim, col_dim])
```

- In the following code, we are creating a one-filled tensor:

```
|     ones_tsr = tf.ones([row_dim, col_dim])
```

- In the following code, we are creating a constant-filled tensor:

```
|     filled_tsr = tf.fill([row_dim, col_dim], 42)
```

- In the following code, we are creating a tensor out of an existing constant:

```
|     constant_tsr = tf.constant([1,2,3])
```

 Note that the `tf.constant()` function can be used to broadcast a value into an array, mimicking the behavior of `tf.fill()` by writing `tf.constant(42, [row_dim, col_dim])`.

2. Tensors of similar shape:

We can also initialize variables based on the shape of other tensors, as follows:

```
|     zeros_similar = tf.zeros_like(constant_tsr)
|     ones_similar = tf.ones_like(constant_tsr)
```

 Note that since these tensors depend on prior tensors, we must initialize them in order. Attempting to initialize all the tensors all at once will result in an error. See the There's more... subsection at the end of the next section, on variables and placeholders.

3. Sequence tensors:

TensorFlow allows us to specify tensors that contain defined intervals. The following functions behave very similarly to the NumPy's `linspace()` outputs and `range()` outputs. See the following function:

```
|     linear_tsr = tf.linspace(start=0, stop=1, start=3)
```

The resultant tensor has a sequence of [0.0, 0.5, 1.0]. Note that this function includes the specified stop value. See the following function for

more information:

```
| integer_seq_tsr = tf.range(start=6, limit=15, delta=3)
```

The result is the sequence [6, 9, 12]. Note that this function does not include the limit value.

4. **Random tensors:** The following generated random numbers are from a uniform distribution:

```
| randunif_tsr = tf.random_uniform([row_dim, col_dim], minval=0, maxval=1)
```

Note that this random uniform distribution draws from the interval that includes the `minval` but not the `maxval` ($\text{minval} \leq x < \text{maxval}$).

To get a tensor with random draws from a normal distribution, you can run the following code:

```
| randnorm_tsr = tf.random_normal([row_dim, col_dim], mean=0.0, stddev=1.0)
```

There are also times where we want to generate normal random values that are assured within certain bounds. The `truncated_normal()` function always picks normal values within two standard deviations of the specified mean:

```
| runcnorm_tsr = tf.truncated_normal([row_dim, col_dim], mean=0.0, stddev=1.0)
```

We might also be interested in randomizing entries of arrays. To accomplish this, there are two functions that can help us: `random_shuffle()` and `random_crop()`. The following code performs this:

```
| shuffled_output = tf.random_shuffle(input_tensor)
| cropped_output = tf.random_crop(input_tensor, crop_size)
```

Later on in this book, we will be interested in randomly cropping images of size (height, width, 3) where there are three color spectrums. To fix a dimension in the `cropped_output`, you must give it the maximum size in that dimension:

```
| cropped_image = tf.random_crop(my_image, [height/2, width/2, 3])
```

How it works...

Once we have decided how to create the tensors, we may also create the corresponding variables by wrapping the tensor in the `variable()` function, as follows (more on this in the next section):

```
|   my_var = tf.Variable(tf.zeros([row_dim, col_dim]))
```

There's more...

We are not limited to the built-in functions: we can convert any NumPy array into a Python list, or a constant into a tensor using the `convert_to_tensor()` function. Note that this function also accepts tensors as an input in case we wish to generalize a computation inside a function.

Using placeholders and variables

Placeholders and variables are key tools in with regard to using computational graphs in TensorFlow. We must understand the difference between them and when to best use them to our advantage.

Getting ready

One of the most important distinctions to make with data is whether it is a placeholder or a variable. Variables are the model parameters of the algorithm, and TensorFlow keeps track of how to change these to optimize the algorithm. Placeholders are objects that allow you to feed in data of a specific type and shape, or that depend on the results of the computational graph, such as the expected outcome of a computation.

How to do it...

The main way to create a variable is by using the `variable()` function, which takes a tensor as an input and outputs a variable. This is only the declaration, and we still need to initialize the variable. Initializing is what puts the variable with the corresponding methods on the computational graph. Here is an example of creating and initializing a variable:

```
my_var = tf.Variable(tf.zeros([2,3]))
sess = tf.Session()
initialize_op = tf.global_variables_initializer()
sess.run(initialize_op)
```

To see what the computational graph looks like after creating and initializing a variable, see the following part of this recipe. Placeholders are just holding the position for data to be fed into the graph. Placeholders get data from a `feed_dict` argument in the session. To put a placeholder into the graph, we must perform at least one operation on the placeholder. In the following code snippet, we initialize the graph, declare `x` to be a placeholder (of a predefined size), and define `y` as the identity operation on `x`, which just returns `x`. We then create data to feed into the `x` placeholder and run the identity operation. The code is shown here, and the resultant graph is in the following section:

```
sess = tf.Session()
x = tf.placeholder(tf.float32, shape=[2,2])
y = tf.identity(x)
x_vals = np.random.rand(2,2)
sess.run(y, feed_dict={x: x_vals})
# Note that sess.run(x, feed_dict={x: x_vals}) will result in a self-referencing err
```



It is worth noting that TensorFlow will not return a self-referenced placeholder in the feed dictionary. In other words, running `sess.run(x, feed_dict={x: x_vals})` in the following graph will return an error.

How it works...

The computational graph of initializing a variable as a tensor of zeros is shown in the following diagram:

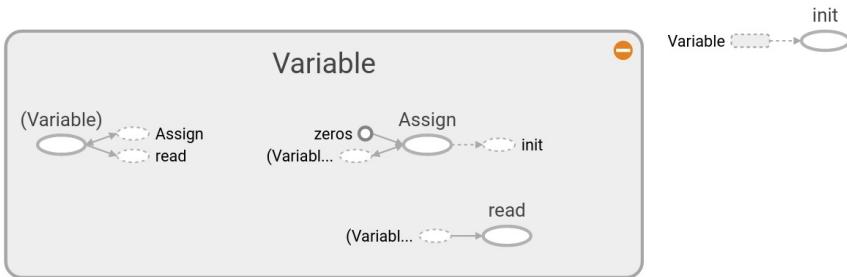


Figure 1: Variable

Here, we can see what the computational graph looks like in detail with just one variable, initialized all to zeros. The grey shaded region is a very detailed view of the operations and constants that were involved. The main computational graph with less detail is the smaller graph outside of the grey region in the upper-right corner. For more details on creating and visualizing graphs, see the first section of [Chapter 10, Taking TensorFlow to Production](#). Similarly, the computational graph of feeding a NumPy array into a placeholder can be seen in the following diagram:

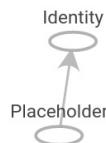


Figure 2: The computational graph of an initialized placeholder

The grey shaded region is a very detailed view of the operations and constants that were involved. The main computational graph with less detail is the smaller graph outside of the grey region in the upper-right corner.

There's more...

During the run of the computational graph, we have to tell TensorFlow when to initialize the variables we have created. While each variable has an `initializer` method, the most common way to do this is with the helper function, that is, `global_variables_initializer()`. This function creates an operation in the graph that initializes all the variables we have created, as follows:

```
|   initializer_op = tf.global_variables_initializer()
```

But if we want to initialize a variable based on the results of initializing another variable, we have to initialize variables in the order we want, as follows:

```
sess = tf.Session()
first_var = tf.Variable(tf.zeros([2,3]))
sess.run(first_var.initializer)
second_var = tf.Variable(tf.zeros_like(first_var))
# 'second_var' depends on the 'first_var'
sess.run(second_var.initializer)
```

Working with matrices

Understanding how TensorFlow works with matrices is very important in understanding the flow of data through computational graphs.



It is worth emphasizing the importance of matrices in machine learning (and mathematics in general). Most machine learning algorithms are computationally expressed as matrix operations. This book does not cover the mathematical background on matrix properties and matrix algebra (linear algebra), so the reader is strongly encouraged to learn enough about matrices to be comfortable with matrix algebra.

Getting ready

Many algorithms depend on matrix operations. TensorFlow gives us easy-to-use operations to perform such matrix calculations. For all of the following examples, we first create a graph session by running the following code:

```
| import tensorflow as tf  
| sess = tf.Session()
```

How to do it...

We will proceed with the recipe as follows:

1. **Creating matrices:** We can create two-dimensional matrices from NumPy arrays or nested lists, as we described in the *Creating and using tensors* recipe. We can also use the tensor creation functions and specify a two-dimensional shape for functions such as `zeros()`, `ones()`, `truncated_normal()`, and so on. TensorFlow also allows us to create a diagonal matrix from a one-dimensional array or list with the `diag()` function, as follows:

```
identity_matrix = tf.diag([1.0, 1.0, 1.0])
A = tf.truncated_normal([2, 3])
B = tf.fill([2,3], 5.0)
C = tf.random_uniform([3,2])
D = tf.convert_to_tensor(np.array([[1., 2., 3.],[-3., -7., -1.],[0., 5., -2.]]))
print(sess.run(identity_matrix))
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
print(sess.run(A))
[[ 0.96751703  0.11397751 -0.3438891 ]
 [-0.10132604 -0.8432678   0.29810596]]
print(sess.run(B))
[[ 5.  5.  5.]
 [ 5.  5.  5.]]
print(sess.run(C))
[[ 0.33184157  0.08907614]
 [ 0.53189191  0.67605299]
 [ 0.95889051  0.67061249]]
print(sess.run(D))
[[ 1.  2.  3.]
 [-3. -7. -1.]
 [ 0.  5. -2.]]
```



Note that if we were to run `sess.run(C)` again, we would reinitialize the random variables and end up with different random values.

2. **Addition, subtraction, and multiplication:** To add, subtract, or multiply matrices of the same dimension, TensorFlow uses the following function:

```
print(sess.run(A+B))
[[ 4.61596632  5.39771316  4.4325695 ]
 [ 3.26702736  5.14477345  4.98265553]]
print(sess.run(B-B))
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
Multiplication
print(sess.run(tf.matmul(B, identity_matrix)))
[[ 5.  5.  5.]
 [ 5.  5.  5.]]
```

It is important to note that the `matmul()` function has arguments that specify whether or not to transpose the arguments before multiplication or whether each matrix is sparse.



Note that matrix division is not explicitly defined. While many define matrix division as multiplying by the inverse, it is fundamentally different compared to real-numbered division.

3. The transpose:

Transpose a matrix (flip the columns and rows) as follows:

```
print(sess.run(tf.transpose(C)))
[[ 0.67124544  0.26766731  0.99068872]
 [ 0.25006068  0.86560275  0.58411312]]
```

Again, it is worth mentioning that reinitializing gives us different values than before.

4. Determinant:

To calculate the determinant, use the following:

```
print(sess.run(tf.matrix_determinant(D)))
-38.0
```

5. Inverse:

To find the inverse of a square matrix, see the following:

```
print(sess.run(tf.matrix_inverse(D)))
[[-0.5         -0.5         -0.5        ]
 [ 0.15789474  0.05263158  0.21052632]
 [ 0.39473684  0.13157895  0.02631579]]
```



The inverse method is based on the Cholesky decomposition, only if the matrix is symmetric positive definite. If the matrix is not symmetric positive definite then it is based on the LU decomposition.

6. Decompositions:

For the Cholesky decomposition, use the following:

```
print(sess.run(tf.cholesky(identity_matrix)))
[[ 1.  0.  1.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

7. Eigenvalues and eigenvectors:

For eigenvalues and eigenvectors, use the following code:

```
print(sess.run(tf.self_adjoint_eig(D)))
[[-10.65907521  -0.22750691   2.88658212]
 [ 0.21749542    0.63250104  -0.74339638]
 [ 0.84526515    0.2587998   0.46749277]
 [-0.4880805     0.73004459   0.47834331]]
```

Note that the `self_adjoint_eig()` function outputs the eigenvalues in the first row

and the subsequent vectors in the remaining vectors. In mathematics, this is known as the eigendecomposition of a matrix.

How it works...

TensorFlow provides all the tools for us to get started with numerical computations and adding such computations to our graphs. This notation might seem quite heavy for simple matrix operations. Remember that we are adding these operations to the graph and telling TensorFlow which tensors to run through those operations. While this might seem verbose now, it helps us understand the notation in later chapters when this way of computation will make it easier to accomplish our goals.

Declaring operations

Now, we must learn what other operations we can add to a TensorFlow graph.

Getting ready

Besides the standard arithmetic operations, TensorFlow provides us with more operations that we should be aware of and how to use them before proceeding. Again, we can create a graph session by running the following code:

```
| import tensorflow as tf  
| sess = tf.Session()
```

How to do it...

TensorFlow has the standard operations on tensors, that is, `add()`, `sub()`, `mul()`, and `div()`. Note that all of the operations in this section will evaluate the inputs element-wise, unless specified otherwise:

1. TensorFlow provides some variations of `div()` and the relevant functions.
2. It is worth mentioning that `div()` returns the same type as the inputs. This means that it really returns the floor of the division (akin to Python 2) if the inputs are integers. To return the Python 3 version, which casts integers into floats before dividing and always returning a float, TensorFlow provides the `truediv()` function, as follows:

```
| print(sess.run(tf.div(3, 4)))  
|   0  
| print(sess.run(tf.truediv(3, 4)))  
|   0.75
```

3. If we have floats and want integer division, we can use the `floordiv()` function. Note that this will still return a float, but it will be rounded down to the nearest integer. This function is as follows:

```
| print(sess.run(tf.floordiv(3.0, 4.0)))  
|   0.0
```

4. Another important function is `mod()`. This function returns the remainder after division. It is as follows:

```
| print(sess.run(tf.mod(22.0, 5.0)))  
|   2.0
```

5. The cross product between two tensors is achieved by the `cross()` function. Remember that the cross product is only defined for two three-dimensional vectors, so it only accepts two three-dimensional tensors. The following code illustrates this use:

```
| print(sess.run(tf.cross([1., 0., 0.], [0., 1., 0.])))  
|   [ 0.  0.  1.0]
```

6. Here is a compact list of the more common math functions. All of these functions operate element-wise:

<code>abs()</code>	Absolute value of one input tensor
<code>ceil()</code>	Ceiling function of one input tensor
<code>cos()</code>	Cosine function of one input tensor
<code>exp()</code>	Base e exponential of one input tensor
<code>floor()</code>	Floor function of one input tensor
<code>inv()</code>	Multiplicative inverse ($1/x$) of one input tensor
<code>log()</code>	Natural logarithm of one input tensor
<code>maximum()</code>	Element-wise max of two tensors
<code>minimum()</code>	Element-wise min of two tensors
<code>neg()</code>	Negative of one input tensor
<code>pow()</code>	The first tensor raised to the second tensor element-wise
<code>round()</code>	Rounds one input tensor

rsqrt()	One over the square root of one tensor
sign()	Returns -1, 0, or 1, depending on the sign of the tensor
sin()	Sine function of one input tensor
sqrt()	Square root of one input tensor
square()	Square of one input tensor

7. **Specialty mathematical functions:** There are some special math functions that get used in machine learning that are worth mentioning, and TensorFlow has built-in functions for them. Again, these functions operate element-wise, unless specified otherwise:

digamma()	Psi function, the derivative of the <code>lgamma()</code> function
erf()	Gaussian error function, element-wise, of one tensor
erfc()	Complementary error function of one tensor
igamma()	Lower regularized incomplete gamma function

<code>igammac()</code>	Upper regularized incomplete gamma function
<code>lbeta()</code>	Natural logarithm of the absolute value of the beta function
<code>lgamma()</code>	Natural logarithm of the absolute value of the gamma function
<code>squared_difference()</code>	Computes the square of the differences between two tensors

How it works...

It is important to know which functions are available to us so that we can add them to our computational graphs. We will mainly be concerned with the preceding functions. We can also generate many different custom functions as compositions of the preceding, as follows:

```
| # Tangent function (tan(pi/4)=1)
| print(sess.run(tf.tan(3.1416/4.)))
| 1.0
```

There's more...

If we wish to add other operations to our graphs that are not listed here, we must create our own from the preceding functions. Here is an example of an operation that wasn't used previously that we can add to our graph. We chose to add a custom polynomial function, $3x^2 - x + 10$, using the following code:

```
| def custom_polynomial(value):
|     return tf.sub(3 * tf.square(value), value) + 10
| print(sess.run(custom_polynomial(11)))
362
```

Implementing activation functions

Activation functions are the key for neural networks to approximate non-linear outputs and adapt to non-linear features. They introduce non-linear operations into neural networks. If we are careful as to which activation functions are selected and where we put them, they are very powerful operations that we can tell TensorFlow to fit and optimize.

Getting ready

When we start to use neural networks, we will use activation functions regularly because activation functions are an essential part of any neural network. The goal of an activation function is just to adjust weight and bias. In TensorFlow, activation functions are non-linear operations that act on tensors. They are functions that operate in a similar way to the previous mathematical operations. Activation functions serve many purposes, but the main concept is that they introduce a non-linearity into the graph while normalizing the outputs. Start a TensorFlow graph with the following commands:

```
| import tensorflow as tf  
| sess = tf.Session()
```

How to do it...

The activation functions live in the **neural network (nn)** library in TensorFlow. Besides using built-in activation functions, we can also design our own using TensorFlow operations. We can import the predefined activation functions (`import tensorflow.nn as nn`) or be explicit and write `nn` in our function calls. Here, we choose to be explicit with each function call:

1. The rectified linear unit, known as ReLU, is the most common and basic way to introduce non-linearity into neural networks. This function is just called `max(0, x)`. It is continuous, but not smooth. It appears as follows:

```
| print(sess.run(tf.nn.relu([-3., 3., 10.])))  
| [ 0.  3.  10.]
```

2. There are times where we will want to cap the linearly increasing part of the preceding ReLU activation function. We can do this by nesting the `max(0, x)` function into a `min()` function. The implementation that TensorFlow has is called the ReLU6 function. This is defined as `min(max(0, x), 6)`. This is a version of the hard-sigmoid function and is computationally faster, and does not suffer from vanishing (infinitesimally near zero) or exploding values. This will come in handy when we discuss deeper neural networks in [chapter 8, Convolutional Neural Networks](#), and [chapter 9, Recurrent Neural Networks](#). It appears as follows:

```
| print(sess.run(tf.nn.relu6([-3., 3., 10.])))  
| [ 0.  3.  6.]
```

3. The sigmoid function is the most common continuous and smooth activation function. It is also called a logistic function and has the form $1 / (1 + \exp(-x))$. The sigmoid function is not used very often because of its tendency to zero-out the backpropagation terms during training. It appears as follows:

```
| print(sess.run(tf.nn.sigmoid([-1., 0., 1.])))  
| [ 0.26894143  0.5           0.7310586 ]
```



We should be aware that some activation functions are not zero-centered, such as the sigmoid. This will require us to zero-mean data prior to using it in most computational graph algorithms.

4. Another smooth activation function is the hyper tangent. The hyper tangent function is very similar to the sigmoid except that instead of having a range between 0 and 1, it has a range between -1 and 1. This function has the form of the ratio of the hyperbolic sine over the hyperbolic cosine. Another way to write this is $((\exp(x) - \exp(-x))/(\exp(x) + \exp(-x)))$. This activation function is as follows:

```
| print(sess.run(tf.nn.tanh([-1., 0., 1.])))
[-0.76159418  0.          0.76159418 ]
```

5. The `softsign` function also gets used as an activation function. The form of this function is $x/(|x| + 1)$. The `softsign` function is supposed to be a continuous (but not smooth) approximation to the sign function. See the following code:

```
| print(sess.run(tf.nn.softsign([-1., 0., -1.])))
[-0.5  0.   0.5]
```

6. Another function, the `softplus` function, is a smooth version of the ReLU function. The form of this function is $\log(\exp(x) + 1)$. It appears as follows:

```
| print(sess.run(tf.nn.softplus([-1., 0., -1.])))
[ 0.31326166  0.69314718  1.31326163]
```



The `softplus` function goes to infinity as the input increases, whereas the `softsign` function goes to 1. As the input gets smaller, however, the `softplus` function approaches zero and the `softsign` function goes to -1.

7. The **Exponential Linear Unit (ELU)** is very similar to the `softplus` function except that the bottom asymptote is -1 instead of 0. The form is $(\exp(x) + 1)$ if $x < 0$ else x . It appears as follows:

```
| print(sess.run(tf.nn.elu([-1., 0., -1.])))
[-0.63212055  0.          1.          ]
```

How it works...

These activation functions are ways that we can introduce non-linearities in neural networks or other computational graphs in the future. It is important to note where in our network we are using activation functions. If the activation function has a range between 0 and 1 (sigmoid), then the computational graph can only output values between 0 and 1. If the activation functions are inside and hidden between nodes, then we want to be aware of the effect that the range can have on our tensors as we pass them through. If our tensors were scaled to have a mean of zero, we will want to use an activation function that preserves as much variance as possible around zero. This would imply that we want to choose an activation function such as the hyperbolic tangent (\tanh) or the softsign. If the tensors are all scaled to be positive, then we would ideally choose an activation function that preserves variance in the positive domain.

There's more...

Here are two graphs that illustrate the different activation functions. The following graphs show the ReLU, ReLU6, softplus, exponential LU, sigmoid, softsign, and hyperbolic tangent functions:

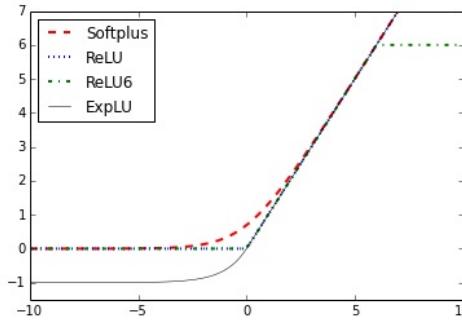


Figure 3: Activation functions of softplus, ReLU, ReLU6, and exponential LU

Here, we can see four of the activation functions: softplus, ReLU, ReLU6, and exponential LU. These functions flatten out to the left of zero and linearly increase to the right of zero, with the exception of ReLU6, which has a maximum value of six:

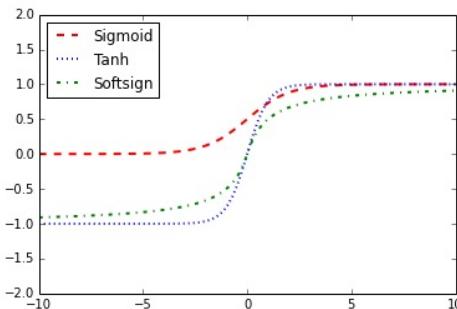


Figure 4: Sigmoid, hyperbolic tangent (tanh), and softsign activation function

Here are the sigmoid, hyperbolic tangent (tanh), and softsign activation functions. These activation functions are all smooth and have a S shape. Note that there are two horizontal asymptotes for these functions.

Working with data sources

For most of this book, we will rely on the use of datasets to fit machine learning algorithms. This section has instructions on how to access each of these datasets through TensorFlow and Python.



*Some of the data sources rely on the maintenance of outside websites so that you can access the data. If these websites change or remove this data, then some of the following code in this section may need to be updated. You may find the updated code at the author's GitHub page: [h
https://github.com/nfmcclure/tensorflow_cookbook](https://github.com/nfmcclure/tensorflow_cookbook).*

Getting ready

In TensorFlow, some of the datasets that we will use are built into Python libraries, some will require a Python script to download, and some will be manually downloaded through the internet. Almost all of these datasets will require an active internet connection so that you can retrieve them.

How to do it...

1. **Iris data:** This dataset is arguably the most classic dataset used in machine learning and maybe all of statistics. It is a dataset that measures sepal length, sepal width, petal length, and petal width of three different types of iris flowers: *Iris setosa*, *Iris virginica*, and *Iris versicolor*. There are 150 measurements overall, which means that there are 50 measurements of each species. To load the dataset in Python, we will use scikit-learn's dataset function, as follows:

```
from sklearn import datasets
iris = datasets.load_iris()
print(len(iris.data))
150
print(len(iris.target))
150
print(iris.data[0]) # Sepal length, Sepal width, Petal length, Petal width
[ 5.1 3.5 1.4 0.2]
print(set(iris.target)) # I. setosa, I. virginica, I. versicolor
{0, 1, 2}
```

2. **Birth weight data:** This data was originally from Baystate Medical Center, Springfield, Mass 1986 (1). This dataset contains the measure of child birth weight and other demographic and medical measurements of the mother and family history. There are 189 observations of eleven variables. The following code shows you how you can access this data in Python:

```
import requests
birthdata_url = 'https://github.com/nfmcclure/tensorflow_cookbook/raw/master/01_
birth_file = requests.get(birthdata_url)
birth_data = birth_file.text.split('\r\n')
birth_header = birth_data[0].split('\t')
birth_data = [[float(x) for x in y.split('\t') if len(x)>=1] for y in birth_data]
print(len(birth_data))
189
print(len(birth_data[0]))
9
```

3. **Boston Housing data:** Carnegie Mellon University maintains a library of datasets in their StatLib Library. This data is easily accessible via The University of California at Irvine's machine learning Repository (<https://archive.ics.uci.edu/ml/index.php>). There are 506 observations of house worth, along with various demographic data and housing attributes (14 variables). The following code shows you how to access this data in Python, via the Keras library:

```

from keras.datasets import boston_housing
(x_train, y_train), (x_test, y_test) = boston_housing.load_data()
housing_header = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD'
print(x_train.shape[0])
404
print(x_train.shape[1])
13

```

4. **MNIST handwriting data:** The MNIST (Mixed National Institute of Standards and Technology) dataset is a subset of the larger NIST handwriting database. The MNIST handwriting dataset is hosted on Yann LeCun's website (<https://yann.lecun.com/exdb/mnist/>). It is a database of 70,000 images of single-digit numbers (0-9) with about 60,000 annotated for a training set and 10,000 for a test set. This dataset is used so often in image recognition that TensorFlow provides built-in functions to access this data. In machine learning, it is also important to provide validation data to prevent overfitting (target leakage). Because of this, TensorFlow sets aside 5,000 images of the train set into a validation set. The following code shows you how to access this data in Python:

```

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
print(len(mnist.train.images))
55000
print(len(mnist.test.images))
10000
print(len(mnist.validation.images))
5000
print(mnist.train.labels[1,:]) # The first label is a 3
[b' 0.  0.  0.  1.  0.  0.  0.  0.  0.]

```

5. **Spam-ham text data.** UCI's machine learning dataset library also holds a spam-ham text message dataset. We can access this .zip file and get the spam-ham text data as follows:

```

import requests
import io
from zipfile import ZipFile
zip_url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00228/smssp
r = requests.get(zip_url)
z = ZipFile(io.BytesIO(r.content))
file = z.read('SMSSpamCollection')
text_data = file.decode()
text_data = text_data.encode('ascii', errors='ignore')
text_data = text_data.decode().split('\n')
text_data = [x.split('\t') for x in text_data if len(x)>=1]
[text_data_target, text_data_train] = [list(x) for x in zip(*text_data)]
print(len(text_data_train))
5574
print(set(text_data_target))
{'ham', 'spam'}
print(text_data_train[1])
Ok lar... Joking wif u oni...

```

6. **Movie review data:** Bo Pang from Cornell has released a movie review dataset that classifies reviews as good or bad (3). You can find the data on the following website: <http://www.cs.cornell.edu/people/pabo/movie-review-data/>. To download, extract, and transform this data, we can run the following code:

```
import requests
import io
import tarfile
movie_data_url = 'http://www.cs.cornell.edu/people/pabo/movie-review-data/rt-polarities'
r = requests.get(movie_data_url)
# Stream data into temp object
stream_data = io.BytesIO(r.content)
tmp = io.BytesIO()
while True:
    s = stream_data.read(16384)
    if not s:
        break
    tmp.write(s)
    stream_data.close()
tmp.seek(0)
# Extract tar file
tar_file = tarfile.open(fileobj=tmp, mode="r:gz")
pos = tar_file.extractfile('rt-polaritydata/rt-polarity.pos')
neg = tar_file.extractfile('rt-polaritydata/rt-polarity.neg')
# Save pos/neg reviews (Also deal with encoding)
pos_data = []
for line in pos:
    pos_data.append(line.decode('ISO-8859-1').encode('ascii', errors='ignore'))
neg_data = []
for line in neg:
    neg_data.append(line.decode('ISO-8859-1').encode('ascii', errors='ignore'))
tar_file.close()
print(len(pos_data))
5331
print(len(neg_data))
5331
# Print out first negative review
print(neg_data[0])
simplistic , silly and tedious .
```

7. **CIFAR-10 image data:** The Canadian Institute For Advanced Research has released an image set that contains 80 million labeled colored images (each image is scaled to 32 x 32 pixels). There are 10 different target classes (airplane, automobile, bird, and so on). CIFAR-10 is a subset that includes 60,000 images. There are 50,000 images in the training set, and 10,000 in the test set. Since we will be using this dataset in multiple ways, and because it is one of our larger datasets, we will not run a script each time we need it. To get this dataset, please navigate to <http://www.cs.toronto.edu/~kriz/cifar.html> and download the CIFAR-10 dataset. We will address how to use this dataset in the appropriate chapters.

8. **The works of Shakespeare text data:** Project Gutenberg (5) is a project

that releases electronic versions of free books. They have compiled all of the works of Shakespeare together. The following code shows you how to access this text file through Python:

```
import requests
shakespeare_url = 'http://www.gutenberg.org/cache/epub/100/pg100.txt'
# Get Shakespeare text
response = requests.get(shakespeare_url)
shakespeare_file = response.content
# Decode binary into string
shakespeare_text = shakespeare_file.decode('utf-8')
# Drop first few descriptive paragraphs.
shakespeare_text = shakespeare_text[7675:]
print(len(shakespeare_text)) # Number of characters
5582212
```

9. **English-German sentence translation data:** The Tatoeba project (<http://tatoeba.org>) collects sentence translations in many languages. Their data has been released under the Creative Commons License. From this data, ManyThings.org (<http://www.manythings.org>) has compiled sentence-to-sentence translations in text files that are available for download. Here, we will use the English-German translation file, but you can change the URL to whichever languages you would like to use:

```
import requests
import io
from zipfile import ZipFile
sentence_url = 'http://www.manythings.org/anki/deu-eng.zip'
r = requests.get(sentence_url)
z = ZipFile(io.BytesIO(r.content))
file = z.read('deu.txt')
# Format Data
eng_ger_data = file.decode()
eng_ger_data = eng_ger_data.encode('ascii', errors='ignore')
eng_ger_data = eng_ger_data.decode().split('\n')
eng_ger_data = [x.split('\t') for x in eng_ger_data if len(x)>=1]
[english_sentence, german_sentence] = [list(x) for x in zip(*eng_ger_data)]
print(len(english_sentence))
137673
print(len(german_sentence))
137673
print(eng_ger_data[10])
['I' won!, 'Ich habe gewonnen!']
```

How it works...

When it comes time to using one of these datasets in a recipe, we will refer you to this section and assume that the data is loaded in such a way as described in the preceding section. If further data transformation or pre-processing is needed, then such code will be provided in the recipe itself.

See also

Here are additional references for the data resources we use in this book:

- Hosmer, D.W., Lemeshow, S., and Sturdivant, R. X. (2013) Applied Logistic Regression: 3rd Edition
- Lichman, M. (2013). UCI machine learning repository <http://archive.ics.uci.edu/ml> Irvine, CA: University of California, School of Information and Computer Science
- Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan, Thumbs up? Sentiment Classification using machine learning techniques, Proceedings of EMNLP 2002 <http://www.cs.cornell.edu/people/pabo/movie-review-data/>
- Krizhevsky. (2009). Learning Multiple Layers of Features from Tiny Images <http://www.cs.toronto.edu/~kriz/cifar.html>
- Project Gutenberg. Accessed April 2016 <http://www.gutenberg.org/>

Additional resources

In this section, you will find additional links, documentation sources, and tutorials that are of great assistance to learning and using TensorFlow.

Getting ready

When learning how to use TensorFlow, it helps to know where to turn for assistance or pointers. This section lists resources to get TensorFlow running and to troubleshoot problems.

How to do it...

Here is a list of TensorFlow resources:

- The code for this book is available online at https://github.com/nfmccclure/tensorflow_cookbook and at the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.
- The official TensorFlow Python API documentation is located at https://www.tensorflow.org/api_docs/python. Here, there is documentation and examples of all of the functions, objects, and methods in TensorFlow.
- TensorFlow's official tutorials are very thorough and detailed. They are located at <https://www.tensorflow.org/tutorials/index.html>. They start covering image recognition models, and work through Word2Vec, RNN models, and sequence-to-sequence models. They also have additional tutorials for generating fractals and solving PDE systems. Note that they are continually adding more tutorials and examples to this collection.
- TensorFlow's official GitHub repository is available via <https://github.com/tensorflow/tensorflow>. Here, you can view the open source code and even fork or clone the most current version of the code if you want. You can also see current filed issues if you navigate to the issues directory.
- A public Docker container that is kept current by TensorFlow is available on Dockerhub at <https://hub.docker.com/r/tensorflow/tensorflow/>.
- A great source for community help is Stack Overflow. There is a tag for TensorFlow. This tag seems to be growing in interest as TensorFlow is gaining more popularity. To view activity on this tag, visit <http://stackoverflow.com/questions/tagged/Tensorflow>.
- While TensorFlow is very agile and can be used for many things, the most common use of TensorFlow is deep learning. To understand the basis for deep learning, how the underlying mathematics works, and to develop more intuition on deep learning, Google has created an online course that's available on Udacity. To sign up and take the video lecture course, visit <https://www.udacity.com/course/deep-learning--ud730>.
- TensorFlow has also made a site where you can visually explore training a neural network while changing the parameters and datasets. Visit <http://playground.tensorflow.org/> to explore how different settings affect the training of

neural networks.

- Geoffrey Hinton teaches an online course called *Neural Networks for Machine Learning* through Coursera <https://www.coursera.org/learn/neural-networks>.
- Stanford University has an online syllabus and detailed course notes for *Convolutional Neural Networks for Visual Recognition* <http://cs231n.stanford.edu/>.

The TensorFlow Way

In this chapter, we will introduce the key components of how TensorFlow operates. Then, we will tie it together to create a simple classifier and evaluate the outcomes. By the end of the chapter, you should have learned about the following:

- Operations in a computational graph
- Layering nested operations
- Working with multiple layers
- Implementing loss functions
- Implementing backpropagation
- Working with batch and stochastic training
- Combining everything together
- Evaluating models

Introduction

Now that we have introduced how TensorFlow creates tensors, and uses variables and placeholders, we will introduce how to act on these objects in a computational graph. From this, we can set up a simple classifier and see how well it performs.



Also, remember that the current and updated code from this book is available online on GitHub at https://github.com/nfmcclure/tensorflow_cookbook.

Operations in a computational graph

Now that we can put objects into our computational graph, we will introduce operations that act on such objects.

Getting ready

To start a graph, we load TensorFlow and create a session, as follows:

```
| import tensorflow as tf  
| sess = tf.Session()
```

How to do it...

In this example, we will combine what we have learned and feed each number in a list into an operation in a graph and print the output:

First, we declare our tensors and placeholders. Here, we will create a NumPy array to feed into our operation:

```
import numpy as np
x_vals = np.array([1., 3., 5., 7., 9.])
x_data = tf.placeholder(tf.float32)
m_const = tf.constant(3.)
my_product = tf.multiply(x_data, m_const)
for x_val in x_vals:
    print(sess.run(my_product, feed_dict={x_data: x_val}))
```

The output of the preceding code is as follows:

```
3.0
9.0
15.0
21.0
27.0
```

How it works...

The code in this section creates the data and operations on the computational graph. The following figure is what the computational graph looks like:

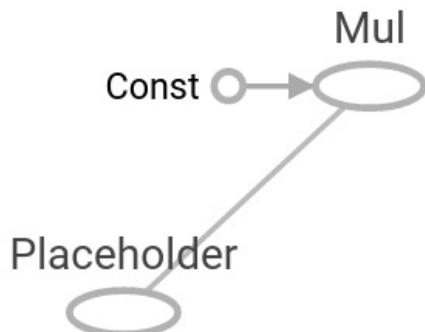


Figure 1: The `x_data` placeholder, along with our multiplicative constant, feeds into the multiplication operation

Layering nested operations

In this recipe, we will learn how to put multiple operations on the same computational graph.

Getting ready

It's important to know how to chain operations together. This will set up layered operations in the computational graph. For a demonstration, we will multiply a placeholder by two matrices and then perform addition. We will feed in two matrices in the form of a three-dimensional NumPy array:

```
| import tensorflow as tf  
| sess = tf.Session()
```

How to do it...

It is also important to note how the data will change shape as it passes through. We will feed in two NumPy arrays of size 3×5 . We will multiply each matrix by a constant of size 5×1 , which will result in a matrix of size 3×1 . We will then multiply this by a 1×1 matrix resulting in a 3×1 matrix again. Finally, we add a 3×1 matrix at the end, as follows:

1. First, we create the data to feed in and the corresponding placeholder:

```
my_array = np.array([[1., 3., 5., 7., 9.],
                     [-2., 0., 2., 4., 6.],
                     [-6., -3., 0., 3., 6.]])
x_vals = np.array([my_array, my_array + 1])
x_data = tf.placeholder(tf.float32, shape=(3, 5))
```

2. Next, we create the constants that we will use for matrix multiplication and addition:

```
m1 = tf.constant([[1.], [0.], [-1.], [2.], [4.]])
m2 = tf.constant([[2.]])
a1 = tf.constant([[10.]])
```

3. Now, we declare the operations and add them to the graph:

```
prod1 = tf.matmul(x_data, m1)
prod2 = tf.matmul(prod1, m2)
add1 = tf.add(prod2, a1)
```

4. Finally, we feed the data through our graph:

```
for x_val in x_vals:
    print(sess.run(add1, feed_dict={x_data: x_val}))
[[ 102.]
 [ 66.]
 [ 58.]]
[[ 114.]
 [ 78.]
 [ 70.]]
```

How it works...

The computational graph we just created can be visualized with TensorBoard. TensorBoard is a feature of TensorFlow that allows us to visualize computational graphs and the values in those graphs. These features are provided natively, unlike other machine learning frameworks. To see how this is done, see the *Visualizing graphs in TensorBoard* recipe in [chapter 11, More with TensorFlow](#). Here is what our layered graph looks as follows:

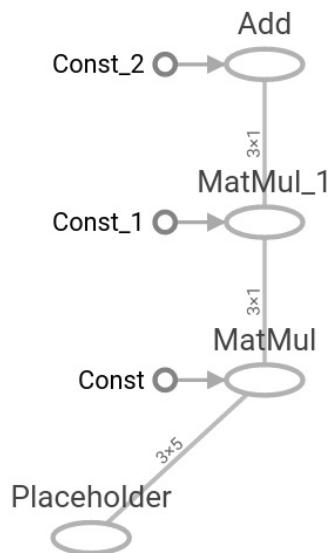


Figure 2: Data size as it propagates upward through the graph

There's more...

We have to declare the data shape and know the outcome shape of the operations before we run data through the graph. This is not always the case. There may be a dimension or two that we do not know beforehand, or some that can vary. To accomplish this, we designate the dimension or dimensions that can vary (or is unknown) as value `None`. For example, to make the prior data placeholder have an unknown amount of columns, we would write the following line:

```
|   x_data = tf.placeholder(tf.float32, shape=(3, None))
```

This allows us to break matrix multiplication rules but we must still obey the fact that the multiplying constant must have the same number of rows. We can either generate this dynamically or reshape `x_data` as we feed data into our graph. This will come in handy in later chapters when we are feeding data in multiple batches of varying batch sizes.



While the use of `None` as a dimension allows us to use variably sized dimensions, it is always recommended to be as explicit as possible when filling out dimensions. If we are standardizing our dimensions to be a fixed size, then we should explicitly write that size as the dimension. The use of `None` as a dimension is recommended to be limited to the batch size of the data (or how many data points we are computing on at once).

Working with multiple layers

Now that we have covered multiple operations, we will cover how to connect various layers that have data propagating through them.

Getting ready

In this recipe, we will introduce how to best connect various layers, including custom layers. The data we will generate and use will be representative of small random images. It is best to understand this type of operation with a simple example and see how we can use some built-in layers to perform calculations. The first layer we will explore is called a **moving window**. We will perform a small moving window average across a 2D image and then the second layer will be a custom operation layer.

In this section, we will see that the computational graph can get large and hard to look at. To address this, we will also introduce ways to name operations and create scopes for layers. To start, load `numpy` and `tensorflow`, and create a graph using the following:

```
| import tensorflow as tf
| import numpy as np
| sess = tf.Session()
```

How to do it...

We proceed with the recipe as follows:

1. First, we create our sample 2D image with NumPy. This image will be a 4×4 pixel image. We will create it in four dimensions; the first and last dimensions will have a size of 1. Note that some TensorFlow image functions will operate on four-dimensional images. Those four dimensions are image number, height, width, and channel, and to make it one image with one channel, we set two of the dimensions to 1, as follows:

```
| x_shape = [1, 4, 4, 1]
| x_val = np.random.uniform(size=x_shape)
```

2. Now, we have to create the placeholder in our graph where we can feed in the sample image, as follows:

```
| x_data = tf.placeholder(tf.float32, shape=x_shape)
```

3. To create a moving window average across our 4×4 image, we will use a built-in function that will convolute a constant across a window of the shape 2×2 . The function we will use is `conv2d()`; this function is quite commonly used in image processing and in TensorFlow. This function takes a piecewise product of the window and a filter we specify. We must also specify a stride for the moving window in both directions. Here, we will compute four moving window averages: the upper-left, upper-right, lower-left, and lower-right four pixels. We do this by creating a 2×2 window and having strides of length 2 in each direction. To take the average, we will convolute the 2×2 window with a constant of 0.25, as follows:

```
| my_filter = tf.constant(0.25, shape=[2, 2, 1, 1])
| my_strides = [1, 2, 2, 1]
| mov_avg_layer= tf.nn.conv2d(x_data, my_filter, my_strides,
|                             padding='SAME', name='Moving_Avg_Window')
```

Note that we are also naming this layer `Moving_Avg_Window` by using the `name` argument of the function.



To figure out the output size of a convolutional layer, we can use the following formula:

Output = $(W - F + 2P)/S + 1$, where W is the input size, F is the filter size, P is the padding of zeros, and S is the stride.

4. Now, we define a custom layer that will operate on the 2×2 output of the

moving window average. The custom function will first multiply the input by another 2×2 matrix tensor, and then add 1 to each entry. After this, we take the sigmoid of each element and return the 2×2 matrix. Since matrix multiplication only operates on two-dimensional matrices, we need to drop the extra dimensions of our image that are of size 1. TensorFlow can do this with the built-in `squeeze()` function. Here, we define the new layer:

```
def custom_layer(input_matrix):
    input_matrix_squeezed = tf.squeeze(input_matrix)
    A = tf.constant([[1., 2.], [-1., 3.]])
    b = tf.constant(1., shape=[2, 2])
    temp1 = tf.matmul(A, input_matrix_squeezed)
    temp = tf.add(temp1, b) # Ax + b
    return tf.sigmoid(temp)
```

5. Now, we have to place the new layer on the graph. We will do this with a named scope so that it is identifiable and collapsible/expandable on the computational graph, as follows:

```
with tf.name_scope('Custom_Layer') as scope:
    custom_layer1 = custom_layer(mov_avg_layer)
```

6. Now, we just feed in the 4×4 image to replace the placeholder and tell TensorFlow to run the graph, as follows:

```
print(sess.run(custom_layer1, feed_dict={x_data: x_val}))
[[ 0.91914582  0.96025133]
 [ 0.87262219  0.9469803 ]]
```

How it works...

The visualized graph looks better with the naming of operations and scoping of layers. We can collapse and expand the custom layer because we created it in a named scope. In the following diagram, see the collapsed version on the left and the expanded version on the right:

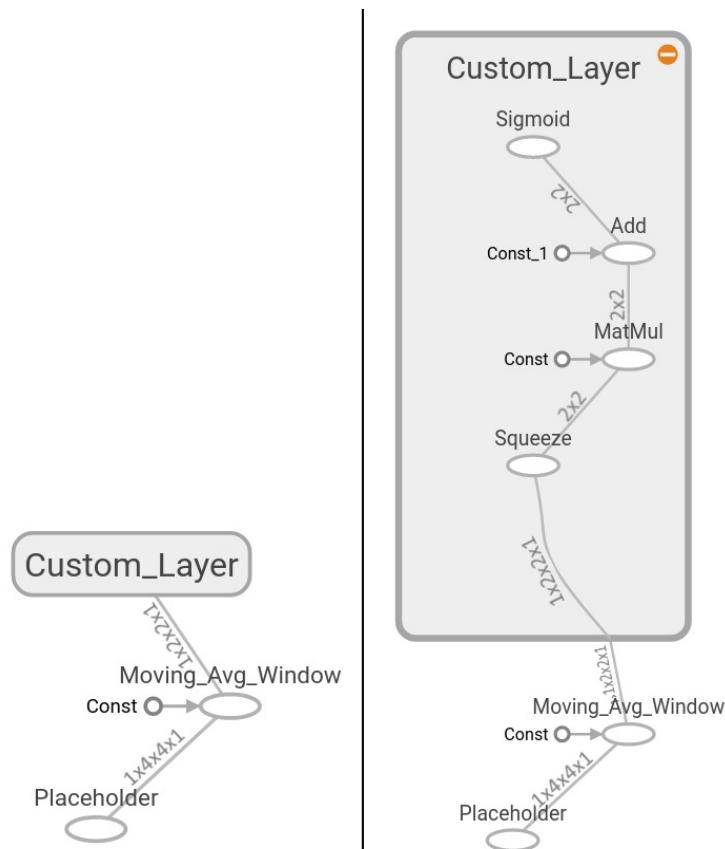


Figure 3: Computational graph with two layers

The first layer is named **Moving_Avg_Window**. The second is a collection of operations called **Custom_Layer**. It is collapsed on the left and expanded on the right.

Implementing loss functions

Loss functions are very important for machine learning algorithms. They measure the distance between the model outputs and the target (truth) values. In this recipe, we show various loss function implementations in TensorFlow.

Getting ready

In order to optimize our machine learning algorithms, we will need to evaluate the outcomes. Evaluating outcomes in TensorFlow depends on specifying a loss function. A loss function tells TensorFlow how good or bad the predictions are compared to the desired result. In most cases, we will have a set of data and a target on which to train our algorithm. The loss function compares the target to the prediction and gives a numerical distance between the two.

For this recipe, we will cover the main loss functions that we can implement in TensorFlow.

To see how the different loss functions operate, we will plot them in this recipe. We will first start a computational graph and load `matplotlib`, a Python plotting library, as follows:

```
| import matplotlib.pyplot as plt  
| import tensorflow as tf
```

How to do it...

- First, we will talk about loss functions for regression, which means predicting a continuous dependent variable. To start, we will create a sequence of our predictions and a target as a tensor. We will output the results across 500 x values between -1 and 1. See the How it works... section for a plot of the outputs. Use the following code:

```
| x_vals = tf.linspace(-1., 1., 500)
| target = tf.constant(0.)
```

- The L2 norm loss is also known as the **Euclidean loss function**. It is just the square of the distance to the target. Here, we will compute the loss function as if the target is zero. The L2 norm is a great loss function because it is very curved near the target and algorithms can use this fact to converge to the target more slowly the closer it gets to zero. We can implement this as follows:

```
| l2_y_vals = tf.square(target - x_vals)
| l2_y_out = sess.run(l2_y_vals)
```

i TensorFlow has a built-in form of the L2 norm, called `nn.l2_loss()`. This function is actually half the L2 norm. In other words, it is the same as the previous one but divided by 2.

- The L1 norm loss is also known as the **absolute loss function**. Instead of squaring the difference, we take the absolute value. The L1 norm is better for outliers than the L2 norm because it is not as steep for larger values. One issue to be aware of is that the L1 norm is not smooth at the target, and this can result in algorithms not converging well. It appears as follows:

```
| l1_y_vals = tf.abs(target - x_vals)
| l1_y_out = sess.run(l1_y_vals)
```

- Pseudo-Huber loss is a continuous and smooth approximation to the **Huber loss function**. This loss function attempts to take the best of the L1 and L2 norms by being convex near the target and less steep for extreme values. The form depends on an extra parameter, delta, which dictates how steep it will be. We will plot two forms, `delta1 = 0.25` and `delta2 = 5`, to show the difference, as follows:

```
| delta1 = tf.constant(0.25)
| phuber1_y_vals = tf.multiply(tf.square(delta1), tf.sqrt(1. +
```

```

        tf.square((target - x_vals)/delta1)) - 1.)
phuber1_y_out = sess.run(phuber1_y_vals)
delta2 = tf.constant(5.)
phuber2_y_vals = tf.multiply(tf.square(delta2), tf.sqrt(1. +
            tf.square((target - x_vals)/delta2)) - 1.))
phuber2_y_out = sess.run(phuber2_y_vals)

```

Now, we move on to loss functions for classification problems.

Classification loss functions are used to evaluate loss when predicting categorical outcomes. Usually, the output of our model for a class category is a real-value number between 0 and 1. Then, we choose a cutoff (0.5 is commonly chosen) and classify the outcome as being in that category if the number is above the cutoff. Here, we consider various loss functions for categorical outputs:

- To start, we will need to redefine our predictions (`x_vals`) and `target`. We will save the outputs and plot them in the next section. Use the following:

```

x_vals = tf.linspace(-3., 5., 500)
target = tf.constant(1.)
targets = tf.fill([500], 1.)

```

- Hinge loss is mostly used for support vector machines, but can be used in neural networks as well. It is meant to compute a loss among two target classes, 1 and -1. In the following code, we are using the target value 1, so the closer our predictions are to 1, the lower the loss value:

```

hinge_y_vals = tf.maximum(0., 1. - tf.multiply(target, x_vals))
hinge_y_out = sess.run(hinge_y_vals)

```

- Cross-entropy loss for a binary case is also sometimes referred to as the **logistic loss function**. It comes about when we are predicting the two classes 0 or 1. We wish to measure a distance from the actual class (0 or 1) to the predicted value, which is usually a real number between 0 and 1. To measure this distance, we can use the cross-entropy formula from information theory, as follows:

```

xentropy_y_vals = - tf.multiply(target, tf.log(x_vals)) - tf.multiply((1. - targ
xentropy_y_out = sess.run(xentropy_y_vals)

```

- Sigmoid cross-entropy loss is very similar to the previous loss function except we transform the x values using the sigmoid function before we put them in the cross-entropy loss, as follows:

```

xentropy_sigmoid_y_vals = tf.nn.sigmoid_cross_entropy_with_logits_v2(logits=x_va

```

```
|     xentropy_sigmoid_y_out = sess.run(xentropy_sigmoid_y_vals)
```

9. Weighted cross-entropy loss is a weighted version of the sigmoid cross-entropy loss. We provide a weight on the positive target. For an example, we will weight the positive target by 0.5, as follows:

```
|     weight = tf.constant(0.5)
|     xentropy_weighted_y_vals = tf.nn.weighted_cross_entropy_with_logits(logits=x_val
|     xentropy_weighted_y_out = sess.run(xentropy_weighted_y_vals)
```

10. Softmax cross-entropy loss operates on non-normalized outputs. This function is used to measure a loss when there is only one target category instead of multiple. Because of this, the function transforms the outputs into a probability distribution via the softmax function and then computes the loss function from a true probability distribution, as follows:

```
|     unscaled_logits = tf.constant([[1., -3., 10.]])
|     target_dist = tf.constant([[0.1, 0.02, 0.88]])
|     softmax_xentropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits=unscaled_lo
|     print(sess.run(softmax_xentropy))
[ 1.16012561]
```

11. Sparse softmax cross-entropy loss is the same as the previous, except instead of the target being a probability distribution, it is an index of which category is true. Instead of a sparse all-zero target vector with one value of 1, we just pass in the index of the category that is the true value, as follows:

```
|     unscaled_logits = tf.constant([[1., -3., 10.]])
|     sparse_target_dist = tf.constant([2])
|     sparse_xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=unscaled
|     print(sess.run(sparse_xentropy))
[ 0.00012564]
```

How it works...

Here is how to use `matplotlib` to plot the regression loss functions:

```
x_array = sess.run(x_vals)
plt.plot(x_array, l2_y_out, 'b-', label='L2 Loss')
plt.plot(x_array, l1_y_out, 'r--', label='L1 Loss')
plt.plot(x_array, phuber1_y_out, 'k-.', label='P-Huber Loss (0.25)')
plt.plot(x_array, phuber2_y_out, 'g:', label='P-Huber Loss (5.0)')
plt.ylim(-0.2, 0.4)
plt.legend(loc='lower right', prop={'size': 11})
plt.show()
```

We get the following plot as output from the preceding code:

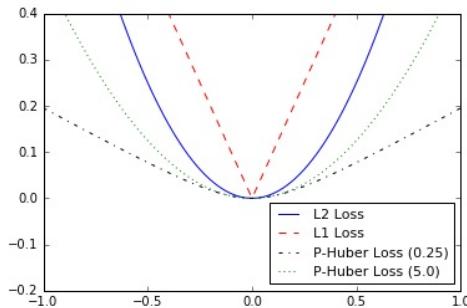


Figure 4: Plotting various regression loss functions

And here is how to use `matplotlib` to plot the various classification loss functions:

```
x_array = sess.run(x_vals)
plt.plot(x_array, hinge_y_out, 'b''', label='Hinge Loss''')
plt.plot(x_array, xentropy_y_out, 'r--''', label='Cross' Entropy Loss')
plt.plot(x_array, xentropy_sigmoid_y_out, 'k-''', label='Cross' Entropy Sigmoid
plt.plot(x_array, xentropy_weighted_y_out, g:'', label='Weighted' Cross Enropy
plt.ylim(-1.5, 3)
plt.legend(loc='lower right''', prop={'size'''': 11})
plt.show()
```

We get the following plot from the preceding code:

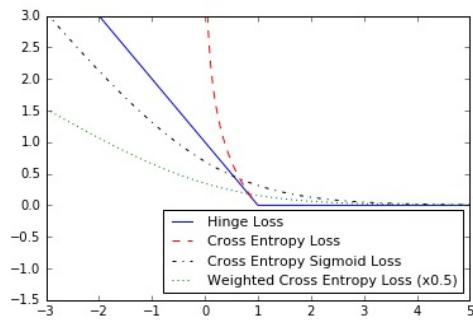


Figure 5: Plots of classification loss functions

There's more...

Here is a table summarizing the different loss functions that we have described:

Loss function	Use	Benefits	Disadvantages
L2	Regression	More stable	Less robust
L1	Regression	More robust	Less stable
Pseudo-Huber	Regression	More robust and stable	One more parameter
Hinge	Classification	Creates a max margin for use in SVM	Unbounded loss affected by outliers
Cross-entropy	Classification	More stable	Unbounded loss, less robust

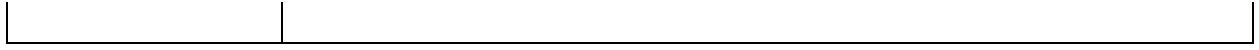
The remaining classification loss functions all have to do with the type of cross-entropy loss. The cross-entropy sigmoid los function is for use on unscaled logits and is preferred over computing the sigmoid and then the cross-entropy, because TensorFlow has better built-in ways to handle numerical edge cases. The same goes for softmax cross-entropy and sparse softmax cross-entropy.



Most of the classification loss functions described here are for two-class predictions. This can be extended to multiple classes by summing the cross-entropy terms over each prediction/target.

There are also many other metrics to look at when evaluating a model. Here is a list of some more to consider:

Model metric	Description
R-squared (coefficient of determination)	For linear models, this is the proportion of variance in the dependent variable that is explained by the independent data. For models with a larger number of features, consider using adjusted R squared.
Root mean squared error	For continuous models, this measures the difference between prediction and actual via the square root of the average squared error.
Confusion matrix	For categorical models, we look at a matrix of predicted categories versus actual categories. A perfect model has all the counts along the diagonal.
Recall	For categorical models, this is the fraction of true positives over all predicted positives.
Precision	For categorical models, this is the fraction of true positives over all actual positives.
F-score	For categorical models, this is the harmonic mean of precision and recall.



Implementing backpropagation

One of the benefits of using TensorFlow is that it can keep track of operations and automatically update model variables based on back propagation. In this recipe, we will introduce how to use this aspect to our advantage when training machine learning models.

Getting ready

Now, we will introduce how to change our variables in the model in such a way that a loss function is minimized. We have learned how to use objects and operations, and create loss functions that will measure the distance between our predictions and targets. Now, we just have to tell TensorFlow how to back propagate errors through our computational graph to update the variables and minimize the loss function. This is done via declaring an optimization function. Once we have an optimization function declared, TensorFlow will go through and figure out the back propagation terms for all of our computations in the graph. When we feed data in and minimize the loss function, TensorFlow will modify our variables in the graph accordingly.

For this recipe, we will do a very simple regression algorithm. We will sample random numbers from a normal distribution, with mean 1 and standard deviation 0.1. Then, we will run the numbers through one operation, which will be to multiply them by a variable, A. From this, the loss function will be the L2 norm between the output and the target, which will always be the value 10. Theoretically, the best value for A will be the number 10, since our data will have mean 1.

The second example is a very simple binary classification algorithm. Here, we will generate 100 numbers from two normal distributions, $N(-1,1)$ and $N(3,1)$. All the numbers from $N(-1, 1)$ will be in target class 0, and all the numbers from $N(3, 1)$ will be in target class 1. The model to differentiate these numbers will be a sigmoid function of a translation. In other words, the model will be $\text{sigmoid}(x + A)$ where A is the variable we will fit. Theoretically, A will be equal to -1. We arrive at this number because if m_1 and m_2 are the means of the two normal functions, the value added to them to translate them equidistant to zero will be $-(m_1 + m_2)/2$. We will see how TensorFlow can arrive at that number in the second example.

While specifying a good learning rate helps the convergence of algorithms, we must also specify a type of optimization. From the preceding two examples, we

are using standard gradient descent. This is implemented with the `GradientDescentOptimizer()` TensorFlow function.

How to do it...

Here is how the regression example works:

1. We start by loading the `numpy` and `tensorflow` numerical Python packages:

```
| import numpy as np  
| import tensorflow as tf
```

2. Now, we start a graph session:

```
| sess = tf.Session()
```

3. Next, we create the data, placeholders, and the `A` variable:

```
| x_vals = np.random.normal(1, 0.1, 100)  
| y_vals = np.repeat(10., 100)  
| x_data = tf.placeholder(shape=[1], dtype=tf.float32)  
| y_target = tf.placeholder(shape=[1], dtype=tf.float32)  
| A = tf.Variable(tf.random_normal(shape=[1]))
```

4. We add the multiplication operation to our graph:

```
| my_output = tf.mul(x_data, A)
```

5. Next, we add our L2 `loss` function between the multiplication output and the target data:

```
| loss = tf.square(my_output - y_target)
```

6. Now, we have to declare a way to optimize the variables in our graph. We declare an optimization algorithm. Most optimization algorithms need to know how far to step in each iteration. This distance is controlled by the learning rate. If our learning rate is too big, our algorithm might overshoot the minimum, but if our learning rate is too small, our algorithm might take too long to converge; this is related to the vanishing and exploding gradient problem. The learning rate has a big influence on convergence and we will discuss this at the end of the section. While here we use the standard gradient descent algorithm, there are many different optimization algorithms that operate differently and can do better or worse depending on the problem. For a great overview of different optimization algorithms, see the paper by Sebastian Ruder in the *See also* section at the end of this

recipe:

```
| my_opt = tf.train.GradientDescentOptimizer(learning_rate=0.02)
| train_step = my_opt.minimize(loss)
```

7. Now we can initialize our model variables:

```
| init = tf.global_variable_initializer()
| sess.run(init)
```



There is a lot of theory on which learning rates are best. This is one of the harder things to figure out in machine learning algorithms. Good papers to read about how learning rates are related to specific optimization algorithms are listed in the See also section at the end of this recipe.

8. The final step is to loop through our training algorithm and tell TensorFlow to train many times. We will do this 101 times and print out results every 25th iteration. To train, we will select a random x and y entry and feed it through the graph. TensorFlow will automatically compute the loss, and slightly change the A bias to minimize the loss:

```
for i in range(100):
    rand_index = np.random.choice(100)
    rand_x = [x_vals[rand_index]]
    rand_y = [y_vals[rand_index]]
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i + 1) % 25 == 0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        print('Loss = ' + str(sess.run(loss, feed_dict={x_data: rand_x, y_target
# Here is the output:
Step #25 A = [ 6.23402166]
Loss = 16.3173
Step #50 A = [ 8.50733757]
Loss = 3.56651
Step #75 A = [ 9.37753201]
Loss = 3.03149
Step #100 A = [ 9.80041122]
Loss = 0.0990248
```

Now, we will introduce the code for the simple classification example. We can use the same TensorFlow script if we reset the graph first. Remember, we will attempt to find an optimal translation, A , which will translate the two distributions to the origin and the sigmoid function will split the two into two different classes:

9. First, we reset the graph and reinitialize the graph session:

```
| from tensorflow.python.framework import ops
| ops.reset_default_graph()
| sess = tf.Session()
```

10. Next, we pull in the data from two different normal distributions, $N(-1, 1)$ and $N(3, 1)$. We will also generate the target labels, placeholders for the data, and bias variable, A :

```
x_vals = np.concatenate((np.random.normal(-1, 1, 50), np.random.normal(3, 1, 50))
y_vals = np.concatenate((np.repeat(0., 50), np.repeat(1., 50)))
x_data = tf.placeholder(shape=[1], dtype=tf.float32)
y_target = tf.placeholder(shape=[1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(mean=10, shape=[1]))
```



We initialized A to around a value of 10, far from the theoretical value of -1. We did this on purpose to show how the algorithm converges from a value of 10 to the optimal value, -1.

11. Next, we add the translation operation to the graph. Remember that we do not have to wrap this in a sigmoid function because the loss function will do that for us:

```
| my_output = tf.add(x_data, A)
```

12. Because the specific loss function expects batches of data that have an extra dimension associated with them (an added dimension, which is the batch number), we will add an extra dimension to the output with the `expand_dims()` function. In the next section, we will discuss how to use variable-sized batches in training. For now, we will again just use one random data point at a time:

```
| my_output_expanded = tf.expand_dims(my_output, 0)
y_target_expanded = tf.expand_dims(y_target, 0)
```

13. Next, we will initialize our one variable, A :

```
| init = tf.initialize_all_variables()
sess.run(init)
```

14. Now, we declare our loss function. We will use cross-entropy with unscaled logits, which transforms them with a sigmoid function. TensorFlow has this all in one function for us, in the neural network package called `nn.sigmoid_cross_entropy_with_logits()`. As stated before, it expects the arguments to have specific dimensions, so we have to use the expanded outputs and targets accordingly:

```
| xentropy = tf.nn.sigmoid_cross_entropy_with_logits( my_output_expanded, y_target
```

15. As with the regression example, we need to add an optimizer function to the graph so that TensorFlow knows how to update the bias variable in the graph:

```
|     my_opt = tf.train.GradientDescentOptimizer(0.05)
|     train_step = my_opt.minimize(xentropy)
```

16. Finally, we loop through a randomly selected data point several hundred times and update the `A` variable accordingly. Every 200 iterations we will print out the value of `A` and the loss:

```
for i in range(1400):
    rand_index = np.random.choice(100)
    rand_x = [x_vals[rand_index]]
    rand_y = [y_vals[rand_index]]
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i + 1) % 200 == 0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        print('Loss = ' + str(sess.run(xentropy, feed_dict={x_data: rand_x, y_ta
Step #200 A = [ 3.59597969]
Loss = [[ 0.00126199]]
Step #400 A = [ 0.50947344]
Loss = [[ 0.01149425]]
Step #600 A = [-0.50994617]
Loss = [[ 0.14271219]]
Step #800 A = [-0.76606178]
Loss = [[ 0.18807337]]
Step #1000 A = [-0.90859312]
Loss = [[ 0.02346182]]
Step #1200 A = [-0.86169094]
Loss = [[ 0.05427232]]
Step #1400 A = [-1.08486211]
Loss = [[ 0.04099189]]
```

How it works...

For a recap and explanation, for both examples we did the following:

1. Created the data. Both examples needed to load data through a placeholder.
2. Initialized placeholders and variables. These were very similar placeholders for the data. The variables were very similar, they both had a multiplicative matrix, A , but the first classification algorithm had a bias term to find the split in the data.
3. Created a loss function, we used the L2 loss for regression and the cross-entropy loss for classification.
4. Defined an optimization algorithm. Both algorithms used gradient descent.
5. Iterated across random data samples to iteratively update our variables.

There's more...

As we mentioned before, the optimization algorithm is sensitive to the choice of learning rate. It is important to summarize the effect of this choice in a concise manner:

Learning rate size	Advantages/disadvantages	Uses
Smaller learning rate	Converges slower but more accurate results	If solution is unstable, try lowering the learning rate first
Larger learning rate	Less accurate, but converges faster	For some problems, helps prevent solutions from stagnating

Sometimes, the standard gradient descent algorithm can get stuck or slow down significantly. This can happen when the optimization is stuck in the flat spot of a saddle. To combat this, there is another algorithm that takes into account a momentum term, which adds on a fraction of the prior step's gradient descent value. TensorFlow has this built in with the `MomentumOptimizer()` function.

Another variant is to vary the optimizer step for each variable in our models. Ideally, we would like to take larger steps for smaller moving variables and shorter steps for faster changing variables. We will not go into the mathematics of this approach, but a common implementation of this idea is called the **Adagrad algorithm**. This algorithm takes into account the whole history of the variable gradients. The function in TensorFlow for this is called `AdagradOptimizer()`.

Sometimes, Adagrad forces the gradients to zero too soon because it takes into

account the whole history. A solution to this is to limit how many steps we use. Doing this is called the Adadelta algorithm. We can apply this by using the `AdadeltaOptimizer()` function.

There are a few other implementations of different gradient descent algorithms. For these, we would refer the reader to the TensorFlow documentation at: https://www.tensorflow.org/api_guides/python/train.

See also

For some references on optimization algorithms and learning rates, see the following papers and articles:

- See also recipes from this chapter as follows:
 - In *Implementing Loss Functions* section.
 - In *Implementing Back Propagation* section.
- Kingma, D., Jimmy, L. Adam: *A Method for Stochastic Optimization*. ICLR 2015 <https://arxiv.org/pdf/1412.6980.pdf>
- Ruder, S. *An Overview of Gradient Descent Optimization Algorithms*. 2016 <https://arxiv.org/pdf/1609.04747v1.pdf>
- Zeiler, M. *ADADelta: An Adaptive Learning Rate Method*. 2012 <http://www.mattzeiler.com/pubs/googleTR2012/googleTR2012.pdf>

Working with batch and stochastic training

While TensorFlow updates our model variables according to back propagation, it can operate on anything from one-datum observation to a large batch of data at once. Operating on one training example can make for a very erratic learning process, while using too large a batch can be computationally expensive. Choosing the right type of training is crucial for getting our machine learning algorithms to converge to a solution.

Getting ready

In order for TensorFlow to compute the variable gradients for back propagation to work, we have to measure the loss on a sample or multiple samples.

Stochastic training only works on one randomly sampled data-target pair at a time, just as we did in the previous recipe. Another option is to put a larger portion of the training examples in at a time and average the loss for the gradient calculation. The sizes of the training batch can vary, up to and including the whole dataset at once. Here, we will show how to extend the prior regression example, which used stochastic training, to batch training.

We will start by loading `numpy`, `matplotlib`, and `tensorflow`, and starting a graph session, as follows:

```
| import matplotlib as plt
| import numpy as np
| import tensorflow as tf
| sess = tf.Session()
```

How to do it...

We proceed with the recipe as follows:

1. We will start by declaring a batch size. This will be how many data observations we will feed through the computational graph at one time:

```
|     batch_size = 20
```

2. Next, we declare the data, placeholders, and variable in the model. The change we make here is that we change the shape of the placeholders. They are now two dimensions, where the first dimension is `None` and the second will be the number of data points in the batch. We could have explicitly set it to 20, but we can generalize and use the `None` value. Again, as mentioned in [Chapter 1, Getting Started with TensorFlow](#), we still have to make sure that the dimensions work out in the model, and this does not allow us to perform any illegal matrix operations:

```
|     x_vals = np.random.normal(1, 0.1, 100)
|     y_vals = np.repeat(10., 100)
|     x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
|     y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
|     A = tf.Variable(tf.random_normal(shape=[1,1]))
```

3. Now, we add our operation to the graph, which will now be matrix multiplication instead of regular multiplication. Remember that matrix multiplication is not commutative, so we have to enter the matrices in the correct order in the `matmul()` function:

```
|     my_output = tf.matmul(x_data, A)
```

4. Our `loss` function will change because we have to take the mean of all the L2 losses of each data point in the batch. We do this by wrapping our prior loss output in TensorFlow's `reduce_mean()` function:

```
|     loss = tf.reduce_mean(tf.square(my_output - y_target))
```

5. We declare our optimizer just as we did before and initialize our model variables, as follows:

```
|     my_opt = tf.train.GradientDescentOptimizer(0.02)
```

```
train_step = my_opt.minimize(loss)
init = tf.global_variables_initializer()
sess.run(init)
```

6. Finally, we will loop through and iterate on the training step to optimize the algorithm. This part is different from before because we want to be able to plot the loss over time and compare the batch versus stochastic training convergence. So, we initialize a list to store the loss function every five intervals:

```
loss_batch = []
for i in range(100):
    rand_index = np.random.choice(100, size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i + 1) % 5 == 0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
        print('Loss = ' + str(temp_loss))
        loss_batch.append(temp_loss)
```

7. Here is the final output of the 100 iterations. Note that the value of A has an extra dimension because it now has to be a 2D matrix:

```
Step #100 A = [[ 9.86720943]]
Loss = 0.
```

How it works...

Batch training and stochastic training differ in their optimization methods and their convergence. Finding a good batch size can be difficult. To see how convergence differs between batch and stochastic, the reader is encouraged to change the batch size to various levels. Here is the code to save and record the stochastic loss in the training loop. Just substitute this code in the previous recipe:

```
loss_stochastic = []
for i in range(100):
    rand_index = np.random.choice(100)
    rand_x = [x_vals[rand_index]]
    rand_y = [y_vals[rand_index]]
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i + 1) % 5 == 0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
        print('Loss = ' + str(temp_loss))
        loss_stochastic.append(temp_loss)
```

Here is the code to produce the plot of both the stochastic and batch losses for the same regression problem:

```
plt.plot(range(0, 100, 5), loss_stochastic, 'b-', label='Stochastic Loss')
plt.plot(range(0, 100, 5), loss_batch, 'r--', label='Batch Loss, size=20')
plt.legend(loc='upper right', prop={'size': 11})
plt.show()
```

We get the following plot:

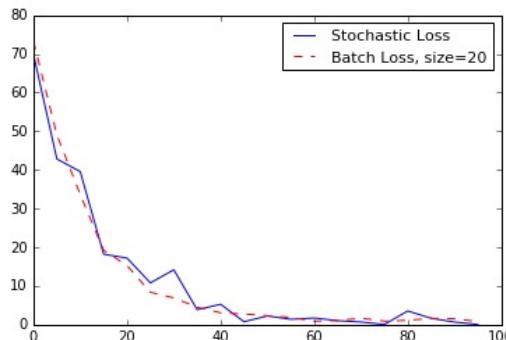


Figure 6: Stochastic loss and batch loss (batch size = 20) plotted over 100 iterations. Note that the batch loss is much smoother and the stochastic loss is much more erratic.

There's more...

Type of training	Advantages	Disadvantages
Stochastic	Randomness may help move out of local minimums.	Generally, needs more iterations to converge.
Batch	Finds minimums quicker.	Takes more resources to compute.

Combining everything together

In this section, we will combine everything we have illustrated so far and create a classifier for the iris dataset.

Getting ready

The iris dataset is described in more detail in the *Working with data sources* recipe in [chapter 1](#), *Getting Started with TensorFlow*. We will load this data and make a simple binary classifier to predict whether a flower is the species Iris setosa or not. To be clear, this dataset has three species, but we will only predict whether a flower is a single species, I. setosa or not, giving us a binary classifier. We will start by loading the libraries and data, then transform the target accordingly.

How to do it...

We proceed with the recipe as follows:

1. First, we load the libraries needed and initialize the computational graph. Note that we also load `matplotlib` here, because we would like to plot the resultant line afterward:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
import tensorflow as tf
sess = tf.Session()
```

2. Next, we load the iris data. We will also need to transform the target data to be just 1 or 0, whether the target is setosa or not. Since the iris dataset marks setosa as a 0, we will change all targets with the value 0 to 1, and the other values all to 0. We will also only use two features, petal length and petal width. These two features are the third and fourth entry in each `x-value`:

```
iris = datasets.load_iris()
binary_target = np.array([1. if x==0 else 0. for x in iris.target])
iris_2d = np.array([[x[2], x[3]] for x in iris.data])
```

3. Let's declare our batch size, data placeholders, and model variables. Remember that the data placeholders for variable batch sizes have `None` as the first dimension:

```
batch_size = 20
x1_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
x2_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(shape=[1, 1]))
b = tf.Variable(tf.random_normal(shape=[1, 1]))
```



Note that we can increase the performance (speed) of the algorithm by decreasing the bytes for floats by using `dtype=tf.float32` instead.

4. Here, we define the linear model. The model will take the form $x2 = x1 * A + b$, and if we want to find points above or below that line, we see whether they are above or below zero when plugged into the equation $x2 - x1 * A - b$. We will do this by taking the sigmoid of that equation and predicting 1 or 0 from that equation. Remember that TensorFlow has `loss` functions with the sigmoid built in, so we just need to

define the output of the model prior to the sigmoid function:

```
| my_mult = tf.matmul(x2_data, A)
| my_add = tf.add(my_mult, b)
| my_output = tf.sub(x1_data, my_add)
```

5. Now, we add our sigmoid cross-entropy loss function with TensorFlow's built-in `sigmoid_cross_entropy_with_logits()` function:

```
| xentropy = tf.nn.sigmoid_cross_entropy_with_logits(my_output, y_target)
```

6. We also have to tell TensorFlow how to optimize our computational graph by declaring an optimizing method. We will want to minimize the cross-entropy loss. We will also choose `0.05` as our learning rate:

```
| my_opt = tf.train.GradientDescentOptimizer(0.05)
| train_step = my_opt.minimize(xentropy)
```

7. Now, we create a variable initialization operation and tell TensorFlow to execute it:

```
| init = tf.global_variables_initializer()
| sess.run(init)
```

8. Now, we will train our linear model with 1,000 iterations. We will feed in the three data points that we require: petal length, petal width, and the target variable. Every 200 iterations, we will print the variable values:

```
| for i in range(1000):
|     rand_index = np.random.choice(len(iris_2d), size=batch_size)
|     rand_x = iris_2d[rand_index]
|     rand_x1 = np.array([[x[0]] for x in rand_x])
|     rand_x2 = np.array([[x[1]] for x in rand_x])
|     rand_y = np.array([[y] for y in binary_target[rand_index]])
|     sess.run(train_step, feed_dict={x1_data: rand_x1, x2_data: rand_x2, y_target: rand_y})
|     if (i + 1) % 200 == 0:
|         print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + ', b = ' + str(b))

Step #200 A = [[ 8.67285347]], b = [[-3.47147632]]
Step #400 A = [[ 10.25393486]], b = [[-4.62928772]]
Step #600 A = [[ 11.152668]], b = [[-5.4077611]]
Step #800 A = [[ 11.81016064]], b = [[-5.96689034]]
Step #1000 A = [[ 12.41202831]], b = [[-6.34769201]]
```

9. The next set of commands extracts the model variables and plots the line on a graph. The resultant graph is in the *How it works...* section:

```
| [[slope]] = sess.run(A)
| [[intercept]] = sess.run(b)
| x = np.linspace(0, 3, num=50)
| ablineValues = []
```

```
for i in x:  
    ablineValues.append(slope*i+intercept)  
  
setosa_x = [a[1] for i,a in enumerate(iris_2d) if binary_target[i]==1]  
setosa_y = [a[0] for i,a in enumerate(iris_2d) if binary_target[i]==1]  
non_setosa_x = [a[1] for i,a in enumerate(iris_2d) if binary_target[i]==0]  
non_setosa_y = [a[0] for i,a in enumerate(iris_2d) if binary_target[i]==0]  
plt.plot(setosa_x, setosa_y, 'rx', ms=10, mew=2, label='setosa')  
plt.plot(non_setosa_x, non_setosa_y, 'ro', label='Non-setosa')  
plt.plot(x, ablineValues, 'b-')  
plt.xlim([0.0, 2.7])  
plt.ylim([0.0, 7.1])  
plt.suptitle('Linear' Separator For I.setosa', fontsize=20)  
plt.xlabel('Petal Length')  
plt.ylabel('Petal Width')  
plt.legend(loc='lower right')  
plt.show()
```

How it works...

Our goal was to fit a line between the I. setosa points and the other two species using only petal width and petal length. If we plot the points and the resultant line, we see that we have achieved this:

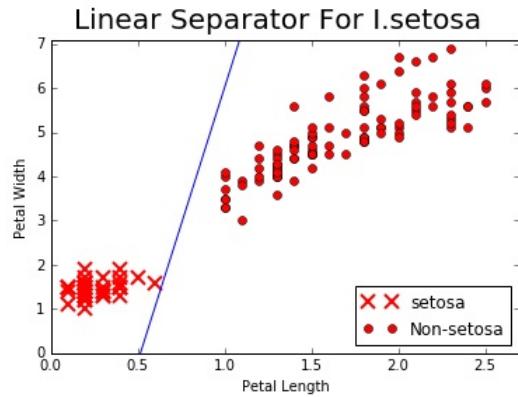


Figure 7: Plot of I. setosa and non-setosa for petal width versus petal length; the solid line is the linear separator that we achieved after 1,000 iterations

There's more...

While we achieved our objective of separating the two classes with a line, it may not be the best model for separating two classes. In [Chapter 4, *Support Vector Machines*](#), we will discuss support vector machines, which are a better way of separating two classes in a feature space.

See also

- For information about the scikit-learn iris dataset implementation, see the documentation at http://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html.

Evaluating models

We have already learned how to train a regression and classification algorithm in TensorFlow. After this, we must be able to evaluate the model's predictions to determine how well it did.

Getting ready

Evaluating models is very important and every subsequent model will have some form of model evaluation. Using TensorFlow, we must build this feature into the computational graph and call it while our model is training and/or after it has finished training.

Evaluating models during training gives us an insight into the algorithm and may give us hints to debug it, improve it, or change models entirely. While evaluation during training isn't always necessary, we will show how to do it with both regression and classification.

After training, we need to quantify how the model performs on the data. Ideally, we have a separate training and test set (and even a validation set) on which we can evaluate the model.

When we want to evaluate a model, we will want to do so on a large batch of data points. If we have implemented batch training, we can reuse our model to make a prediction on such a batch. If we have implemented stochastic training, we may have to create a separate evaluator that can process data in batches.



If we included a transformation on our model output in the `loss` function, for example, `sigmoid_cross_entropy_with_logits()`, we must take that into account when computing predictions for accuracy calculations. Don't forget to include this in your evaluation of the model.

Another important aspect of any model we want to evaluate is if it is a regression or classification model.

Regression models attempt to predict a continuous number. The target is not a category, but a desired number. To evaluate these regression predictions against the actual targets, we need an aggregate measure of the distance between the two. Most of the time, a meaningful loss function will satisfy these criteria. This recipe shows you how to change the simple regression algorithm from before into printing out the loss in the training loop and evaluating the loss at the end. For an example, we will revisit and rewrite our regression example in the prior *Implementing back propagation* recipe in this chapter.

Classification models predict a category based on numerical inputs. The actual targets are a sequence of 1s and 0s, and we must have a measure of how close we are to the truth from our predictions. The loss function for classification models usually isn't that helpful in interpreting how well our model is doing. Usually, we want some sort of classification accuracy, which is commonly the percentage of correctly predicted categories. For this example, we will use the classification example from the prior *Implementing back propagation* recipe in this chapter.

How to do it...

First, we will show how to evaluate the simple regression model that simply fits a constant multiplication to the target, which is 10, as follows:

1. First, we start by loading the libraries and creating the graph, data, variables, and placeholders. There is an additional part to this section that is very important. After we create the data, we will split the data into training and test datasets randomly. This is important because we will always test our models to see whether they are predicting well or not. Evaluating the model both on the training data and test data also lets us see whether the model is overfitting or not:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
sess = tf.Session()
x_vals = np.random.normal(1, 0.1, 100)
y_vals = np.repeat(10., 100)
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
batch_size = 25
train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
A = tf.Variable(tf.random_normal(shape=[1,1]))
```

2. Now, we declare our model, loss function, and optimization algorithm. We will also initialize the model variable `A`. Use the following code:

```
my_output = tf.matmul(x_data, A)
loss = tf.reduce_mean(tf.square(my_output - y_target))
my_opt = tf.train.GradientDescentOptimizer(0.02)
train_step = my_opt.minimize(loss)
init = tf.global_variables_initializer()
sess.run(init)
```

3. We run the training loop just as we have seen previously, as follows:

```
for i in range(100):
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
    rand_x = x_vals_train[rand_index]
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i + 1) % 25 == 0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
```

```

    print('Loss = ' + str(sess.run(loss, feed_dict={x_data: rand_x, y_target
Step #25 A = [[ 6.39879179]]
Loss = 13.7903
Step #50 A = [[ 8.64770794]]
Loss = 2.53685
Step #75 A = [[ 9.40029907]]
Loss = 0.818259
Step #100 A = [[ 9.6809473]]
Loss = 1.10908

```

- Now, to evaluate the model we will output the MSE (loss function) on the training and test sets, as follows:

```

mse_test = sess.run(loss, feed_dict={x_data: np.transpose([x_vals_test]), y_targ
mse_train = sess.run(loss, feed_dict={x_data: np.transpose([x_vals_train]), y_ta
print('MSE' on test:' + str(np.round(mse_test, 2)))
print('MSE' on train:' + str(np.round(mse_train, 2)))
MSE on test:1.35
MSE on train:0.88

```

For the classification example, we will do something very similar. This time, we will need to create our own accuracy function that we can call at the end. One reason for this is that our loss function has the sigmoid built in, and we will need to call the sigmoid separately and test it to see whether our classes are correct:

- In the same script, we can just reload the graph and create our data, variables, and placeholders. Remember that we will also need to separate the data and targets into training and test sets. Use the following code:

```

from tensorflow.python.framework import ops
ops.reset_default_graph()
sess = tf.Session()
batch_size = 25
x_vals = np.concatenate((np.random.normal(-1, 1, 50), np.random.normal(2, 1, 50))
y_vals = np.concatenate((np.repeat(0., 50), np.repeat(1., 50)))
x_data = tf.placeholder(shape=[1, None], dtype=tf.float32)
y_target = tf.placeholder(shape=[1, None], dtype=tf.float32)
train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
A = tf.Variable(tf.random_normal(mean=10, shape=[1]))

```

- We will now add the model and the loss function to the graph, initialize variables, and create the optimization procedure, as follows:

```

my_output = tf.add(x_data, A)
init = tf.initialize_all_variables()
sess.run(init)
xentropy = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(my_output, y_t

```

```

my_opt = tf.train.GradientDescentOptimizer(0.05)
train_step = my_opt.minimize(xentropy)

```

- Now, we run our training loop, as follows:

```

for i in range(1800):
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
    rand_x = [x_vals_train[rand_index]]
    rand_y = [y_vals_train[rand_index]]
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i+1)%200==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)))
        print('Loss = ' + str(sess.run(xentropy, feed_dict={x_data: rand_x, y_ta
Step #200 A = [ 6.64970636]
Loss = 3.39434
Step #400 A = [ 2.2884655]
Loss = 0.456173
Step #600 A = [ 0.29109824]
Loss = 0.312162
Step #800 A = [-0.20045301]
Loss = 0.241349
Step #1000 A = [-0.33634067]
Loss = 0.376786
Step #1200 A = [-0.36866501]
Loss = 0.271654
Step #1400 A = [-0.3727718]
Loss = 0.294866
Step #1600 A = [-0.39153299]
Loss = 0.202275
Step #1800 A = [-0.36630616]
Loss = 0.358463

```

- To evaluate the model, we will create our own prediction operation. We wrap the prediction operation in a squeeze function because we want to make the predictions and targets the same shape. Then, we test for equality with the equal function. After that, we are left with a tensor of true and false values that we cast to float32 and take the mean. This will result in an accuracy value. We will evaluate this function for both the training and test sets, as follows:

```

y_prediction = tf.squeeze(tf.round(tf.nn.sigmoid(tf.add(x_data, A))))
correct_prediction = tf.equal(y_prediction, y_target)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
acc_value_test = sess.run(accuracy, feed_dict={x_data: [x_vals_test], y_target:
acc_value_train = sess.run(accuracy, feed_dict={x_data: [x_vals_train], y_target
print('Accuracy' on train set: ' + str(acc_value_train))
print('Accuracy' on test set: ' + str(acc_value_test))
Accuracy on train set: 0.925
Accuracy on test set: 0.95

```

- Often, seeing the model results (accuracy, MSE, and so on) will help us to evaluate the model. We can easily graph the model and data here because it is one-dimensional. Here is how to visualize the model and data with two separate histograms using matplotlib:

```
A_result = sess.run(A)
bins = np.linspace(-5, 5, 50)
plt.hist(x_vals[0:50], bins, alpha=0.5, label='N(-1,1)', color='white')
plt.hist(x_vals[50:100], bins[0:50], alpha=0.5, label='N(2,1)', color='red')
plt.plot((A_result, A_result), (0, 8), 'k--', linewidth=3, label='A = ' + str(np.
plt.legend(loc='upper right')
plt.title('Binary Classifier, Accuracy=' + str(np.round(acc_value, 2)))
plt.show()
```

How it works...

This results in a graph that shows the predicted best separator for the two classes among a histogram of the two separate data classes.

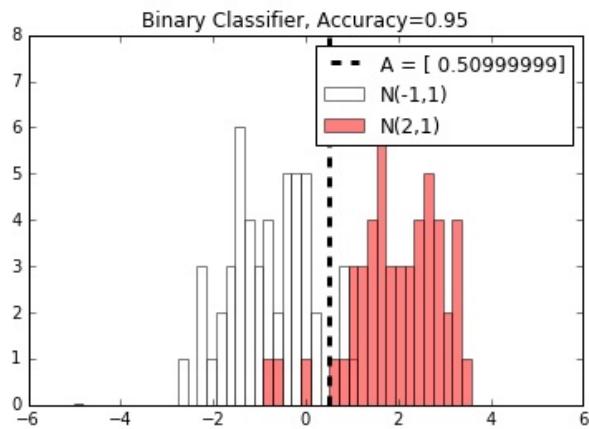


Figure 8: Visualization of the data and the end model, A. The two normal values are centered at -1 and 2, making the theoretical best split at 0.5. Here, the model found the best split very close to that number.

Linear Regression

In this chapter, we will cover recipes involving linear regression. We start with the mathematical formulation for solving linear regression with matrices and move on to implementing standard linear regression and variants with the TensorFlow paradigm. We will cover the following areas:

- Using the matrix inverse method
- Implementing a decomposition method
- Learning the TensorFlow way of regression
- Understanding loss functions in linear regression
- Implementing deming regression
- Implementing lasso and ridge regression
- Implementing elastic net regression
- Implementing logistic regression

Introduction

Linear regression may be one of the most important algorithms in statistics, machine learning, and science in general. It's one of the most widely used algorithms, and it is very important to understand how to implement it and its various flavors. One of the advantages that linear regression has over many other algorithms is that it is very interpretable. We end up with a number for each feature that directly represents how that feature influences the target or dependent variable. In this chapter, we will introduce how linear regression is classically implemented, and then move on to how to best implement it in the TensorFlow paradigm.



Remember that all the code is available on GitHub at https://github.com/nfmcclure/tensorflow_cookbook as well as the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

Using the matrix inverse method

In this recipe, we will use TensorFlow to solve two-dimensional linear regression with the matrix inverse method.

Getting ready

Linear regression can be represented as a set of matrix equations, say $Ax = b$. Here, we are interested in solving the coefficients in matrix x . We have to be careful if our observation matrix (design matrix) A is not square. The solution to solving x can be expressed as $x = (A^T A)^{-1} A^T b$. To show that this is indeed the case, we will generate two-dimensional data, solve it in TensorFlow, and plot the result.

How to do it...

We proceed with the recipe as follows:

1. First, we load the necessary libraries, initialize the graph, and create the data. See the following code:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
sess = tf.Session()
x_vals = np.linspace(0, 10, 100)
y_vals = x_vals + np.random.normal(0, 1, 100)
```

2. Next, we create the matrices to use in the inverse method. We create the A matrix first, which will be a column of x data and a column of 1s. Then, we create the b matrix from the y data. Use the following code:

```
x_vals_column = np.transpose(np.matrix(x_vals))
ones_column = np.transpose(np.matrix(np.repeat(1, 100)))
A = np.column_stack((x_vals_column, ones_column))
b = np.transpose(np.matrix(y_vals))
```

3. We then turn our A and b matrices into tensors, as follows:

```
A_tensor = tf.constant(A)
b_tensor = tf.constant(b)
```

4. Now that we have our matrices set up, we can use TensorFlow to solve this via the matrix inverse method, as follows:

```
tA_A = tf.matmul(tf.transpose(A_tensor), A_tensor)
tA_A_inv = tf.matrix_inverse(tA_A)
product = tf.matmul(tA_A_inv, tf.transpose(A_tensor))
solution = tf.matmul(product, b_tensor)
solution_eval = sess.run(solution)
```

5. We now extract the coefficients from the solution, the slope and y-intercept, with the following code:

```
slope = solution_eval[0][0]
y_intercept = solution_eval[1][0]
print('slope: ' + str(slope))
print('y_intercept: ' + str(y_intercept))
slope: 0.955707151739
y_intercept: 0.174366829314
best_fit = []
```

```
for i in x_vals:  
    best_fit.append(slope*i+y_intercept)  
  
plt.plot(x_vals, y_vals, 'o', label='Data')  
plt.plot(x_vals, best_fit, 'r-', label='Best fit line', linewidth=3)  
plt.legend(loc='upper left')  
plt.show()
```

We get the plot for the preceding code as follows:

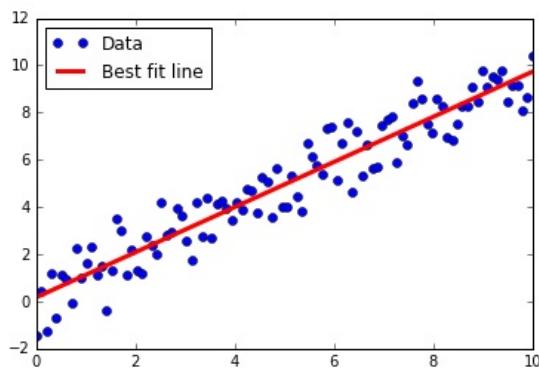


Figure 1: Data points and a best-fit line obtained via the matrix inverse method

How it works...

Unlike prior recipes, or most recipes in this book, the solution here is found only through matrix operations. Most TensorFlow algorithms that we will use are implemented via a training loop and take advantage of automatic backpropagation to update model variables. Here, we illustrate the versatility of TensorFlow by implementing a direct solution to fitting a model to data.



We have used a two-dimensional data example here to show the plot of the fit with the data. It is important to note that the formula used for solving the coefficients, $x = (A^T A)^{-1} A^T b$, will scale to as many features in the data as needed (barring any co-linearity issues).

Implementing a decomposition method

For this recipe, we will implement a matrix decomposition method for linear regression. Specifically, we will use Cholesky decomposition, for which relevant functions exist in TensorFlow.

Getting ready

Implementing the inverse methods from the previous recipe can be numerically inefficient in most cases, especially when the matrices get very large. Another approach is to decompose the `A` matrix and perform matrix operations on the decompositions instead. One such approach is to use the built-in Cholesky decomposition method in TensorFlow.

One reason people are so interested in decomposing a matrix into more matrices is that the resultant matrices will have assured properties that allow us to use certain methods efficiently. Cholesky decomposition decomposes a matrix into a lower and upper triangular matrix, say L and L' , such that these matrices are transposes of each other. For further information on the properties of this decomposition, there are many resources available that describe it and how to arrive at it. Here, we will solve the system $Ax = b$ by writing it as $LL'x = b$. We will first solve $Ly = b$ for y , and then solve $L'x = y$ to arrive at our coefficient matrix, x .

How to do it...

We proceed with the recipe as follows:

1. We will set up the system in exactly the same way as the previous recipe. We will import libraries, initialize the graph, and create the data. Then, we will obtain our A matrix and b matrix in the say way we did previously:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.python.framework import ops
ops.reset_default_graph()
sess = tf.Session()
x_vals = np.linspace(0, 10, 100)
y_vals = x_vals + np.random.normal(0, 1, 100)
x_vals_column = np.transpose(np.matrix(x_vals))
ones_column = np.transpose(np.matrix(np.repeat(1, 100)))
A = np.column_stack((x_vals_column, ones_column))
b = np.transpose(np.matrix(y_vals))
A_tensor = tf.constant(A)
b_tensor = tf.constant(b)
```

2. Next, we find the Cholesky decomposition of our square matrix, $A^T A$:

```
tA_A = tf.matmul(tf.transpose(A_tensor), A_tensor)
L = tf.cholesky(tA_A)
tA_b = tf.matmul(tf.transpose(A_tensor), b)
sol1 = tf.matrix_solve(L, tA_b)
sol2 = tf.matrix_solve(tf.transpose(L), sol1)
```



Note that the TensorFlow function, `cholesky()`, only returns the lower diagonal part of the decomposition. This is fine, as the upper diagonal matrix is just the transpose of the lower one.

3. Now that we have the solution, we extract the coefficients:

```
solution_eval = sess.run(sol2)
slope = solution_eval[0][0]
y_intercept = solution_eval[1][0]
print('slope: ' + str(slope))
print('y_intercept: ' + str(y_intercept))
slope: 0.956117676145
y_intercept: 0.136575513864
best_fit = []
for i in x_vals:
    best_fit.append(slope*i+y_intercept)
plt.plot(x_vals, y_vals, 'o', label='Data')
plt.plot(x_vals, best_fit, 'r-', label='Best fit line', linewidth=3)
plt.legend(loc='upper left')
plt.show()
```

The plot is as follows:

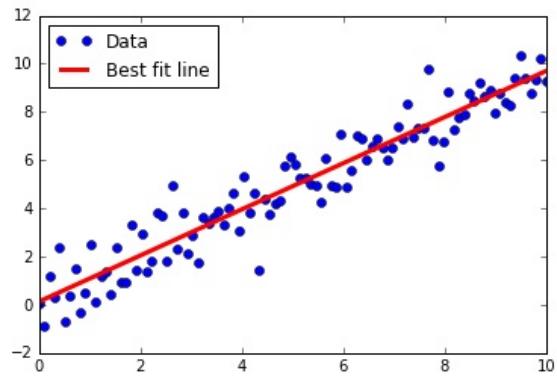


Figure 2: Data points and best fit line obtained via Cholesky decomposition

How it works...

As you can see, we arrive at a very similar answer to the prior recipe. Keep in mind that this way of decomposing a matrix and then performing our operations on the pieces is sometimes much more efficient and numerically stable, especially when considering large matrices of data.

Learning the TensorFlow way of linear regression

While using matrices and decomposition methods are very powerful, TensorFlow has another way to solve for the slope and intercept. TensorFlow can do this iteratively, gradually learning the best linear regression parameters that will minimize the loss.

Getting ready

In this recipe, we will loop through batches of data points and let TensorFlow update the slope and y intercept. Instead of generated data, we will use the iris dataset that is built into the scikit-learn library. Specifically, we will find an optimal line through data points where the x value is the petal width and the y value is the sepal length. We chose these two because there appears to be a linear relationship between them, as we will see in the graphs at the end. We will also talk more about the effects of different loss functions in the next section, but for this recipe we will use the L2 loss function.

How to do it...

We proceed with the recipe as follows:

1. We start by loading the necessary libraries, creating a graph, and loading the data:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
from tensorflow.python.framework import ops
ops.reset_default_graph()
sess = tf.Session()
iris = datasets.load_iris()
x_vals = np.array([x[3] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])
```

2. We then declare our learning rate, batch size, placeholders, and model variables:

```
learning_rate = 0.05
batch_size = 25
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(shape=[1,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
```

3. Next, we write the formula for the linear model $y = Ax + b$:

```
|   model_output = tf.add(tf.matmul(x_data, A), b)
```

4. Then, we declare our L2 loss function (which includes the mean over the batch), initialize the variables, and declare our optimizer. Note that we chose 0.05 as our learning rate:

```
loss = tf.reduce_mean(tf.square(y_target - model_output))
init = tf.global_variables_initializer()
sess.run(init)
my_opt = tf.train.GradientDescentOptimizer(learning_rate)
train_step = my_opt.minimize(loss)
```

5. We can now loop through and train the model on randomly selected batches. We will run it for 100 loops and print out the variable and loss values every 25 iterations. Note that, here, we are also saving the loss of every iteration so that we can view them afterward:

```

loss_vec = []
for i in range(100):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss)
    if (i+1)%25==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + ' b = ' + str(s
        print('Loss = ' + str(temp_loss))

Step #25 A = [[ 2.17270374]] b = [[ 2.85338426]]
Loss = 1.08116
Step #50 A = [[ 1.70683455]] b = [[ 3.59916329]]
Loss = 0.796941
Step #75 A = [[ 1.32762754]] b = [[ 4.08189011]]
Loss = 0.466912
Step #100 A = [[ 1.15968263]] b = [[ 4.38497639]]
Loss = 0.281003

```

6. Next, we will extract the coefficients we found and create a best-fit line to put in the graph:

```

[slope] = sess.run(A)
[y_intercept] = sess.run(b)
best_fit = []
for i in x_vals:
    best_fit.append(slope*i+y_intercept)

```

7. Here, we will create two plots. The first will be the data with the fitted line overlaid. The second is the L2 loss function over the 100 iterations. Here is the code to generate the two plots.

```

plt.plot(x_vals, y_vals, 'o', label='Data Points')
plt.plot(x_vals, best_fit, 'r-', label='Best fit line', linewidth=3)
plt.legend(loc='upper left')
plt.title('Sepal Length vs Petal Width')
plt.xlabel('Petal Width')
plt.ylabel('Sepal Length')
plt.show()
plt.plot(loss_vec, 'k-')
plt.title('L2 Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('L2 Loss')
plt.show()

```

This code generates the following fitted data and loss plots.

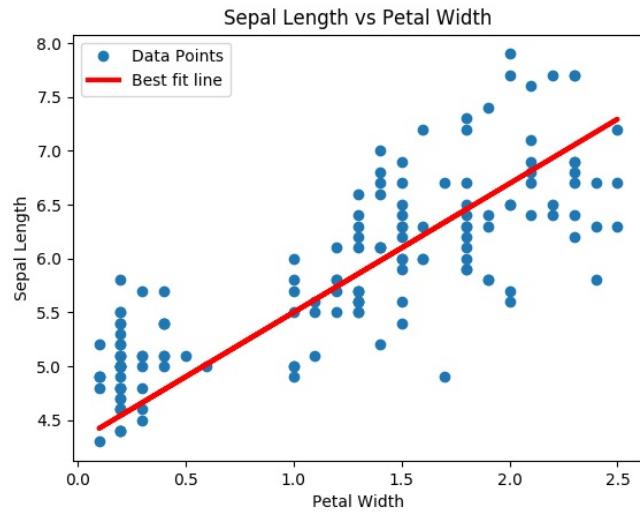


Figure 3: Data points from the iris dataset (sepal length versus petal width) overlaid with the optimal line fit found in TensorFlow.

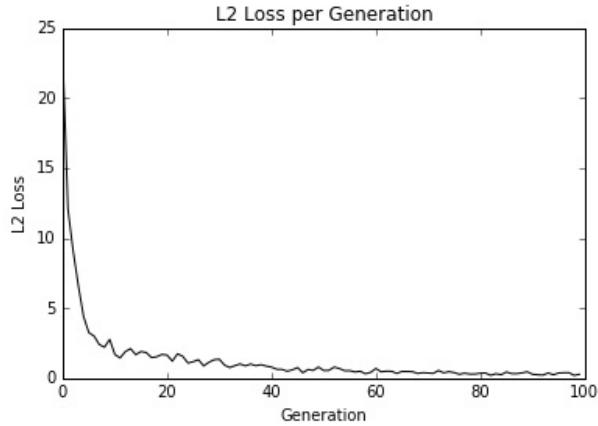


Figure 4: L2 loss of fitting the data with our algorithm; note the jitter in the loss function, which can be decreased with a larger batch size or increased with a smaller batch size.



Here is a good place to note how to see whether the model is overfitting or underfitting the data. If the accuracy is greater on the training set and lower on the test set, then we are overfitting the data. If the accuracy is still increasing on both test and training sets, then the model is underfitting and we should continue training.

How it works...

The optimal line found is not guaranteed to be the line of best fit. Convergence to the line of best fit depends on the number of iterations, batch size, learning rate, and loss function. It is always good practice to observe the loss function over time as it can help you troubleshoot problems or hyperparameter changes.

Understanding loss functions in linear regression

It is important to know the effect of loss functions in algorithm convergence. Here, we will illustrate how the L1 and L2 loss functions affect convergence in linear regression.

Getting ready

We will use the same iris data set as in the prior recipe, but we will change our loss functions and learning rates to see how convergence changes.

How to do it...

We proceed with the recipe as follows:

1. The start of the program is the same as the last recipe, until we get to our loss function. We load the necessary libraries, start a session, load the data, create placeholders, and define our variables and model. One thing to note is that we are pulling out our learning rate and model iterations. We are doing this because we want to show the effect of quickly changing these parameters. Use the following code:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
sess = tf.Session()
iris = datasets.load_iris()
x_vals = np.array([x[3] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])
batch_size = 25
learning_rate = 0.1 # Will not converge with learning rate at 0.4
iterations = 50
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(shape=[1,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
model_output = tf.add(tf.matmul(x_data, A), b)
```

2. Our loss function will change to the L1-loss (`loss_l1`), as follows:

```
| loss_l1 = tf.reduce_mean(tf.abs(y_target - model_output))
```

3. Now, we resume by initializing the variables, declaring our optimizer, and iterating the data through the training loop. Note that we are also saving our loss at every generation to measure the convergence. Use the following code:

```
init = tf.global_variables_initializer()
sess.run(init)
my_opt_l1 = tf.train.GradientDescentOptimizer(learning_rate)
train_step_l1 = my_opt_l1.minimize(loss_l1)
loss_vec_l1 = []
for i in range(iterations):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step_l1, feed_dict={x_data: rand_x, y_target: rand_y})
    temp_loss_l1 = sess.run(loss_l1, feed_dict={x_data: rand_x, y_target: rand_y})
```

```
loss_vec_l1.append(temp_loss_l1)
if (i+1)%25==0:
    print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + ' b = ' + str(s

plt.plot(loss_vec_l1, 'k-', label='L1 Loss')
plt.plot(loss_vec_l2, 'r--', label='L2 Loss')
plt.title('L1 and L2 Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('L1 Loss')
plt.legend(loc='upper right')
plt.show()
```

How it works...

When choosing a loss function, we must also choose a corresponding learning rate that will work with our problem. Here, we will illustrate two situations, one in which L2 is preferred and one in which L1 is preferred.

If our learning rate is small, our convergence will take more time. But if our learning rate is too large, we will have issues with our algorithm never converging. Here is a plot of the loss function of the L1 and L2 loss for the iris linear regression problem when the learning rate is 0.05:

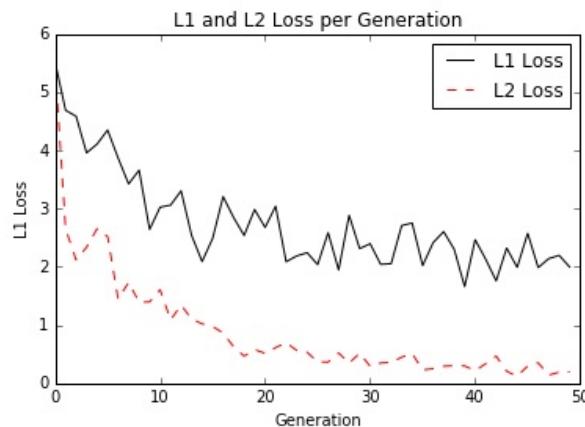


Figure 5: L1 and L2 loss with a learning rate of 0.05 for the iris linear regression problem

With a learning rate of 0.05, it would appear that **L2 loss** is preferred, as it converges to a lower loss. Here is a graph of the loss functions when we increase the learning rate to 0.4:

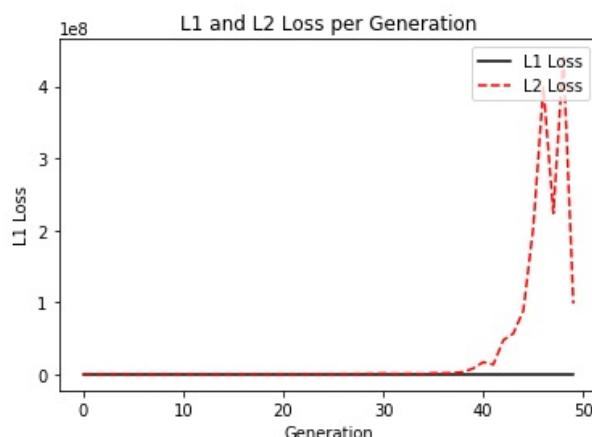


Figure 6: L1 and L2 loss on the iris linear regression problem with a learning rate of 0.4; note that the L1 loss is not visible because of the high scale of the y axis

Here, we can see that a high learning rate can overshoot in the L2 norm, whereas the L1 norm converges.

There's more...

To understand what is happening, we should look at how a large learning rate and small learning rate act on L1 norms and L2 norms. To visualize this, we look at a one-dimensional representation of learning steps on both norms, as follows:

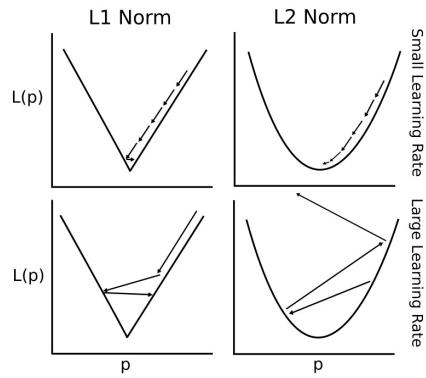


Figure 7: What can happen with the L1 and L2 norm with larger and smaller learning rates

Implementing deming regression

In this recipe, we will implement deming regression, which means we will need a different way to measure the distance between the model line and data points.



*Deming regression goes by several names. It is also known as total regression, **orthogonal distance regression (ODR)**, and shortest-distance regression.*

Getting ready

If least squares linear regression minimizes the vertical distance to the line, deming regression minimizes the total distance to the line. This type of regression minimizes the error in both y and x values.

See the following diagram for a comparison:

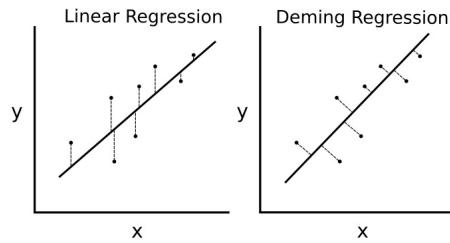


Figure 8: Difference between regular linear regression and deming regression; linear regression on the left minimizes the vertical distance to the line, and deming regression on the right minimizes the total distance to the line

To implement deming regression, we have to modify the loss function. The loss function in regular linear regression minimizes the vertical distance. Here, we want to minimize the total distance. Given a slope and intercept of a line, the perpendicular distance to a point is a known geometric formula. We just have to substitute this formula in and tell TensorFlow to minimize it.

How to do it...

We proceed with the recipe as follows:

1. The code is very similar to the prior recipe, except when we get to the loss function. We begin by loading the libraries; starting a session; loading the data; declaring the batch size; and creating the placeholders, variables, and model output, as follows:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
sess = tf.Session()
iris = datasets.load_iris()
x_vals = np.array([x[3] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])
batch_size = 50
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(shape=[1,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
model_output = tf.add(tf.matmul(x_data, A), b)
```

2. The loss function is a geometric formula that consists of a numerator and denominator. For clarity, we will write these out separately. Given a line, $y = mx + b$, and a point, (x_0, y_0) , the perpendicular distance between the two can be written as follows:

$$d = \frac{|y_0 - (mx_0 + b)|}{\sqrt{m^2 + 1}}$$

```
deming_numerator = tf.abs(tf.sub(y_target, tf.add(tf.matmul(x_data, A), b)))
deming_denominator = tf.sqrt(tf.add(tf.square(A), 1))
loss = tf.reduce_mean(tf.truediv(deming_numerator, deming_denominator))
```

3. We now initialize our variables, declare our optimizer, and loop through the training set to arrive at our parameters, as follows:

```
init = tf.global_variables_initializer()
sess.run(init)
my_opt = tf.train.GradientDescentOptimizer(0.1)
train_step = my_opt.minimize(loss)
loss_vec = []
for i in range(250):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
```

```

rand_y = np.transpose([y_vals[rand_index]])
sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
loss_vec.append(temp_loss)
if (i+1)%50==0:
    print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + ' b = ' + str(s
    print('Loss = ' + str(temp_loss))

```

4. We can plot the output with the following code:

```

[slope] = sess.run(A)
[y_intercept] = sess.run(b)
best_fit = []
for i in x_vals:
    best_fit.append(slope*i+y_intercept)

plt.plot(x_vals, y_vals, 'o', label='Data Points')
plt.plot(x_vals, best_fit, 'r-', label='Best fit line', linewidth=3)
plt.legend(loc='upper left')
plt.title('Sepal Length vs petal Width')
plt.xlabel('petal Width')
plt.ylabel('Sepal Length')
plt.show()

```

We get the following plot for the preceding code:

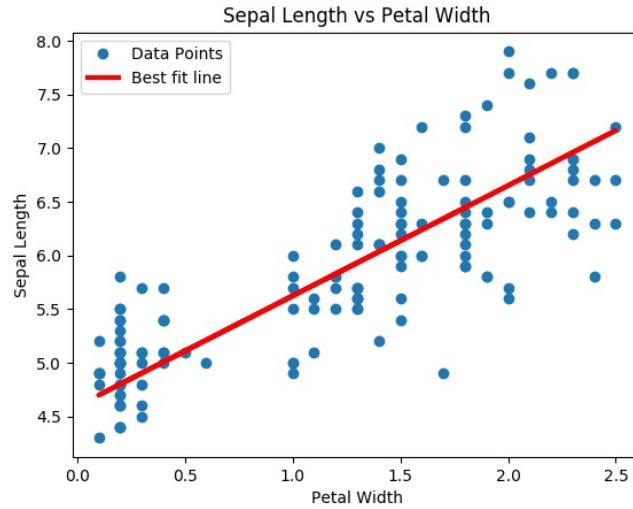


Figure 9: Solution to deming regression on the iris dataset

How it works...

The recipe here for deming regression is almost identical to regular linear regression. The key difference is how we measure the loss between the predictions and the data points. Instead of a vertical loss, we have a perpendicular loss (or total loss) for the y and x values.



This type of regression is used when we assume that the errors in the x and y values are similar. We can also scale the x and y axes in the distance calculation by the difference in the errors, according to our assumptions.

Implementing lasso and ridge regression

There are also ways to limit the influence of coefficients on the regression output. These methods are called regularization methods, and two of the most common regularization methods are lasso and ridge regression. We cover how to implement both of these in this recipe.

Getting ready

Lasso and ridge regression are very similar to regular linear regression, except we add regularization terms to limit the slopes (or partial slopes) in the formula. There may be multiple reasons for this, but a common one is that we wish to restrict the features that have an impact on the dependent variable. This can be accomplished by adding a term to the loss function that depends on the value of our slope, a .

For lasso regression, we must add a term that greatly increases our loss function if the slope, a , gets above a certain value. We could use TensorFlow's logical operations, but they do not have a gradient associated with them. Instead, we will use a continuous approximation to a step function, called the continuous heavy step function, that is scaled up and over to the regularization cutoff we choose. We will show how to do lasso regression in this recipe.

For ridge regression, we just add a term to the L2 norm, which is the scaled L2 norm of the slope coefficient. This modification is simple and is shown in the *There's more...* section at the end of this recipe.

How to do it...

We proceed with the recipe as follows:

1. We will use the iris dataset again and set up our script the same way as before. We first load the libraries; start a session; load the data; declare the batch size; and create the placeholders, variables, and model output as follows:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
from tensorflow.python.framework import ops
ops.reset_default_graph()
sess = tf.Session()
iris = datasets.load_iris()
x_vals = np.array([x[3] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])
batch_size = 50
learning_rate = 0.001
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(shape=[1,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
model_output = tf.add(tf.matmul(x_data, A), b)
```

2. We add the loss function, which is a modified continuous Heaviside step function. We also set the cutoff for lasso regression at 0.9. This means that we want to restrict the slope coefficient to be less than 0.9. Use the following code:

```
lasso_param = tf.constant(0.9)
heavyside_step = tf.truediv(1., tf.add(1., tf.exp(tf.multiply(-100., tf.subtract
regularization_param = tf.mul(heavyside_step, 99.))
loss = tf.add(tf.reduce_mean(tf.square(y_target - model_output)), regularization
```

3. We now initialize our variables and declare our optimizer, as follows:

```
init = tf.global_variables_initializer()
sess.run(init)
my_opt = tf.train.GradientDescentOptimizer(learning_rate)
train_step = my_opt.minimize(loss)
```

4. We run the training loop a fair bit longer because it can take a while to converge. We can see that the slope coefficient is less than 0.9. Use the following code:

```
loss_vec = []
for i in range(1500):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss[0])
    if (i+1)%300==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + ' b = ' + str(s
        print('Loss = ' + str(temp_loss))

Step #300 A = [[ 0.82512331]] b = [[ 2.30319238]]
Loss = [[ 6.84168959]]
Step #600 A = [[ 0.8200165]] b = [[ 3.45292258]]
Loss = [[ 2.02759886]]
Step #900 A = [[ 0.81428504]] b = [[ 4.08901262]]
Loss = [[ 0.49081498]]
Step #1200 A = [[ 0.80919558]] b = [[ 4.43668795]]
Loss = [[ 0.40478843]]
Step #1500 A = [[ 0.80433637]] b = [[ 4.6360755]]
Loss = [[ 0.23839757]]
```

How it works...

We implement lasso regression by adding a continuous Heaviside step function to the loss function of linear regression. Because of the steepness of the step function, we have to be careful with step size. Too big a step size and it will not converge. For ridge regression, see the change required in the next section.

There's more...

For ridge regression, we change the loss function to look as follows:

```
| ridge_param = tf.constant(1.)
| ridge_loss = tf.reduce_mean(tf.square(A))
loss = tf.expand_dims(tf.add(tf.reduce_mean(tf.square(y_target - model_output)), tf.mult
```

Implementing elastic net regression

Elastic net regression is a type of regression that combines lasso regression with ridge regression by adding L1 and L2 regularization terms to the loss function.

Getting ready

Implementing elastic net regression should be straightforward after the previous two recipes, so we will implement this in multiple linear regression on the iris dataset, instead of sticking to the two-dimensional data as before. We will use petal length, petal width, and sepal width to predict sepal length.

How to do it...

We proceed with the recipe as follows:

1. First, we load the necessary libraries and initialize a graph, as follows:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
sess = tf.Session()
```

2. Now, we load the data. This time, each element of x data will be a list of three values instead of one. Use the following code:

```
iris = datasets.load_iris()
x_vals = np.array([[x[1], x[2], x[3]] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])
```

3. Next, we declare the batch size, placeholders, variables, and model output. The only difference here is that we change the size specifications of the x data placeholder to take three values instead of one, as follows:

```
batch_size = 50
learning_rate = 0.001
x_data = tf.placeholder(shape=[None, 3], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(shape=[3,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
model_output = tf.add(tf.matmul(x_data, A), b)
```

4. For elastic net, the loss function has the L1 and L2 norms of the partial slopes. We create these terms and then add them into the loss function, as follows:

```
elastic_param1 = tf.constant(1.)
elastic_param2 = tf.constant(1.)
l1_a_loss = tf.reduce_mean(tf.abs(A))
l2_a_loss = tf.reduce_mean(tf.square(A))
e1_term = tf.multiply(elastic_param1, l1_a_loss)
e2_term = tf.multiply(elastic_param2, l2_a_loss)
loss = tf.expand_dims(tf.add(tf.add(tf.reduce_mean(tf.square(y_target - model_out
```

5. Now, we can initialize the variables, declare our optimization function, run the training loop, and fit our coefficients, as follows:

```

init = tf.global_variables_initializer()
sess.run(init)
my_opt = tf.train.GradientDescentOptimizer(learning_rate)
train_step = my_opt.minimize(loss)
loss_vec = []
for i in range(1000):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = x_vals[rand_index]
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss[0])
    if (i+1)%250==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + ' b = ' + str(s
        print('Loss = ' + str(temp_loss))

```

6. Here is the output of the code:

```

Step #250 A = [[ 0.42095602]
 [ 0.1055888 ]
 [ 1.77064979]] b = [[ 1.76164341]]
Loss = [ 2.87764359]
Step #500 A = [[ 0.62762028]
 [ 0.06065864]
 [ 1.36294949]] b = [[ 1.87629771]]
Loss = [ 1.8032167]
Step #750 A = [[ 0.67953539]
 [ 0.102514 ]
 [ 1.06914485]] b = [[ 1.95604002]]
Loss = [ 1.33256555]
Step #1000 A = [[ 0.6777274 ]
 [ 0.16535147]
 [ 0.8403284 ]] b = [[ 2.02246833]]
Loss = [ 1.21458709]

```

7. Now, we can observe the loss over the training iterations to be sure that the algorithm converged, as follows:

```

plt.plot(loss_vec, 'k-')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()

```

We get the following plot for the preceding code:

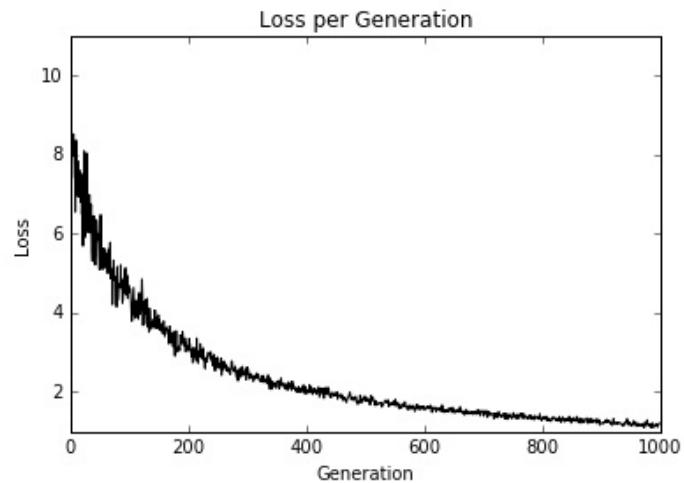


Figure 10: Elastic net regression loss plotted over the 1,000 training iterations

How it works...

Elastic net regression is implemented here as well as multiple linear regression. We can see that, with these regularization terms in the loss function, the convergence is slower than in prior recipes. Regularization is as simple as adding the appropriate terms in the loss functions.

Implementing logistic regression

For this recipe, we will implement logistic regression to predict the probability of low birth weight in our sample population.

Getting ready

Logistic regression is a way to turn linear regression into a binary classification. This is accomplished by transforming the linear output into a sigmoid function that scales the output between zero and one. The target is a zero or one, which indicates whether a data point is in one class or another. Since we are predicting a number between zero and one, the prediction is classified into class value 1 if the prediction is above a specified cutoff value, and class 0 otherwise. For the purpose of this example, we will specify that cutoff to be 0.5, which will make the classification as simple as rounding the output.

The data we will use for this example will be the low birth weight data obtained from the author's GitHub repository (https://github.com/nfmcclure/tensorflow_cookbook/raw/master/01_Introduction/07_Working_with_Data_Sources/birthweight_data/birthweight.dat). We will be predicting low birth weight from several other factors.

How to do it...

We proceed with the recipe as follows:

1. We start by loading the libraries, including the `request` library, because we will access the low birth weight data through a hyperlink. We also initiate a session:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import requests
from sklearn import datasets
from sklearn.preprocessing import normalize
from tensorflow.python.framework import ops
ops.reset_default_graph()
sess = tf.Session()
```

2. Next, we load the data through the `request` module and specify which features we want to use. We have to be specific because one feature is the actual birth weight and we don't want to use this to predict whether the birth weight is greater or less than a specific amount. We also do not want to use the ID column as a predictor either:

```
birth_weight_file = 'birth_weight.csv'
# Download data and create data file if file does not exist in current directory
if not os.path.exists(birth_weight_file):
    birthdata_url = 'https://github.com/nfmccclure/tensorflow_cookbook/raw/master'
    birth_file = requests.get(birthdata_url)
    birth_data = birth_file.text.split('\r\n')
    birth_header = birth_data[0].split('\t')
    birth_data = [[float(x) for x in y.split('\t') if len(x)>=1] for y in birth_
    with open(birth_weight_file, 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(birth_header)
        writer.writerows(birth_data)

# Read birth weight data into memory
birth_data = []
with open(birth_weight_file, newline='') as csvfile:
    csv_reader = csv.reader(csvfile)
    birth_header = next(csv_reader)
    for row in csv_reader:
        birth_data.append(row)
    birth_data = [[float(x) for x in row] for row in birth_data]
# Pull out target variable
y_vals = np.array([x[0] for x in birth_data])
# Pull out predictor variables (not id, not target, and not birthweight)
x_vals = np.array([x[1:8] for x in birth_data])
```

- First, we split the dataset into test and train sets:

```

train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]

```

- Logistic regression convergence works better when the features are scaled between 0 and 1 (min-max scaling). So, next we will scale each feature:

```

def normalize_cols(m, col_min=np.array([None]), col_max=np.array([None])):
    if not col_min[0]:
        col_min = m.min(axis=0)
    if not col_max[0]:
        col_max = m.max(axis=0)
    return (m-col_min) / (col_max - col_min), col_min, col_max

x_vals_train, train_min, train_max = np.nan_to_num(normalize_cols(x_vals_train))
x_vals_test = np.nan_to_num(normalize_cols(x_vals_test, train_min, train_max))

```



Note that we split the dataset into train and test before we scale the dataset. This is an important distinction to make. We want to make sure that the test set does not influence the training set at all. If we scale the whole set before splitting, then we cannot guarantee that they don't influence each other. We make sure to save the scaling from the train set to scale the test set.

- Next, we declare the batch size, placeholders, variables, and logistic model. We do not wrap the output in a sigmoid because that operation is built into the loss function. Also note that there are seven input features for every observation so our size of the `x_data` placeholder is `[None, 7]`.

```

batch_size = 25
x_data = tf.placeholder(shape=[None, 7], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
A = tf.Variable(tf.random_normal(shape=[7,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
model_output = tf.add(tf.matmul(x_data, A), b)

```

- Now, we declare our loss function, which has the sigmoid function, initialize our variables, and declare our optimizing function:

```

loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(model_output, y_ta
init = tf.global_variables_initializer()
sess.run(init)
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)

```

- Along with recording the loss function, we will also want to record the classification accuracy on the training and test set. So, we will create a

prediction function that returns the accuracy for any size of batch:

```
prediction = tf.round(tf.sigmoid(model_output))
predictions_correct = tf.cast(tf.equal(prediction, y_target), tf.float32)
accuracy = tf.reduce_mean(predictions_correct)
```

8. Now, we can start our training loop and record the loss and accuracy:

```
loss_vec = []
train_acc = []
test_acc = []
for i in range(1500):
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
    rand_x = x_vals_train[rand_index]
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss)
    temp_acc_train = sess.run(accuracy, feed_dict={x_data: x_vals_train, y_target: y_vals_train})
    train_acc.append(temp_acc_train)
    temp_acc_test = sess.run(accuracy, feed_dict={x_data: x_vals_test, y_target: y_vals_test})
    test_acc.append(temp_acc_test)
```

9. Here is the code to look at the plots of loss and accuracy:

```
plt.plot(loss_vec, 'k-')
plt.title('Cross' Entropy Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Cross' Entropy Loss')
plt.show()
plt.plot(train_acc, 'k-', label='Train Set Accuracy')
plt.plot(test_acc, 'r--', label='Test Set Accuracy')
plt.title('Train and Test Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()
```

How it works...

Here is the loss over the iterations and train and test accuracy. Since the dataset is only 189 observations, the train and test accuracy plots will change owing to the random splitting of the dataset. The first figure is the cross-entropy loss:

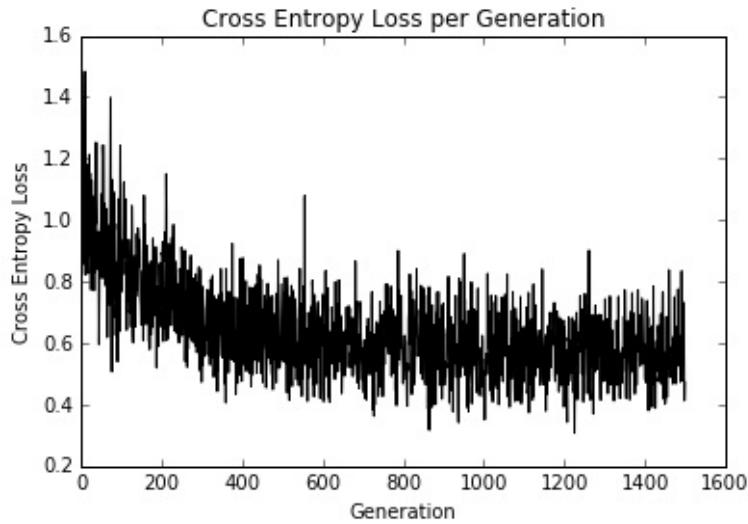


Figure 11: Cross-entropy loss plotted over the course of 1,500 iterations

The second figure shows us the accuracy of the train and test sets:

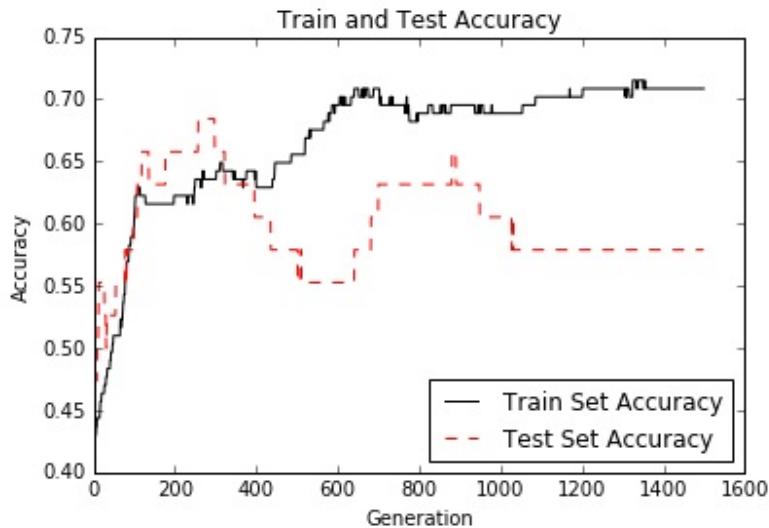


Figure 12: Test and train set accuracy plotted over 1,500 generations

Support Vector Machines

This chapter will cover some important recipes regarding how to use, implement, and evaluate **Support Vector Machines (SVM)** in TensorFlow. The following areas will be covered:

- Working with a linear SVM
- Reduction to linear regression
- Working with kernels in TensorFlow
- Implementing a non-linear SVM
- Implementing a multi-class SVM



Both the prior-covered logistic regression and most of the SVMs in this chapter are binary predictors. While logistic regression tries to find any separating line that maximizes the distance (probabilistically), SVMs also try to minimize the error while maximizing the margin between classes. In general, if the problem has a large number of features compared to training examples, try logistic regression or a linear SVM. If the number of training examples is larger, or the data is not linearly separable, a SVM with a Gaussian kernel may be used.

Also, remember that all the code for this chapter is available at https://github.com/nfmcclure/tensorflow_cookbook and at the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

Introduction

SVMs are a method of binary classification. The basic idea is to find a linear separating line in two dimensions (or hyperplane for more dimensions) between the two classes. We first assume that the binary class targets are -1 or 1, instead of the prior 0 or 1 targets. Since there may be many lines that separate two classes, we define the best linear separator that maximizes the distance between both classes:

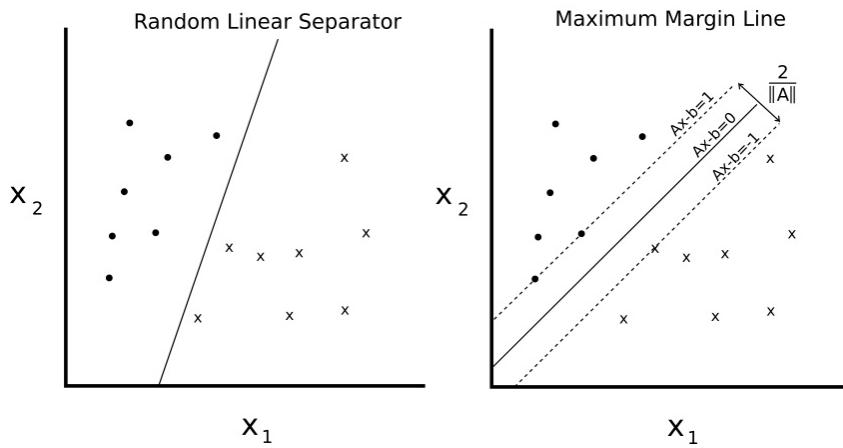


Figure 1

Given two separable classes, o and x , we wish to find the equation for the linear separator between the two. The left-hand graph shows that there are many lines that separate the two classes. The right-hand graph shows the unique maximum margin line. The margin width is given by $\frac{2}{\|A\|}$. This line is found by minimizing the L2 norm of A .

We can write such a hyperplane as follows:

$$Ax - b = 0$$

Here, A is a vector of our partial slopes and x is a vector of inputs. The width of the maximum margin can be shown to be 2 divided by the L2 norm of A . There are many proofs out there of this fact, but for a geometric idea, solving the perpendicular distance from a 2D point to a line may provide motivation for moving forward.

For linearly separable binary class data, to maximize the margin, we minimize the L2 norm of A , $\|A\|$. We must also subject this minimum to the following constraint:

$$y_i (Ax_i - b) \geq 1, \quad \forall i$$

The preceding constraint assures us that all the points from the corresponding classes are on the same side of the separating line.

Since not all datasets are linearly separable, we can introduce a loss function for points that cross the margin lines. For n data points, we introduce what is called the soft margin loss function, as follows:

$$\frac{1}{n} n \sum_{i=1}^n \max(0, 1 - y_i(Ax_i - b)) + \alpha \|A\|^2$$

Note that the product, $y_i(Ax_i - b)$, is always greater than 1 if the point is on the correct side of the margin. This makes the left-hand term of the loss function equal to 0, and the only influence on the loss function is the size of the margin.

The preceding loss function will seek a linearly separable line, but will allow for points crossing the margin line. This can be a hard or soft allowance, depending on the value of α . Larger values of α result in more emphasis on widening the margin, and smaller values of α result in the model acting more such like a hard margin, while allowing data points to cross the margin, if need be.

In this chapter, we will set up a soft margin SVM and show how to extend it to non-linear cases and multiple classes.

Working with a linear SVM

For this example, we will create a linear separator from the iris dataset. We know from prior chapters that the sepal length and petal width create a linear separable binary dataset for predicting whether a flower is I. setosa or not.

Getting ready

To implement a soft separable SVM in TensorFlow, we will implement the specific loss function, as follows:

$$\frac{1}{n} n \sum_{i=1}^n \max(0, 1 - y_i(Ax_i - b)) + \alpha \|A\|^2$$

Here, A is the vector of partial slopes, b is the intercept, x_i is a vector of inputs, y_i is the actual class, (-1 or 1), and α is the soft separability regularization parameter.

How to do it...

We proceed with the recipe as follows:

1. We start by loading the necessary libraries. This will include the `scikit-learn` dataset library for access to the iris dataset. Use the following code:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
```

 To set up scikit-learn for this exercise, we just need to type `$pip install -U scikit-learn`. Note that it also comes installed with Anaconda as well.

2. Next, we start a graph session and load the data as we need it. Remember that we are loading the first and fourth variable in the iris dataset as they are the sepal length and sepal width. We are loading the target variable, which will take on the value 1 for I. setosa and -1 otherwise. Use the following code:

```
sess = tf.Session()
iris = datasets.load_iris()
x_vals = np.array([[x[0], x[3]] for x in iris.data])
y_vals = np.array([1 if y==0 else -1 for y in iris.target])
```

3. We should now split the dataset into train and test sets. We will evaluate the accuracy on both the training and test sets. Since we know this dataset is linearly separable, we should expect to get 100% accuracy on both sets. To split up the data, use the following code:

```
train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```

4. Next, we set our batch size, placeholders, and model variables. It is important to mention that, with this SVM algorithm, we want very large batch sizes to help with convergence. We can imagine that, with very small batch sizes, the maximum margin line would jump around slightly. Ideally, we would also slowly decrease the learning rate as well, but this will suffice for now. Also, the `A` variable will take on the `2x1` shape because we have

two predictor variables: sepal length and petal width. To set this up, we use the following code:

```
batch_size = 100

x_data = tf.placeholder(shape=[None, 2], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)

A = tf.Variable(tf.random_normal(shape=[2,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
```

5. We now declare our model output. For correctly classified points, this will return numbers that are greater than or equal to 1 if the target is I. setosa and less than or equal to -1 otherwise. The model output uses the following code:

```
| model_output = tf.subtract(tf.matmul(x_data, A), b)
```

6. Next, we will put together and declare the necessary components for the maximum margin loss. First, we will declare a function that will calculate the L2 norm of a vector. Then, we add the margin parameter, α . We then declare our classification loss and add together the two terms. Use the following code:

```
l2_norm = tf.reduce_sum(tf.square(A))
alpha = tf.constant([0.1])
classification_term = tf.reduce_mean(tf.maximum(0., tf.subtract(1., tf.multiply(
    loss = tf.add(classification_term, tf.multiply(alpha, l2_norm)))
```

7. Now, we declare our prediction and accuracy functions so that we can evaluate the accuracy on both the training and test sets, as follows:

```
| prediction = tf.sign(model_output)
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction, y_target), tf.float32))
```

8. Here, we will declare our optimization function and initialize our model variables; we do this in the following code:

```
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)

init = tf.global_variables_initializer()
sess.run(init)
```

9. We can now start our training loop, keeping in mind that we want to record our loss and training accuracy on both the training and testing sets, as follows:

```

loss_vec = []
train_accuracy = []
test_accuracy = []
for i in range(500):
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
    rand_x = x_vals_train[rand_index]
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss)

    train_acc_temp = sess.run(accuracy, feed_dict={x_data: x_vals_train, y_target: y_vals_train})
    train_accuracy.append(train_acc_temp)

    test_acc_temp = sess.run(accuracy, feed_dict={x_data: x_vals_test, y_target: y_vals_test})
    test_accuracy.append(test_acc_temp)

    if (i+1)%100==0:
        print('Step #' + str(i+1) + ' A = ' + str(sess.run(A)) + ' b = ' + str(sess.run(b)))
        print('Loss = ' + str(temp_loss))

```

10. The output of the script during training should look like the following:

```

Step #100 A = [[-0.10763293]
[-0.65735245]] b = [[-0.68752676]]
Loss = [ 0.48756418]
Step #200 A = [[-0.0650763 ]
[-0.89443302]] b = [[-0.73912662]]
Loss = [ 0.38910741]
Step #300 A = [[-0.02090022]
[-1.12334013]] b = [[-0.79332656]]
Loss = [ 0.28621092]
Step #400 A = [[ 0.03189624]
[-1.34912157]] b = [[-0.8507266]]
Loss = [ 0.22397576]
Step #500 A = [[ 0.05958777]
[-1.55989814]] b = [[-0.9000265]]
Loss = [ 0.20492229]

```

11. In order to plot the outputs (fit, loss, and accuracy), we have to extract the coefficients and separate the x values into I. setosa and Non-setosa, as follows:

```

[[a1], [a2]] = sess.run(A)
[[b]] = sess.run(b)
slope = -a2/a1
y_intercept = b/a1

x1_vals = [d[1] for d in x_vals]

best_fit = []
for i in x1_vals:
    best_fit.append(slope*i+y_intercept)

setosa_x = [d[1] for i,d in enumerate(x_vals) if y_vals[i]==1]
setosa_y = [d[0] for i,d in enumerate(x_vals) if y_vals[i]==1]
not_setosa_x = [d[1] for i,d in enumerate(x_vals) if y_vals[i]==-1]
not_setosa_y = [d[0] for i,d in enumerate(x_vals) if y_vals[i]==-1]

```

12. The following is the code to plot the data with the linear separator fit, accuracies, and loss:

```
plt.plot(setosa_x, setosa_y, 'o', label='I. setosa')
plt.plot(not_setosa_x, not_setosa_y, 'x', label='Non-setosa')
plt.plot(x1_vals, best_fit, 'r-', label='Linear Separator', linewidth=3)
plt.ylim([0, 10])
plt.legend(loc='lower right')
plt.title('Sepal Length vs Petal Width')
plt.xlabel('Petal Width')
plt.ylabel('Sepal Length')
plt.show()

plt.plot(train_accuracy, 'k-', label='Training Accuracy')
plt.plot(test_accuracy, 'r--', label='Test Accuracy')
plt.title('Train and Test Set Accuracies')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

plt.plot(loss_vec, 'k-')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```

i Using TensorFlow in this manner to implement the SVD algorithm may result in slightly different outcomes each run. The reasons for this include the random train/test set splitting and the selection of different batches of points on each training batch. Also, it would be ideal to slowly lower the learning rate after each generation.

The resulting plots are as follows:

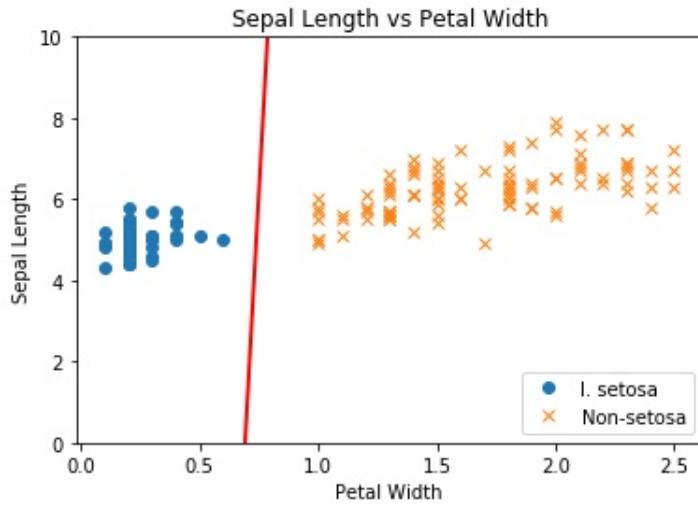


Figure 2: Final linear SVM fit with the two classes plotted

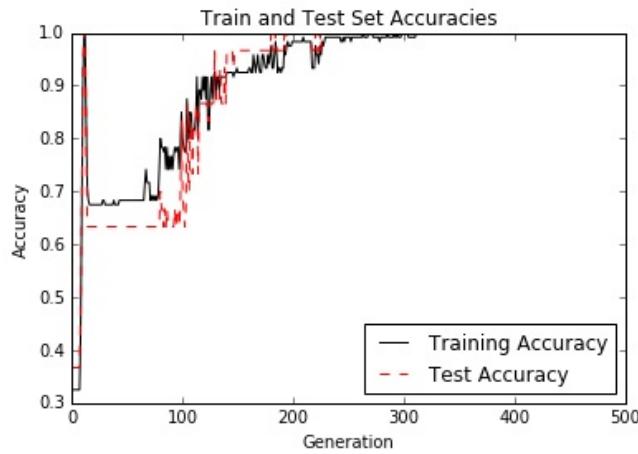


Figure 3: Test and train set accuracy over iterations; we do get 100% accuracy because the two classes are linearly separable

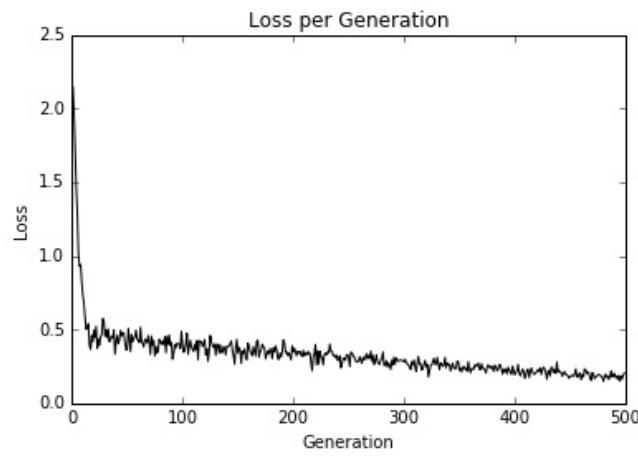


Figure 4: Plot of the maximum margin loss over 500 iterations

How it works...

In this recipe, we have shown that implementing a linear SVD model is possible with using the maximum margin loss function.

Reduction to linear regression

SVM can be used to fit linear regression. In this section, we will explore how to do this with TensorFlow.

Getting ready

The same *maximum margin* concept can be applied toward fitting linear regression. Instead of maximizing the margin that separates the classes, we can think about maximizing the margin that contains the most (x, y) points. To illustrate this, we will use the same iris dataset, and show that we can use this concept to fit a line between sepal length and petal width.

The corresponding loss function will be similar to $\max(0, |y_i - (Ax_i + b)| - \epsilon)$. Here, ϵ is half of the width of the margin, which makes the loss equal to 0 if a point lies in this region.

How to do it...

We proceed with the recipe as follows:

1. First, we load the necessary libraries, start a graph, and load the iris dataset. After that, we will split the dataset into train and test sets to visualize the loss on both. Use the following code:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
sess = tf.Session()
iris = datasets.load_iris()
x_vals = np.array([x[3] for x in iris.data])
y_vals = np.array([y[0] for y in iris.data])
train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```

 **TIP** For this example, we have split the data into train and test sets. It is also common to split the data into three datasets, which includes the validation set. We can use this validation set to verify that we are not overfitting models as we train them.

2. Let's declare our batch size, placeholders, and variables, and create our linear model, as follows:

```
batch_size = 50

x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)

A = tf.Variable(tf.random_normal(shape=[1,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))

model_output = tf.add(tf.matmul(x_data, A), b)
```

3. Now, we declare our loss function. The loss function, as described in the preceding text, is implemented to follow with $\epsilon = 0.5$. Remember that epsilon is part of our loss function, which allows for a soft margin instead of a hard margin:

```
epsilon = tf.constant([0.5])
loss = tf.reduce_mean(tf.maximum(0., tf.subtract(tf.abs(tf.subtract(model_output
```

4. We create an optimizer and initialize our variables next, as follows:

```

my_opt = tf.train.GradientDescentOptimizer(0.075)
train_step = my_opt.minimize(loss)

init = tf.global_variables_initializer()
sess.run(init)

```

- Now, we iterate through 200 training iterations and save the training and test loss for plotting later:

```

train_loss = []
test_loss = []
for i in range(200):
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
    rand_x = np.transpose([x_vals_train[rand_index]])
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})

    temp_train_loss = sess.run(loss, feed_dict={x_data: np.transpose([x_vals_train])})
    train_loss.append(temp_train_loss)

    temp_test_loss = sess.run(loss, feed_dict={x_data: np.transpose([x_vals_test])})
    test_loss.append(temp_test_loss)
    if (i+1)%50==0:
        print('-----')
        print('Generation: ' + str(i))
        print('A = ' + str(sess.run(A)) + ' b = ' + str(sess.run(b)))
        print('Train Loss = ' + str(temp_train_loss))
        print('Test Loss = ' + str(temp_test_loss))

```

- This results in the following output:

```

Generation: 50
A = [[ 2.20651722]] b = [[ 2.71290684]]
Train Loss = 0.609453
Test Loss = 0.460152
-----
Generation: 100
A = [[ 1.6440177]] b = [[ 3.75240564]]
Train Loss = 0.242519
Test Loss = 0.208901
-----
Generation: 150
A = [[ 1.27711761]] b = [[ 4.3149066]]
Train Loss = 0.108192
Test Loss = 0.119284
-----
Generation: 200
A = [[ 1.05271816]] b = [[ 4.53690529]]
Train Loss = 0.0799957
Test Loss = 0.107551

```

- We can now extract the coefficients we found, and get values for the best-fit line. For plotting purposes, we will also get values for the margins as well. Use the following code:

```

[[slope]] = sess.run(A)
[[y_intercept]] = sess.run(b)

```

```

[width] = sess.run(epsilon)

best_fit = []
best_fit_upper = []
best_fit_lower = []
for i in x_vals:
    best_fit.append(slope*i+y_intercept)
    best_fit_upper.append(slope*i+y_intercept+width)
    best_fit_lower.append(slope*i+y_intercept-width)

```

8. Finally, here is the code to plot the data with the fitted line and the train-test loss:

```

plt.plot(x_vals, y_vals, 'o', label='Data Points')
plt.plot(x_vals, best_fit, 'r-', label='SVM Regression Line', linewidth=3)
plt.plot(x_vals, best_fit_upper, 'r--', linewidth=2)
plt.plot(x_vals, best_fit_lower, 'r--', linewidth=2)
plt.ylim([0, 10])
plt.legend(loc='lower right')
plt.title('Sepal Length vs Petal Width')
plt.xlabel('Petal Width')
plt.ylabel('Sepal Length')
plt.show()
plt.plot(train_loss, 'k-', label='Train Set Loss')
plt.plot(test_loss, 'r--', label='Test Set Loss')
plt.title('L2 Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('L2 Loss')
plt.legend(loc='upper right')
plt.show()

```

The plot from the preceding code is as follows:

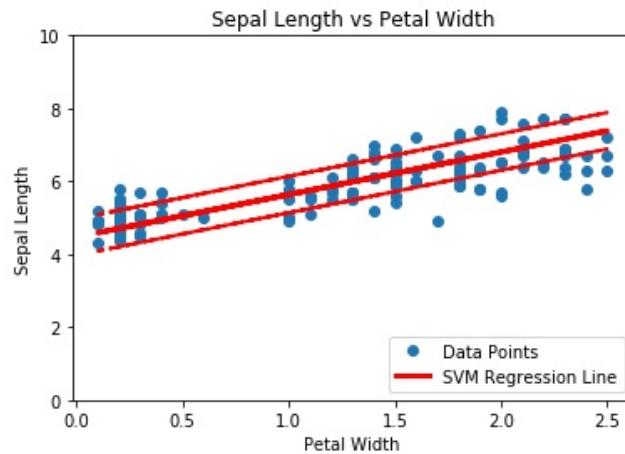


Figure 5: SVM regression with a 0.5 margin on iris data (sepal length versus petal width)

Here is the train and test loss over the training iterations:

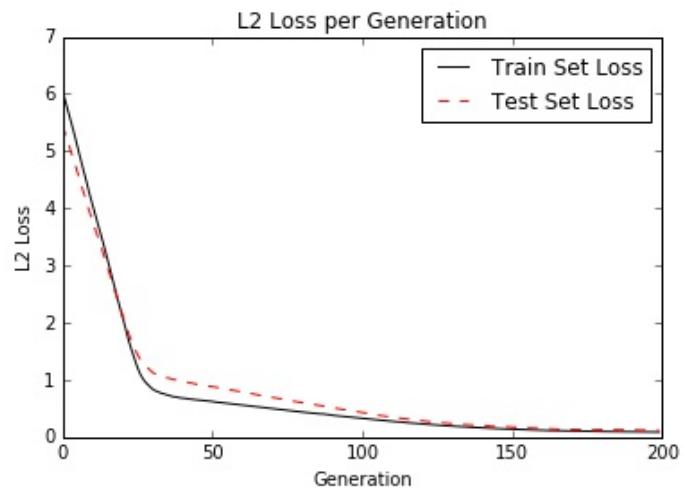


Figure 6: SVM regression loss per generation on both the train and test sets

How it works...

Intuitively, we can think of SVM regression as a function that is trying to fit as many points in the 2ϵ width margin from the line as possible. The fitting of this line is somewhat sensitive to this parameter. If we choose too small an epsilon, the algorithm will not be able to fit many points in the margin. If we choose too large an epsilon, there will be many lines that are able to fit all the data points in the margin. We prefer a smaller epsilon, since nearer points to the margin contribute less loss than further-away points.

Working with kernels in TensorFlow

The prior SVMs worked with linear separable data. If we separate non-linear data, we can change how we project the linear separator onto the data. This is done by changing the kernel in the SVM loss function. In this chapter, we introduce how to change kernels and separate non-linear separable data.

Getting ready

In this recipe, we will motivate the usage of kernels in Support Vector Machines. In the linear SVM section, we solved the soft margin with a specific loss function. A different approach to this method is to solve what is called the dual of the optimization problem. It can be shown that the dual for the linear SVM problem is given by the following formula:

$$\max \sum_{i=1}^n b_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n y_i b_i (x_i \cdot x_j) y_j b_j$$

To this, the following applies:

$$\sum_{i=1}^n b_i y_i = 0 \wedge 0 \leq b_i \leq \frac{1}{2n\Upsilon}$$

Here, the variable in the model will be the b vector. Ideally, this vector will be quite sparse, only taking on values near 1 and -1 for the corresponding *support vectors* of our dataset. Our data point vectors are indicated by x_i and our targets (1 or -1) are represented by y_i .

The kernel in the preceding equations is the dot product, $x_i \cdot x_j$, which gives us the linear kernel. This kernel is a square matrix filled with the i, j dot products of the data points.

Instead of just doing the dot product between data points, we can expand them with more complicated functions into higher dimensions, in which the classes may be linear separable. This may seem needlessly complicated, but we may, for instance, select a function, k , that has the following property:

$$k(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

Here, k is called a **kernel** function. A more common kernel is the Gaussian

kernel (also known as the radial basis function kernel or the RBF kernel) is used. This kernel is described with the following equation:

$$k(x_i, x_j) = e^{-\gamma} \|x_i - x_j\|^2$$

In order to make predictions on this kernel, say at a point p_i , we just substitute in the prediction point at the appropriate equation in the kernel, as follows:

$$k(x_i, p_j) = e^{-\gamma} \|x_i - p_j\|^2$$

In this section, we will discuss how to implement the Gaussian kernel. We will also make a note where to make the substitution for implementing the linear kernel, where appropriate. The dataset we will use will be manually created to show where the Gaussian kernel would be more appropriate to use over the linear kernel.

How to do it...

We proceed with the recipe as follows:

1. First, we load the necessary libraries and start a graph session, as follows:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
sess = tf.Session()
```

2. Now, we generate the data. The data we will generate will be two concentric rings of data; each ring will belong to a different class. We have to make sure that the classes are -1 or 1 only. Then we will split the data into x and y values for each class for plotting purposes. To do this, use the following code:

```
(x_vals, y_vals) = datasets.make_circles(n_samples=500, factor=.5, noise=.1)
y_vals = np.array([1 if y==1 else -1 for y in y_vals])
class1_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==1]
class1_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==1]
class2_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==-1]
class2_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==-1]
```

3. Next, we declare our batch size, and placeholders, and create our model variable, b . For SVMs we tend to want larger batch sizes because we want a very stable model that won't fluctuate much with each training generation. Also, note that we have an extra placeholder for the prediction points. To visualize the results, we will create a color grid to see which areas belong to which class at the end. We do this as follows:

```
batch_size = 250
x_data = tf.placeholder(shape=[None, 2], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
prediction_grid = tf.placeholder(shape=[None, 2], dtype=tf.float32)
b = tf.Variable(tf.random_normal(shape=[1,batch_size]))
```

4. We will now create the Gaussian kernel. This kernel can be expressed as matrix operations, as follows:

```
gamma = tf.constant(-50.0)
dist = tf.reduce_sum(tf.square(x_data), 1)
dist = tf.reshape(dist, [-1,1])
sq_dists = tf.add(tf.subtract(dist, tf.multiply(2., tf.matmul(x_data, tf.transpo
```

| my_kernel = tf.exp(tf.multiply(gamma, tf.abs(sq_dists)))
 | Note the usage of broadcasting in the `sq_dists` line of the `add` and `subtract` operations.
 | Also, note that the linear kernel can be expressed as `my_kernel = tf.matmul(x_data, tf.transpose(x_data))`.

- Now, we declare the dual problem as previously stated in this recipe. At the end, instead of maximizing, we will be minimizing the negative of the loss function with a `tf.negative()` function. We accomplish this with the following code:

```
model_output = tf.matmul(b, my_kernel)
first_term = tf.reduce_sum(b)
b_vec_cross = tf.matmul(tf.transpose(b), b)
y_target_cross = tf.matmul(y_target, tf.transpose(y_target))
second_term = tf.reduce_sum(tf.multiply(my_kernel, tf.multiply(b_vec_cross, y_target)))
loss = tf.negative(tf.subtract(first_term, second_term))
```

- We now create the prediction and accuracy functions. First, we must create a prediction kernel, similar to step 4, but instead of a kernel of the points with itself, we have the kernel of the points with the prediction data. The prediction is then the sign of the output of the model. This is implemented as follows:

```
rA = tf.reshape(tf.reduce_sum(tf.square(x_data), 1), [-1,1])
rB = tf.reshape(tf.reduce_sum(tf.square(prediction_grid), 1), [-1,1])
pred_sq_dist = tf.add(tf.subtract(rA, tf.multiply(2., tf.matmul(x_data, tf.transpose(prediction_grid)))), pred_kernel = tf.exp(tf.multiply(gamma, tf.abs(pred_sq_dist))))
```

| prediction_output = tf.matmul(tf.transpose(y_target), b), pred_kernel
| prediction = tf.sign(prediction_output - tf.reduce_mean(prediction_output))
| accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.squeeze(prediction), tf.squeeze(y_target)), tf.float32))

| To implement the linear prediction kernel, we can write `pred_kernel = tf.matmul(x_data, tf.transpose(prediction_grid))`.

- Now, we can create an optimization function and initialize all the variables, as follows:

```
my_opt = tf.train.GradientDescentOptimizer(0.001)
train_step = my_opt.minimize(loss)
init = tf.global_variables_initializer()
sess.run(init)
```

- Next, we start the training loop. We will record the loss vector and the batch accuracy for each generation. When we run the accuracy, we have to put in all three placeholders, but we feed in the `x` data twice to get the prediction on the points, as follows:

```
loss_vec = []
batch_accuracy = []
```

```

for i in range(500):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = x_vals[rand_index]
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss)

    acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x,
                                             y_target: rand_y,
                                             prediction_grid:rand_x})
    batch_accuracy.append(acc_temp)

    if (i+1)%100==0:
        print('Step #' + str(i+1))
        print('Loss = ' + str(temp_loss))

```

9. This results in the following output:

```

Step #100
Loss = -28.0772
Step #200
Loss = -3.3628
Step #300
Loss = -58.862
Step #400
Loss = -75.1121
Step #500
Loss = -84.8905

```

10. In order to see the output class on the whole space, we will create a mesh of prediction points in our system and run the prediction on all of them, as follows:

```

x_min, x_max = x_vals[:, 0].min() - 1, x_vals[:, 0].max() + 1
y_min, y_max = x_vals[:, 1].min() - 1, x_vals[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                      np.arange(y_min, y_max, 0.02))
grid_points = np.c_[xx.ravel(), yy.ravel()]
[grid_predictions] = sess.run(prediction, feed_dict={x_data: x_vals,
                                                    y_target: np.transpose([y_val]),
                                                    prediction_grid: grid_points})
grid_predictions = grid_predictions.reshape(xx.shape)

```

11. The following is the code to plot the result, batch accuracy, and loss:

```

plt.contourf(xx, yy, grid_predictions, cmap=plt.cm.Paired, alpha=0.8)
plt.plot(class1_x, class1_y, 'ro', label='Class 1')
plt.plot(class2_x, class2_y, 'kx', label='Class -1')
plt.legend(loc='lower right')
plt.ylim([-1.5, 1.5])
plt.xlim([-1.5, 1.5])
plt.show()

plt.plot(batch_accuracy, 'k-', label='Accuracy')
plt.title('Batch Accuracy')
plt.xlabel('Generation')

```

```

plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

plt.plot(loss_vec, 'k-')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()

```

For succinctness, we will show only the results graph, but we can also separately run the plotting code and view the loss and accuracy as well.

The following screenshot illustrates how bad a linear separable fit is on our non-linear data:

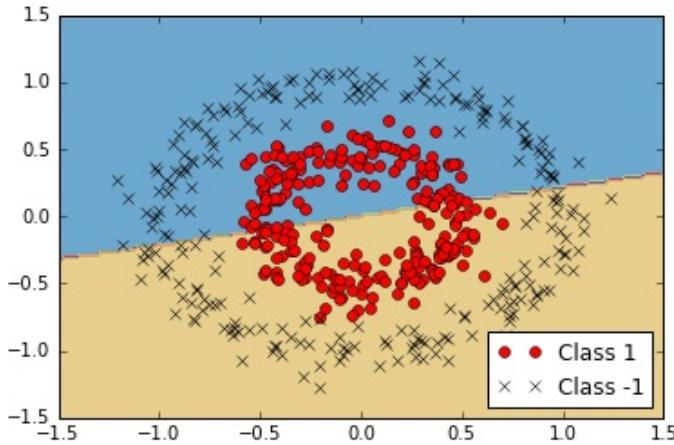


Figure 7: Linear SVM on non-linear separable data

The following screenshot shows us how much better a Gaussian kernel can fit the non-linear data:

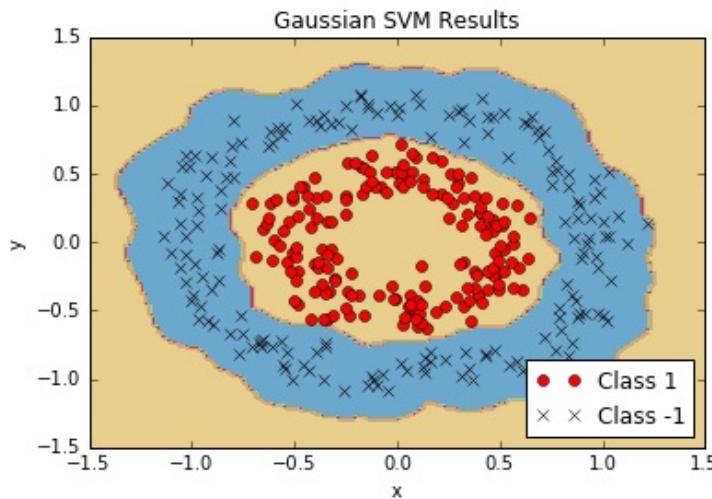


Figure 8: Non-linear SVM with Gaussian kernel results on non-linear ring data

If we use a Gaussian kernel to separate our non-linear ring data, we get a much better fit.

How it works...

There are two important pieces of the code to know about: how we implemented the kernel, and how we implemented the loss function for the SVM dual optimization problem. We have shown how to implement the linear and Gaussian kernel, and that the Gaussian kernel can separate non-linear datasets.

We should also mention that there is another parameter, the gamma value in the Gaussian kernel. This parameter controls how much influence points have on the curvature of the separation. Small values are commonly chosen, but it depends heavily on the dataset. Ideally, this parameter is chosen with statistical techniques such as cross validation.



For prediction/evaluation on new points, we use the following command: `sess.run(prediction, feed_dict:{x_data: x_vals, y_data: np.transpose([y_vals])})`. This evaluation must include the original dataset (`x_vals` and `y_vals`) because SVMs are defined with the support vectors, given by which points are designated to be on the margin or not.

There's more...

There are many more kernels that we could implement if we so choose. Here is a list of a few more common non-linear kernels:

- Polynomial homogeneous kernel:

$$k(x_i, x_j) = (x_i \cdot x_j)^d$$

- Polynomial inhomogeneous kernel:

$$k(x_i, x_j) = (x_i \cdot x_j + 1)^d$$

- Hyperbolic tangent kernel:

$$k(x_i, x_j) = \tanh(ax_i \cdot x_j + k)$$

Implementing a non-linear SVM

For this recipe, we will apply a non-linear kernel to split a dataset.

Getting ready

In this section, we will implement the preceding Gaussian kernel SVM on real data. We will load the iris dataset and create a classifier for I. setosa (versus Non-setosa). We will see the effect of various gamma values on the classification.

How to do it...

We proceed with the recipe as follows:

1. We first load the necessary libraries, which includes the `scikit-learn` datasets so that we can load the iris data. Then, we will start a graph session. Use the following code:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
sess = tf.Session()
```

2. Next, we will load the iris data, extract the sepal length and petal width, and separate the x and y values for each class (for plotting purposes later), as follows:

```
iris = datasets.load_iris()
x_vals = np.array([[x[0], x[3]] for x in iris.data])
y_vals = np.array([1 if y==0 else -1 for y in iris.target])
class1_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==1]
class1_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==1]
class2_x = [x[0] for i,x in enumerate(x_vals) if y_vals[i]==-1]
class2_y = [x[1] for i,x in enumerate(x_vals) if y_vals[i]==-1]
```

3. Now, we declare our batch size (larger batches preferred), placeholders, and the model variable, b , as follows:

```
batch_size = 100

x_data = tf.placeholder(shape=[None, 2], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
prediction_grid = tf.placeholder(shape=[None, 2], dtype=tf.float32)

b = tf.Variable(tf.random_normal(shape=[1,batch_size]))
```

4. Next, we declare our Gaussian kernel. This kernel is dependent on the gamma value, and we will illustrate the effects of various gamma values on the classification later in this recipe. Use the following code:

```
gamma = tf.constant(-10.0)
dist = tf.reduce_sum(tf.square(x_data), 1)
dist = tf.reshape(dist, [-1,1])
sq_dists = tf.add(tf.subtract(dist, tf.multiply(2., tf.matmul(x_data, tf.transpose(x_data)))), tf.abs(sq_dists)))
my_kernel = tf.exp(tf.multiply(gamma, sq_dists))

# We now compute the loss for the dual optimization problem, as follows:
```

```

model_output = tf.matmul(b, my_kernel)
first_term = tf.reduce_sum(b)
b_vec_cross = tf.matmul(tf.transpose(b), b)
y_target_cross = tf.matmul(y_target, tf.transpose(y_target))
second_term = tf.reduce_sum(tf.multiply(my_kernel, tf.multiply(b_vec_cross, y_target)))
loss = tf.negative(tf.subtract(first_term, second_term))

```

- In order to perform predictions using an SVM, we must create a prediction kernel function. After that, we also declare an accuracy calculation, which will just be a percentage of points correctly classified using the following code:

```

rA = tf.reshape(tf.reduce_sum(tf.square(x_data), 1), [-1, 1])
rB = tf.reshape(tf.reduce_sum(tf.square(prediction_grid), 1), [-1, 1])
pred_sq_dist = tf.add(tf.subtract(rA, tf.mul(2., tf.matmul(x_data, tf.transpose(pred_kernel))), pred_kernel = tf.exp(tf.multiply(gamma, tf.abs(pred_sq_dist)))))

prediction_output = tf.matmul(tf.transpose(y_target), b), pred_kernel
prediction = tf.sign(prediction_output - tf.reduce_mean(prediction_output))
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.squeeze(prediction), tf.squeeze(y_
```

- Next, we declare our optimization function and initialize the variables, as follows:

```

my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)
init = tf.initialize_all_variables()
sess.run(init)

```

- Now, we can start the training loop. We run the loop for 300 iterations and will store the loss value and the batch accuracy. To do this, we use the following implementation:

```

loss_vec = []
batch_accuracy = []
for i in range(300):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = x_vals[rand_index]
    rand_y = np.transpose([y_vals[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss)

    acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x,
                                             y_target: rand_y,
                                             prediction_grid: rand_x})
    batch_accuracy.append(acc_temp)

```

- In order to plot the decision boundary, we will create a mesh of x, y points and evaluate the prediction function we created on all of these points, as follows:

```

x_min, x_max = x_vals[:, 0].min() - 1, x_vals[:, 0].max() + 1
y_min, y_max = x_vals[:, 1].min() - 1, x_vals[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                      np.arange(y_min, y_max, 0.02))
grid_points = np.c_[xx.ravel(), yy.ravel()]
[grid_predictions] = sess.run(prediction, feed_dict={x_data: x_vals,
                                                    y_target: np.transpose([y_val
                                                                prediction_grid: grid_points]}
grid_predictions = grid_predictions.reshape(xx.shape)

```

- For succinctness, we will only show how to plot the points with the decision boundaries. For the plot and effect of gamma, see the next section in this recipe. Use the following code:

```

plt.contourf(xx, yy, grid_predictions, cmap=plt.cm.Paired, alpha=0.8)
plt.plot(class1_x, class1_y, 'ro', label='I. setosa')
plt.plot(class2_x, class2_y, 'kx', label='Non-setosa')
plt.title('Gaussian SVM Results on Iris Data')
plt.xlabel('Petal Length')
plt.ylabel('Sepal Width')
plt.legend(loc='lower right')
plt.ylim([-0.5, 3.0])
plt.xlim([3.5, 8.5])
plt.show()

```

How it works...

Here is the classification of I. setosa results for four different gamma values (1, 10, 25, and 100). Note how the higher the gamma value, the more of an effect each individual point has on the classification boundary:

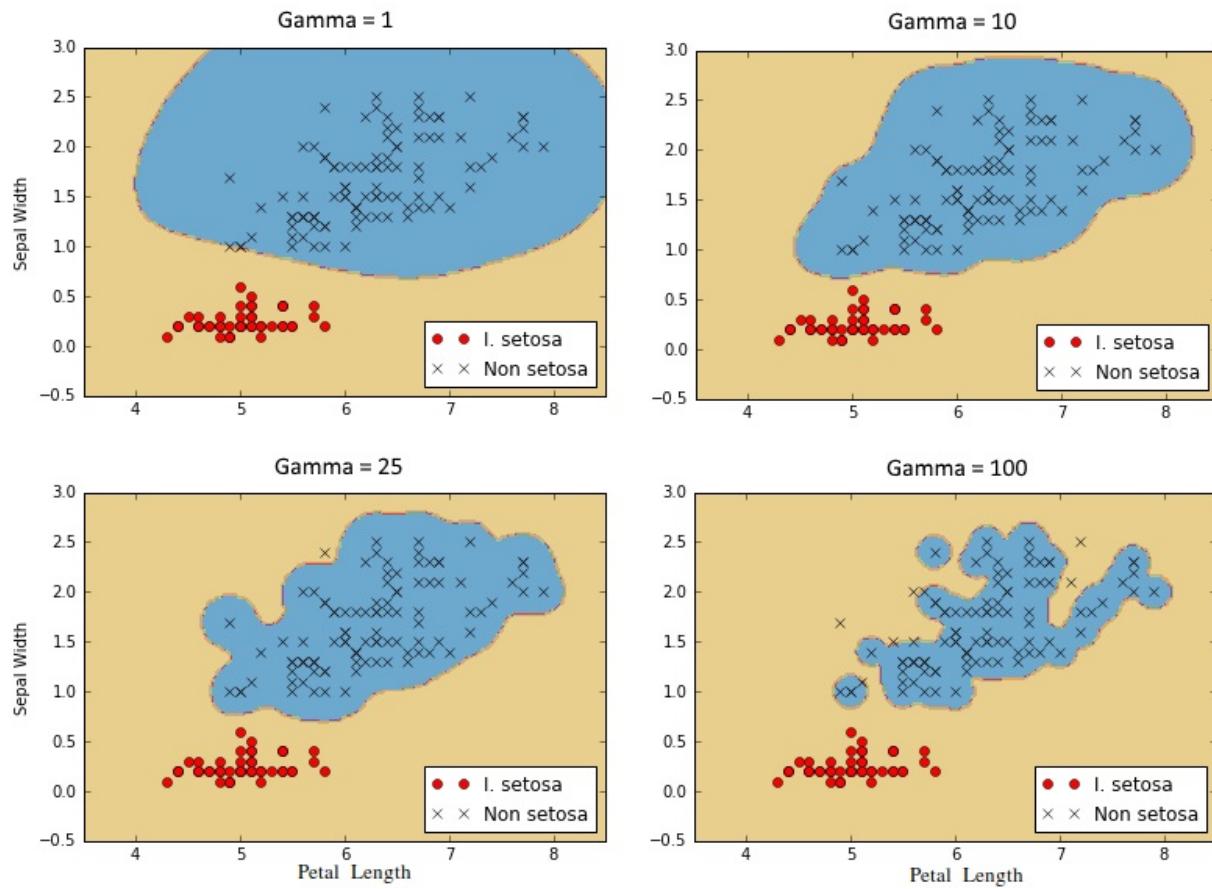


Figure 9: Classification results of I. setosa using a Gaussian kernel SVM with four different values of gamma

Implementing a multi-class SVM

We can also use SVMs to categorize multiple classes instead of just two. In this recipe, we will use a multi-class SVM to categorize the three types of flowers in the iris dataset.

Getting ready

By design, SVM algorithms are binary classifiers. However, there are a few strategies employed to get them to work on multiple classes. The two main strategies are called One versus all, and One versus one.

One versus one is a strategy where a binary classifier is created for each possible pair of classes. Then, a prediction is made for a point for the class that has the most votes. This can be computationally hard, as we must create $k!/(k - 2)!2!$ classifiers for k classes.

Another way to implement multi-class classifiers is to do a one versus all strategy where we create a classifier for each of the k classes. The predicted class of a point will be the class that creates the largest SVM margin. This is the strategy we will implement in this section.

Here, we will load the iris dataset and perform a multi-class non-linear SVM with a Gaussian kernel. The iris dataset is ideal because there are three classes (I. setosa, I. virginica, and I. versicolor). We will create three Gaussian kernel SVMs for each class and make the prediction of points where the highest margin exists.

How to do it...

We proceed with the recipe as follows:

1. First, we load the libraries we need and start a graph, as follows:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
sess = tf.Session()
```

2. Next, we will load the iris dataset and split apart the targets for each class. We will only be using sepal length and petal width to illustrate because we want to be able to plot the outputs. We also separate the x and y values for each class for plotting purposes at the end. Use the following code:

```
iris = datasets.load_iris()
x_vals = np.array([[x[0], x[3]] for x in iris.data])
y_vals1 = np.array([1 if y==0 else -1 for y in iris.target])
y_vals2 = np.array([1 if y==1 else -1 for y in iris.target])
y_vals3 = np.array([1 if y==2 else -1 for y in iris.target])
y_vals = np.array([y_vals1, y_vals2, y_vals3])
class1_x = [x[0] for i,x in enumerate(x_vals) if iris.target[i]==0]
class1_y = [x[1] for i,x in enumerate(x_vals) if iris.target[i]==0]
class2_x = [x[0] for i,x in enumerate(x_vals) if iris.target[i]==1]
class2_y = [x[1] for i,x in enumerate(x_vals) if iris.target[i]==1]
class3_x = [x[0] for i,x in enumerate(x_vals) if iris.target[i]==2]
class3_y = [x[1] for i,x in enumerate(x_vals) if iris.target[i]==2]
```

3. The biggest change we have in this example, as compared to the *Implementing a Non-Linear SVM* recipe, is that a lot of the dimensions will change (we have three classifiers now instead of one). We will also make use of matrix broadcasting and reshaping techniques to calculate all three SVMs at once. Since we are doing this all at once, our y_{target} placeholder now has the dimensions of $[3, \text{None}]$, and our model variable, b , will be initialized to be of size $[3, \text{batch_size}]$. Use the following code:

```
batch_size = 50

x_data = tf.placeholder(shape=[None, 2], dtype=tf.float32)
y_target = tf.placeholder(shape=[3, None], dtype=tf.float32)
prediction_grid = tf.placeholder(shape=[None, 2], dtype=tf.float32)

b = tf.Variable(tf.random_normal(shape=[3,batch_size]))
```

4. Next, we calculate the Gaussian kernel. Since this is only dependent on the input x -data, this code doesn't change from the prior recipe. Use the following code:

```
gamma = tf.constant(-10.0)
dist = tf.reduce_sum(tf.square(x_data), 1)
dist = tf.reshape(dist, [-1,1])
sq_dists = tf.add(tf.subtract(dist, tf.multiply(2., tf.matmul(x_data, tf.transpose(x_data)))), tf.abs(sq_dists)))
```

5. One big change is that we will do batch matrix multiplication. We will end up with three-dimensional matrices and we will want to broadcast matrix multiplication across the third index. Our data and target matrices are not set up for this. In order for an operation such as $\mathbf{x}^T \cdot \mathbf{x}$ to work across an extra dimension, we create a function to expand such matrices, reshape the matrix into a transpose, and then call TensorFlow's `batch_matmul` across the extra dimension. Use the following code:

```
def reshape_matmul(mat):
    v1 = tf.expand_dims(mat, 1)
    v2 = tf.reshape(v1, [3, batch_size, 1])
    return tf.batch_matmul(v2, v1)
```

6. With this function created, we can now compute the dual loss function, as follows:

```
model_output = tf.matmul(b, my_kernel)
first_term = tf.reduce_sum(b)
b_vec_cross = tf.matmul(tf.transpose(b), b)
y_target_cross = reshape_matmul(y_target)

second_term = tf.reduce_sum(tf.multiply(my_kernel, tf.multiply(b_vec_cross, y_target)))
loss = tf.reduce_sum(tf.negative(tf.subtract(first_term, second_term)))
```

7. Now, we can create the prediction kernel. Note that we have to be careful with the `reduce_sum` function and not reduce across all three SVM predictions, so we have to tell TensorFlow not to sum everything up with a second index argument. Use the following code:

```
rA = tf.reshape(tf.reduce_sum(tf.square(x_data), 1), [-1,1])
rB = tf.reshape(tf.reduce_sum(tf.square(prediction_grid), 1), [-1,1])
pred_sq_dist = tf.add(tf.subtract(rA, tf.multiply(2., tf.matmul(x_data, tf.transpose(prediction_grid)))), tf.abs(pred_sq_dist)))
pred_kernel = tf.exp(tf.multiply(gamma, tf.abs(pred_sq_dist)))
```

8. When we are done with the prediction kernel, we can create predictions. A big change here is that the predictions are not the `sign()` of the output. Since we are implementing a one versus all strategy, the prediction is the classifier that has the largest output. To accomplish this, we use TensorFlow's built-in

`argmax()` function, as follows:

```
prediction_output = tf.matmul(tf.mul(y_target, b), pred_kernel)
prediction = tf.argmax(prediction_output - tf.expand_dims(tf.reduce_mean(prediction, axis=1), 1))
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction, tf.argmax(y_target, 1)), tf.float32))
```

9. Now that we have the kernel, loss and prediction capabilities up, we just have to declare our optimization function and initialize our variables, as follows:

```
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)
init = tf.global_variables_initializer()
sess.run(init)
```

10. This algorithm converges relatively quickly, so we won't have to run the training loop for more than 100 iterations. We do so with the following code:

```
loss_vec = []
batch_accuracy = []
for i in range(100):
    rand_index = np.random.choice(len(x_vals), size=batch_size)
    rand_x = x_vals[rand_index]
    rand_y = y_vals[:,rand_index]
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})

    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss)

    acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x, y_target: rand_y})
    batch_accuracy.append(acc_temp)

    if (i+1)%25==0:
        print('Step #' + str(i+1))
        print('Loss = ' + str(temp_loss))

Step #25
Loss = -2.8951
Step #50
Loss = -27.9612
Step #75
Loss = -26.896
Step #100
Loss = -30.2325
```

11. We can now create the prediction grid of points and run the prediction function on all of them, as follows:

```
x_min, x_max = x_vals[:, 0].min() - 1, x_vals[:, 0].max() + 1
y_min, y_max = x_vals[:, 1].min() - 1, x_vals[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                      np.arange(y_min, y_max, 0.02))
grid_points = np.c_[xx.ravel(), yy.ravel()]
grid_predictions = sess.run(prediction, feed_dict={x_data: rand_x,
```

```

y_target: rand_y,
prediction_grid: grid_points}
grid_predictions = grid_predictions.reshape(xx.shape)

```

12. The following is the code to plot the results, batch accuracy, and loss function. For succinctness, we will only display the end result:

```

plt.contourf(xx, yy, grid_predictions, cmap=plt.cm.Paired, alpha=0.8)
plt.plot(class1_x, class1_y, 'ro', label='I. setosa')
plt.plot(class2_x, class2_y, 'kx', label='I. versicolor')
plt.plot(class3_x, class3_y, 'gv', label='I. virginica')
plt.title('Gaussian SVM Results on Iris Data')
plt.xlabel('Petal Length')
plt.ylabel('Sepal Width')
plt.legend(loc='lower right')
plt.ylim([-0.5, 3.0])
plt.xlim([3.5, 8.5])
plt.show()

plt.plot(batch_accuracy, 'k-', label='Accuracy')
plt.title('Batch Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

plt.plot(loss_vec, 'k-')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()

```

We then get the following plot:

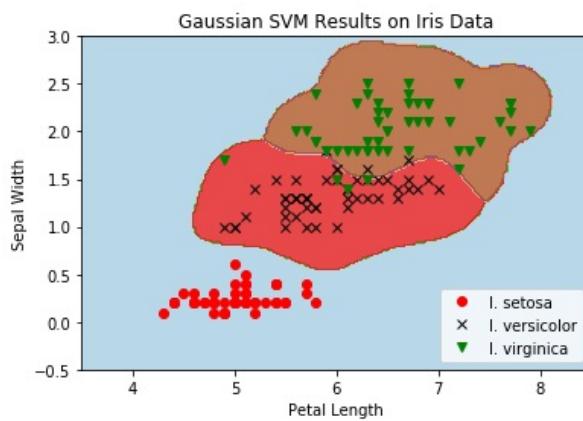


Figure 10: Multi-class (three classes) non-linear Gaussian SVM results on the iris dataset with gamma = 10

We observe preceding screenshot which shows all three iris classes, and a grid of points classified for each class.

How it works...

The important point to note in this recipe is how we changed our algorithm to optimize over three SVM models at once. Our model parameter, b , has an extra dimension to take into account all three models. Here, we can see that the extension of an algorithm to multiple similar algorithms was made relatively easy owing to TensorFlow's built-in capabilities to deal with extra dimensions.

The next chapter will cover nearest neighbor methods, which are a very powerful algorithm used for prediction purposes.

Nearest-Neighbor Methods

This chapter will focus on nearest-neighbor methods, and how to implement them in TensorFlow. We will start with an introduction to the methods, and we will illustrate how to implement various forms. The chapter will end with examples of address matching and image recognition.

In this chapter, we will cover the following:

- Working with nearest-neighbors
- Working with text based distances
- Computing mixed distance functions
- Using an address matching example
- Using nearest-neighbors for image recognition



Note that all of the latest code is available at: https://github.com/nfmccclure/tensorflow_cookbook and at the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

Introduction

Nearest-neighbor methods are rooted in a distance-based conceptual idea. We consider our training set a model, and make predictions on new points based on how close they are to points in the training set. A naive method is to make the prediction class the same as the closest training data point class. But since most datasets contain a degree of noise, a more common method is to take a weighted average of a set of k -nearest-neighbors. This method is called **k-nearest-neighbors (k-NN)**.

Given a training dataset (x_1, x_2, \dots, x_n) with corresponding targets (y_1, y_2, \dots, y_n) , we can make a prediction on a point, z , by looking at a set of nearest-neighbors. The actual method of prediction depends on whether we are performing regression (continuous y_i) or classification (discrete y_i).

For discrete classification targets, the prediction can be given by a maximum voting scheme, weighted by the distance to the prediction point:

$$f(z) = \max_j \sum_{i=1}^k \varphi(d_{ij}) I_{ij}$$

Our prediction here, $f(z)$, is the maximum weighted value over all classes, j , where the weighted distance from the prediction point to the training point, i , is given by $\varphi(d_{ij})$. The I_{ij} is just an indicator function, if point i is in class j . The indicator function takes the value 1 if point i is in class j , and if not, it takes on the value 0. Also, k is the number of nearest points to consider.

For continuous regression targets, the prediction is given by a weighted average of all k points nearest to the prediction:

$$f(z) = \frac{1}{k} \sum_{i=1}^k \varphi(d_i)$$

It is clear that the prediction is heavily dependent on the choice of the distance metric, d .

Common specifications of the distance metric are L1 and L2 distances, as follows:

- $d_{L1}(x_i, x_j) = |x_i - x_j| = |x_{i1} - x_{j1}| + |x_{i2} - x_{j2}| + \dots$
- $d_{L2}(x_i, x_j) = \|x_i - x_j\| = \sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + \dots}$

There are many different specifications of distance metrics that we can choose. In this chapter, we will explore the L1 and L2 metrics, as well as edit and textual distances.

We also have to choose how to weight the distances. A straightforward way to weight the distances is by the distance itself. Points that are further away from our prediction should have less of an impact than nearer points. The most common way to weight is by the normalized inverse of the distance. We will implement this method in the next recipe.



Note that k-NN is an aggregating method. For regression, we are performing a weighted average of neighbors. Because of this, predictions will be less extreme and less varied than the actual targets. The magnitude of this effect will be determined by k , the number of neighbors in the algorithm.

Working with nearest-neighbors

We will start this chapter by implementing nearest-neighbors to predict housing values. This is a great way to start with nearest-neighbors, because we will be dealing with numerical features and continuous targets.

Getting ready

To illustrate how making predictions with nearest-neighbors works in TensorFlow, we will use the Boston housing dataset. Here, we will be predicting the median neighborhood housing value as a function of several features.

Since we consider the training set the trained model, we will find the k-NNs to the prediction points, and will calculate a weighted average of the target value.

How to do it...

We proceed with the recipe as follows:

1. We will start by loading the required libraries and starting a graph session. We will use the `requests` module to load the necessary Boston housing data from the UCI machine learning repository:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import requests

sess = tf.Session()
```

2. Next, we will load the data using the `requests` module:

```
housing_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/housing'
housing_header = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
cols_used = ['CRIM', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS', 'TAX', 'PTRATIO', 'B',
num_features = len(cols_used)
# Request data
housing_file = requests.get(housing_url)
# Parse Data
housing_data = [[float(x) for x in y.split(' ') if len(x)>=1] for y in housing_f
```

3. Next, we will separate the data into our dependent and independent features. We will be predicting the last variable, `MEDV`, which is the median value for the group of houses. We also will not use the `ZN`, `CHAS`, and `RAD` features, because of their uninformative or binary natures:

```
y_vals = np.transpose([np.array([y[13] for y in housing_data])])
x_vals = np.array([[x for i,x in enumerate(y) if housing_header[i] in cols_used]
x_vals = (x_vals - x_vals.min(0)) / x_vals.ptp(0)
```

4. Now, we will split the `x` and `y` values into the train and test sets. We will create the training set by selecting about 80% of the rows at random, and will leave the remaining 20% for the test set:

```
train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
```

5. Next, we will declare our k value and batch size:

```
|   k = 4
|   batch_size=len(x_vals_test)
```

6. We will declare our placeholders next. Remember that there are no model variables to train, as the model is determined exactly by our training set:

```
|   x_data_train = tf.placeholder(shape=[None, num_features], dtype=tf.float32)
|   x_data_test = tf.placeholder(shape=[None, num_features], dtype=tf.float32)
|   y_target_train = tf.placeholder(shape=[None, 1], dtype=tf.float32)
|   y_target_test = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

7. Next, we will create our distance function for a batch of test points. Here, we are illustrating the use of the L1 distance:

 *distance = tf.reduce_sum(tf.abs(tf.subtract(x_data_train, tf.expand_dims(x_data_test, 1))))*

Note that the L2 distance function can be used as well. We would change the distance formula to $distance = tf.sqrt(tf.reduce_sum(tf.square(tf.subtract(x_data_train, tf.expand_dims(x_data_test, 1))), reduction_indices=1))$.

8. Now, we will create our prediction function. To do this, we will use the `top_k()` function, which returns the values and indices of the largest values in a tensor. Since we want the indices of the smallest distances, we will instead find the k -biggest negative distances. We will also declare the predictions and the **mean squared error (MSE)** of the target values:

```
|   top_k_xvals, top_k_indices = tf.nn.top_k(tf.negative(distance), k=k)
|   x_sums = tf.expand_dims(tf.reduce_sum(top_k_xvals, 1), 1)
|   x_sums_repeated = tf.matmul(x_sums, tf.ones([1, k], tf.float32))
|   x_val_weights = tf.expand_dims(tf.divide(top_k_xvals, x_sums_repeated), 1)

|   top_k_yvals = tf.gather(y_target_train, top_k_indices)
|   prediction = tf.squeeze(tf.batch_matmul(x_val_weights, top_k_yvals), squeeze_dims=1)
|   mse = tf.divide(tf.reduce_sum(tf.square(tf.subtract(prediction, y_target_test))),
```

9. Now, we will loop through our test data and store the prediction and accuracy values:

```
num_loops = int(np.ceil(len(x_vals_test)/batch_size))

for i in range(num_loops):
    min_index = i*batch_size
    max_index = min((i+1)*batch_size, len(x_vals_train))
    x_batch = x_vals_test[min_index:max_index]
    y_batch = y_vals_test[min_index:max_index]
    predictions = sess.run(prediction, feed_dict={x_data_train: x_vals_train, x_data_test: x_vals_test})
    batch_mse = sess.run(mse, feed_dict={x_data_train: x_vals_train, x_data_test: x_vals_test})

    print('Batch #' + str(i+1) + ' MSE: ' + str(np.round(batch_mse,3)))
```

| Batch #1 MSE: 23.153

10. Additionally, we can look at a histogram of the actual target values compared with the predicted values. One reason to look at this is to note the fact that, with an averaging method, we have trouble predicting the extreme ends of the targets:

```
bins = np.linspace(5, 50, 45)
plt.hist(predictions, bins, alpha=0.5, label='Prediction')
plt.hist(y_batch, bins, alpha=0.5, label='Actual')
plt.title('Histogram of Predicted and Actual Values')
plt.xlabel('Med Home Value in $1,000s')
plt.ylabel('Frequency')
plt.legend(loc='upper right')
plt.show()
```

We will then get the histogram plot, as follows:

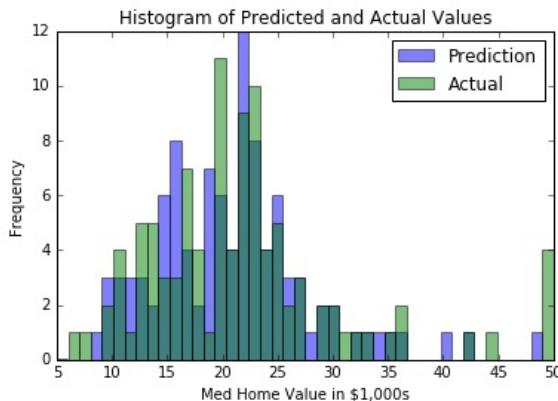


Figure 1: A histogram of the predicted values and actual target values for k-NN (where $k=4$)

One hard thing to determine is the best value of k . For the preceding diagram and predictions, we used $k=4$ for our model. We chose this specifically because it gives us the lowest MSE. This is verified by cross-validation. If we use cross-validation across multiple values of k , we will see that $k=4$ gives us a minimum MSE. We illustrate this in the following diagram. It is also worthwhile to plot the variance in the predicted values, to show that it will decrease with the more neighbors that we average over:

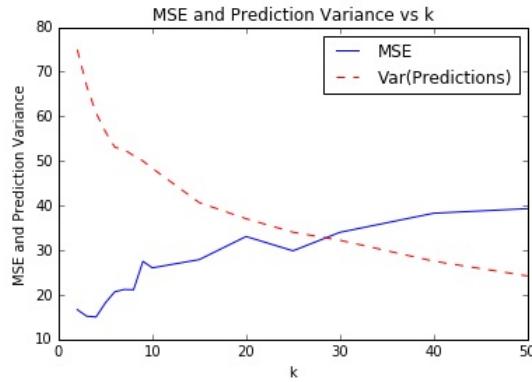


Figure 2: The MSE for k-NN predictions for various values of k . We also plot the variance of the predicted values on the test set. Note that the variance decreases as k increases.

How it works...

With the nearest-neighbors algorithm, the model is the training set. Because of this, we do not have to train any variables in our model. The only parameter, k , was determined via cross-validation, to minimize our MSE.

There's more...

For the weighting of the k-NN, we chose to weight directly by the distance. There are other options that we could consider as well. Another common method is to weight by the inverse squared distance.

Working with text based distances

Nearest-neighbors is more versatile than just dealing with numbers. As long as we have a way to measure distances between features, we can apply the nearest-neighbors algorithm. In this recipe, we will introduce how to measure text distances with TensorFlow.

Getting ready

In this recipe, we will illustrate how to use TensorFlow's text distance metric, the Levenshtein distance (the edit distance), between strings. This will be important later in this chapter, as we expand the nearest-neighbor methods to include features with text.

The Levenshtein distance is the minimal number of edits to get from one string to another string. The allowed edits are inserting a character, deleting a character, or substituting a character with a different one. For this recipe, we will use TensorFlow's Levenshtein distance function, `edit_distance()`. It is worthwhile to illustrate the use of this function, because its use will be applicable in later chapters.



Note that TensorFlow's `edit_distance()` function only accepts sparse tensors. We will have to create our strings as sparse tensors of individual characters.

How to do it...

1. First, we will load TensorFlow and initialize a graph:

```
| import tensorflow as tf  
| sess = tf.Session()
```

2. Then, we will illustrate how to calculate the edit distance between two words, 'bear' and 'beer'. First, we will create a list of characters from our strings with Python's `list()` function. Next, we will create a sparse 3D matrix from that list. We have to tell TensorFlow the character indices, the shape of the matrix, and which characters we want in the tensor. After that, we can decide whether we would like to go with the total edit distance (`normalize=False`) or the normalized edit distance (`normalize=True`), where we divide the edit distance by the length of the second word:

```
| hypothesis = list('bear')  
| truth = list('beers')  
| h1 = tf.SparseTensor([[0,0,0], [0,0,1], [0,0,2], [0,0,3]],  
|                      hypothesis, [1,1,1])  
| t1 = tf.SparseTensor([[0,0,0], [0,0,1], [0,0,2], [0,0,3],[0,0,4]], truth, [1,1,1]  
|  
| print(sess.run(tf.edit_distance(h1, t1, normalize=False)))
```

i *TensorFlow's documentation treats the two strings as a proposed (hypothesis) string and a ground truth string. We will continue this notation here, with the `h` and `t` tensors. The function, `SparseTensorValue()`, is a way to create a sparse tensor in TensorFlow. It accepts the indices, values, and shape of the sparse tensor we wish to create.*

3. Next, we will illustrate how to compare two words, `bear` and `beer`, with another word, `beers`. In order to achieve this, we must replicate `beers` in order to have the same amount of comparable words:

```
| hypothesis2 = list('bearbeer')  
| truth2 = list('beersbeers')  
| h2 = tf.SparseTensor([[0,0,0], [0,0,1], [0,0,2], [0,0,3], [0,1,0], [0,1,1], [0,1  
| t2 = tf.SparseTensor([[0,0,0], [0,0,1], [0,0,2], [0,0,3], [0,0,4], [0,1,0], [0,1  
|  
| print(sess.run(tf.edit_distance(h2, t2, normalize=True)))
```

i `[[0.40000001 0.2]]`

4. A more efficient way to compare a set of words against another word is shown in this example. We will create the indices and list of characters

beforehand, for both the hypothesis and the ground truth string:

```
hypothesis_words = ['bear', 'bar', 'tensor', 'flow']
truth_word = ['beers''']
num_h_words = len(hypothesis_words)
h_indices = [[xi, 0, yi] for xi,x in enumerate(hypothesis_words) for yi,y in enumerate(truth_word)]
h_chars = list(''.join(hypothesis_words))
h3 = tf.SparseTensor(h_indices, h_chars, [num_h_words,1,1])
truth_word_vec = tf.SparseTensor(h3, num_h_words)
t_indices = [[xi, 0, yi] for xi,x in enumerate(truth_word_vec) for yi,y in enumerate(hypothesis_words)]
t_chars = list(''.join(truth_word_vec))
t3 = tf.SparseTensor(t_indices, t_chars, [num_h_words,1,1])

print(sess.run(tf.edit_distance(h3, t3, normalize=True)))

[[ 0.40000001]
 [ 0.60000002]
 [ 0.80000001]
 [ 1. ]]
```

5. Now, we will illustrate how to calculate the edit distance between two word lists, using placeholders. The concept is the same, except that we will be feeding in `sparseTensorValue()` instead of sparse tensors. First, we will create a function that creates the sparse tensors from a word list:

```
def create_sparse_vec(word_list):
    num_words = len(word_list)
    indices = [[xi, 0, yi] for xi,x in enumerate(word_list) for yi,y in enumerate(word_list)]
    chars = list(''.join(word_list))
    return(tf.SparseTensorValue(indices, chars, [num_words,1,1]))

hyp_string_sparse = create_sparse_vec(hypothesis_words)
truth_string_sparse = create_sparse_vec(truth_word)

hyp_input = tf.sparse_placeholder(dtype=tf.string)
truth_input = tf.sparse_placeholder(dtype=tf.string)

edit_distances = tf.edit_distance(hyp_input, truth_input, normalize=True)

feed_dict = {hyp_input: hyp_string_sparse,
            truth_input: truth_string_sparse}

print(sess.run(edit_distances, feed_dict=feed_dict))

[[ 0.40000001]
 [ 0.60000002]
 [ 0.80000001]
 [ 1. ]]
```

How it works...

In this recipe, we showed that we can measure text distances in several ways using TensorFlow. This will be extremely useful for performing nearest-neighbors on data that has text features. We will see more of this later in the chapter, when we perform address matching.

There's more...

There are other text distance metrics that we should discuss. Here is a definition table describing other text distances between two strings, s_1 and s_2 :

Name	Description	Formula
Hamming distance	Number of equal character positions. Only valid if the strings are equal lengths.	$D(s_1, s_2) = \sum_i I_i$, where I is an indicator function of equal characters.
Cosine distance	The dot product of the k -gram differences divided by the L2 norm of the k -gram differences.	$D(s_1, s_2) = 1 - \frac{k(s_1) \cdot k(s_2)}{\ k(s_1)\ \ k(s_2)\ }$
Jaccard distance	Number of characters in common, divided by the total union of characters in both strings.	$D(s_1, s_2) = \frac{ s_1 \cap s_2 }{ s_1 \cup s_2 }$



Computing with mixed distance functions

When dealing with data observations that have multiple features, we should be aware that features can be scaled differently, on different scales. In this recipe, we will account for that to improve our housing value predictions.

Getting ready

It is important to extend the nearest-neighbor algorithm, to take into account variables that are scaled differently. In this example, we will illustrate how to scale the distance function for different variables. Specifically, we will scale the distance function as a function of the feature variance.

The key to weighting the distance function is to use a weight matrix. The distance function, written with matrix operations, becomes the following formula:

$$D(x, y) = \sqrt{(x - y)^T \cdot A \cdot (x - y)}$$

Here, A is a diagonal weight matrix that we will use to scale the distance metric for each feature.

In this recipe, we will try to improve our MSE on the Boston housing value dataset. This dataset is a great example of features that are on different scales, and the nearest-neighbor algorithm will benefit from scaling the distance function.

How to do it...

We will proceed with the recipe as follows:

1. First, we will load the necessary libraries and start a graph session:

```
| import matplotlib.pyplot as plt  
| import numpy as np  
| import tensorflow as tf  
| import requests  
| sess = tf.Session()
```

2. Next, we will load the data and store it in a NumPy array. Again, note that we will only use certain columns for prediction. We do not use the id, nor variables that have very low variance:

```
| housing_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/housing  
| housing_header = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD  
| cols_used = ['CRIM', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS', 'TAX', 'PTRATIO', 'B',  
| num_features = len(cols_used)  
| housing_file = requests.get(housing_url)  
| housing_data = [[float(x) for x in y.split(' ') if len(x)>=1] for y in housing_f  
| y_vals = np.transpose([np.array([y[13] for y in housing_data])])  
| x_vals = np.array([[x for i,x in enumerate(y) if housing_header[i] in cols_used]
```

3. Now, we will scale the x values to be between 0 and 1, with min-max scaling:

```
| x_vals = (x_vals - x_vals.min(0)) / x_vals.ptp(0)
```

4. Then, we will create the diagonal-weight matrix that will provide the scaling of the distance metric by the standard deviation of the features:

```
| weight_diagonal = x_vals.std(0)  
| weight_matrix = tf.cast(tf.diag(weight_diagonal), dtype=tf.float32)
```

5. Now, we will split the data into a training and a test set. We will also declare k , the amount of nearest-neighbors, and make the batch size equal to the test set size:

```
| train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)  
| test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))  
| x_vals_train = x_vals[train_indices]  
| x_vals_test = x_vals[test_indices]  
| y_vals_train = y_vals[train_indices]  
| y_vals_test = y_vals[test_indices]
```

```
| k = 4  
| batch_size=len(x_vals_test)
```

6. We will declare the placeholders that we need next. We have four placeholders—the x -inputs and y -targets for both the training and test sets:

```
| x_data_train = tf.placeholder(shape=[None, num_features], dtype=tf.float32)  
| x_data_test = tf.placeholder(shape=[None, num_features], dtype=tf.float32)  
| y_target_train = tf.placeholder(shape=[None, 1], dtype=tf.float32)  
| y_target_test = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

7. Now, we can declare our distance function. For readability, we will break the distance function up into its components. Note that we will have to tile the weight matrix by the batch size and use the `batch_matmul()` function to perform batch matrix multiplication across the batch size:

```
| subtraction_term = tf.subtract(x_data_train, tf.expand_dims(x_data_test,1))  
| first_product = tf.batch_matmul(subtraction_term, tf.tile(tf.expand_dims(weight_,  
| second_product = tf.batch_matmul(first_product, tf.transpose(subtraction_term, p  
| distance = tf.sqrt(tf.batch_matrix_diag_part(second_product)))
```

8. After we calculate all of the training distances for each test point, we will need to return the top k -NNs. We can do this with the `top_k()` function. Since this function returns the largest values, and we want the smallest distances, we return the largest of the negative distance values. Then, we will make predictions as the weighted average of the distances of the top k neighbors:

```
| top_k_xvals, top_k_indices = tf.nn.top_k(tf.neg(distance), k=k)  
| x_sums = tf.expand_dims(tf.reduce_sum(top_k_xvals, 1),1)  
| x_sums_repeated = tf.matmul(x_sums, tf.ones([1, k], tf.float32))  
| x_val_weights = tf.expand_dims(tf.div(top_k_xvals,x_sums_repeated), 1)  
| top_k_yvals = tf.gather(y_target_train, top_k_indices)  
| prediction = tf.squeeze(tf.batch_matmul(x_val_weights,top_k_yvals), squeeze_dims
```

9. To evaluate our model, we will calculate the MSE of our predictions:

```
| mse = tf.divide(tf.reduce_sum(tf.square(tf.subtract(prediction, y_target_test))))
```

10. Now, we can loop through our test batches and calculate the MSE for each:

```
| num_loops = int(np.ceil(len(x_vals_test)/batch_size))  
| for i in range(num_loops):  
|     min_index = i*batch_size  
|     max_index = min((i+1)*batch_size, len(x_vals_train))  
|     x_batch = x_vals_test[min_index:max_index]  
|     y_batch = y_vals_test[min_index:max_index]  
|     predictions = sess.run(prediction, feed_dict={x_data_train: x_vals_train, x_  
|     batch_mse = sess.run(mse, feed_dict={x_data_train: x_vals_train, x_data_test  
|     print('Batch #' + str(i+1) + ' MSE: ' + str(np.round(batch_mse,3)))
```

```
| Batch #1 MSE: 21.322
```

11. As a final comparison, we can plot the distribution of housing values for the actual test set and the predictions on the test set with the following code:

```
bins = np.linspace(5, 50, 45)
plt.hist(predictions, bins, alpha=0.5, label='Prediction')
plt.hist(y_batch, bins, alpha=0.5, label='Actual')
plt.title('Histogram of Predicted and Actual Values')
plt.xlabel('Med Home Value in $1,000s')
plt.ylabel('Frequency')
plt.legend(loc='upper right')
plt.show()
```

We will get the following histogram for the preceding code:

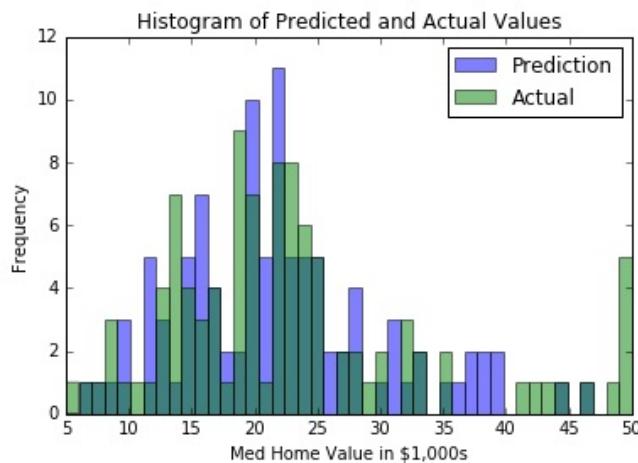


Figure 3: The two histograms of the predicted and actual housing values on the Boston dataset; this time, we have scaled the distance function differently for each feature

How it works...

We decreased our MSE on the test set by introducing a method of scaling the distance functions for each feature. Here, we scaled the distance functions by a factor of the feature's standard deviation. This provides a more accurate view of measuring which points are the closest neighbors. From this, we also took the weighted average of the top k neighbors as a function of distance, to get the housing value prediction.

There's more...

This scaling factor can also be used to down-weight or up-weight features in the nearest-neighbor distance calculation. This can be useful in situations where we trust some features more than others.

Using an address matching example

Now that we have measured numerical and text distances, we will spend some time learning how to combine them to measure distances between observations that have both text and numerical features.

Getting ready

Nearest-neighbor is a great algorithm to use for address matching. Address matching is a type of record matching in which we have addresses in multiple datasets and would like to match them up. In address matching, we may have typos in the address, different cities, or different ZIP Codes, but they may all refer to the same address. Using the nearest-neighbor algorithm across the numerical and character components of an address may help us to identify addresses that are actually the same.

In this example, we will generate two datasets. Each dataset will comprise a street address and a ZIP Code. But one dataset has a high number of typos in the street address. We will take the non-typo dataset as our gold standard, and will return one address from it for each typo address that is the closest as a function of string distance (for the street) and numerical distance (for the ZIP Code).

The first part of the code will focus on generating the two datasets. Then, the second part of the code will run through the test set and return the closest address from the training set.

How to do it...

We will proceed with the recipe as follows:

1. We will start by loading the necessary libraries:

```
import random
import string
import numpy as np
import tensorflow as tf
```

2. We will now create the reference dataset. To show succinct output, we will only make each dataset be comprised of 10 addresses (but it can be run with many more):

```
n = 10
street_names = ['abbey', 'baker', 'canal', 'donner', 'elm']
street_types = ['rd', 'st', 'ln', 'pass', 'ave']
rand_zips = [random.randint(65000, 65999) for i in range(5)]
numbers = [random.randint(1, 9999) for i in range(n)]
streets = [random.choice(street_names) for i in range(n)]
street_suffs = [random.choice(street_types) for i in range(n)]
zips = [random.choice(rand_zips) for i in range(n)]
full_streets = [str(x) + ' ' + y + ' ' + z for x,y,z in zip(numbers, streets, street_suffs)]
reference_data = [list(x) for x in zip(full_streets, zips)]
```

3. To create the test set, we will need a function that will randomly create a typo in a string and return the resultant string:

```
def create_typos(s, prob=0.75):
    if random.uniform(0,1) < prob:
        rand_ind = random.choice(range(len(s)))
        s_list = list(s)
        s_list[rand_ind]=random.choice(string.ascii_lowercase)
        s = ''.join(s_list)
    return s

typo_streets = [create_typos(x) for x in streets]
typo_full_streets = [str(x) + ' ' + y + ' ' + z for x,y,z in zip(numbers, typo_streets, zips)]
test_data = [list(x) for x in zip(typo_full_streets, zips)]
```

4. Now, we can initialize a graph session and declare the placeholders that we need. We will need four placeholders in each test and reference set, and we will need an address and ZIP Code placeholder:

```
sess = tf.Session()
test_address = tf.sparse_placeholder( dtype=tf.string)
test_zip = tf.placeholder(shape=[None, 1], dtype=tf.float32)
ref_address = tf.sparse_placeholder(dtype=tf.string)
ref_zip = tf.placeholder(shape=[None, n], dtype=tf.float32)
```

- Now, we will declare the numerical zip distance and the edit distance for the address string:

```
zip_dist = tf.square(tf.subtract(ref_zip, test_zip))
address_dist = tf.edit_distance(test_address, ref_address, normalize=True)
```

- We will now convert the zip distance and the address distance into similarities. For the similarities, we want a similarity of 1 when the two inputs are exactly the same, and near 0 when they are very different. For the zip distance, we can do this by taking the distances, subtracting from the max, and then dividing by the range of the distances. For the address similarity, since the distance is already scaled between 0 and 1, we just subtract it from 1 to get the similarity:

```
zip_max = tf.gather(tf.squeeze(zip_dist), tf.argmax(zip_dist, 1))
zip_min = tf.gather(tf.squeeze(zip_dist), tf.argmin(zip_dist, 1))
zip_sim = tf.divide(tf.subtract(zip_max, zip_dist), tf.subtract(zip_max, zip_min))
address_sim = tf.subtract(1., address_dist)
```

- To combine the two similarity functions, we will take a weighted average of the two. For this recipe, we put equal weight on the address and the ZIP Code. We can change this, depending on how much we trust each feature. We will then return the index of the highest similarity of the reference set:

```
address_weight = 0.5
zip_weight = 1. - address_weight
weighted_sim = tf.add(tf.transpose(tf.multiply(address_weight, address_sim)), tf
top_match_index = tf.argmax(weighted_sim, 1)
```

- In order to use the edit distance in TensorFlow, we have to convert the address strings to a sparse vector. In a prior recipe in this chapter, *Working with text-based distances*, we created the following function, and we will use it in this recipe as well:

```
def sparse_from_word_vec(word_vec):
    num_words = len(word_vec)
    indices = [[xi, 0, yi] for xi,x in enumerate(word_vec) for yi,y in enumerate
    chars = list(''.join(word_vec))
    # Now we return our sparse vector
    return tf.SparseTensorValue(indices, chars, [num_words,1,1])
```

- We will need to separate the addresses and ZIP Codes in the reference dataset so that we can feed them into the placeholders when we loop through the test set:

```
reference_addresses = [x[0] for x in reference_data]
reference_zips = np.array([[x[1] for x in reference_data]])
```

10. We need to create the sparse tensor set of reference addresses, by using the function that we created in step 8:

```
| sparse_ref_set = sparse_from_word_vec(reference_addresses)
```

11. Now, we can loop through each entry of the test set and return the index of the reference set that it is the closest to. We will print off both test and reference sets for each entry. As you can see, we have great results in this generated dataset:

```
for i in range(n):
    test_address_entry = test_data[i][0]
    test_zip_entry = [[test_data[i][1]]]

    # Create sparse address vectors
    test_address_repeated = [test_address_entry] * n
    sparse_test_set = sparse_from_word_vec(test_address_repeated)

    feeddict={test_address: sparse_test_set,
              test_zip: test_zip_entry,
              ref_address: sparse_ref_set,
              ref_zip: reference_zips}
    best_match = sess.run(top_match_index, feed_dict=feeddict)
    best_street = reference_addresses[best_match[0]]
    [best_zip] = reference_zips[0][best_match]
    [[test_zip_]] = test_zip_entry
    print('Address: ' + str(test_address_entry) + ', ' + str(test_zip_))
    print('Match : ' + str(best_street) + ', ' + str(best_zip))
```

We will get the following results:

```
Address: 8659 beker ln, 65463
Match : 8659 baker ln, 65463
Address: 1048 eanal ln, 65681
Match : 1048 canal ln, 65681
Address: 1756 vaker st, 65983
Match : 1756 baker st, 65983
Address: 900 abbjy pass, 65983
Match : 900 abbey pass, 65983
Address: 5025 canal rd, 65463
Match : 5025 canal rd, 65463
Address: 6814 elh st, 65154
Match : 6814 elm st, 65154
Address: 3057 cagal ave, 65463
Match : 3057 canal ave, 65463
Address: 7776 iaker ln, 65681
Match : 7776 baker ln, 65681
Address: 5167 caker rd, 65154

Match : 5167 baker rd, 65154
Address: 8765 donnor st, 65154
Match : 8765 donner st, 65154
```

How it works...

One of the difficult things to figure out in address matching problems like this one is the value of the weights and how to scale the distances. This might take some exploration and insight into the data itself. Also, when dealing with addresses, we should consider components other than those used here. We can consider the street number as a separate component from the street address, and can even have other components, such as the city and state.



When dealing with numerical address components, note that they can be treated as numbers (with a numerical distance) or as characters (with an edit distance). It is up to you to choose which. Note that we could consider using an edit distance with the ZIP Code, if we think that typos in the ZIP Code come from human errors and not, say, computer mapping errors.

To get a feel for how typos affect the results, we encourage the reader to change the typo function to make more typos or more frequent typos and increase the dataset size to see how well this algorithm works.

Using nearest-neighbors for image recognition

Nearest-neighbors can also be used for image recognition. The hello world of image recognition datasets is the MNIST handwritten digit dataset. Since we will be using this dataset for various neural network image recognition algorithms in later chapters, it will be great to compare the results to a non-neural network algorithm.

Getting ready

The MNIST digit dataset is composed of thousands of labeled images that are 28x28 pixels in size. Although this is considered a small image, it has a total of 784 pixels (or features) for the nearest-neighbor algorithm. We will compute the nearest-neighbor prediction for this categorical problem by considering the mode prediction of the nearest k neighbors ($k=4$, in this example).

How to do it...

We will proceed with the recipe as follows:

1. We will start by loading the necessary libraries. Note that we will also import the Python Image Library (PIL), to be able to plot a sample of the predicted outputs. And TensorFlow has a built-in method to load the MNIST dataset that we will use, as follows:

```
import random
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from PIL import Image
from tensorflow.examples.tutorials.mnist import input_data
```

2. Now, we will start a graph session and load the MNIST data in a one-hot encoded form:

```
sess = tf.Session()
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

i One-hot encoding is a numerical representation of categorical values that are better suited for numerical computations. Here, we have 10 categories (the numbers 0-9), and represent them as a 0-1 vector of length 10. For example, the 0 category is denoted by the vector 1,0,0,0,0,0,0,0,0,0, the 1 vector is denoted by 0,1,0,0,0,0,0,0,0,0, and so on.

3. Because the MNIST dataset is large and computing the distances between 784 features on tens of thousands of inputs would be computationally difficult, we will sample a smaller set of images to train on. Also, we will choose a test set number that is divisible by 6, only for plotting purposes, as we will plot the last batch of six images to see a sample of the results:

```
train_size = 1000
test_size = 102
rand_train_indices = np.random.choice(len(mnist.train.images), train_size, replace=False)
rand_test_indices = np.random.choice(len(mnist.test.images), test_size, replace=False)
x_vals_train = mnist.train.images[rand_train_indices]
x_vals_test = mnist.test.images[rand_test_indices]
y_vals_train = mnist.train.labels[rand_train_indices]
y_vals_test = mnist.test.labels[rand_test_indices]
```

4. We will declare our k value and batch size:

```
k = 4
batch_size=6
```

- Now, we will initialize the placeholders that we will feed into the graph:

```
x_data_train = tf.placeholder(shape=[None, 784], dtype=tf.float32)
x_data_test = tf.placeholder(shape=[None, 784], dtype=tf.float32)
y_target_train = tf.placeholder(shape=[None, 10], dtype=tf.float32)
y_target_test = tf.placeholder(shape=[None, 10], dtype=tf.float32)
```

- We will then declare our distance metric. Here, we will use the L1 metric (absolute value):

 `distance = tf.reduce_sum(tf.abs(tf.subtract(x_data_train, tf.expand_dims(x_data_`
Note that we can also make our distance function use the L2 distance, by using the following code instead: `distance = tf.sqrt(tf.reduce_sum(tf.square(tf.subtract(x_data_train, tf.expand_dims(x_data_test, 1))), reduction_indices=1)).`

- Now, we will find the top k images that are the closest and predict the mode. The mode will be performed on one-hot encoded indices, counting which occurs the most:

```
top_k_xvals, top_k_indices = tf.nn.top_k(tf.negative(distance), k=k)
prediction_indices = tf.gather(y_target_train, top_k_indices)
count_of_predictions = tf.reduce_sum(prediction_indices, reduction_indices=1)
prediction = tf.argmax(count_of_predictions)
```

- We can now loop through our test set, compute the predictions, and store them, as follows:

```
num_loops = int(np.ceil(len(x_vals_test)/batch_size))
test_output = []
actual_vals = []
for i in range(num_loops):
    min_index = i*batch_size
    max_index = min((i+1)*batch_size, len(x_vals_train))
    x_batch = x_vals_test[min_index:max_index]
    y_batch = y_vals_test[min_index:max_index]
    predictions = sess.run(prediction, feed_dict={x_data_train: x_vals_train, x_
    test_output.extend(predictions)
    actual_vals.extend(np.argmax(y_batch, axis=1))
```

- Now that we have saved the actual and predicted output, we can calculate the accuracy. This will change due to our random sampling of the test/training datasets, but we should end up with accuracy values around 80%–90%:

```
accuracy = sum([1./test_size for i in range(test_size) if test_output[i]==actual
print('Accuracy on test set: ' + str(accuracy))
Accuracy on test set: 0.8333333333333325
```

- Here is the code to plot the preceding batch results:

```

actuals = np.argmax(y_batch, axis=1)
Nrows = 2
Ncols = 3
for i in range(len(actuals)):
    plt.subplot(Nrows, Ncols, i+1)
    plt.imshow(np.reshape(x_batch[i], [28,28]), cmap='Greys_r')
    plt.title('Actual: ' + str(actuals[i]) + ' Pred: ' + str(predictions[i]), fontweight='bold')
    frame = plt.gca()
    frame.axes.get_xaxis().set_visible(False)
    frame.axes.get_yaxis().set_visible(False)

```

The results will be as follows:

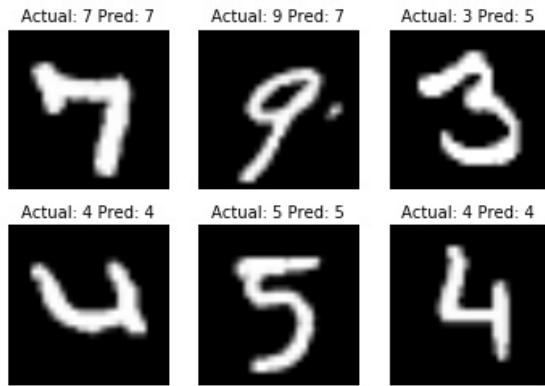


Figure 4: The last batch of six images that we ran our nearest-neighbor prediction on. We can see that we do not get all of the images exactly correct.

How it works...

Given enough computation time and computational resources, we could have made the test and training sets bigger. This probably would have increased our accuracy, and is also a common way to prevent overfitting. Also, note that this algorithm warrants further exploration on the ideal k value to choose. The k value can be chosen after a set of cross-validation experiments on the dataset.

There's more...

We can also use the nearest-neighbor algorithm for evaluating unseen numbers from the user. Please see the online repository for a way to use this model to evaluate user input digits, at https://github.com/nfmccclure/tensorflow_cookbook.

In this chapter, we explored how to use k-NN algorithms for regression and classification. We discussed the different usages of distance functions, and how to mix them together. We encourage the reader to explore different distance metrics, weights, and k values to optimize the accuracy of these methods.

Neural Networks

In this chapter, we will introduce neural networks and how to implement them in TensorFlow. Most of the subsequent chapters will be based on neural networks, so learning how to use them in TensorFlow is very important. We will start by introducing the basic concepts of neural networking before working up to multilayer networks. In the last section, we will create a neural network that will learn how to play Tic Tac Toe.

In this chapter, we'll cover the following recipes:

- Implementing operational gates
- Working with gates and activation functions
- Implementing a one-layer neural network
- Implementing different layers
- Using multilayer networks
- Improving predictions of linear models
- Learning to play Tic Tac Toe

The reader can find all of the code from this chapter online, at https://github.com/fmcclure/tensorflow_cookbook and at the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

Introduction

Neural networks are currently breaking records in tasks such as image and speech recognition, reading handwriting, understanding text, image segmentation, dialog systems, autonomous car driving, and so much more. While some of these aforementioned tasks will be covered in later chapters, it is important to introduce neural networks as an easy-to-implement machine learning algorithm, so that we can expand on it later.

The concept of a neural network has been around for decades. However, it only recently gained traction because we now have the computational power to train large networks because of advances in processing power, algorithm efficiency, and data sizes.

A neural network is basically a sequence of operations applied to a matrix of input data. These operations are usually collections of additions and multiplications followed by the application of non-linear functions. One example that we have already seen is logistic regression, which we looked at in [Chapter 3, Linear Regression](#). Logistic regression is the sum of partial slope-feature products followed by the application of the sigmoid function, which is non-linear. Neural networks generalize this a bit more by allowing any combination of operations and non-linear functions, which includes the application of absolute value, maximum, minimum, and so on.

The important trick with neural networks is called **back propagation**. Back propagation is a procedure that allows us to update model variables based on the learning rate and the output of the loss function. We used back propagation to update our model variables in [chapter 3, Linear Regression](#), and [Chapter 4, Support Vector Machines](#).

Another important feature to take note of regarding neural networks is the non-linear activation function. Since most neural networks are just combinations of addition and multiplication operations, they will not be able to model non-linear data sets. To address this issue, we have used non-linear activation functions in our neural networks. This will allow the neural network to adapt to most non-

linear situations.

It is important to remember that, as we have seen in many of the algorithms covered, neural networks are sensitive to the hyper-parameters we choose. In this chapter, we will explore the impact of different learning rates, loss functions, and optimization procedures.



There are more resources for learning about neural networks that cover the topic in greater depth and more detail. These resources are as follows:

- The seminal paper describing back propagation is *Efficient Back Prop* by Yann LeCun et. al. The PDF is located here: <http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>.
- CS231, *Convolutional Neural Networks for Visual Recognition*, by Stanford University. Class resources are available here: <http://cs231n.stanford.edu/>.
- CS224d, *Deep Learning for Natural Language Processing*, by Stanford University. Class resources are available here: <http://cs224d.stanford.edu/>.
- *Deep Learning*, a book by the MIT Press. Goodfellow, et. al. 2016. The book is located here: <http://www.deeplearningbook.org>.
- There is an online book called *Neural Networks and Deep Learning* by Michael Nielsen, which is located here: <http://neuralnetworksanddeeplearning.com/>.
- For a more pragmatic approach and introduction to neural networks, Andrej Karpathy has written a great summary with JavaScript examples called *A Hacker's Guide to Neural Networks*. The write-up is located here: <http://karpathy.github.io/neuralsnets/>.
- Another site that summarizes deep learning well is called *Deep Learning for Beginners* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville. The web page can be found here: <http://randommekk.github.io/deep/deeplearning.html>.

Implementing operational gates

One of the most fundamental concepts of neural networks is an operating as an operational gate. In this section, we will start with a multiplication operation as a gate, before moving on to consider nested gate operations.

Getting ready

The first operational gate we will implement is $f(x) = a \cdot x$. To optimize this gate, we declare the a input as a variable and the x input as a placeholder. This means that TensorFlow will try to change the a value and not the x value. We will create the loss function as the difference between the output and the target value, which is 50.

The second, nested operational gate will be $f(x) = a \cdot x + b$. Again, we will declare a and b as variables and x as a place holder. We optimize the output towards the target value of 50 again. The interesting thing to note is that the solution for this second example is not unique. There are many combinations of model variables that will allow the output to be 50. With neural networks, we do not care so much about the values of the intermediate model variables, but instead place more emphasis on the desired output.

Think of these operations as operational gates on our computational graph. The following diagram depicts the two preceding examples:

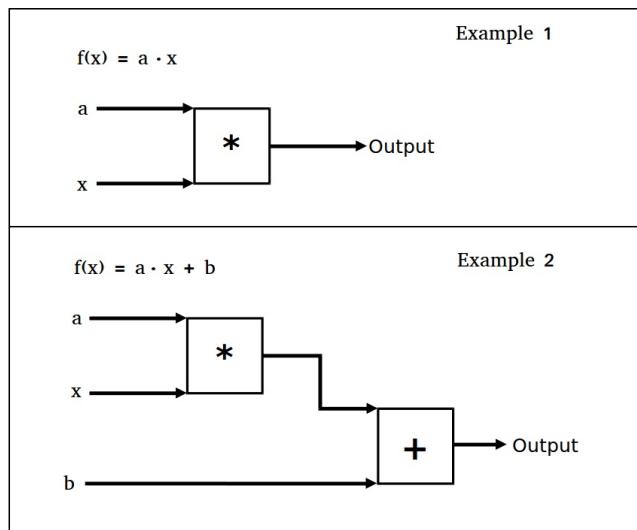


Figure 1: Two operational gate examples in this section

How to do it...

To implement the first operational gate, $f(x) = a \cdot x$, in TensorFlow and train the output towards the value of 50, follow these steps:

1. Start off by loading `TensorFlow` and creating a graph session as follows:

```
| import tensorflow as tf  
| sess = tf.Session()
```

2. Now we will need to declare our model variable, input data, and placeholder. We make our input data equal to the value `5`, so that the multiplication factor to get 50 will be 10 (that is, $5 \times 10 = 50$), as follows:

```
| a = tf.Variable(tf.constant(4.))  
| x_val = 5.  
| x_data = tf.placeholder(dtype=tf.float32)
```

3. Next, we add the operation to our computational graph with the following input:

```
| multiplication = tf.multiply(a, x_data)
```

4. We will now declare the loss function as the L2 distance between the output and the desired target value of `50`, as follows:

```
| loss = tf.square(tf.subtract(multiplication, 50.))
```

5. Now we initialize our model variable and declare our optimizing algorithm as the standard gradient descent, as follows:

```
| init = tf.global_variables_initializer()  
| sess.run(init)  
| my_opt = tf.train.GradientDescentOptimizer(0.01)  
| train_step = my_opt.minimize(loss)
```

6. We can now optimize our model output towards the desired value of `50`. We do this by continually feeding in the input value of `5` and back-propagating the loss to update the model variable towards the value of `10`, shown as follows:

```
| print('Optimizing a Multiplication Gate Output to 50.')
```

```

for i in range(10):
    sess.run(train_step, feed_dict={x_data: x_val})
    a_val = sess.run(a)
    mult_output = sess.run(multiplication, feed_dict={x_data: x_val})
    print(str(a_val) + ' * ' + str(x_val) + ' = ' + str(mult_output))

```

7. The preceding step should result in the following output:

```

Optimizing a Multiplication Gate Output to 50.
7.0 * 5.0 = 35.0
8.5 * 5.0 = 42.5
9.25 * 5.0 = 46.25
9.625 * 5.0 = 48.125
9.8125 * 5.0 = 49.0625
9.90625 * 5.0 = 49.5312
9.95312 * 5.0 = 49.7656
9.97656 * 5.0 = 49.8828
9.98828 * 5.0 = 49.9414
9.99414 * 5.0 = 49.9707

```

Next, we will do the same with the two-nested operational gate,
 $f(x) = a \cdot x + b$.

8. We will start in exactly the same way as the preceding example, but will initialize two model variables, `a` and `b`, as follows:

```

from tensorflow.python.framework import ops
ops.reset_default_graph()
sess = tf.Session()

a = tf.Variable(tf.constant(1.))
b = tf.Variable(tf.constant(1.))
x_val = 5.
x_data = tf.placeholder(dtype=tf.float32)

two_gate = tf.add(tf.multiply(a, x_data), b)

loss = tf.square(tf.subtract(two_gate, 50.))

my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)

init = tf.global_variables_initializer()
sess.run(init)

```

9. We now optimize the model variables to train the output towards the target value of 50, shown as follows:

```

print('Optimizing Two Gate Output to 50.')
for i in range(10):
    # Run the train step
    sess.run(train_step, feed_dict={x_data: x_val})
    # Get the a and b values
    a_val, b_val = (sess.run(a), sess.run(b))
    # Run the two-gate graph output

```

```
| two_gate_output = sess.run(two_gate, feed_dict={x_data: x_val})  
| print(str(a_val) + ' * ' + str(x_val) + ' + ' + str(b_val) + ' = ' + str(two
```

10. The preceding step should result in the following output:

```
Optimizing Two Gate Output to 50.  
5.4 * 5.0 + 1.88 = 28.88  
7.512 * 5.0 + 2.3024 = 39.8624  
8.52576 * 5.0 + 2.50515 = 45.134  
9.01236 * 5.0 + 2.60247 = 47.6643  
9.24593 * 5.0 + 2.64919 = 48.8789  
9.35805 * 5.0 + 2.67161 = 49.4619  
9.41186 * 5.0 + 2.68237 = 49.7417  
9.43769 * 5.0 + 2.68754 = 49.876  
9.45009 * 5.0 + 2.69002 = 49.9405  
9.45605 * 5.0 + 2.69121 = 49.9714
```



It is important to note here that the solution to the second example is not unique. This does not matter as much in neural networks, as all parameters are adjusted towards reducing the loss.

The final solution here will depend on the initial values of a and b. If these were randomly initialized, instead of to the value of 1, we would see different ending values for the model variables for each iteration.

How it works...

We achieved the optimization of a computational gate via TensorFlow's implicit back propagation. TensorFlow keeps track of our model's operations and variable values and makes adjustments in respect of our optimization algorithm specification and the output of the loss function.

We can keep expanding the operational gates while keeping track of which inputs are variables and which inputs are data. This is important to keep track of, because TensorFlow will change all variables to minimize the loss but not the data, which is declared as placeholders.

The implicit ability to keep track of the computational graph and update the model variables automatically with every training step is one of the great features of TensorFlow and what makes it so powerful.

Working with gates and activation functions

Now that we can link together operational gates, we want to run the computational graph output through an activation function. In this section, we will introduce common activation functions.

Getting ready

In this section, we will compare and contrast two different activation functions: **sigmoid** and **rectified linear unit (ReLU)**. Recall that the two functions are given by the following equations:

$$\begin{aligned} \text{sigmoid}(x) &= \sigma(x) = \frac{1}{1 + e^x} \\ \text{ReLU}(x) &= \max(0, x) \end{aligned}$$

In this example, we will create two one-layer neural networks with the same structure, except that one will feed through the sigmoid activation and one will feed through the ReLU activation. The loss function will be governed by the L2 distance from the value 0.75. We will randomly pull batch data from a normal distribution (*Normal(mean=2, sd=0.1)*) and then optimize the output towards 0.75.

How to do it...

We proceed with the recipe as follows:

1. We will start by loading the necessary libraries and initializing a graph. This is also a good point at which we can bring up how to set a random seed with TensorFlow. Since we will be using a random number generator from NumPy and TensorFlow, we need to set a random seed for both. With the same random seeds set, we should be able to replicate the results. We do this with the following input:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
sess = tf.Session()
tf.set_random_seed(5)
np.random.seed(42)
```

2. Now we need to declare our batch size, model variables, data, and a placeholder for feeding the data in. Our computational graph will consist of feeding in our normally-distributed data into two similar neural networks that differ only by the activation function at the end, shown as follows:

```
batch_size = 50
a1 = tf.Variable(tf.random_normal(shape=[1,1]))
b1 = tf.Variable(tf.random_uniform(shape=[1,1]))
a2 = tf.Variable(tf.random_normal(shape=[1,1]))
b2 = tf.Variable(tf.random_uniform(shape=[1,1]))
x = np.random.normal(2, 0.1, 500)
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

3. Next, we'll declare our two models, the sigmoid activation model and the ReLU activation model, as follows:

```
sigmoid_activation = tf.sigmoid(tf.add(tf.matmul(x_data, a1), b1))
relu_activation = tf.nn.relu(tf.add(tf.matmul(x_data, a2), b2))
```

4. The loss functions will be the average L2 norm between the model output and the value of 0.75, as follows:

```
loss1 = tf.reduce_mean(tf.square(tf.subtract(sigmoid_activation, 0.75)))
loss2 = tf.reduce_mean(tf.square(tf.subtract(relu_activation, 0.75)))
```

5. Now we need to declare our optimization algorithm and initialize our

variables, shown as follows:

```
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step_sigmoid = my_opt.minimize(loss1)
train_step_relu = my_opt.minimize(loss2)
init = tf.global_variable_initializer()
sess.run(init)
```

6. Now we'll loop through our training for 750 iterations for both models, as shown in the following code block. We will also save the loss output and the activation output values for plotting later on:

```
loss_vec_sigmoid = []
loss_vec_relu = []
activation_sigmoid = []
activation_relu = []
for i in range(750):
    rand_indices = np.random.choice(len(x), size=batch_size)
    x_vals = np.transpose([x[rand_indices]])
    sess.run(train_step_sigmoid, feed_dict={x_data: x_vals})
    sess.run(train_step_relu, feed_dict={x_data: x_vals})

    loss_vec_sigmoid.append(sess.run(loss1, feed_dict={x_data: x_vals}))
    loss_vec_relu.append(sess.run(loss2, feed_dict={x_data: x_vals}))

    activation_sigmoid.append(np.mean(sess.run(sigmoid_activation, feed_dict={x_data: x_vals})))
    activation_relu.append(np.mean(sess.run(rectified_linear_activation, feed_dict={x_data: x_vals}))
```

7. To plot the loss and the activation outputs, we need to input the following code:

```
plt.plot(activation_sigmoid, 'k-', label='Sigmoid Activation')
plt.plot(activation_relu, 'r--', label='ReLU Activation')
plt.ylim([0, 1.0])
plt.title('Activation Outputs')
plt.xlabel('Generation')
plt.ylabel('Outputs')
plt.legend(loc='upper right')
plt.show()
plt.plot(loss_vec_sigmoid, 'k-', label='Sigmoid Loss')
plt.plot(loss_vec_relu, 'r--', label='ReLU Loss')
plt.ylim([0, 1.0])
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```

The activation output needs to be plot as shown in the following diagram:

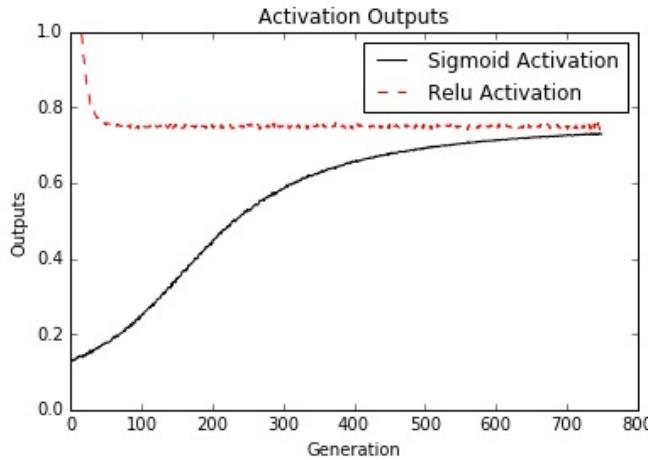


Figure 2: Computational graph outputs from the network with the sigmoid activation and a network with the ReLU activation

The two neural networks work with a similar architecture and target (0.75) but with two different activation functions, sigmoid, and ReLU. It is important to notice how much more rapidly the ReLU activation network converges to the desired target of 0.75 than the sigmoid activation, as shown in the following diagram:

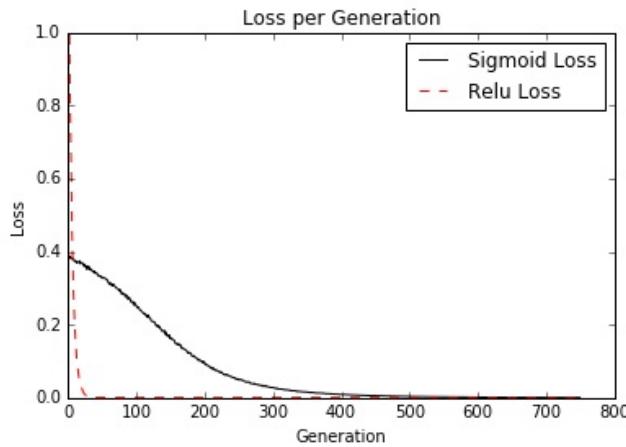


Figure 3: This figure depicts the loss value of the sigmoid and the ReLU activation networks. Notice how extreme the ReLU loss is at the beginning of the iterations

How it works...

Because of the form of the ReLU activation function, it returns the value of zero much more often than the sigmoid function. We consider this behavior as a type of sparsity. This sparsity results in a speeding up of convergence, but a loss of controlled gradients. On the other hand, the sigmoid function has very well-controlled gradients and does not risk the extreme values that the ReLU activation does, as illustrated in the following diagram:

Activation function	Advantages	Disadvantages
Sigmoid	Less extreme outputs	Slower convergence
ReLU	Converges quicker	Extreme output values possible

There's more...

In this section, we compared the ReLU activation function and the sigmoid activation function for neural networks. There are many other activation functions that are commonly-used for neural networks, but most fall into either one of two categories; the first category contains functions that are shaped like the sigmoid function, such as arctan, hypertangent, heaviside step, and so on; the second category contains functions that are shaped such as the ReLU function, such as softplus, leaky ReLU, and so on. Most of what we discussed in this section about comparing the two functions will hold true for activations in either category. However, it is important to note that the choice of the activation function has a big impact on the convergence and the output of neural networks.

Implementing a one-layer neural network

We have all of the tools needed to implement a neural network that operates on real data, so in this section we will create a neural network with one layer that operates on the `iris` dataset.

Getting ready

In this section, we will implement a neural network with one hidden layer. It will be important to understand that a fully connected neural network is based mostly on matrix multiplication. As such, it is important the dimensions of the data and matrix are lined up correctly.

Since this is a regression problem, we will use mean squared error as the loss function.

How to do it...

We proceed with the recipe as follows:

1. To create the computational graph, we'll start by loading the following necessary libraries:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn import datasets
```

2. Now we'll load the `Iris` data and store the length as the target value. Then we'll start a graph session with the following code:

```
iris = datasets.load_iris()
x_vals = np.array([x[0:3] for x in iris.data])
y_vals = np.array([x[3] for x in iris.data])
sess = tf.Session()
```

3. Since the dataset is smaller, we will want to set a seed to make the results reproducible, as follows:

```
seed = 2
tf.set_random_seed(seed)
np.random.seed(seed)
```

4. To prepare the data, we'll create a 80-20 train-test split and normalize the x features to be between 0 and 1 via min-max scaling, shown as follows:

```
train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]

def normalize_cols(m):
    col_max = m.max(axis=0)
    col_min = m.min(axis=0)
    return (m-col_min) / (col_max - col_min)

x_vals_train = np.nan_to_num(normalize_cols(x_vals_train))
x_vals_test = np.nan_to_num(normalize_cols(x_vals_test))
```

5. Now we will declare the batch size and placeholders for the data and target with the following code:

```
batch_size = 50
x_data = tf.placeholder(shape=[None, 3], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

- The important part is to declare our model variables with the appropriate shape. We can declare the size of our hidden layer to be any size we wish; in the following code block, we have set it to have five hidden nodes:

```
hidden_layer_nodes = 5
A1 = tf.Variable(tf.random_normal(shape=[3,hidden_layer_nodes]))
b1 = tf.Variable(tf.random_normal(shape=[hidden_layer_nodes]))
A2 = tf.Variable(tf.random_normal(shape=[hidden_layer_nodes,1]))
b2 = tf.Variable(tf.random_normal(shape=[1]))
```

- We'll now declare our model in two steps. The first step will be creating the hidden layer output and the second will be creating the `final_output` of the model, as follows:



As a note, our model goes from three input features to five hidden nodes, and finally to one output value.

```
hidden_output = tf.nn.relu(tf.add(tf.matmul(x_data, A1), b1))
final_output = tf.nn.relu(tf.add(tf.matmul(hidden_output, A2), b2))
```

- Our mean squared error as a `loss` function is as follows:

```
loss = tf.reduce_mean(tf.square(y_target - final_output))
```

- Now we'll declare our optimizing algorithm and initialize our variables with the following code:

```
my_opt = tf.train.GradientDescentOptimizer(0.005)
train_step = my_opt.minimize(loss)
init = tf.global_variables_initializer()
sess.run(init)
```

- Next, we loop through our training iterations. We'll also initialize two lists that we can store our train and `test_loss` function. In every loop, we also want to randomly select a batch from the training data for fitting to the model, shown as follows:

```
# First we initialize the loss vectors for storage.
loss_vec = []
test_loss = []
for i in range(500):
    # We select a random set of indices for the batch.
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
    # We then select the training values
    rand_x = x_vals_train[rand_index]
    rand_y = np.transpose([y_vals_train[rand_index]])
    # Now we run the training step
```

```

sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
# We save the training loss
temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
loss_vec.append(np.sqrt(temp_loss))

# Finally, we run the test-set loss and save it.
test_temp_loss = sess.run(loss, feed_dict={x_data: x_vals_test, y_target: np
test_loss.append(np.sqrt(test_temp_loss))
if (i+1)%50==0:
    print('Generation: ' + str(i+1) + '. Loss = ' + str(temp_loss))

```

11. We can plot the losses with `matplotlib` and the following code:

```

plt.plot(loss_vec, 'k-', label='Train Loss')
plt.plot(test_loss, 'r--', label='Test Loss')
plt.title('Loss (MSE) per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()

```

We proceed with the recipe by plotting the following diagram:

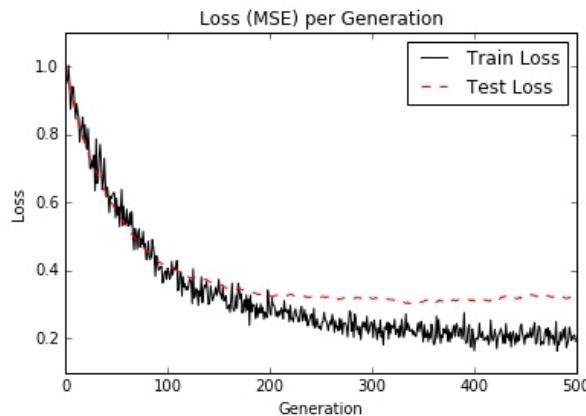


Figure 4: We plot the loss (MSE) of the train and test set. Notice that we are slightly over-fitting the model after 200 generations, as the test MSE does not drop any further but the training MSE does

How it works...

Our model has now been visualized as a neural network diagram, as shown in the following diagram:

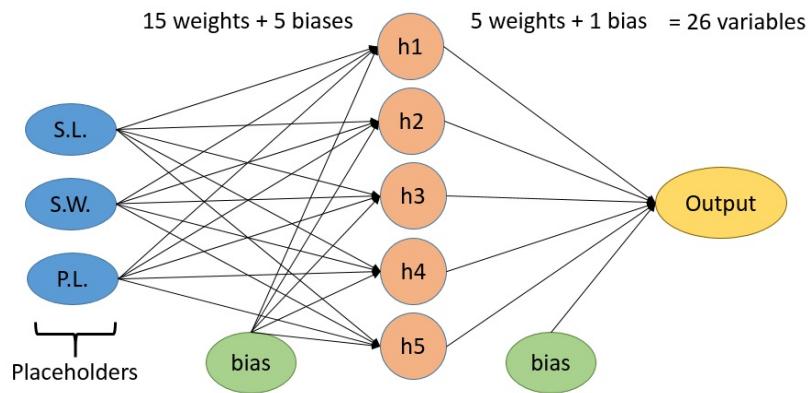


Figure 5: The preceding figure is a visualization of our neural network that has five nodes in the hidden layer. We are feeding in three values: the sepal length (S.L.), the sepal width (S.W.), and the petal length (P.L.). The target will be the petal width. In total, there will be 26 total variables in the model

There's more...

Note that we can identify when the model starts overfitting on the training data by viewing the `loss` function on the test and train sets. We can also see that the train set loss is not as smooth as that in the test set. This is because of two reasons: the first is that we are using a smaller batch size than the test set, although not by much; the second cause is due to the fact that we are training on the train set and the test set does not impact the variables of the model.

Implementing different layers

It is important to know how to implement different layers. In the preceding recipe, we implemented fully-connected layers. In this recipe, we will further expand our knowledge of various layers.

Getting ready

We have explored how to connect data inputs and a fully-connected hidden layer, but there are more types of layer that are built-in functions inside TensorFlow. The most popular layers that are used are convolutional layers and maxpool layers. We will show you how to create and use such layers with input data and with fully-connected data. First, we will look at how to use these layers on one-dimensional data, and then on two-dimensional data.

While neural networks can be layered in any fashion, one of the most common uses is to use convolutional layers and fully-connected layers to first create features. If we have too many features, it is common to have a maxpool layer. After these layers, non-linear layers are commonly introduced as activation functions. **Convolutional neural networks (CNNs)**, which we will consider in [chapter 8](#), *Convolutional Neural Networks*, usually have convolutional, maxpool, activation, convolutional, maxpool, and activation forms.

How to do it...

We will first look at one-dimensional data. We need to generate a random array of data for this task using the following steps:

1. We'll start by loading the libraries we need and starting a graph session, as follows:

```
import tensorflow as tf
import numpy as np
sess = tf.Session()
```

2. Now we can initialize our data (a NumPy array of length 25) and create the placeholder that we will feed it through with the following code:

```
data_size = 25
data_1d = np.random.normal(size=data_size)
x_input_1d = tf.placeholder(dtype=tf.float32, shape=[data_size])
```

3. Next, we will define a function that will make a convolutional layer. Then we will declare a random filter and create the convolutional layer, as follows:

Note that many of TensorFlow's layer functions are designed to deal with 4D data (4D = [batch size, width, height, and channels]). We need to modify our input data and the output data to extend or collapse the extra dimensions needed. For our example data, we have a batch size of 1, a width of 1, a height of 25, and a channel size of 1. To expand dimensions, we use the `expand_dims()` function, and to collapse dimensions, we use the `squeeze()` function. Also note that we can calculate the output dimensions of convolutional layers with the `output_size=(w-F+2P)/S+1` formula, where w is the input size, F is the filter size, P is the padding size, and S is the stride size.

```
i
def conv_layer_1d(input_1d, my_filter):
    # Make 1d input into 4d
    input_2d = tf.expand_dims(input_1d, 0)
    input_3d = tf.expand_dims(input_2d, 0)
    input_4d = tf.expand_dims(input_3d, 3)
    # Perform convolution
    convolution_output = tf.nn.conv2d(input_4d, filter=my_filter, strides=[1,1,1,1])
    # Now drop extra dimensions
    conv_output_1d = tf.squeeze(convolution_output)
    return(conv_output_1d)

my_filter = tf.Variable(tf.random_normal(shape=[1,5,1,1]))
my_convolution_output = conv_layer_1d(x_input_1d, my_filter)
```

4. TensorFlow's activation functions will act element-wise by default. This

means we just have to call our activation function on the layer of interest. We do this by creating an activation function and then initializing it on the graph, shown as follows:

```
def activation(input_1d):
    return tf.nn.relu(input_1d)
my_activation_output = activation(my_convolution_output)
```

- Now we'll declare a maxpool layer function. This function will create a maxpool on a moving window across our one-dimensional vector. For this example, we will initialize it to have a width of 5, shown as follows:

 *TensorFlow's maxpool arguments are very similar to those of the convolutional layer. While a maxpool argument does not have a filter, it does have size, stride, and padding options. Since we have a window of 5 with valid padding (no zero padding), then our output array will have 4 fewer entries.*

```
def max_pool(input_1d, width):
    # First we make the 1d input into 4d.
    input_2d = tf.expand_dims(input_1d, 0)
    input_3d = tf.expand_dims(input_2d, 0)
    input_4d = tf.expand_dims(input_3d, 3)
    # Perform the max pool operation
    pool_output = tf.nn.max_pool(input_4d, ksize=[1, 1, width, 1], strides=[1, 1
    pool_output_1d = tf.squeeze(pool_output)
    return pool_output_1d

my_maxpool_output = max_pool(my_activation_output, width=5)
```

- The final layer that we will connect is the fully-connected layer. Here, we want to create a versatile function that inputs a 1D array and outputs the number of values indicated. Also remember that to do matrix multiplication with a 1D array, we must expand the dimensions into 2D, as shown in the following code block:

```
def fully_connected(input_layer, num_outputs):
    # Create weights
    weight_shape = tf.squeeze(tf.stack([tf.shape(input_layer), [num_outputs]]))
    weight = tf.random_normal(weight_shape, stddev=0.1)
    bias = tf.random_normal(shape=[num_outputs])
    # Make input into 2d
    input_layer_2d = tf.expand_dims(input_layer, 0)
    # Perform fully connected operations
    full_output = tf.add(tf.matmul(input_layer_2d, weight), bias)
    # Drop extra dimensions
    full_output_1d = tf.squeeze(full_output)
    return full_output_1d

my_full_output = fully_connected(my_maxpool_output, 5)
```

- Now we'll initialize all the variables, run the graph, and print the output of each of the layers, as follows:

```

init = tf.global_variable_initializer()
sess.run(init)
feed_dict = {x_input_1d: data_1d}
# Convolution Output
print('Input = array of length 25')
print('Convolution w/filter, length = 5, stride size = 1, results in an array of')
print(sess.run(my_convolution_output, feed_dict=feed_dict))
# Activation Output
print('Input = the above array of length 21')
print('ReLU element wise returns the array of length 21:')
print(sess.run(my_activation_output, feed_dict=feed_dict))
# Maxpool Output
print('Input = the above array of length 21')
print('MaxPool, window length = 5, stride size = 1, results in the array of leng')
print(sess.run(my_maxpool_output, feed_dict=feed_dict))
# Fully Connected Output
print('Input = the above array of length 17')
print('Fully connected layer on all four rows with five outputs:')
print(sess.run(my_full_output, feed_dict=feed_dict))

```

8. The preceding step should result in the following output:

```

Input = array of length 25
Convolution w/filter, length = 5, stride size = 1, results in an array of length
[-0.91608119  1.53731811 -0.7954089   0.5041104   1.88933098
 -1.81099761  0.56695032  1.17945457 -0.66252393 -1.90287709
  0.87184119  0.84611893 -5.25024986 -0.05473572  2.19293165
 -4.47577858 -1.71364677  3.96857905 -2.0452652  -1.86647367
 -0.12697852]
Input = the above array of length 21
ReLU element wise returns the array of length 21:
[ 0.          1.53731811  0.          0.5041104   1.88933098
  0.          0.          1.17945457  0.          0.
  0.87184119  0.84611893  0.          0.          2.19293165
  0.          0.          3.96857905  0.          0.
  0.          ]
Input = the above array of length 21
MaxPool, window length = 5, stride size = 1, results in the array of length 17:
[ 1.88933098  1.88933098  1.88933098  1.88933098  1.88933098
  1.17945457  1.17945457  1.17945457  0.87184119  0.87184119
  2.19293165  2.19293165  2.19293165  3.96857905  3.96857905
  3.96857905  3.96857905]
Input = the above array of length 17
Fully connected layer on all four rows with five outputs:
[ 1.23588216 -0.42116445  1.44521213  1.40348077 -0.79607368]

```



One-dimensional data is very important to consider for neural networks. Time series, signal processing, and some text embeddings are considered to be one-dimensional and are frequently used in neural networks.

We will now consider the same types of layer in an equivalent order but for two-dimensional data:

1. We will start by clearing and resetting the computational graph, as follows:

```

ops.reset_default_graph()
sess = tf.Session()

```

- Then we will initialize our input array so that it is a 10x10 matrix, and we will then initialize a placeholder for the graph with the same shape, as follows:

```
data_size = [10,10]
data_2d = np.random.normal(size=data_size)
x_input_2d = tf.placeholder(dtype=tf.float32, shape=data_size)
```

- Just as in the one-dimensional example, we now need to declare a convolutional layer function. Since our data has a height and width already, we just need to expand it in two dimensions (a batch size of 1, and a channel size of 1) so that we can operate on it with the `conv2d()` function. For the filter, we will use a random 2x2 filter, a stride of 2 in both directions, and valid padding (in other words, no zero padding). Because our input matrix is 10x10, our convolutional output will be 5x5, shown as follows:

```
def conv_layer_2d(input_2d, my_filter):
    # First, change 2d input to 4d
    input_3d = tf.expand_dims(input_2d, 0)
    input_4d = tf.expand_dims(input_3d, 3)
    # Perform convolution
    convolution_output = tf.nn.conv2d(input_4d, filter=my_filter, strides=[1,2,2,1])
    # Drop extra dimensions
    conv_output_2d = tf.squeeze(convolution_output)
    return(conv_output_2d)

my_filter = tf.Variable(tf.random_normal(shape=[2,2,1,1]))
my_convolution_output = conv_layer_2d(x_input_2d, my_filter)
```

- The activation function works on an element-wise basis, so we can now create an activation operation and initialize it on the graph with the following code:

```
def activation(input_2d):
    return tf.nn.relu(input_2d)
my_activation_output = activation(my_convolution_output)
```

- Our maxpool layer is very similar to the one-dimensional case, except we have to declare a width and height for the maxpool window. Just as with our convolutional 2D layer, we only have to expand out into in two dimensions, shown as follows:

```
def max_pool(input_2d, width, height):
    # Make 2d input into 4d
    input_3d = tf.expand_dims(input_2d, 0)
    input_4d = tf.expand_dims(input_3d, 3)
    # Perform max pool
    pool_output = tf.nn.max_pool(input_4d, ksize=[1, height, width, 1], strides=[1,1,1,1])
    # Drop extra dimensions
    pool_output_2d = tf.squeeze(pool_output)
```

```

    return pool_output_2d
my_maxpool_output = max_pool(my_activation_output, width=2, height=2)

```

- Our fully-connected layer is very similar to the one-dimensional output. We should also note here that the 2D input to this layer is considered as one object, so we want each of the entries connected to each of the outputs. In order to accomplish this, we need to fully flatten out the two-dimensional matrix and then expand it for matrix multiplication, as follows:

```

def fully_connected(input_layer, num_outputs):
    # Flatten into 1d
    flat_input = tf.reshape(input_layer, [-1])
    # Create weights
    weight_shape = tf.squeeze(tf.stack([tf.shape(flat_input), [num_outputs]]))
    weight = tf.random_normal(weight_shape, stddev=0.1)
    bias = tf.random_normal(shape=[num_outputs])
    # Change into 2d
    input_2d = tf.expand_dims(flat_input, 0)
    # Perform fully connected operations
    full_output = tf.add(tf.matmul(input_2d, weight), bias)
    # Drop extra dimensions
    full_output_2d = tf.squeeze(full_output)
    return full_output_2d

my_full_output = fully_connected(my_maxpool_output, 5)

```

- Now we need to initialize our variables and create a feed dictionary for our operations using the following code:

```

init = tf.global_variables_initializer()
sess.run(init)

feed_dict = {x_input_2d: data_2d}

```

- The outputs for each of the layers should look as follows:

```

# Convolution Output
print('Input = [10 X 10] array')
print('2x2 Convolution, stride size = [2x2], results in the [5x5] array:')
print(sess.run(my_convolution_output, feed_dict=feed_dict))
# Activation Output
print('Input = the above [5x5] array')
print('ReLU element wise returns the [5x5] array:')
print(sess.run(my_activation_output, feed_dict=feed_dict))
# Max Pool Output
print('Input = the above [5x5] array')
print('MaxPool, stride size = [1x1], results in the [4x4] array:')
print(sess.run(my_maxpool_output, feed_dict=feed_dict))
# Fully Connected Output
print('Input = the above [4x4] array')
print('Fully connected layer on all four rows with five outputs:')
print(sess.run(my_full_output, feed_dict=feed_dict))

```

- The preceding step should result in the following output:

```
Input = [10 X 10] array
2x2 Convolution, stride size = [2x2], results in the [5x5] array:
[[ 0.37630892 -1.41018617 -2.58821273 -0.32302785  1.18970704]
 [-4.33685207  1.97415686  1.0844903   -1.18965471  0.84643292]
 [ 5.23706436  2.46556497 -0.95119286  1.17715418  4.1117816 ]
 [ 5.86972761  1.2213701   1.59536231  2.66231227  2.28650784]
 [-0.88964868 -2.75502229  4.3449688   2.67776585 -2.23714781]]
Input = the above [5x5] array
ReLU element wise returns the [5x5] array:
[[ 0.37630892  0.          0.          0.          1.18970704]
 [ 0.          1.97415686  1.0844903   0.          0.84643292]
 [ 5.23706436  2.46556497  0.          1.17715418  4.1117816 ]
 [ 5.86972761  1.2213701   1.59536231  2.66231227  2.28650784]
 [ 0.          0.          4.3449688   2.67776585  0.          ]]
Input = the above [5x5] array
MaxPool, stride size = [1x1], results in the [4x4] array:
[[ 1.97415686  1.97415686  1.0844903   1.18970704]
 [ 5.23706436  2.46556497  1.17715418  4.1117816 ]
 [ 5.86972761  2.46556497  2.66231227  4.1117816 ]
 [ 5.86972761  4.3449688   4.3449688   2.67776585]]
Input = the above [4x4] array
Fully connected layer on all four rows with five outputs:
[-0.6154139  -1.96987963 -1.88811922  0.20010889  0.32519674]
```

How it works...

We should now know how to use the convolutional and maxpool layers in TensorFlow with one-dimensional and two-dimensional data. Regardless of the shape of the input, we ended up with the same size output. This is important for illustrating the flexibility of neural network layers. This section should also impress upon us again the importance of shapes and sizes in neural network operations.

Using a multilayer neural network

We will now apply our knowledge of different layers to real data by using a multilayer neural network on the low birth weight dataset.

Getting ready

Now that we know how to create neural networks and work with layers, we will apply this methodology with the aim of predicting the birth weight in the low birth weight dataset. We'll create a neural network with three hidden layers. The low birth weight dataset includes the actual birth weight and an indicator variable for whether the birth weight is above or below 2,500 grams. In this example, we'll make the target the actual birth weight (regression) and then see what the accuracy is on the classification at the end. At the end, our model should be able to identify whether the birth weight will be <2,500 grams.

How to do it...

We proceed with the recipe as follows:

1. We will start by loading the libraries and initializing our computational graph as follows:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import os
import csv
import requests
import numpy as np
sess = tf.Session()
```

2. We'll now load the data from the website using the `requests` module. After this, we will split the data into features of interest and the target value, shown as follows:

```
# Name of data file
birth_weight_file = 'birth_weight.csv'
birthdata_url = 'https://github.com/nfmcclure/tensorflow_cookbook/raw/master' \
    '/01_Introduction/07_Working_with_Data_Sources/birthweight_data/birthweight.dat'

# Download data and create data file if file does not exist in current directory
if not os.path.exists(birth_weight_file):
    birth_file = requests.get(birthdata_url)
    birth_data = birth_file.text.split('\r\n')
    birth_header = birth_data[0].split('\t')
    birth_data = [[float(x) for x in y.split('\t') if len(x) >= 1]
                  for y in birth_data[1:] if len(y) >= 1]
    with open(birth_weight_file, "w") as f:
        writer = csv.writer(f)
        writer.writerow([birth_header])
        writer.writerows(birth_data)

# Read birth weight data into memory
birth_data = []
with open(birth_weight_file, newline='') as csvfile:
    csv_reader = csv.reader(csvfile)
    birth_header = next(csv_reader)
    for row in csv_reader:
        birth_data.append(row)

birth_data = [[float(x) for x in row] for row in birth_data]

# Pull out target variable
y_vals = np.array([x[0] for x in birth_data])
# Pull out predictor variables (not id, not target, and not birthweight)
x_vals = np.array([x[1:8] for x in birth_data])
```

3. To help with repeatability, we now need to set the random seed for both

NumPy and TensorFlow. Then we declare our batch size as follows:

```
seed = 4
tf.set_random_seed(seed)
np.random.seed(seed)
batch_size = 100
```

4. Next, we split the data into an 80-20 train-test split. After this, we need to normalize our input features so that they are between 0 and 1 with min-max scaling, shown as follows:

```
train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]

# Normalize by column (min-max norm)
def normalize_cols(m, col_min=np.array([None]), col_max=np.array([None])):
    if not col_min[0]:
        col_min = m.min(axis=0)
    if not col_max[0]:
        col_max = m.max(axis=0)
    return (m-col_min) / (col_max - col_min), col_min, col_max

x_vals_train, train_min, train_max = np.nan_to_num(normalize_cols(x_vals_train))
x_vals_test, _, _ = np.nan_to_num(normalize_cols(x_vals_test), train_min, train_max)
```

 *Normalizing input features is a common feature transformation, and especially useful for neural networks. It will help with convergence if our data is centered near 0 to 1 for the activation functions.*

5. Since we have multiple layers that have similar initialized variables, we now need to create a function to initialize both the weights and the bias. We do that with the following code:

```
def init_weight(shape, st_dev):
    weight = tf.Variable(tf.random_normal(shape, stddev=st_dev))
    return weight

def init_bias(shape, st_dev):
    bias = tf.Variable(tf.random_normal(shape, stddev=st_dev))
    return bias
```

6. We now need to initialize our placeholders. There will be eight input features and one output, the birth weight in grams, as follows:

```
x_data = tf.placeholder(shape=[None, 8], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

7. The fully-connected layer will be used three times for all three hidden

layers. To prevent repeated code, we will create a layer function for use when we initialize our model, shown as follows:

```
def fully_connected(input_layer, weights, biases):
    layer = tf.add(tf.matmul(input_layer, weights), biases)
    return tf.nn.relu(layer)
```

- Now it's time to create our model. For each layer (and output layer), we will initialize a weight matrix, bias matrix, and the fully-connected layer. For this example, we will use hidden layers of sizes 25, 10, and 3:



The model that we are using will have 522 variables to fit. To arrive at this number, we can see that between the data and the first hidden layer we have $8 \times 25 + 25 = 225$ variables. If we continue in this way and add them up, we'll have $225 + 260 + 33 + 4 = 522$ variables. This is significantly larger than the nine variables that we used in the logistic regression model on this data.

```
# Create second layer (25 hidden nodes)
weight_1 = init_weight(shape=[8, 25], st_dev=10.0)
bias_1 = init_bias(shape=[25], st_dev=10.0)
layer_1 = fully_connected(x_data, weight_1, bias_1)

# Create second layer (10 hidden nodes)
weight_2 = init_weight(shape=[25, 10], st_dev=10.0)
bias_2 = init_bias(shape=[10], st_dev=10.0)
layer_2 = fully_connected(layer_1, weight_2, bias_2)

# Create third layer (3 hidden nodes)
weight_3 = init_weight(shape=[10, 3], st_dev=10.0)
bias_3 = init_bias(shape=[3], st_dev=10.0)
layer_3 = fully_connected(layer_2, weight_3, bias_3)
# Create output layer (1 output value)
weight_4 = init_weight(shape=[3, 1], st_dev=10.0)
bias_4 = init_bias(shape=[1], st_dev=10.0)
final_output = fully_connected(layer_3, weight_4, bias_4)
```

- We'll now use the L1 loss function (the absolute value), declare our optimizer (using Adam optimization), and initialize our variables as follows:

```
loss = tf.reduce_mean(tf.abs(y_target - final_output))
my_opt = tf.train.AdamOptimizer(0.05)
train_step = my_opt.minimize(loss)
init = tf.global_variables_initializer()
sess.run(init)
```



While the learning rate we used in the preceding step for the Adam optimization function is 0.05, there is research that suggests a lower learning rate consistently produces better results. For this recipe, we have used a larger learning rate because of the consistency of the data and the need for quick convergence.

- Next, we need to train our model for 200 iterations. We'll also include code that will store the `train` and `test` loss, select a random batch size, and print

the status every 25 generations, shown as follows:

```
# Initialize the loss vectors
loss_vec = []
test_loss = []
for i in range(200):
    # Choose random indices for batch selection
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
    # Get random batch
    rand_x = x_vals_train[rand_index]
    rand_y = np.transpose([y_vals_train[rand_index]])
    # Run the training step
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    # Get and store the train loss
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss)
    # Get and store the test loss
    test_temp_loss = sess.run(loss, feed_dict={x_data: x_vals_test, y_target: np
    test_loss.append(test_temp_loss)
    if (i+1)%25==0:
        print('Generation: ' + str(i+1) + '. Loss = ' + str(temp_loss))
```

11. The preceding step should result in the following output:

```
Generation: 25. Loss = 5922.52
Generation: 50. Loss = 2861.66
Generation: 75. Loss = 2342.01
Generation: 100. Loss = 1880.59
Generation: 125. Loss = 1394.39
Generation: 150. Loss = 1062.43
Generation: 175. Loss = 834.641
Generation: 200. Loss = 848.54
```

12. The following is a snippet of code that plots the train and test loss with `matplotlib`:

```
plt.plot(loss_vec, 'k-', label='Train Loss')
plt.plot(test_loss, 'r--', label='Test Loss')
plt.title('Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```

We proceed with the recipe by plotting the following diagram:

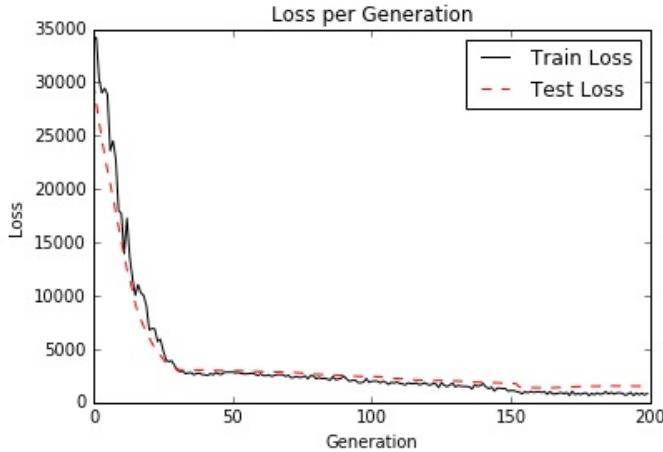


Figure 6: In the preceding figure, we plot the train and test losses for our neural network that we trained to predict birth weight in grams. Notice that we have arrived at a good model after approximately 30 generations

13. We now want to compare our birth weight results to our prior logistic results. With logistic linear regression (as seen in the *Implementing Logistic Regression* recipe in [Chapter 3, Linear Regression](#)), we achieved a result of around 60% accuracy after thousands of iterations. To compare this with what we did in the preceding section, we need to output the train and test regression results and turn them into classification results by creating an indicator if they are above or below 2,500 grams. To find out the model's accuracy, we need to use the following code:

```

actuals = np.array([x[1] for x in birth_data])
test_actuals = actuals[test_indices]
train_actuals = actuals[train_indices]
test_preds = [x[0] for x in sess.run(final_output, feed_dict={x_data: x_vals_tes})
train_preds = [x[0] for x in sess.run(final_output, feed_dict={x_data: x_vals_tr})
test_preds = np.array([1.0 if x<2500.0 else 0.0 for x in test_preds])
train_preds = np.array([1.0 if x<2500.0 else 0.0 for x in train_preds])
# Print out accuracies
test_acc = np.mean([x==y for x,y in zip(test_preds, test_actuals)])
train_acc = np.mean([x==y for x,y in zip(train_preds, train_actuals)])
print('On predicting the category of low birthweight from regression output (<25')
print('Test Accuracy: {}'.format(test_acc))
print('Train Accuracy: {}'.format(train_acc))

```

14. The preceding step should result in the following output:

```

Test Accuracy: 0.631578947368421
Train Accuracy: 0.7019867549668874

```

How it works...

In this recipe, we created a regression neural network with three fully-connected hidden layers to predict the birth weight of the low birth weight data set. When comparing this to a logistic output to predict above or below 2,500 grams, we achieved similar results and achieved them in fewer generations. In the next recipe, we will try to improve our logistic regression by making it a multiple-layer, logistic-type neural network.

Improving the predictions of linear models

In the preceding recipes, we have noted that the number of parameters we are fitting far exceeds the equivalent linear models. In this recipe, we will attempt to improve our logistic model of low birth weight by using a neural network.

Getting ready

For this recipe, we will load the low birth weight data and use a neural network with two hidden fully-connected layers with sigmoid activations to fit the probability of a low birth weight.

How to do it

We proceed with the recipe as follows:

1. We start by loading the libraries and initializing our computational graph as follows:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import requests
sess = tf.Session()
```

2. Next, we load, extract, and normalize our data as in the preceding recipe, except that here we are going to using the low birth weight indicator variable as our target instead of the actual birth weight, shown as follows:

```
# Name of data file
birth_weight_file = 'birth_weight.csv'
birthdata_url = 'https://github.com/nfmccclure/tensorflow_cookbook/raw/master' \
    '/01_Introduction/07_Working_with_Data_Sources/birthweight_data/'

# Download data and create data file if file does not exist in current directory
if not os.path.exists(birth_weight_file):
    birth_file = requests.get(birthdata_url)
    birth_data = birth_file.text.split('\r\n')
    birth_header = birth_data[0].split('\t')
    birth_data = [[float(x) for x in y.split('\t') if len(x) >= 1]
                  for y in birth_data[1:] if len(y) >= 1]
    with open(birth_weight_file, "w") as f:
        writer = csv.writer(f)
        writer.writerow([birth_header])
        writer.writerows(birth_data)

# read birth weight data into memory
birth_data = []
with open(birth_weight_file, newline='') as csvfile:
    csv_reader = csv.reader(csvfile)
    birth_header = next(csv_reader)
    for row in csv_reader:
        birth_data.append(row)

birth_data = [[float(x) for x in row] for row in birth_data]

# Pull out target variable
y_vals = np.array([x[0] for x in birth_data])
# Pull out predictor variables (not id, not target, and not birthweight)
x_vals = np.array([x[1:8] for x in birth_data])

train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
```

```

y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]

def normalize_cols(m, col_min=np.array([None]), col_max=np.array([None])):
    if not col_min[0]:
        col_min = m.min(axis=0)
    if not col_max[0]:
        col_max = m.max(axis=0)
    return (m - col_min) / (col_max - col_min), col_min, col_max

x_vals_train, train_min, train_max = np.nan_to_num(normalize_cols(x_vals_train))
x_vals_test, _, _ = np.nan_to_num(normalize_cols(x_vals_test, train_min, train_max))

```

3. Next, we need to declare our batch size and our placeholders for the data as follows:

```

batch_size = 90
x_data = tf.placeholder(shape=[None, 7], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)

```

4. As previously, we now need to declare functions that initialize a variable and a layer in our model. To create a better logistic function, we need to create a function that returns a logistic layer on an input layer. In other words, we will just use a fully-connected layer and return a sigmoid element for each layer. It is important to remember that our loss function will have the final sigmoid included, so we want to specify on our last layer that we will not return the sigmoid of the output, shown as follows:

```

def init_variable(shape):
    return tf.Variable(tf.random_normal(shape=shape))
# Create a logistic layer definition
def logistic(input_layer, multiplication_weight, bias_weight, activation = True)
    linear_layer = tf.add(tf.matmul(input_layer, multiplication_weight), bias_we

    if activation:
        return tf.nn.sigmoid(linear_layer)
    else:
        return linear_layer

```

5. Now we will declare three layers (two hidden layers and an output layer). We will start by initializing a weight and bias matrix for each layer and defining the layer operations as follows:

```

# First logistic layer (7 inputs to 14 hidden nodes)
A1 = init_variable(shape=[7,14])
b1 = init_variable(shape=[14])
logistic_layer1 = logistic(x_data, A1, b1)

# Second logistic layer (14 hidden inputs to 5 hidden nodes)
A2 = init_variable(shape=[14,5])
b2 = init_variable(shape=[5])
logistic_layer2 = logistic(logistic_layer1, A2, b2)
# Final output layer (5 hidden nodes to 1 output)

```

```

A3 = init_variable(shape=[5,1])
b3 = init_variable(shape=[1])
final_output = logistic(logistic_layer2, A3, b3, activation=False)

```

6. Next, we declare our loss (cross entropy) and optimization algorithm, and initialize the following variables:

```

# Create loss function
loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=final_outpu
# Declare optimizer
my_opt = tf.train.AdamOptimizer(learning_rate = 0.002)
train_step = my_opt.minimize(loss)
# Initialize variables
init = tf.global_variables_initializer()
sess.run(init)

```

Cross entropy is a way of measuring distances between probabilities. Here, we want to measure the difference between certainty (0 or 1) and our model probability ($0 < x < 1$).

 *TensorFlow implements cross entropy with the sigmoid function built in. This is also important as part of the hyperparameter tuning, as we are more likely to find the best loss function, learning rate, and optimization algorithm for the problem at hand. For brevity in this recipe, we do not include hyperparameter tuning.*

7. In order to evaluate and compare our model to previous models, we need to create a prediction and accuracy operation on the graph. This will allow us to feed in the whole test set and determine the accuracy, as follows:

```

prediction = tf.round(tf.nn.sigmoid(final_output))
predictions_correct = tf.cast(tf.equal(prediction, y_target), tf.float32)
accuracy = tf.reduce_mean(predictions_correct)

```

8. We are now ready to start our training loop. We will train for 1,500 generations and save the model loss and train and test accuracies for plotting later. Our training loop is started with the following code:

```

# Initialize loss and accuracy vectors loss_vec = [] train_acc = [] test_acc = []
for i in range(1500):
    # Select random indices for batch selection
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
    # Select batch
    rand_x = x_vals_train[rand_index]
    rand_y = np.transpose([y_vals_train[rand_index]])
    # Run training step
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    # Get training loss
    temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
    loss_vec.append(temp_loss)
    # Get training accuracy
    temp_acc_train = sess.run(accuracy, feed_dict={x_data: x_vals_train, y_targe
    train_acc.append(temp_acc_train)
    # Get test accuracy
    temp_acc_test = sess.run(accuracy, feed_dict={x_data: x_vals_test, y_target:
    test_acc.append(temp_acc_test)
    if (i+1)%150==0:
        print('Loss = ' + str(temp_loss)))

```

9. The preceding step should result in the following output:

```
Loss = 0.696393
Loss = 0.591708
Loss = 0.59214
Loss = 0.505553
Loss = 0.541974
Loss = 0.512707
Loss = 0.590149
Loss = 0.502641
Loss = 0.518047
Loss = 0.502616
```

10. The following code blocks illustrate how to plot the cross entropy loss and train and test set accuracies with `matplotlib`:

```
# Plot loss over time
plt.plot(loss_vec, 'k-')
plt.title('Cross Entropy Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Cross Entropy Loss')
plt.show()
# Plot train and test accuracy
plt.plot(train_acc, 'k-', label='Train Set Accuracy')
plt.plot(test_acc, 'r--', label='Test Set Accuracy')
plt.title('Train and Test Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()
```

We get the plot for cross entropy loss per generation as follows:

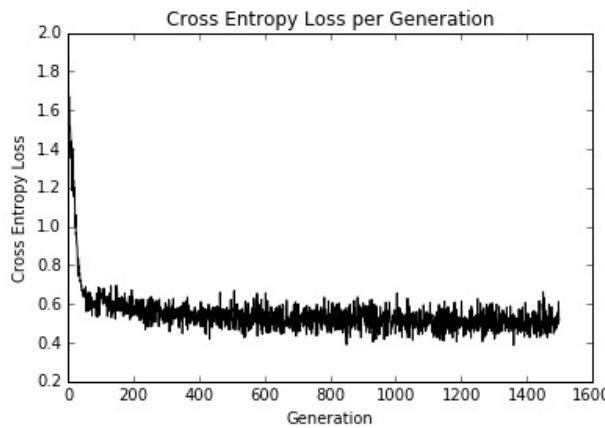


Figure 7: Training loss over 1500 iterations

Within approximately 50 generations, we have reached a good model. As we continue to train, we can see that very little is gained over the remaining iterations, as shown in the following diagram:

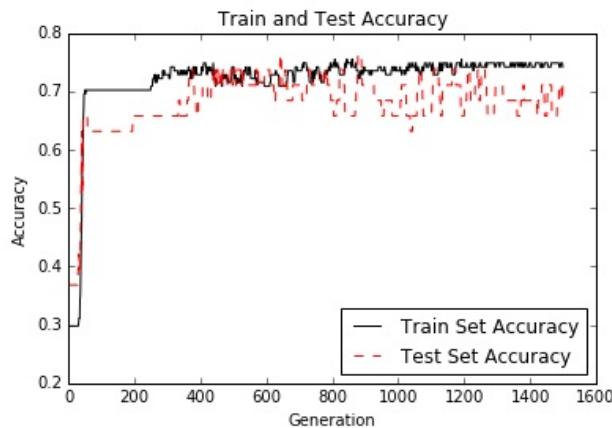


Figure 8: Accuracy for the train set and test set

As you can see in the preceding diagram, we arrived at a good model very quickly.

How it works...

When considering using neural networks to model data, you have to consider the advantages and disadvantages. While our model has converged faster than previous models, and perhaps with greater accuracy, this comes with a price; we are training many more model variables and there is a greater chance of overfitting. To check if overfitting is occurring, we look at the accuracy of the test and train sets. If the accuracy of the training set continues to increase while the accuracy on the test set stays the same or even decreases slightly, we can assume over-fitting is occurring.

To combat underfitting, we can increase our model depth or train the model for more iterations. To address overfitting, we can add more data or add regularization techniques to our model.

It is also important to note that our model variables are not as interpretable as a linear model. Neural network models have coefficients that are harder to interpret than linear models, as they explain the significance of features within the model.

Learning to play Tic Tac Toe

To show how adaptable neural networks can be, we will now attempt to use a neural network in order to learn the optimal moves for Tic Tac Toe. We will approach this knowing that Tic Tac Toe is a deterministic game and that the optimal moves are already known.

Getting ready

To train our model, we will use a list of board positions followed by the best optimal response for a number of different boards. We can reduce the amount of boards to train on by considering only board positions that are different with regard to symmetry. The non-identity transformations of a Tic Tac Toe board are a rotation (in either direction) of 90 degrees, 180 degrees, and 270 degrees, a horizontal reflection, and a vertical reflection. Given this idea, we will use a shortlist of boards with the optimal move, apply two random transformations, and then feed that into our neural network for learning.



Since Tic Tac Toe is a deterministic game, it is worth noting that whoever goes first should either win or draw. We will hope for a model that can respond to our moves optimally and ultimately result in a draw.

If we annotate Xs by 1, Os by -1, and empty spaces by 0, then the following diagram illustrates how we can consider a board position and an optimal move as a row of data:

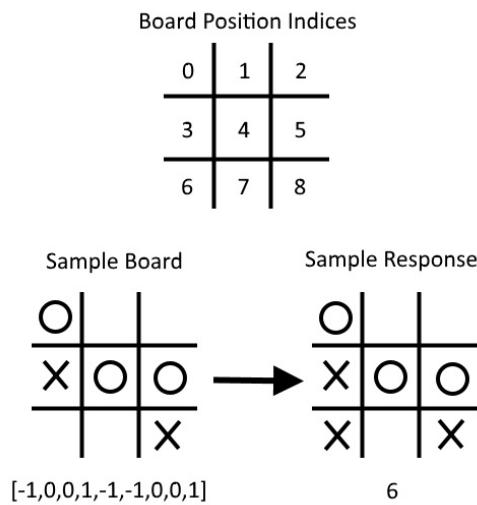


Figure 9: Here, we illustrate how to consider a board and an optimal move as a row of data. Note that X = 1, O = -1, and empty spaces are 0, and we start indexing at 0

In addition to the model loss, to check how our model is performing we will do two things. The first check we will perform is to remove a position and an optimal move row from our training set. This will allow us to see if the neural

network model can generalize a move it hasn't seen before. The second method we will take to evaluate our model is to actually play a game against it at the end.

The list of possible boards and optimal moves can be found on the GitHub directory for this recipe here: https://github.com/nfmcclure/tensorflow_cookbook/tree/master/06_Neural_Networks/08_Learning_Tic_Tac_Toe and in the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

How to do it...

We proceed with the recipe as follows:

1. We need to start by loading the necessary libraries for this script, as follows:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import csv
import random
import numpy as np
import random
```

2. Next, we declare the following batch size for training our model:

```
| batch_size = 50
```

3. To make visualizing the boards a bit easier, we will create a function that outputs Tic Tac Toe boards with Xs and Os. This is done with the following code:

```
def print_board(board):
    symbols = ['0', ' ', 'X']
    board_plus1 = [int(x) + 1 for x in board]
    board_line1 = ' {} | {} | {}'.format(symbols[board_plus1[0]],
                                           symbols[board_plus1[1]],
                                           symbols[board_plus1[2]])
    board_line2 = ' {} | {} | {}'.format(symbols[board_plus1[3]],
                                           symbols[board_plus1[4]],
                                           symbols[board_plus1[5]])
    board_line3 = ' {} | {} | {}'.format(symbols[board_plus1[6]],
                                           symbols[board_plus1[7]],
                                           symbols[board_plus1[8]])
    print(board_line1)
    print('_____')
    print(board_line2)
    print('_____')
    print(board_line3)
```

4. Now we have to create a function that will return a new board and an optimal response position under a transformation. This is done with the following code:

```
def get_symmetry(board, response, transformation):
    '''
    :param board: list of integers 9 long:
    opposing mark = -1
    friendly mark = 1
```

```

empty space = 0
:param transformation: one of five transformations on a board:
    rotate180, rotate90, rotate270, flip_v, flip_h
:return: tuple: (new_board, new_response)
'''

if transformation == 'rotate180':
    new_response = 8 - response
    return board[::-1], new_response

elif transformation == 'rotate90':
    new_response = [6, 3, 0, 7, 4, 1, 8, 5, 2].index(response)
    tuple_board = list(zip(*[board[6:9], board[3:6], board[0:3]]))
    return [value for item in tuple_board for value in item], new_response

elif transformation == 'rotate270':
    new_response = [2, 5, 8, 1, 4, 7, 0, 3, 6].index(response)
    tuple_board = list(zip(*[board[0:3], board[3:6], board[6:9]]))[::-1]
    return [value for item in tuple_board for value in item], new_response

elif transformation == 'flip_v':
    new_response = [6, 7, 8, 3, 4, 5, 0, 1, 2].index(response)
    return board[6:9] + board[3:6] + board[0:3], new_response

elif transformation == 'flip_h':
# flip_h = rotate180, then flip_v
    new_response = [2, 1, 0, 5, 4, 3, 8, 7, 6].index(response)
    new_board = board[::-1]
    return new_board[6:9] + new_board[3:6] + new_board[0:3], new_response

else:
    raise ValueError('Method not implemented.')

```

5. The list of boards and their optimal responses is in a .csv file in the directory, available at the github repository https://github.com/nfmccleure/tensorflow_cookbook or the Packt repository <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>. We will create a function that will load the file with the boards and responses and will store it as a list of tuples, as follows:

```

def get_moves_from_csv(csv_file):
    '''
    :param csv_file: csv file location containing the boards w/ responses
    :return: moves: list of moves with index of best response
    '''

    moves = []
    with open(csv_file, 'rt') as csvfile:
        reader = csv.reader(csvfile, delimiter=',')
        for row in reader:
            moves.append(([int(x) for x in row[0:9]], int(row[9])))
    return moves

```

6. Now we need to tie everything together to create a function that will return a randomly-transformed board and response. This is done with the following code:

```

def get_rand_move(moves, rand_transforms=2):
    # This function performs random transformations on a board.
    (board, response) = random.choice(moves)
    possible_transforms = ['rotate90', 'rotate180', 'rotate270', 'flip_v', 'flip_h']
    for i in range(rand_transforms):
        random_transform = random.choice(possible_transforms)
        (board, response) = get_symmetry(board, response, random_transform)
    return board, response

```

7. Next, we need to initialize our graph session, load our data, and create a training set as follows:

```

sess = tf.Session()
moves = get_moves_from_csv('base_tic_tac_toe_moves.csv')
# Create a train set:
train_length = 500
train_set = []
for t in range(train_length):
    train_set.append(get_rand_move(moves))

```

8. Remember that we want to remove one board and an optimal response from our training set to see if the model can generalize making the best move. The best move for the following board will be to play at index number 6:

```

test_board = [-1, 0, 0, 1, -1, -1, 0, 0, 1]
train_set = [x for x in train_set if x[0] != test_board]

```

9. We can now create functions to create our model variables and our model operations. Note that we do not include the `softmax()` activation function in the following model because it is included in the loss function:

```

def init_weights(shape):
    return tf.Variable(tf.random_normal(shape))

def model(X, A1, A2, bias1, bias2):
    layer1 = tf.nn.sigmoid(tf.add(tf.matmul(X, A1), bias1))
    layer2 = tf.add(tf.matmul(layer1, A2), bias2)
    return layer2

```

10. Now we need to declare our placeholders, variables, and model as follows:

```

X = tf.placeholder(dtype=tf.float32, shape=[None, 9])
Y = tf.placeholder(dtype=tf.int32, shape=[None])
A1 = init_weights([9, 81])
bias1 = init_weights([81])
A2 = init_weights([81, 9])
bias2 = init_weights([9])
model_output = model(X, A1, A2, bias1, bias2)

```

11. Next, we need to declare our `loss` function, which will be the average softmax of the final output logits (unstandardized output). Then we will declare our training step and optimizer. We also need to create a prediction

operation if we want to be able to play against our model in the future, as follows:

```
loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits=mode
train_step = tf.train.GradientDescentOptimizer(0.025).minimize(loss)
prediction = tf.argmax(model_output, 1)
```

12. We can now initialize our variables and loop through the training of our neural network with the following code:

```
# Initialize variables
init = tf.global_variables_initializer()
sess.run(init)
loss_vec = []
for i in range(10000):
    # Select random indices for batch
    rand_indices = np.random.choice(range(len(train_set)), batch_size, replace=False)
    # Get batch
    batch_data = [train_set[i] for i in rand_indices]
    x_input = [x[0] for x in batch_data]
    y_target = np.array([y[1] for y in batch_data])
    # Run training step
    sess.run(train_step, feed_dict={X: x_input, Y: y_target})
    # Get training loss
    temp_loss = sess.run(loss, feed_dict={X: x_input, Y: y_target})
    loss_vec.append(temp_loss)

    if i%500==0:
        print('iteration ' + str(i) + ' Loss: ' + str(temp_loss))
```

13. The following is the code needed to plot the loss over the model training:

```
plt.plot(loss_vec, 'k-', label='Loss')
plt.title('Loss (MSE) per Generation')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```

We should get the following plot for the loss per generation:

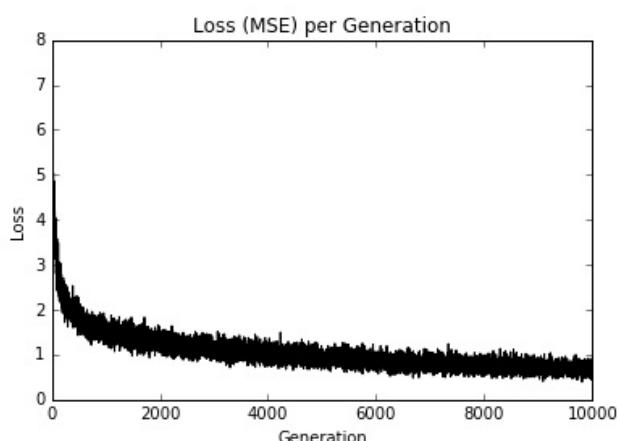


Figure 10: A Tic-Tac-Toe train set loss over 10,000 iterations

In the preceding diagram, we have plotted the loss over the training steps.

14. To test the model, we need to see how it performs on the test board that we removed from the training set. We are hoping that the model can generalize and predict the optimal index for moving, which will be index number 6. Most of the time the model will succeed, shown as follows:

```
test_boards = [test_board]
feed_dict = {X: test_boards}
logits = sess.run(model_output, feed_dict=feed_dict)
predictions = sess.run(prediction, feed_dict=feed_dict)
print(predictions)
```

15. The preceding step should result in the following output:

```
| [6]
```

16. In order to evaluate our model, we need to play against our trained model. To do this, we have to create a function that will check for a win. This way, our program will know when to stop asking for more moves. This is done with the following code:

```
def check(board):
    wins = [[0,1,2], [3,4,5], [6,7,8], [0,3,6], [1,4,7], [2,5,8], [0,4,8], [2,4,
    for i in range(len(wins)):
        if board[wins[i][0]]==board[wins[i][1]]==board[wins[i][2]]==1.:
            return 1
        elif board[wins[i][0]]==board[wins[i][1]]==board[wins[i][2]]==-1.:
            return -1
    return 0
```

17. Now we can loop through and play a game with our model. We start with a blank board (all zeros), we ask the user to input an index (0-8) of where to play, and we then feed that into the model for a prediction. For the model's move, we take the largest available prediction that is also an open space. From this game, we can see that our model is not perfect, as follows:

```
game_tracker = [0., 0., 0., 0., 0., 0., 0., 0., 0.]
win_logical = False
num_moves = 0
while not win_logical:
    player_index = input('Input index of your move (0-8): ')
    num_moves += 1
    # Add player move to game
    game_tracker[int(player_index)] = 1.

    # Get model's move by first getting all the logits for each index
    [potential_moves] = sess.run(model_output, feed_dict={X: [game_tracker]})
    # Now find allowed moves (where game tracker values = 0.0)
    allowed_moves = [ix for ix,x in enumerate(game_tracker) if x==0.0]
```

```
# Find best move by taking argmax of logits if they are in allowed moves
model_move = np.argmax([x if ix in allowed_moves else -999.0 for ix,x in enumerate(logits)])
```

```
# Add model move to game
game_tracker[int(model_move)] = -1.
print('Model has moved')
print_board(game_tracker)
# Now check for win or too many moves
if check(game_tracker)==1 or num_moves>=5:
    print('Game Over!')
    win_logical = True
```

18. The preceding step should result in the following interactive output:

```
Input index of your move (0-8): 4
Model has moved
 0 |   |
  _____
      | X |
  _____
      |   |
Input index of your move (0-8): 6
Model has moved
 0 |   |
  _____
      | X |
  _____
      X |   | 0
Input index of your move (0-8): 2
Model has moved
 0 |   | X
  _____
      0 | X |
  _____
      X |   | 0
Game Over!
```

How it works...

In this section, we trained a neural network to play Tic Tac Toe by feeding in board positions and a nine-dimensional vector, and predicted the optimal response. We only had to feed in a few possible Tic Tac Toe boards and apply random transformations to each board to increase the training set size.

To test our algorithm, we removed all instances of one specific board and saw whether our model could generalize to predict the optimal response. Finally, we played a sample game against our model. While it is not perfect yet, there are still different architectures and training procedures that can be applied to improve it.

Natural Language Processing

In this chapter, we will introduce you to working with text in TensorFlow. We will start by introducing how word embeddings work using the bag-of-words method, and then we will move on to implementing more advanced embeddings such as word2vec and doc2vec.

In this chapter, we will be covering the following topics:

- Working with bag-of-words
- Implementing TF-IDF
- Working with Skip-Gram embeddings
- Working with CBOW embeddings
- Making predictions with word2vec
- Using doc2vec for sentiment analysis

As a note, the reader will find all of the code for this chapter online at https://github.com/nfmccclure/tensorflow_cookbook and at the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

Introduction

So far, we have only considered machine learning algorithms that mostly operate on numerical inputs. If we want to use text, we must find a way to convert the text into numbers. There are many ways to do this, and we will explore a few common ways to do so in this chapter.

If we consider the sentence *TensorFlow makes machine learning easy*, we could convert the words to numbers in the order that we observe them. This would make the sentence become *1 2 3 4 5*. Then when we see a new sentence, *machine learning is easy*, we can translate this as *3 4 0 5*, denoting words we haven't seen with an index of zero. With these two examples, we have limited our vocabulary to six numbers. With large pieces of text, we can choose how many words we want to keep, and usually keep the most frequent words, labeling everything else with a zero index.

If the word *learning* has a numerical value of 4, and the word *makes* has a numerical value of 2, then it would be natural to assume that *learning* is twice that of *makes*. Since we do not want this type of numerical relationship between words, we can assume that these numbers represent categories and not relational numbers.

Another problem is that these two sentences are different sizes. Each observation we make (sentences, in this case) needs to have the same size input as the model we wish to create. To get around this, we have to create each sentence in a sparse vector that has a value of 1 in a specific index if that word occurs in that index:

TensorFlow	makes	machine	learning	easy
1	2	3	4	5

```
|first_sentence = [0,1,1,1,1,1]
```

To further explain the preceding vector, our vocabulary consists of six different words (five known words and an unknown word). For each of these words, we either have a zero or 1 value. A zero indicates that the word doesn't occur in our sentence, and a 1 indicates that it occurs at least once. So a value of zero means that the word does not occur, and a value of 1 means that it occurs

machine	learning	is	easy
3	4	0	5

```
| second_sentence = [1, 0, 0, 1, 1, 1]
```

A disadvantage to this method is that we lose any indication of word order. The two sentences *TensorFlow makes machine learning easy* and *machine learning makes TensorFlow easy* would result in the same sentence vector.

It is also worthwhile to note that the length of these vectors is equal to the size of the vocabulary that we pick. It is common to pick a very large vocabulary, so these sentence vectors can be very sparse. This type of embedding is called **bag-of-words**. We will implement this in the next section.

Another drawback is that the word *is* and *TensorFlow* have the same numerical index value: 1. It makes sense that the word *is* might be less important than the occurrence of the word *TensorFlow*.

We will explore different types of embeddings in this chapter that attempt to address these ideas, but first we will start with an implementation of the bag-of-words algorithm.

Working with bag-of-words embeddings

In this section, we will start by showing you how to work with a bag-of-words embedding in TensorFlow. This mapping is what we introduced in the introduction. Here, we will show you how to use this type of embedding for spam prediction.

Getting ready

To illustrate how to use bag-of-words with a text dataset, we will use a spam-ham phone text database from the UCI machine learning data repository (<https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection>). This is a collection of phone text messages that are spam or not-spam (ham). We will download this data, store it for future use, and then proceed with the bag-of-words method to predict if a text is spam or not. The model that will operate on the bag-of-words algorithm will be a logistic model with no hidden layers. We will use stochastic training, with a batch size of 1, and compute the accuracy on a held-out test set at the end.

How to do it...

For this example, we will start by getting the data, normalizing and splitting the text, running it through an embedding function, and training the logistic function to predict spam:

1. The first task will be to import the necessary libraries for this task. Among the usual libraries, we will need a `.zip` file library to unzip the data from the UCI machine learning website we retrieve it from:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import os
import numpy as np
import csv
import string
import requests
import io
from zipfile import ZipFile
from tensorflow.contrib import learn
sess = tf.Session()
```

2. Instead of downloading the text data every time the script is run, we will save it and check if the file has been saved before. This prevents us from repeatedly downloading the data over and over if we want to change the script's parameters. After downloading this data, we will extract the input and target data and change the target to `1` for spam and `0` for ham:

```
save_file_name = os.path.join('temp', 'temp_spam_data.csv')
if os.path.isfile(save_file_name):
    text_data = []
    with open(save_file_name, 'r') as temp_output_file:
        reader = csv.reader(temp_output_file)
        for row in reader:
            text_data.append(row)
else:
    zip_url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00228/sm
r = requests.get(zip_url)
z = ZipFile(io.BytesIO(r.content))
file = z.read('SMSSpamCollection')
# Format Data
text_data = file.decode()
text_data = text_data.encode('ascii', errors='ignore')
text_data = text_data.decode().split('\n')
text_data = [x.split('\t') for x in text_data if len(x)>=1]

# And write to csv
with open(save_file_name, 'w') as temp_output_file:
    writer = csv.writer(temp_output_file)
    writer.writerows(text_data)
```

```

texts = [x[1] for x in text_data]
target = [x[0] for x in text_data]
# Relabel 'spam' as 1, 'ham' as 0
target = [1 if x=='spam' else 0 for x in target]

```

3. To reduce the potential vocabulary size, we normalize the text. To do this, we remove the influence of capitalization and numbers in the text. Use the following code for this:

```

# Convert to lower case
texts = [x.lower() for x in texts]
# Remove punctuation
texts = [''.join(c for c in x if c not in string.punctuation) for x in texts]
# Remove numbers
texts = [''.join(c for c in x if c not in '0123456789') for x in texts]
# Trim extra whitespace
texts = [' '.join(x.split()) for x in texts]

```

4. We must also determine the maximum sentence size. To do this, we will look at a histogram of text lengths in the dataset. We can see that a good cut-off might be around 25 words. Use the following code for this:

```

# Plot histogram of text lengths
text_lengths = [len(x.split()) for x in texts]
text_lengths = [x for x in text_lengths if x < 50]
plt.hist(text_lengths, bins=25)
plt.title('Histogram of # of Words in Texts')
sentence_size = 25
min_word_freq = 3

```

From this, we will get the following plot:

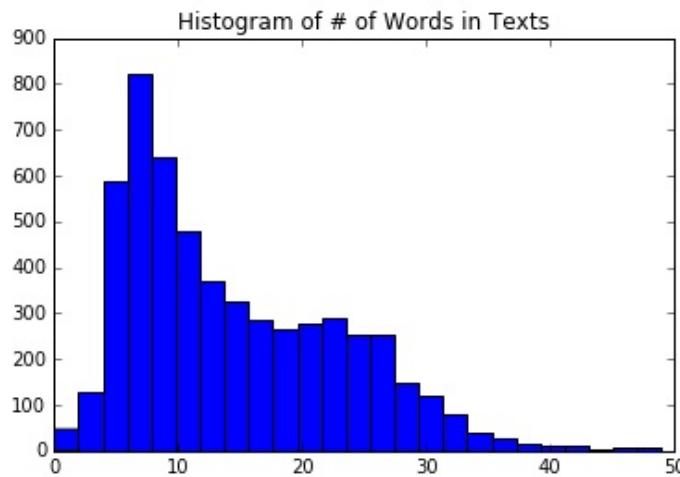


Figure 1: A histogram of the number of words in each text in our data.

i We use this to establish a maximum length of words to consider in each text. We set this to 25 words, but it can easily be set to 30 or 40 as well.

5. TensorFlow has a built-in processing tool for determining vocabulary

embedding called `vocabularyProcessor()`, which is in the `learn.preprocessing` library. Note that you may get a deprecated warning with this function:

```
vocab_processor = learn.preprocessing.VocabularyProcessor(sentence_size, min_fre
vocab_processor.fit_transform(texts)
transformed_texts = np.array([x for x in vocab_processor.transform(texts)])
embedding_size = len(np.unique(transformed_texts))
```

6. Now we will split the data into 80-20 train and test sets:

```
train_indices = np.random.choice(len(texts), round(len(texts)*0.8), replace=False)
test_indices = np.array(list(set(range(len(texts))) - set(train_indices)))
texts_train = [x for ix, x in enumerate(texts) if ix in train_indices]
texts_test = [x for ix, x in enumerate(texts) if ix in test_indices]
target_train = [x for ix, x in enumerate(target) if ix in train_indices]
target_test = [x for ix, x in enumerate(target) if ix in test_indices]
```

7. Next, we declare the embedding matrix for the words. Sentence words will be translated into indices. These indices will be translated into one-hot-encoded vectors that we can create with an identity matrix, which will be the size of our word embeddings. We will use this matrix to look up the sparse vector for each word and add them together for the sparse sentence vector. Use the following code to do this:

```
| identity_mat = tf.diag(tf.ones(shape=[embedding_size]))
```

8. Since we will end up performing logistic regression to predict the probability of spam, we need to declare our logistic regression variables. Then we can declare our data placeholders as well. It is important to note that the `x_data` input placeholder should be an integer type because it will be used to look up the row index of our identity matrix. TensorFlow requires this lookup to be an integer:

```
A = tf.Variable(tf.random_normal(shape=[embedding_size,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
# Initialize placeholders
x_data = tf.placeholder(shape=[sentence_size], dtype=tf.int32)
y_target = tf.placeholder(shape=[1, 1], dtype=tf.float32)
```

9. Now we will use TensorFlow's embedding lookup function, which will map the indices of the words in the sentence to the one-hot-encoded vectors of our identity matrix. When we have that matrix, we create the sentence vector by summing up the aforementioned word vectors. Use the following code to do this:

```
| x_embed = tf.nn.embedding_lookup(identity_mat, x_data)
| x_col_sums = tf.reduce_sum(x_embed, 0)
```

- Now that we have our fixed-length sentence vectors for each sentence, we want to perform logistic regression. To do this, we will need to declare the actual model operations. Since we are doing this one data point at a time (stochastic training), we will expand the dimensions of our input and perform linear regression operations on it. Remember that TensorFlow has a loss function that includes the sigmoid function, so we do not need to include it in our output here:

```
| x_col_sums_2D = tf.expand_dims(x_col_sums, 0)
| model_output = tf.add(tf.matmul(x_col_sums_2D, A), b)
```

- We will now declare the loss function, prediction operation, and optimization function to train the model. Use the following code to do this:

```
| loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=model_outpu
| # Prediction operation
| prediction = tf.sigmoid(model_output)
| # Declare optimizer
| my_opt = tf.train.GradientDescentOptimizer(0.001)
| train_step = my_opt.minimize(loss)
```

- Next, we will initialize our graph variables before we start the training generations:

```
| init = tf.global_variables_initializer()
| sess.run(init)
```

- Now we will start the iteration over the sentences. TensorFlow's `vocab_processor.fit()` function is a generator that operates one sentence at a time. We will use this to our advantage so that we can perform stochastic training on our logistic model. To get a better idea of the accuracy trend, we will keep a trailing average of the past 50 training steps. If we just plotted the current one, we would see either 1 or 0, depending on whether we predicted that training data point correctly or not. Use the following code to do this:

```
| loss_vec = []
| train_acc_all = []
| train_acc_avg = []
| for ix, t in enumerate(vocab_processor.fit_transform(texts_train)):
|     y_data = [[target_train[ix]]]
|
|     sess.run(train_step, feed_dict={x_data: t, y_target: y_data})
|
|     temp_loss = sess.run(loss, feed_dict={x_data: t, y_target: y_data})
|     loss_vec.append(temp_loss)
|
|     if (ix+1)%10==0:
|         print('Training Observation #{}: Loss= {}'.format(ix+1, temp_loss))
```

```

# Keep trailing average of past 50 observations accuracy
# Get prediction of single observation
[[temp_pred]] = sess.run(prediction, feed_dict={x_data:t, y_target:y_data})
# Get True/False if prediction is accurate
train_acc_temp = target_train[ix]==np.round(temp_pred)
train_acc_all.append(train_acc_temp)
if len(train_acc_all) >= 50:
    train_acc_avg.append(np.mean(train_acc_all[-50:]))

```

14. This results in the following output:

```

Starting Training Over 4459 Sentences.
Training Observation #10: Loss = 5.45322
Training Observation #20: Loss = 3.58226
Training Observation #30: Loss = 0.0
...
Training Observation #4430: Loss = 1.84636
Training Observation #4440: Loss = 1.46626e-05
Training Observation #4450: Loss = 0.045941

```

15. To get the test set accuracy, we repeat the preceding process, but only on the prediction operation, not the training operation with the test sets:

```

print('Getting Test Set Accuracy')
test_acc_all = []
for ix, t in enumerate(vocab_processor.fit_transform(texts_test)):
    y_data = [[target_test[ix]]]

    if (ix+1)%50==0:
        print('Test Observation #{}'.format(ix+1))

    # Keep trailing average of past 50 observations accuracy
    # Get prediction of single observation
    [[temp_pred]] = sess.run(prediction, feed_dict={x_data:t, y_target:y_data})
    # Get True/False if prediction is accurate
    test_acc_temp = target_test[ix]==np.round(temp_pred)
    test_acc_all.append(test_acc_temp)
print('\nOverall Test Accuracy: {}'.format(np.mean(test_acc_all)))

Getting Test Set Accuracy For 1115 Sentences.
Test Observation #10
Test Observation #20
Test Observation #30
...
Test Observation #1000
Test Observation #1050
Test Observation #1100
Overall Test Accuracy: 0.8035874439461883

```

How it works...

For this example, we worked with the spam-ham text data from the UCI machine learning repository. We used TensorFlow's vocabulary processing functions to create a standardized vocabulary to work with and create sentence vectors which were the sum of each text's word vectors. We used this sentence vector with logistic regression and obtained an 80% accuracy model to predict whether a specific text is spam.

There's more...

It is worth mentioning the motivation behind limiting the sentence's (or text) size. In this example, we limited the text size to 25 words. This is a common practice with bag-of-words because it limits the effect of text length on the prediction. As you can imagine, if we find a word, for example, *meeting*, that is predictive of a text being ham (not spam), then a spam message might get through by putting in many occurrences of that word at the end. In fact, this is a common problem with imbalanced target data. Imbalanced data might occur in this situation, since spam may be hard to find and ham may be easy to find. Because of this fact, the vocabulary that we create might be heavily skewed toward words represented in the ham part of our data (more ham means more words are represented in ham than spam). If we allow an unlimited length for texts, then spammers might take advantage of this and create very long texts, which have a higher probability of triggering non-spam word factors in our logistic model.

In the next section, we will attempt to tackle this problem in a better way by using the frequency of word occurrence to determine the values of word embeddings.

Implementing TF-IDF

Since we can choose the embedding for each word, we might decide to change the weighting on certain words. One such strategy is to up weight useful words and downweight overly common or rare words. The embedding we will explore in this recipe is an attempt to achieve this.

Getting ready

TF-IDF is an acronym that stands for **Text Frequency - Inverse Document Frequency**. This term is essentially the product of text frequency and inverse document frequency for each word.

In the preceding recipe, we introduced the bag-of-words methodology, which assigned a value of 1 for every occurrence of a word in a sentence. This is probably not ideal as each category of sentence (spam and ham in the preceding recipe's example) most likely has the same frequency of *the*, *and*, and other words, whereas words such as *Viagra* and *sale* probably should have increased importance in figuring out whether or not the text is spam.

First, we want to take into consideration word frequency. Here, we consider the frequency with which a word occurs in an individual entry. The purpose of this part (TF) is to find terms that appear to be important in each entry.

But words such as *the* and *and* may appear very frequently in every entry. We want to downweight the importance of these words, so it makes sense that multiplying the preceding text frequency (TF) by the inverse of the whole document frequency might help find important words. However, since a collection of texts (a corpus) may be quite large, it is common to take the logarithm of the inverse document frequency. This leaves us with the following formula for TF-IDF for each word in each document entry:

$$w_{tf-idf} = w_{tf} \cdot \log\left(\frac{1}{w_{df}}\right)$$

Here w_{tf} is the word frequency by document, and w_{df} is the total frequency of such words across all documents. It makes sense that high values of TF-IDF might indicate words that are very important in determining what a document is about.

Creating the TF-IDF vectors requires us to load all the text into memory and count the occurrences of each word before we can start training our model.

Because of this, it is not implemented fully in TensorFlow, so we will use scikit-learn to create our TF-IDF embedding, but TensorFlow to fit the logistic model.

How to do it...

We will proceed with the recipe as follows:

1. We will start by loading the necessary libraries. This time, we are loading the scikit-learn TF-IDF preprocessing library for our texts. Use the following code to do this:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import csv
import numpy as np
import os
import string
import requests
import io
import nltk
from zipfile import ZipFile
from sklearn.feature_extraction.text import TfidfVectorizer
```

2. We will start a graph session and declare our batch size and maximum feature size for our vocabulary:

```
sess = tf.Session()
batch_size= 200
max_features = 1000
```

3. Next, we will load the data, either from the web or from our `temp` data folder if we have saved it before. Use the following code to do this:

```
save_file_name = os.path.join('temp','temp_spam_data.csv')
if os.path.isfile(save_file_name):
    text_data = []
    with open(save_file_name, 'r') as temp_output_file:
        reader = csv.reader(temp_output_file)
        for row in reader:
            text_data.append(row)
else:
    zip_url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00228/sm
r = requests.get(zip_url)
z = ZipFile(io.BytesIO(r.content))
file = z.read('SMSSpamCollection')
# Format Data
text_data = file.decode()
text_data = text_data.encode('ascii',errors='ignore')
text_data = text_data.decode().split('\n')
text_data = [x.split('\t') for x in text_data if len(x)>=1]

# And write to csv
with open(save_file_name, 'w') as temp_output_file:
    writer = csv.writer(temp_output_file)
```

```

writer.writerows(text_data)
texts = [x[1] for x in text_data]
target = [x[0] for x in text_data]
# Relabel 'spam' as 1, 'ham' as 0
target = [1. if x=='spam' else 0. for x in target]

```

- Just like in the preceding recipe, we will decrease our vocabulary size by converting everything to lower case, removing punctuation and getting rid of numbers:

```

# Lower case
texts = [x.lower() for x in texts]
# Remove punctuation
texts = [''.join(c for c in x if c not in string.punctuation) for x in texts]
# Remove numbers
texts = [''.join(c for c in x if c not in '0123456789') for x in texts]
# Trim extra whitespace
texts = [' '.join(x.split()) for x in texts]

```

- In order to use scikit-learn's TF-IDF processing functions, we have to tell it how to tokenize our sentences. By this, we just mean how to break up a sentence into its corresponding words. A great tokenizer is already built for us: the `nltk` package does a great job of breaking up sentences into the corresponding words:

```

def tokenizer(text):
    words = nltk.word_tokenize(text)
    return words
# Create TF-IDF of texts
tfidf = TfidfVectorizer(tokenizer=tokenizer, stop_words='english', max_features=sparse_tfidf_texts = tfidf.fit_transform(texts)

```

- Next, we will break up our dataset into test and train sets. Use the following code to do this:

```

train_indices = np.random.choice(sparse_tfidf_texts.shape[0], round(0.8*sparse_tfidf_texts.shape[0]))
test_indices = np.array(list(set(range(sparse_tfidf_texts.shape[0])) - set(train_indices)))
texts_train = sparse_tfidf_texts[train_indices]
texts_test = sparse_tfidf_texts[test_indices]
target_train = np.array([x for ix, x in enumerate(target) if ix in train_indices])
target_test = np.array([x for ix, x in enumerate(target) if ix in test_indices])

```

- Now we declare our model variables for logistic regression and our data placeholders:

```

A = tf.Variable(tf.random_normal(shape=[max_features,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
# Initialize placeholders
x_data = tf.placeholder(shape=[None, max_features], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)

```

- We can now declare the model operations and the loss function. Remember

that the sigmoid part of the logistic regression is in our loss function. Use the following code to do this:

```
model_output = tf.add(tf.matmul(x_data, A), b)
loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=model_outpu
```

9. We will add the prediction and accuracy functions to the graph so that we can see the accuracy of the train and test sets as our model is training:

```
prediction = tf.round(tf.sigmoid(model_output))
predictions_correct = tf.cast(tf.equal(prediction, y_target), tf.float32)
accuracy = tf.reduce_mean(predictions_correct)
```

10. Then we will declare an optimizer and initialize our graph variables:

```
my_opt = tf.train.GradientDescentOptimizer(0.0025)
train_step = my_opt.minimize(loss)
# Initialize Variables
init = tf.global_variables_initializer()
sess.run(init)
```

11. We will now train our model over 10,000 generations and record the test/train loss and accuracy every 100 generations, printing out the status every 500 generations. Use the following code to do this:

```
train_loss = []
test_loss = []
train_acc = []
test_acc = []
i_data = []
for i in range(10000):
    rand_index = np.random.choice(texts_train.shape[0], size=batch_size)
    rand_x = texts_train[rand_index].todense()
    rand_y = np.transpose([target_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})

    # Only record loss and accuracy every 100 generations
    if (i+1)%100==0:
        i_data.append(i+1)
        train_loss_temp = sess.run(loss, feed_dict={x_data: rand_x, y_target: ra
        train_loss.append(train_loss_temp)

        test_loss_temp = sess.run(loss, feed_dict={x_data: texts_test.todense()},
        test_loss.append(test_loss_temp)

        train_acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x, y_target:
        train_acc.append(train_acc_temp)

        test_acc_temp = sess.run(accuracy, feed_dict={x_data: texts_test.todense
        test_acc.append(test_acc_temp)
if (i+1)%500==0:
    acc_and_loss = [i+1, train_loss_temp, test_loss_temp, train_acc_temp, te
    acc_and_loss = [np.round(x,2) for x in acc_and_loss]
    print('Generation # {}. Train Loss (Test Loss): {:.2f} ({:.2f}). Train A
```

12. This results in the following output:

```
Generation # 500. Train Loss (Test Loss): 0.69 (0.73). Train Acc (Test Acc): 0.6
Generation # 1000. Train Loss (Test Loss): 0.62 (0.63). Train Acc (Test Acc): 0.
...
Generation # 9500. Train Loss (Test Loss): 0.39 (0.45). Train Acc (Test Acc): 0.
Generation # 10000. Train Loss (Test Loss): 0.48 (0.45). Train Acc (Test Acc): 0
```

And the following is the graph to plot the accuracy and loss for both the train and test sets:

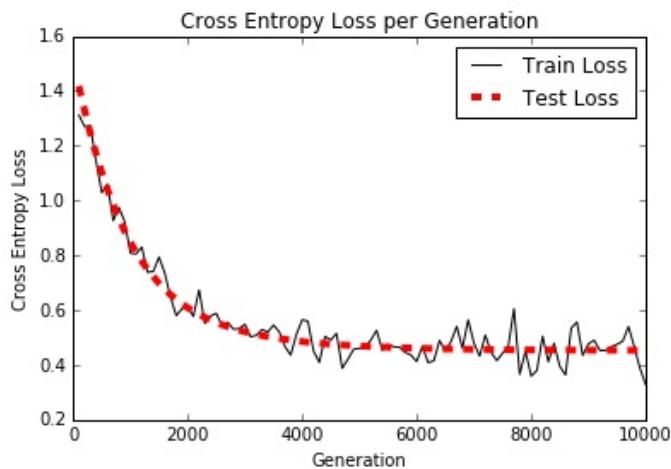


Figure 2: Cross entropy loss for our logistic spam model built from TF-IDF values

The train and test accuracy plot is as follows:

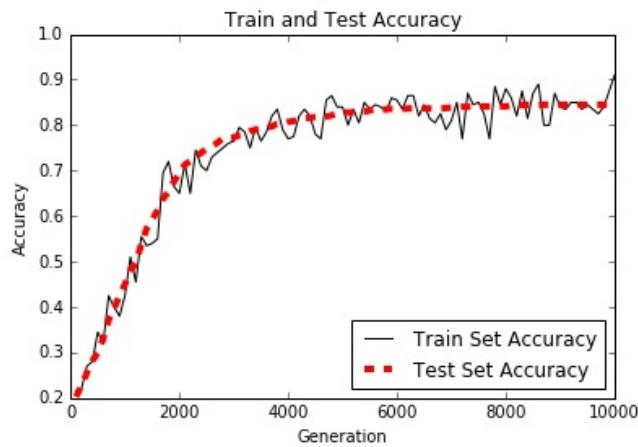


Figure 3: Train and test set accuracy for the logistic spam model built from TF-IDF values

How it works...

Using TF-IDF values for the model has increased our prediction over the preceding bag-of-words model from 80% accuracy to almost 90% accuracy. We achieved this by using scikit-learn's TF-IDF vocabulary processing functions and using those TF-IDF values for logistic regression.

There's more...

While we might have addressed the issue of word importance, we have not addressed the issue of word ordering. Both bag-of-words and TF-IDF have no features that take into account word ordering in a sentence. We will attempt to address this in the next few sections, which will introduce us to word2vec techniques.

Working with Skip-Gram embeddings

In the previous recipes, we decided on our textual embeddings before training the model. With neural networks, we can make the embedding values part of the training procedure. The first such method we will explore is called **Skip-Gram embedding**.

Getting ready

Prior to this recipe, we have not considered the order of words to be relevant in creating word embeddings. In early 2013, Tomas Mikolov and other researchers at Google authored a paper about creating word embeddings that addressed this issue (<https://arxiv.org/abs/1301.3781>), and they named their method **word2vec**.

The basic idea is to create word embeddings that capture the relational aspect of words. We seek to understand how various words are related to each other. Some examples of how these embeddings might behave are as follows:

$$\text{king} - \text{man} + \text{woman} = \text{queen}$$

$$\text{India pale ale} - \text{hops} + \text{malt} = \text{stout}$$

We might achieve such a numerical representation of words if we only consider their positional relationship to each other. If we could analyze a large enough source of coherent documents, we might find that the words *king*, *man*, and *queen* are mentioned closely to each other in our texts. If we also know that *man* and *woman* are related in a different way, then we might conclude that *man* is to *king* as *woman* is to *queen*, and so on.

To go about finding such an embedding, we will use a neural network that predicts surrounding words from a given input word. We could, just as easily, switch that and try to predict a target word given a set of surrounding words, but we will start with the preceding method. Both are variations of the word2vec procedure, but the preceding method of predicting the surrounding words (the context) from a target word is called the Skip-Gram model. In the next recipe, we will implement the other method, predicting the target word from the context, which is called the **continuous bag-of-words method (CBOW)**:

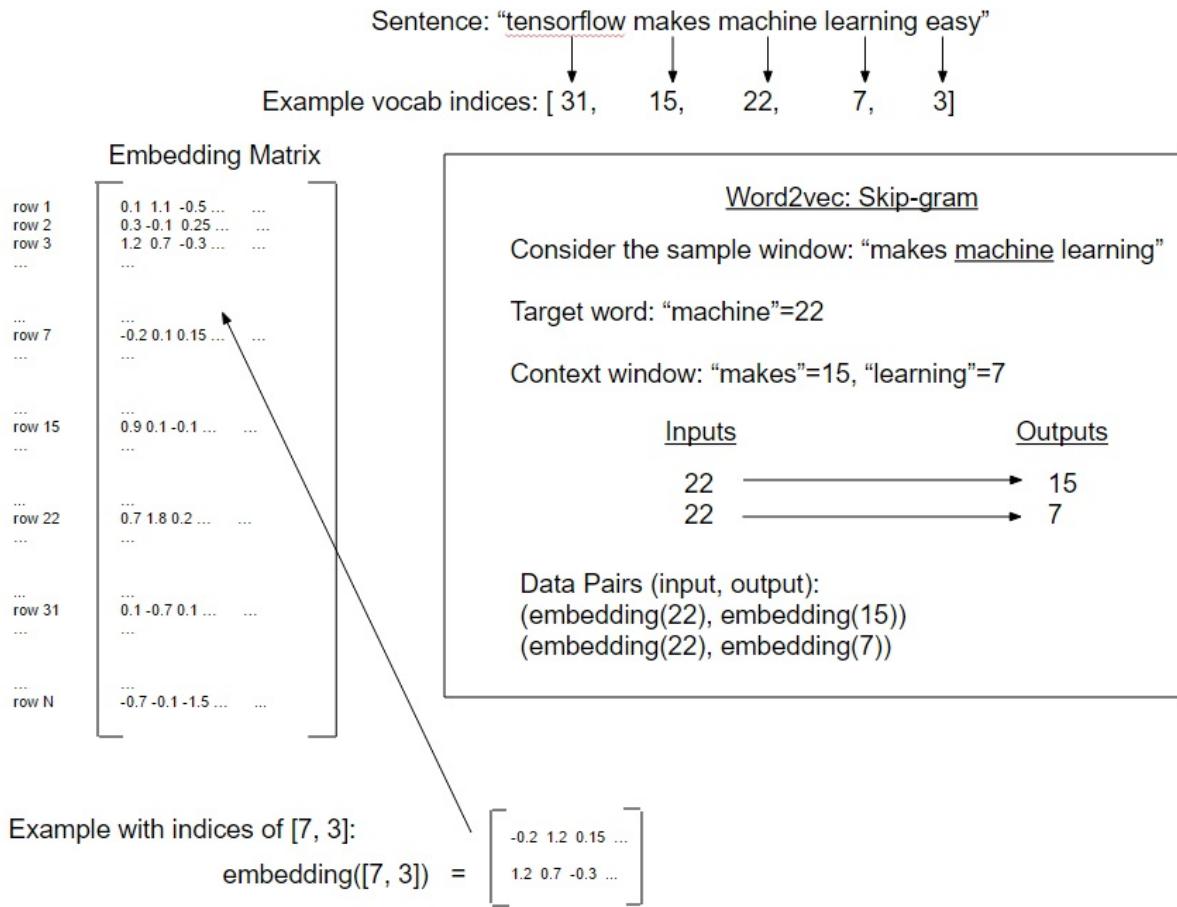


Figure 4: An illustration of the Skip-Gram implementations of word2vec. Skip-Gram predicts a window of context from the target word (window size of 1 on each side).

For this recipe, we will implement the Skip-Gram model on a set of movie review data from Cornell University (<http://www.cs.cornell.edu/people/pabo/movie-review-data/>). The **CBOW** method from word2vec will be implemented in the next recipe.

How to do it...

For this recipe, we will create several helper functions. These functions will load the data, normalize the text, generate the vocabulary, and generate data batches. Only after all this will we then start training our word embeddings. To be clear, we are not predicting any target variables, but we will be fitting word embeddings instead:

1. First, we will load the necessary libraries and start a graph session:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import random
import os
import string
import requests
import collections
import io
import tarfile
import urllib.request
from nltk.corpus import stopwords
sess = tf.Session()
```

2. Then we declare some model parameters. We will look at 50 pairs of word embeddings at a time (batch size). The embedding size of each word will be a vector of length 200, and we will only consider the 10,000 most frequent words (every other word will be classified as unknown). We will train for 50,000 generations and print out the loss every 500 generations. Then we will declare a `num_sampled` variable that we will use in the loss function (which we will explain later), and we also declare our Skip-Gram window size. Here, we set our window size to 2, so we will look at the surrounding two words on each side of the target. We will set our stop words via the Python package called `nltk`. We also want a way to check how our word embeddings are performing, so we will choose some common movie review words and print out the nearest neighbor words from these every 2,000 iterations:

```
batch_size = 50
embedding_size = 200
vocabulary_size = 10000
generations = 50000
print_loss_every = 500
num_sampled = int(batch_size/2)
```

```

window_size = 2
stops = stopwords.words('english')
print_valid_every = 2000
valid_words = ['cliche', 'love', 'hate', 'silly', 'sad']

```

3. Next, we will declare our data loading function, which checks to make sure that we have not downloaded the data before it downloads. Otherwise, it will load the data from the disk if we have it saved before. Use the following code to do this:

```

def load_movie_data():
    save_folder_name = 'temp'
    pos_file = os.path.join(save_folder_name, 'rt-polarity.pos')
    neg_file = os.path.join(save_folder_name, 'rt-polarity.neg')
    # Check if files are already downloaded
    if os.path.exists(save_folder_name):
        pos_data = []
        with open(pos_file, 'r') as temp_pos_file:
            for row in temp_pos_file:
                pos_data.append(row)
        neg_data = []
        with open(neg_file, 'r') as temp_neg_file:
            for row in temp_neg_file:
                neg_data.append(row)
    else: # If not downloaded, download and save
        movie_data_url = 'http://www.cs.cornell.edu/people/pabo/movie-review-dataset'
        stream_data = urllib.request.urlopen(movie_data_url)
        tmp = io.BytesIO()
        while True:
            s = stream_data.read(16384)
            if not s:
                break
            tmp.write(s)
        stream_data.close()
        tmp.seek(0)
        tar_file = tarfile.open(fileobj=tmp, mode='r:gz')
        pos = tar_file.extractfile('rt-polaritydata/rt-polarity.pos')
        neg = tar_file.extractfile('rt-polaritydata/rt-polarity.neg')
        # Save pos/neg reviews
        pos_data = []
        for line in pos:
            pos_data.append(line.decode('ISO-8859-1').encode('ascii', errors='ignore'))
        neg_data = []
        for line in neg:
            neg_data.append(line.decode('ISO-8859-1').encode('ascii', errors='ignore'))
        tar_file.close()
        # Write to file
        if not os.path.exists(save_folder_name):
            os.makedirs(save_folder_name)
        # Save files
        with open(pos_file, 'w') as pos_file_handler:
            pos_file_handler.write(''.join(pos_data))
        with open(neg_file, 'w') as neg_file_handler:
            neg_file_handler.write(''.join(neg_data))
    texts = pos_data + neg_data
    target = [1]*len(pos_data) + [0]*len(neg_data)
    return(texts, target)
texts, target = load_movie_data()

```

4. Next, we will create a normalization function for text. This function will input a list of strings and make it lowercase, remove punctuation, remove numbers, strip out extra whitespace, and remove stop words. Use the following code to do this:

```
def normalize_text(texts, stops):
    # Lower case
    texts = [x.lower() for x in texts]
    # Remove punctuation
    texts = [''.join(c for c in x if c not in string.punctuation) for x in texts]
    # Remove numbers
    texts = [''.join(c for c in x if c not in '0123456789') for x in texts]
    # Remove stopwords
    texts = [' '.join([word for word in x.split() if word not in (stops)]) for x in texts]
    # Trim extra whitespace
    texts = [' '.join(x.split()) for x in texts]

    return(texts)
texts = normalize_text(texts, stops)
```

5. To make sure that all of our movie reviews are informative, we should make sure that they are long enough to contain important word relationships. We will arbitrarily set this to three or more words:

```
target = [target[ix] for ix, x in enumerate(texts) if len(x.split()) > 2]
texts = [x for x in texts if len(x.split()) > 2]
```

6. To build our vocabulary, we will create a function that creates a dictionary of words with their count. Any word that is uncommon enough to not make our vocabulary size cut-off will be labeled `RARE`. Use the following code to do this:

```
def build_dictionary(sentences, vocabulary_size):
    # Turn sentences (list of strings) into lists of words
    split_sentences = [s.split() for s in sentences]
    words = [x for sublist in split_sentences for x in sublist]
    # Initialize list of [word, word_count] for each word, starting with unknown
    count = [['RARE', -1]]
    # Now add most frequent words, limited to the N-most frequent (N=vocabulary
    count.extend(collections.Counter(words).most_common(vocabulary_size-1))
    # Now create the dictionary
    word_dict = {}
    # For each word, that we want in the dictionary, add it, then make it the va
    for word, word_count in count:
        word_dict[word] = len(word_dict)
    return(word_dict)
```

7. We need a function that will convert a list of sentences into lists of word indices that we can pass into our embedding lookup function. Use the following code to do this:

```
def text_to_numbers(sentences, word_dict):
```

```

# Initialize the returned data
data = []
for sentence in sentences:
    sentence_data = []
    # For each word, either use selected index or rare word index
    for word in sentence:
        if word in word_dict:
            word_ix = word_dict[word]
        else:
            word_ix = 0
        sentence_data.append(word_ix)
    data.append(sentence_data)
return data

```

- Now we can actually create our dictionary and transform our list of sentences into lists of word indices:

```

word_dictionary = build_dictionary(texts, vocabulary_size)
word_dictionary_rev = dict(zip(word_dictionary.values(), word_dictionary.keys()))
text_data = text_to_numbers(texts, word_dictionary)

```

- From the preceding word dictionary, we can look up the index for the validation words we chose in step 2. Use the following code to do this:

```
| valid_examples = [word_dictionary[x] for x in valid_words]
```

- We will now create a function that will return our Skip-Gram batches. We want to train on pairs of words where one word is the training input (from the target word at the center of our window) and the other word is selected from the window. For example, the sentence *the cat in the hat* may result in (input, output) pairs such as the following: (*the, in*), (*cat, in*), (*the, in*), (*hat, in*) if it was the target word, and we had a window size of 2 in each direction:

```

def generate_batch_data(sentences, batch_size, window_size, method='skip_gram'):
    # Fill up data batch
    batch_data = []
    label_data = []
    while len(batch_data) < batch_size:
        # select random sentence to start
        rand_sentence = np.random.choice(sentences)
        # Generate consecutive windows to look at
        window_sequences = [rand_sentence[max((ix - window_size), 0):(ix + window_size)] for ix in range(len(rand_sentence))]
        # Denote which element of each window is the center word of interest
        label_indices = [ix if ix < window_size else window_size for ix, x in enumerate(window_sequences)]
        # Pull out center word of interest for each window and create a tuple for each window
        if method == 'skip_gram':
            batch_and_labels = [(x[y], x[:y] + x[(y+1):]) for x, y in zip(window_sequences, label_indices)]
            # Make it into a big list of tuples (target word, surrounding word)
            tuple_data = [(x, y_) for x, y in batch_and_labels for y_ in y]
        else:
            raise ValueError('Method {} not implemented yet.'.format(method))
    return batch_data, tuple_data

```

```

        # extract batch and labels
        batch, labels = [list(x) for x in zip(*tuple_data)]
        batch_data.extend(batch[:batch_size])
        label_data.extend(labels[:batch_size])
    # Trim batch and label at the end
    batch_data = batch_data[:batch_size]
    label_data = label_data[:batch_size]

    # Convert to numpy array
    batch_data = np.array(batch_data)
    label_data = np.transpose(np.array([label_data]))

    return batch_data, label_data

```

11. We can now initialize our embedding matrix, declare our placeholders, and initialize our embedding lookup function. Use the following code to do this:

```

embeddings = tf.Variable(tf.random_uniform([vocabulary_size,
                                           embedding_size], -1.0, 1.0))
# Create data/target placeholders
x_inputs = tf.placeholder(tf.int32, shape=[batch_size])
y_target = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

# Lookup the word embedding:
embed = tf.nn.embedding_lookup(embeddings, x_inputs)

```

12. The loss function should be something such as a `softmax`, which calculates the loss on predicting the wrong word category. But since our target has 10,000 different categories, it is very sparse. This sparsity causes problems regarding fitting or converging for a model. To tackle this, we will use a loss function called noise-contrastive error. This NCE loss function turns our problem into a binary prediction by predicting the word class versus random noise predictions. The `num_sampled` parameter specifies how much of the batch to turn into random noise:

```

nce_weights = tf.Variable(tf.truncated_normal([vocabulary_size,
                                              embedding_size], stddev=1.0 / np.sqrt(embedding_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
loss = tf.reduce_mean(tf.nn.nce_loss(weights=nce_weights,
                                      biases=nce_biases,
                                      inputs=embed,
                                      labels=y_target,
                                      num_sampled=num_sampled,
                                      num_classes=vocabulary_size))

```

13. Now we need to create a way to find nearby words to our validation words. We will do this by computing the cosine similarity between the validation set and all of our word embeddings, and then we can print out the closest set of words for each validation word. Use the following code to do this:

```

norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keepdims=True))

```

```

normalized_embeddings = embeddings / norm
valid_embeddings = tf.nn.embedding_lookup(normalized_embeddings, valid_dataset)
similarity = tf.matmul(valid_embeddings, normalized_embeddings, transpose_b=True

```

14. We now declare our optimization function and initialize our model variables:

```

optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0).minimize(loss)
init = tf.global_variables_initializer()
sess.run(init)

```

15. Now we can train our embeddings and print off the loss and the closest words to our validation set during training. Use the following code to do this:

```

loss_vec = []
loss_x_vec = []
for i in range(generations):
    batch_inputs, batch_labels = generate_batch_data(text_data, batch_size, word
    feed_dict = {x_inputs : batch_inputs, y_target : batch_labels}
    # Run the train step
    sess.run(optimizer, feed_dict=feed_dict)
    # Return the loss
    if (i+1) % print_loss_every == 0:
        loss_val = sess.run(loss, feed_dict=feed_dict)
        loss_vec.append(loss_val)
        loss_x_vec.append(i+1)
        print("Loss at step {} : {}".format(i+1, loss_val))

    # Validation: Print some random words and top 5 related words
    if (i+1) % print_valid_every == 0:
        sim = sess.run(similarity, feed_dict=feed_dict)
        for j in range(len(valid_words)):
            valid_word = word_dictionary_rev[valid_examples[j]]
            top_k = 5 # number of nearest neighbors
            nearest = (-sim[j, :]).argsort()[1:top_k+1]
            log_str = "Nearest to {}:".format(valid_word)
            for k in range(top_k):
                close_word = word_dictionary_rev[nearest[k]]
                log_str = "%s %s," % (log_str, close_word)
            print(log_str)

```



In the preceding code, we take the negative of the similarity matrix before calling the `argsort` method on it. We do this because we want to find the indices from the highest similarity values to the lowest similarity values and not the other way around.

16. This results in the following output:

```

Loss at step 500 : 13.387781143188477
Loss at step 1000 : 7.240757465362549
Loss at step 49500 : 0.9395825862884521
Loss at step 50000 : 0.30323168635368347
Nearest to cliche: walk, intrigue, brim, eileen, dumber,
Nearest to love: plight, fiction, complete, lady, bartleby,
Nearest to hate: style, throws, players, fearlessness, astringent,
Nearest to silly: delivers, meow, regain, nicely, anger,
Nearest to sad: dizzying, variety, existing, environment, tunney,

```

How it works...

We have trained a word2vec model via the `skip-gram` method on a corpus of movie review data. We downloaded the data, converted the words to an index with a dictionary, and used those index numbers as an embedding lookup, which we trained so that nearby words could be predictive of each other.

There's more...

At first glance, we might expect the set of nearby words for the validation set to be synonyms. This is not quite the case because very rarely do synonyms actually appear next to each other in sentences. What we are really getting at is predicting which words are in proximity to each other in our dataset. We hope that using an embedding such as this will make prediction easier.

In order to use these embeddings, we must make them reusable and save them. We will do this in the next recipe by implementing CBOW embeddings.

Working with CBOW embeddings

In this recipe, we will implement the CBOW (continuous bag-of-words) method of word2vec. It is very similar to the skip-gram method, except we are predicting a single target word from a surrounding window of context words.

Getting ready

In this recipe, we will implement the `CBOW` method of word2vec. It is very similar to the `skip-Gram` method, except we are predicting a single target word from a surrounding window of context words.

In the previous example, we treated each combination of window and target as a group of paired inputs and outputs, but with CBOW we will add the surrounding window embeddings together to get one embedding to predict the target word embedding:

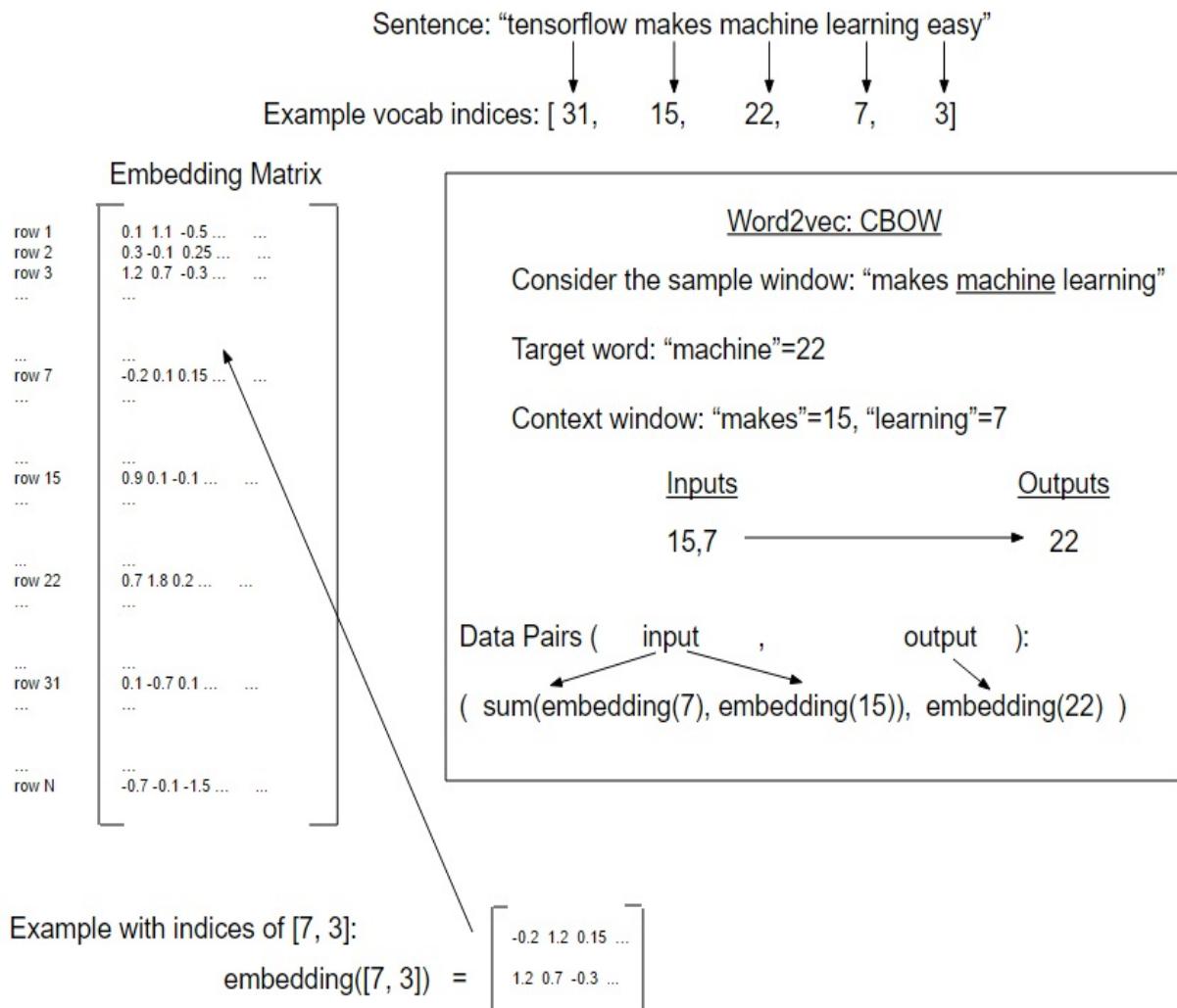


Figure 5: A depiction of how the CBOW embedding data is created out of a window on an example sentence (window size = 1 on each side)

Most of the code will stay the same, except we will need to change how we create the embeddings and how we generate the data from the sentences.

To make the code easier to read, we have moved all the major functions to a separate file called `text_helpers.py` in the same directory. This function holds the data loading, text normalization, dictionary creation, and batch generation functions. These functions are exactly as they appear in the *Working with Skip-Gram Embeddings* recipe, except where noted.

How to do it...

We will proceed with the recipe as follows:

1. We will start by loading the necessary libraries, including the aforementioned `text_helpers.py` script, where we will put our functions for text loading and manipulation. We will then start a graph session:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import random
import os
import pickle
import string
import requests
import collections
import io
import tarfile
import urllib.request
import text_helpers
from nltk.corpus import stopwords
sess = tf.Session()
```

2. We want to make sure that our temporary data and parameter saving folder exists before we start saving to it. Use the following code to check this:

```
# Make a saving directory if it doesn't exist
data_folder_name = 'temp'
if not os.path.exists(data_folder_name):
    os.makedirs(data_folder_name)
```

3. We will then declare the parameters of our model, which will be similar to what we did for the `Skip-Gram` method in the previous recipe:

```
# Declare model parameters
batch_size = 500
embedding_size = 200
vocabulary_size = 2000
generations = 50000
model_learning_rate = 0.001
num_sampled = int(batch_size/2)
window_size = 3
# Add checkpoints to training
save_embeddings_every = 5000
print_valid_every = 5000
print_loss_every = 100
# Declare stop words
stops = stopwords.words('english')
# We pick some test words. We are expecting synonyms to appear
valid_words = ['love', 'hate', 'happy', 'sad', 'man', 'woman']
```

- We have moved the data loading and text normalization functions to a separate file that we imported at the start, this file is available at both the github repository, https://github.com/nfmccleure/tensorflow_cookbook/tree/master/07_Natural_Language_Processing/05_Working_With_CBOW_Embeddings, and the Packt repository, <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>. Now we can call them. We also only want reviews that have three or more words in them. Use the following code for this:

```
| texts, target = text_helpers.load_movie_data(data_folder_name) texts = text_help
```

- Now we will create our vocabulary dictionary, which will help us look up words. We also need a reverse dictionary that looks up words from indices when we want to print out the nearest words to our validation set:

```
| word_dictionary = text_helpers.build_dictionary(texts,
vocabulary_size)
word_dictionary_rev = dict(zip(word_dictionary.values(), word_dictionary.keys()))
text_data = text_helpers.text_to_numbers(texts, word_dictionary)
# Get validation word keys
valid_examples = [word_dictionary[x] for x in valid_words]
```

- Next, we will initialize the word embeddings that we want to fit and declare model data placeholders. Use the following code to do this:

```
| embeddings = tf.Variable(tf.random_uniform([vocabulary_size, embedding_size], -1
# Create data/target placeholders
x_inputs = tf.placeholder(tf.int32, shape=[batch_size,
2*window_size])
y_target = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
```

- We can now create a way to deal with the word embeddings. Since the CBOW model adds up the embeddings of the context window, we will create a loop and add up all of the embeddings in the window:

```
| # Lookup the word embeddings and
# Add together window embeddings:
embed = tf.zeros([batch_size, embedding_size])
for element in range(2*window_size):
    embed += tf.nn.embedding_lookup(embeddings, x_inputs[:, element])
```

- We will use the noise-contrastive error loss function that is built in to TensorFlow because our categorical output is too sparse for the softmax to converge, as follows:

```
| # NCE loss parameters
nce_weights = tf.Variable(tf.truncated_normal([vocabulary_size,
embedding_size], stddev=1.0 / np.sqrt(embedding_size)))
```

```

nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
# Declare loss function (NCE)
loss = tf.reduce_mean(tf.nn.nce_loss(weights=nce_weights,
                                      biases=nce_biases,
                                      inputs=embed,
                                      labels=y_target,
                                      num_sampled=num_sampled,
                                      num_classes=vocabulary_size))

```

- Just like we did in the Skip-Gram recipe, we will use cosine similarity to print off the nearest words to our validation word dataset to get an idea of how our embeddings are working. Use the following code to do this:

```

norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keepdims=True))
normalized_embeddings = embeddings / norm
valid_embeddings = tf.nn.embedding_lookup(normalized_embeddings, valid_dataset)
similarity = tf.matmul(valid_embeddings, normalized_embeddings, transpose_b=True)

```

- To save our embeddings, we must load the TensorFlow `train.Saver` method. This method defaults to saving the whole graph, but we can give it an argument just to save the embedding variable, and we can also give it a specific name. Here, we are giving it the same name as the variable name in our graph:

```
| saver = tf.train.Saver({"embeddings": embeddings})
```

- We will now declare our optimization function and initialize our model variables. Use the following code to do this:

```

optimizer = tf.train.GradientDescentOptimizer(learning_rate=model_learning_rate)
init = tf.global_variables_initializer()
sess.run(init)

```

- Finally, we can loop across our training step, print out the loss, and save the embeddings and dictionary where we specify to:

```

loss_vec = []
loss_x_vec = []
for i in range(generations):
    batch_inputs, batch_labels = text_helpers.generate_batch_data(text_data, batch_size)
    feed_dict = {x_inputs : batch_inputs, y_target : batch_labels}
    # Run the train step
    sess.run(optimizer, feed_dict=feed_dict)
    # Return the loss
    if (i+1) % print_loss_every == 0:
        loss_val = sess.run(loss, feed_dict=feed_dict)
        loss_vec.append(loss_val)
        loss_x_vec.append(i+1)
        print('Loss at step {} : {}'.format(i+1, loss_val))

    # Validation: Print some random words and top 5 related words
    if (i+1) % print_valid_every == 0:
        sim = sess.run(similarity, feed_dict=feed_dict)

```

```

        for j in range(len(valid_words)):
            valid_word = word_dictionary_rev[valid_examples[j]]
            top_k = 5 # number of nearest neighbors
            nearest = (-sim[j, :]).argsort()[1:top_k+1]
            log_str = "Nearest to {}:".format(valid_word)
            for k in range(top_k):
                close_word = word_dictionary_rev[nearest[k]]
                print_str = '{} {}'.format(log_str, close_word)
            print(print_str)

        # Save dictionary + embeddings
        if (i+1) % save_embeddings_every == 0:
            # Save vocabulary dictionary
            with open(os.path.join(data_folder_name, 'movie_vocab.pkl'), 'wb') as f:
                pickle.dump(word_dictionary, f)

            # Save embeddings
            model_checkpoint_path = os.path.join(os.getcwd(), data_folder_name, 'cbow_'
            save_path = saver.save(sess, model_checkpoint_path)
            print('Model saved in file: {}'.format(save_path))

```

13. This results in the following output:

```

Loss at step 100 : 62.04829025268555
Loss at step 200 : 33.182334899902344
...
Loss at step 49900 : 1.6794960498809814
Loss at step 50000 : 1.5071022510528564
Nearest to love: clarity, cult, cliched, literary, memory,
Nearest to hate: bringing, gifted, almost, next, wish,
Nearest to happy: ensemble, fall, courage, uneven, girls,
Nearest to sad: santa, devoid, biopic, genuinely, becomes,
Nearest to man: project, stands, none, soul, away,
Nearest to woman: crush, even, x, team, ensemble,
Model saved in file: .../temp/cbow_movie_embeddings.ckpt

```

14. All but one of the functions in the `text_helpers.py` file have functions that come directly from the previous recipe. We will make a slight addition to the `generate_batch_data()` function by adding a `cbow` method as follows:

```

elif method=='cbow':
    batch_and_labels = [(x[:y] + x[(y+1):], x[y]) for x,y in zip(window_sequence
    # Only keep windows with consistent 2*window_size
    batch_and_labels = [(x,y) for x,y in batch_and_labels if len(x)==2*window_si
    batch, labels = [list(x) for x in zip(*batch_and_labels)]
```

How it works...

This recipe works very similarly to creating embeddings with Skip-Gram. The main difference is how we generate the data and combine the embeddings.

For this recipe, we loaded the data, normalized the text, created a vocabulary dictionary, used the dictionary to look up embeddings, combined the embeddings, and trained a neural network to predict the target word.

There's more...

It is worthwhile to note that the `CBOW` method trains on a summed-up embedding of the surrounding window to predict the target word. One effect of this is that the `CBOW` method from word2vec has a smoothing effect that the `Skip-Gram` method lacks, and it is reasonable to think that this might be preferred for smaller textual datasets.

Making predictions with word2vec

In this recipe, we will use the previously learned embedding strategies to perform classification.

Getting ready

Now that we have created and saved CBOW word embeddings, we need to use them to make sentiment predictions on the movie dataset. In this recipe, we will learn how to load and use pre-trained embeddings and use these embeddings to perform sentiment analysis by training a logistic linear model to predict a good or bad review.

Sentiment analysis is a really hard task to perform because the human language makes it very hard to grasp the subtleties and nuances of the true meaning of what is meant. Sarcasm, jokes, and ambiguous references all make this task exponentially harder. We will create a simple logistic regression on the movie review dataset to see if we can get any information out of the CBOW embeddings we created and saved in the previous recipe. Since the focus of this recipe is in the loading and use of saved embeddings, we will not pursue more complicated models.

How to do it...

We will proceed with the recipe as follows:

1. We will begin by loading the necessary libraries and starting a graph session:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import random
import os
import pickle
import string
import requests
import collections
import io
import tarfile
import urllib.request
import text_helpers
from nltk.corpus import stopwords
sess = tf.Session()
```

2. Now we will declare the model parameters. The embedding size should be the same as the embedding size we used to create the preceding CBOW embeddings. Use the following code to do this:

```
embedding_size = 200
vocabulary_size = 2000
batch_size = 100
max_words = 100
stops = stopwords.words('english')
```

3. We will load and transform the text data from the `text_helpers.py` file that we have created. Use the following code to do this:

```
texts, target = text_helpers.load_movie_data()
# Normalize text
print('Normalizing Text Data')
texts = text_helpers.normalize_text(texts, stops)
# Texts must contain at least 3 words
target = [target[ix] for ix, x in enumerate(texts) if len(x.split()) > 2]
texts = [x for x in texts if len(x.split()) > 2]
train_indices = np.random.choice(len(target), round(0.8*len(target)), replace=False)
test_indices = np.array(list(set(range(len(target))) - set(train_indices)))
texts_train = [x for ix, x in enumerate(texts) if ix in train_indices]
texts_test = [x for ix, x in enumerate(texts) if ix in test_indices]
target_train = np.array([x for ix, x in enumerate(target) if ix in train_indices])
target_test = np.array([x for ix, x in enumerate(target) if ix in test_indices])
```

- We now load the word dictionary that we created while fitting the CBOW embeddings. It is important that we load this so that we have the exact same mapping from word to embedding index, as follows:

```
| dict_file = os.path.join(data_folder_name, 'movie_vocab.pkl')
| word_dictionary = pickle.load(open(dict_file, 'rb'))
```

- We can now convert our loaded sentence data to a numerical `numpy` array with our word dictionary:

```
| text_data_train = np.array(text_helpers.text_to_numbers(texts_train, word_dictio
| text_data_test = np.array(text_helpers.text_to_numbers(texts_test, word_dictiona
```

- Since movie reviews are of different lengths, we standardize them so they're all the same length. In our case, we set it to 100 words. If a review has fewer than 100 words, we will pad it with zeros. Use the following code to do this:

```
| text_data_train = np.array([x[0:max_words] for x in [y+[0]*max_words for y in te
| text_data_test = np.array([x[0:max_words] for x in [y+[0]*max_words for y in tex
```

- Now we will declare our model variables and placeholders for the logistic regression. Use the following code to do this:

```
A = tf.Variable(tf.random_normal(shape=[embedding_size,1]))
b = tf.Variable(tf.random_normal(shape=[1,1]))
# Initialize placeholders
x_data = tf.placeholder(shape=[None, max_words], dtype=tf.int32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
```

- In order for TensorFlow to restore our pre-trained embeddings, we must first give the `saver` method a variable to restore into, so we will create an embedding variable that is the same shape as the embeddings we will load:

```
| embeddings = tf.Variable(tf.random_uniform([vocabulary_size, embedding_size], -1
```

- Now we will put our `embedding_lookup` function on the graph and take the average embedding of all the words in the sentence. Use the following code to do this:

```
| embed = tf.nn.embedding_lookup(embeddings, x_data)
| # Take average of all word embeddings in documents
| embed_avg = tf.reduce_mean(embed, 1)
```

- Next, we will declare our model operations and our loss function, remembering that our loss function has the sigmoid operation built in

already, as follows:

```
model_output = tf.add(tf.matmul(embed_avg, A), b)
# Declare loss function (Cross Entropy loss)
loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=model_outpu
```

11. Now we will add prediction and accuracy functions to the graph so that we can evaluate the accuracy as the model is trained with the following code:

```
prediction = tf.round(tf.sigmoid(model_output))
predictions_correct = tf.cast(tf.equal(prediction, y_target), tf.float32)
accuracy = tf.reduce_mean(predictions_correct)
```

12. We will declare our optimization function and initialize the following model variables:

```
my_opt = tf.train.AdagradOptimizer(0.005)
train_step = my_opt.minimize(loss)
init = tf.global_variables_initializer()
sess.run(init)
```

13. Now that we have a random initialized embedding, we can tell the `saver` method to load our previous CBOW embeddings into our embedding variable. Use the following code to do this:

```
model_checkpoint_path = os.path.join(data_folder_name, 'cbow_movie_embeddings.ckpt')
saver = tf.train.Saver({'embeddings': embeddings})
saver.restore(sess, model_checkpoint_path)
```

14. Now we can start the training generations. Note that we save the training and test loss and accuracy every 100 generations. We will only print out the model status every 500 generations, as follows:

```
train_loss = []
test_loss = []
train_acc = []
test_acc = []
i_data = []
for i in range(10000):
    rand_index = np.random.choice(text_data_train.shape[0], size=batch_size)
    rand_x = text_data_train[rand_index]
    rand_y = np.transpose([target_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})

    # Only record loss and accuracy every 100 generations
    if (i+1)%100==0:
        i_data.append(i+1)
        train_loss_temp = sess.run(loss, feed_dict={x_data: rand_x, y_target: ra
        train_loss.append(train_loss_temp)

        test_loss_temp = sess.run(loss, feed_dict={x_data: text_data_test, y_tar
        test_loss.append(test_loss_temp)
```

```

train_acc_temp = sess.run(accuracy, feed_dict={x_data: rand_x, y_target:
train_acc.append(train_acc_temp)
test_acc_temp = sess.run(accuracy, feed_dict={x_data: text_data_test, y_
test_acc.append(test_acc_temp)
if (i+1)%500==0:
    acc_and_loss = [i+1, train_loss_temp, test_loss_temp, train_acc_temp, te
    acc_and_loss = [np.round(x,2) for x in acc_and_loss]
    print('Generation # {}. Train Loss (Test Loss): {:.2f} ({:.2f}). Train A

```

15. The following output is the result:

```

Generation # 500. Train Loss (Test Loss): 0.70 (0.71). Train Acc (Test Acc): 0.5
Generation # 1000. Train Loss (Test Loss): 0.69 (0.72). Train Acc (Test Acc): 0.
...
Generation # 9500. Train Loss (Test Loss): 0.69 (0.70). Train Acc (Test Acc): 0.
Generation # 10000. Train Loss (Test Loss): 0.70 (0.70). Train Acc (Test Acc): 0

```

16. The following is the code to plot the training and test loss and accuracy, which we saved every 100 generations:

```

# Plot loss over time
plt.plot(i_data, train_loss, 'k-', label='Train Loss')
plt.plot(i_data, test_loss, 'r--', label='Test Loss', linewidth=4)
plt.title('Cross Entropy Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Cross Entropy Loss')
plt.legend(loc='upper right')
plt.show()

# Plot train and test accuracy
plt.plot(i_data, train_acc, 'k-', label='Train Set Accuracy')
plt.plot(i_data, test_acc, 'r--', label='Test Set Accuracy', linewidth=4)
plt.title('Train and Test Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

```

The plot of the cross entropy loss per generation is as follows:

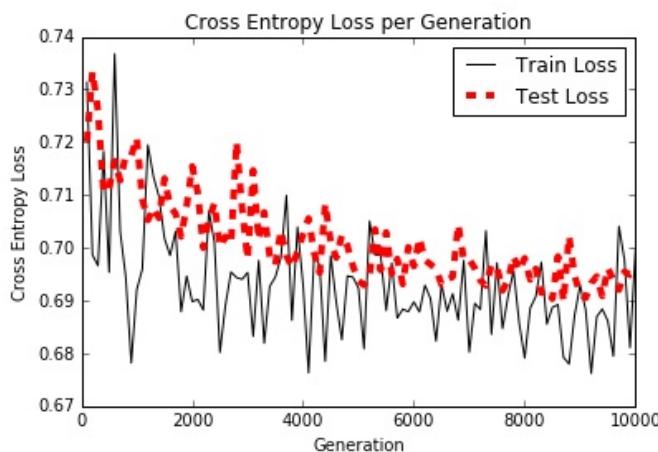


Figure 6: Here we observe the train and test loss over 10,000 generations

The plot of the train and test accuracy for the preceding code is as follows:

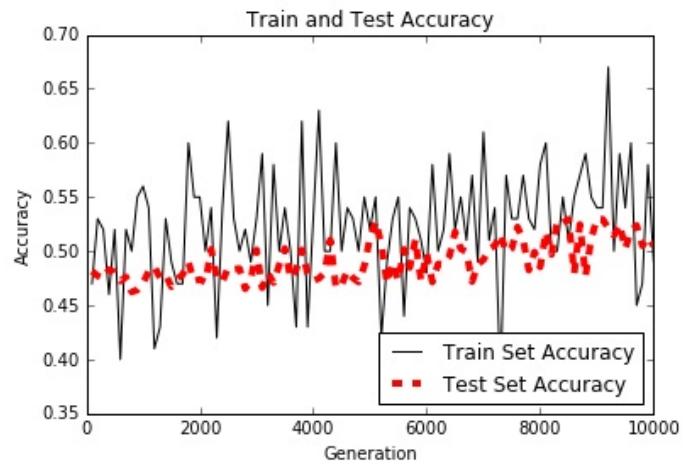


Figure 7: We can observe that the train and test set accuracy is slowly improving over 10,000 generations. It is worthwhile to note that this model performs very poorly and is only slightly better than a random predictor.

How it works...

We loaded our preceding CBOW embeddings and performed logistic regression on the average embedding of a review. The important methods to note here are how we load model variables from the disk onto already initialized variables in our current model. We also have to remember to store and load our vocabulary dictionary created prior to training the embeddings. It is very important to have the same mapping from words to embedding indices when using the same embedding.

There's more...

We can see that we almost achieve 60% accuracy on predicting the sentiment. For example, it is a hard task to know the meaning behind the word *great*; it could be used in a negative or positive context within a review.

To tackle this problem, we want to somehow create embeddings for the documents themselves and address the sentiment issue. Usually, a whole review is positive or a whole review is negative. We can use this to our advantage, and we will look at how to do this in the *Using doc2vec for sentiment analysis* recipe that follows.

Using doc2vec for sentiment analysis

Now that we know how to train word embeddings, we can also extend these methodologies to have a document embedding. We will explore how to do this in the following sections.

Getting ready

In the previous sections about word2vec methods, we managed to capture positional relationships between words. What we have not done is capture the relationship of words to the document (or movie review) that they come from. One extension of word2vec that captures a document effect is called **doc2vec**.

The basic idea of doc2vec is to introduce document embedding, along with the word embeddings that may help to capture the tone of the document. For example, just knowing that the words *movie* and *love* are near to each other may not help us determine the sentiment of the review. The review may be talking about how they love the movie or how they do not love the movie. But if the review is long enough and more negative words are found in the document, maybe we can pick up on an overall tone that may help us predict the words that follow.

Doc2vec simply adds an additional embedding matrix for the documents and uses a window of words plus the document index to predict the next word. All word windows in a document have the same document index. It is worth mentioning that it is important to think about how we will combine the document embedding and the word embeddings. We combine the word embeddings in the word window by summing them. There are two main ways to combine these embeddings with the document embedding: commonly, the document embedding is either added to the word embeddings, or concatenated to the end of the word embeddings. If we add the two embeddings, we limit the document embedding size to be the same size as the word embedding size. If we concatenate, we lift that restriction, but increase the number of variables that the logistic regression must deal with. For illustrative purposes, we will show you how to deal with concatenation in this recipe. But in general, for smaller datasets, addition is the better choice.

The first step will be to fit both the document and word embeddings on the whole corpus of movie reviews. Then we will perform a train-test split, train a logistic model, and see if we can make predicting the review sentiment more accurate.

How to do it...

We will proceed with the recipe as follows:

1. We will start by loading the necessary libraries and starting a graph session, as follows:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import random
import os
import pickle
import string
import requests
import collections
import io
import tarfile
import urllib.request
import text_helpers
from nltk.corpus import stopwords
sess = tf.Session()
```

2. We will load the movie review corpus, just as we did in the previous two recipes. Use the following code to do this:

```
| texts, target = text_helpers.load_movie_data()
```

3. We will declare the model parameters, like so:

```
batch_size = 500
vocabulary_size = 7500
generations = 100000
model_learning_rate = 0.001
embedding_size = 200    # Word embedding size
doc_embedding_size = 100    # Document embedding size
concatenated_size = embedding_size + doc_embedding_size
num_sampled = int(batch_size/2)
window_size = 3          # How many words to consider to the left.
# Add checkpoints to training
save_embeddings_every = 5000
print_valid_every = 5000
print_loss_every = 100
# Declare stop words
stops = stopwords.words('english')
# We pick a few test words.
valid_words = ['love', 'hate', 'happy', 'sad', 'man', 'woman']
```

4. We will normalize the movie reviews and make sure that each movie review is larger than the desired window size. Use the following code to do

this:

```
texts = text_helpers.normalize_text(texts, stops)
# Texts must contain at least as much as the prior window size
target = [target[ix] for ix, x in enumerate(texts) if len(x.split()) > window_size]
texts = [x for x in texts if len(x.split()) > window_size]
assert(len(target)==len(texts))
```

5. Now we will create our word dictionary. It is important to note that we do not have to create a document dictionary. The document indices will just be the index of the document; each document will have a unique index:

```
word_dictionary = text_helpers.build_dictionary(texts, vocabulary_size)
word_dictionary_rev = dict(zip(word_dictionary.values(), word_dictionary.keys()))
text_data = text_helpers.text_to_numbers(texts, word_dictionary)
# Get validation word keys
valid_examples = [word_dictionary[x] for x in valid_words]
```

6. Next, we will define our word embeddings and document embeddings. Then we will declare our noise-contrastive loss parameters. Use the following code to do this:

```
embeddings = tf.Variable(tf.random_uniform([vocabulary_size, embedding_size], -1
doc_embeddings = tf.Variable(tf.random_uniform([len(texts), doc_embedding_size],
# NCE loss parameters
nce_weights = tf.Variable(tf.truncated_normal([vocabulary_size, concatenated_size],
                                              stddev=1.0 / np.sqrt(concatenated_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size])))
```

7. We will now declare our placeholders for the doc2vec indices and target word index. Note that the size of the input indices is the window size plus 1. This is because every data window we generate will have an additional document index with it, as follows:

```
x_inputs = tf.placeholder(tf.int32, shape=[None, window_size + 1])
y_target = tf.placeholder(tf.int32, shape=[None, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
```

8. Now we have to create our embedding function, which adds together the word embeddings and then concatenates the document embedding at the end. Use the following code to do this:

```
embed = tf.zeros([batch_size, embedding_size])
for element in range(window_size):
    embed += tf.nn.embedding_lookup(embeddings, x_inputs[:, element])
doc_indices = tf.slice(x_inputs, [0,window_size],[batch_size,1])
doc_embed = tf.nn.embedding_lookup(doc_embeddings,doc_indices)
# concatenate embeddings
final_embed = tf.concat(axis=1, values=)
```

9. We also need to declare the cosine distance from a set of validation words that we can print out every so often to observe the progress of our doc2vec model. Use the following code to do this:

```

loss = tf.reduce_mean(tf.nn.nce_loss(weights=nce_weights,
                                      biases=nce_biases,
                                      labels=y_target,
                                      inputs=final_embed,
                                      num_sampled=num_sampled,
                                      num_classes=vocabulary_size))

# Create optimizer
optimizer =
    tf.train.GradientDescentOptimizer(learning_rate=model_learning_rate)
train_step = optimizer.minimize(loss)

```

10. We also need to declare the cosine distance from a set of validation words, which we can print out every so often to observe the progress of our doc2vec model. Use the following code to do this:

```

norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1,
                             keep_dims=True))
normalized_embeddings = embeddings / norm
valid_embeddings = tf.nn.embedding_lookup(normalized_embeddings,
                                          valid_dataset)
similarity = tf.matmul(valid_embeddings, normalized_embeddings,
                       transpose_b=True)

```

11. To save our embeddings for later, we will create a model saver function. Then we can initialize the variables, which is the last step before we commence training on the word embeddings:

```

saver = tf.train.Saver({"embeddings": embeddings, "doc_embeddings": doc_embeddings})
init = tf.global_variables_initializer()
sess.run(init)
loss_vec = []
loss_x_vec = []
for i in range(generations):
    batch_inputs, batch_labels = text_helpers.generate_batch_data(text_data, batch_size, window_size, m)
    feed_dict = {x_inputs : batch_inputs, y_target : batch_labels}

    # Run the train step
    sess.run(train_step, feed_dict=feed_dict)

    # Return the loss
    if (i+1) % print_loss_every == 0:
        loss_val = sess.run(loss, feed_dict=feed_dict)
        loss_vec.append(loss_val)
        loss_x_vec.append(i+1)
        print('Loss at step {} : {}'.format(i+1, loss_val))

    # Validation: Print some random words and top 5 related words
    if (i+1) % print_valid_every == 0:
        sim = sess.run(similarity, feed_dict=feed_dict)

```

```

        for j in range(len(valid_words)):
            valid_word = word_dictionary_rev[valid_examples[j]]
            top_k = 5 # number of nearest neighbors
            nearest = (-sim[j, :]).argsort()[1:top_k+1]
            log_str = "Nearest to {}:".format(valid_word)
            for k in range(top_k):
                close_word = word_dictionary_rev[nearest[k]]
                log_str = '{} {}'.format(log_str, close_word)
            print(log_str)

        # Save dictionary + embeddings
        if (i+1) % save_embeddings_every == 0:
            # Save vocabulary dictionary
            with open(os.path.join(data_folder_name, 'movie_vocab.pkl'), 'wb') as f:
                pickle.dump(word_dictionary, f)

            # Save embeddings
            model_checkpoint_path = os.path.join(os.getcwd(), data_folder_name, 'doc2v')
            save_path = saver.save(sess, model_checkpoint_path)
            print('Model saved in file: {}'.format(save_path))

```

12. This results in the following output:

```

Loss at step 100 : 126.176816940307617
Loss at step 200 : 89.608322143554688
...
Loss at step 99900 : 17.733346939086914
Loss at step 100000 : 17.384489059448242
Nearest to love: ride, with, by, its, start,
Nearest to hate: redundant, snapshot, from, performances, extravagant,
Nearest to happy: queen, chaos, them, succumb, elegance,
Nearest to sad: terms, pity, chord, wallet, morality,
Nearest to man: of, teen, an, our, physical,
Nearest to woman: innocuous, scenes, prove, except, lady,
Model saved in file: ../../temp/doc2vec_movie_embeddings.ckpt

```

13. Now that we have trained the doc2vec embeddings, we can use these embeddings in a logistic regression to predict the review sentiment. First, we set some parameters for the logistic regression. Use the following code to do this:

```

max_words = 20 # maximum review word length
logistic_batch_size = 500 # training batch size

```

14. We will now split the dataset into train and test sets:

```

train_indices = np.sort(np.random.choice(len(target),
round(0.8*len(target)), replace=False))
test_indices = np.sort(np.array(list(set(range(len(target))) -
set(train_indices))))
texts_train = [x for ix, x in enumerate(texts) if ix in train_indices]
texts_test = [x for ix, x in enumerate(texts) if ix in test_indices]
target_train = np.array([x for ix, x in enumerate(target) if ix in train_indices])
target_test = np.array([x for ix, x in enumerate(target) if ix in test_indices])

```

15. Next, we will convert the reviews into numerical word indices and pad or

crop each review to 20 words, as follows:

```
|     text_data_train = np.array(text_helpers.text_to_numbers(texts_train, word_dictio
```

16. Now we will declare the parts of the graph that pertain to the logistic regression model. We will add the data placeholders, the variables, model operations, and the loss function, as follows:

```
# Define Logistic placeholders
log_x_inputs = tf.placeholder(tf.int32, shape=[None, max_words + 1])
log_y_target = tf.placeholder(tf.int32, shape=[None, 1])
A = tf.Variable(tf.random_normal(shape=[concatenated_size, 1]))
b = tf.Variable(tf.random_normal(shape=[1, 1]))

# Declare logistic model (sigmoid in loss function)
model_output = tf.add(tf.matmul(log_final_embed, A), b)

# Declare loss function (Cross Entropy loss)
logistic_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=model_output, labels=tf.cast(log_y_target, tf.float32)))
```

17. We need to create another embedding function. The embedding function in the first half is trained on a smaller window of three words (and a document index) to predict the next word. Here, we will do the same but with the 20 word review. Use the following code to do this:

```
# Add together element embeddings in window:
log_embed = tf.zeros([logistic_batch_size, embedding_size])
for element in range(max_words):
    log_embed += tf.nn.embedding_lookup(embeddings, log_x_inputs[:, element])
log_doc_indices = tf.slice(log_x_inputs, [0, max_words], [logistic_batch_size, 1])
log_doc_embed = tf.nn.embedding_lookup(doc_embeddings, log_doc_indices)
# concatenate embeddings
log_final_embed = tf.concat(1, [log_embed, tf.squeeze(log_doc_embed)])
```

18. Next, we will create a prediction and accuracy function on the graph so that we can evaluate the performance of the model as we run through the training generations. Then we will declare an optimizing function and initialize all the variables:

```
prediction = tf.round(tf.sigmoid(model_output))
predictions_correct = tf.cast(tf.equal(prediction, tf.cast(log_y_target, tf.float32)), tf.int32)
accuracy = tf.reduce_mean(predictions_correct)
# Declare optimizer
logistic_opt = tf.train.GradientDescentOptimizer(learning_rate=0.01)
logistic_train_step = logistic_opt.minimize(logistic_loss, var_list=[A, b])
# Initialize Variables
init = tf.global_variables_initializer()
sess.run(init)
```

19. Now we can start the logistic model training:

```

train_loss = []
test_loss = []
train_acc = []
test_acc = []
i_data = []
for i in range(10000):
    rand_index = np.random.choice(text_data_train.shape[0], size=logistic_batch_
    rand_x = text_data_train[rand_index]
    # Append review index at the end of text data
    rand_x_doc_indices = train_indices[rand_index]
    rand_x = np.hstack((rand_x, np.transpose([rand_x_doc_indices])))
    rand_y = np.transpose([target_train[rand_index]])

    feed_dict = {log_x_inputs : rand_x, log_y_target : rand_y}
    sess.run(logistic_train_step, feed_dict=feed_dict)

    # Only record loss and accuracy every 100 generations
    if (i+1)%100==0:
        rand_index_test = np.random.choice(text_data_test.shape[0], size=logisti
        rand_x_test = text_data_test[rand_index_test]
        # Append review index at the end of text data
        rand_x_doc_indices_test = test_indices[rand_index_test]
        rand_x_test = np.hstack((rand_x_test, np.transpose([rand_x_doc_indices_t
        rand_y_test = np.transpose([target_test[rand_index_test]]))

        test_feed_dict = {log_x_inputs: rand_x_test, log_y_target: rand_y_test}

        i_data.append(i+1)
        train_loss_temp = sess.run(logistic_loss, feed_dict=feed_dict)
        train_loss.append(train_loss_temp)

        test_loss_temp = sess.run(logistic_loss, feed_dict=test_feed_dict)
        test_loss.append(test_loss_temp)

        train_acc_temp = sess.run(accuracy, feed_dict=feed_dict)
        train_acc.append(train_acc_temp)

        test_acc_temp = sess.run(accuracy, feed_dict=test_feed_dict)
        test_acc.append(test_acc_temp)
    if (i+1)%500==0:
        acc_and_loss = [i+1, train_loss_temp, test_loss_temp, train_acc_temp, te
        acc_and_loss = [np.round(x,2) for x in acc_and_loss]
        print('Generation # {}. Train Loss (Test Loss): {:.2f} ({:.2f}). Train A

```

20. This results in the following output:

```
| Generation # 500. Train Loss (Test Loss): 5.62 (7.45). Train Acc (Test Acc): 0.5
```

21. We should also note that we have created a separate data batch generating method in the `text_helpers.generate_batch_data()` function called **doc2vec**, which we used in the first part of this recipe to train the doc2vec embeddings. The following is the excerpt from that function that pertains to this method:

```

def generate_batch_data(sentences, batch_size, window_size, method='skip_gram'):
    # Fill up data batch
    batch_data = []
    label_data = []

```

```

while len(batch_data) < batch_size:
    # select random sentence to start
    rand_sentence_ix = int(np.random.choice(len(sentences), size=1))
    rand_sentence = sentences[rand_sentence_ix]
    # Generate consecutive windows to look at
    window_sequences = [rand_sentence[max((ix-window_size),0):(ix+window_size)] for ix in range(len(rand_sentence))]
    # Denote which element of each window is the center word of interest
    label_indices = [ix if ix<window_size else window_size for ix,x in enumerate(window_sequences)]
    # Pull out center word of interest for each window and create a tuple for labels
    if method=='skip_gram':
        ...
    elif method=='cbow':
        ...
    elif method=='doc2vec':
        # For doc2vec we keep LHS window only to predict target word
        batch_and_labels = [(rand_sentence[i:i+window_size], rand_sentence[i]) for i in range(len(rand_sentence))]
        batch, labels = [list(x) for x in zip(*batch_and_labels)]
        # Add document index to batch!! Remember that we must extract the last element of the list
        batch = [x + [rand_sentence_ix] for x in batch]
    else:
        raise ValueError('Method {} not implemented yet.'.format(method))

    # extract batch and labels
    batch_data.extend(batch[:batch_size])
    label_data.extend(labels[:batch_size])
# Trim batch and label at the end
batch_data = batch_data[:batch_size]
label_data = label_data[:batch_size]

# Convert to numpy array
batch_data = np.array(batch_data)
label_data = np.transpose(np.array([label_data]))

return batch_data, label_data

```

How it works...

In this recipe, we performed two training loops. The first was to fit the doc2vec embeddings, and the second loop was to fit the logistic regression on the movie sentiment.

While we did not increase the sentiment prediction accuracy by much (it is still slightly under 60%), we successfully implemented the concatenation version of doc2vec on the movie corpus. To increase our accuracy, we should try different parameters for the doc2vec embeddings and possibly a more complicated model, as logistic regression may not be able to capture all non-linear behavior in natural language.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are responsible for the major breakthroughs in image recognition made in the past few years. In this chapter, we will cover the following topics:

- Implementing a simple CNN
- Implementing an advanced CNN
- Retraining existing CNN models
- Applying StyleNet and the neural-style project
- Implementing DeepDream



As a reminder, the reader can find all of the code for this chapter available online here: https://github.com/nfmccleure/tensorflow_cookbook, as well as the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

Introduction

In mathematics, a convolution is a function that is applied over the output of another function. In our case, we will consider applying a matrix multiplication (filter) across an image. For our purposes, we consider an image to be a matrix of numbers. These numbers may represent pixels or even image attributes. The convolution operation we will apply to these matrices involves moving a filter of fixed width across the image and applying element-wise multiplication to get our result.

See the following diagram for a conceptual understanding of how image convolution can work:

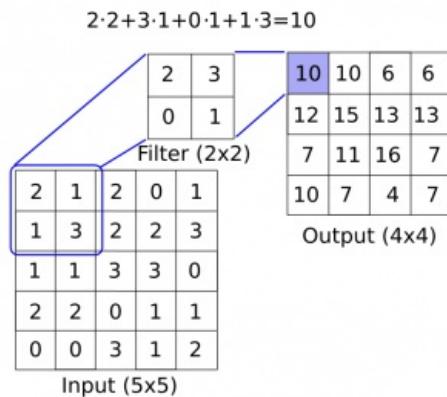


Figure 1: How a convolutional filter applied across an image (length by width by depth) operates to create a new feature layer. Here, we have a 2×2 convolutional filter, operating in the valid spaces of the 5×5 input with a stride of 1 in both directions. The result is a 4×4 matrix

CNNs also have other operations that fulfill more requirements, such as introducing non-linearities (ReLU), or aggregating parameters (max P\pool), and other similar operations. The preceding diagram is an example of applying a convolution operation on a 5×5 array with the convolutional filter being a 2×2 matrix. The step size is 1 and we only consider valid placements. The trainable variables in this operation will be the 2×2 filter weights. After a convolution, it is common to follow up with an aggregation operation, such as max pool. The following diagram provides an example of how a max pool operates, if we take the max of a 2×2 region with a stride of 2 in both directions:

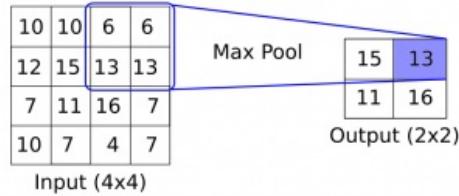


Figure 2: An example of how a Max Pool operation could operate. Here, we have a 2×2 window, operating on the valid spaces of a 4×4 input with a stride of 2 in both directions. The result is a 2×2 matrix

Although we will start by creating our own CNN for image recognition, it is highly recommended you to use existing architectures, as we will do in the remainder of the chapter.



It is common to take a pre-trained network and retrain it with a new dataset with a new fully connected layer at the end. This method is very useful, and we will illustrate it in the Retraining existing CNN models recipe, where we will retrain an existing architecture to improve on our CIFAR-10 predictions.

Implementing a simple CNN

In this recipe, we will develop a four-layer convolutional neural network to improve upon our accuracy in predicting MNIST digits. The first two convolution layers will each be composed of convolution-ReLU-Max Pool operations, and the final two layers will be fully connected layers.

Getting ready

To access the MNIST data, TensorFlow has an `examples.tutorials` package that has great dataset-loading functionalities. After we load the data, we will set up our model variables, create the model, train the model in batches, and then visualize loss, accuracy, and some sample digits.

How to do it...

Perform the following steps:

1. First, we'll load the necessary libraries and start a graph session:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
from tensorflow.python.framework import ops
ops.reset_default_graph()

sess = tf.Session()
```

2. Next, we will load the data and transform the images into 28x28 arrays:

```
data_dir = 'temp'
mnist = input_data.read_data_sets(data_dir, one_hot=False)
train_xdata = np.array([np.reshape(x, (28,28)) for x in mnist.train.images])
test_xdata = np.array([np.reshape(x, (28,28)) for x in mnist.test.images])
train_labels = mnist.train.labels
test_labels = mnist.test.labels
```



Note that the MNIST dataset downloaded here also includes a validation set. This validation set is normally the same size as the test set. If we do any hyperparameter tuning or model selection, it will be wise to load this in as well for additional testing.

3. Now we will set the model parameters. Remember that the depth of the image (number of channels) is 1 because these images are grayscale:

```
batch_size = 100
learning_rate = 0.005
evaluation_size = 500
image_width = train_xdata[0].shape[0]
image_height = train_xdata[0].shape[1]
target_size = max(train_labels) + 1
num_channels = 1
generations = 500
eval_every = 5
conv1_features = 25
conv2_features = 50
max_pool_size1 = 2
max_pool_size2 = 2
fully_connected_size1 = 100
```

4. We can now declare our placeholders for the data. We'll declare our training data variables and our test data variables. We will have different batch sizes for training and evaluation sizes. You may change these, depending on the physical memory that is available for training and evaluating:

```

x_input_shape = (batch_size, image_width, image_height, num_channels)
x_input = tf.placeholder(tf.float32, shape=x_input_shape)
y_target = tf.placeholder(tf.int32, shape=(batch_size))
eval_input_shape = (evaluation_size, image_width, image_height, num_channels)
eval_input = tf.placeholder(tf.float32, shape=eval_input_shape)
eval_target = tf.placeholder(tf.int32, shape=(evaluation_size))

```

5. We'll declare our convolution weights and biases with the parameters we set up in the preceding steps:

```

conv1_weight = tf.Variable(tf.truncated_normal([4, 4, num_channels, conv1_features], stddev=0.1))
conv1_bias = tf.Variable(tf.zeros([conv1_features]), dtype=tf.float32)
conv2_weight = tf.Variable(tf.truncated_normal([4, 4, conv1_features, conv2_features], stddev=0.1))
conv2_bias = tf.Variable(tf.zeros([conv2_features]), dtype=tf.float32)

```

6. Next, we are going to declare our fully connected weights and biases for the last two layers of the model:

```

resulting_width = image_width // (max_pool_size1 * max_pool_size2)
resulting_height = image_height // (max_pool_size1 * max_pool_size2)
full1_input_size = resulting_width * resulting_height * conv2_features
full1_weight = tf.Variable(tf.truncated_normal([full1_input_size, fully_connected_size1], stddev=0.1))
full1_bias = tf.Variable(tf.truncated_normal([fully_connected_size1], stddev=0.1))
full2_weight = tf.Variable(tf.truncated_normal([fully_connected_size1, target_size], stddev=0.1))
full2_bias = tf.Variable(tf.truncated_normal([target_size], stddev=0.1), dtype=tf.float32)

```

7. Now we'll declare our model. We do this first by creating a model function. Note that the function will look in the global scope for the required layer weights and biases. Also, to get the fully connected layer to work, we flatten the output of the second convolutional layer, so we can use it in the fully connected layer:

```

def my_conv_net(input_data):
    # First Conv-ReLU-MaxPool Layer
    conv1 = tf.nn.conv2d(input_data, conv1_weight, strides=[1, 1, 1, 1], padding='SAME')
    relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_bias))
    max_pool1 = tf.nn.max_pool(relu1, ksize=[1, max_pool_size1, max_pool_size1, 1], strides=[1, max_pool_size1, max_pool_size1, 1], padding='SAME')
    # Second Conv-ReLU-MaxPool Layer
    conv2 = tf.nn.conv2d(max_pool1, conv2_weight, strides=[1, 1, 1, 1], padding='SAME')
    relu2 = tf.nn.relu(tf.nn.bias_add(conv2, conv2_bias))
    max_pool2 = tf.nn.max_pool(relu2, ksize=[1, max_pool_size2, max_pool_size2, 1], strides=[1, max_pool_size2, max_pool_size2, 1], padding='SAME')
    # Transform Output into a 1xN layer for next fully connected layer
    final_conv_shape = max_pool2.get_shape().as_list()
    final_shape = final_conv_shape[1] * final_conv_shape[2] * final_conv_shape[3]
    flat_output = tf.reshape(max_pool2, [final_conv_shape[0], final_shape])
    # First Fully Connected Layer
    fully_connected1 = tf.nn.relu(tf.add(tf.matmul(flat_output, full1_weight), full1_bias))
    # Second Fully Connected Layer
    final_model_output = tf.add(tf.matmul(fully_connected1, full2_weight), full2_bias)
    return final_model_output

```

8. Next, we can declare the model on the training and test data:

```

model_output = my_conv_net(x_input)

```

```
|     test_model_output = my_conv_net(eval_input)
```

9. The loss function we will use is the softmax function. We use a sparse softmax because our predictions will be only one category and not multiple categories. We are also going to use a loss function that operates on logits and not scaled probabilities:

```
|     loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits=mode
```

10. Next, we'll create a training and test prediction functions. Then we will also create an accuracy function to determine how accurate the model is on each batch:

```
prediction = tf.nn.softmax(model_output)
test_prediction = tf.nn.softmax(test_model_output)
# Create accuracy function
def get_accuracy(logits, targets):
    batch_predictions = np.argmax(logits, axis=1)
    num_correct = np.sum(np.equal(batch_predictions, targets))
    return 100. * num_correct/batch_predictions.shape[0]
```

11. Now we will create our optimization function, declare the training step, and initialize all the model variables:

```
my_optimizer = tf.train.MomentumOptimizer(learning_rate, 0.9)
train_step = my_optimizer.minimize(loss)
# Initialize Variables
init = tf.global_variables_initializer()
sess.run(init)
```

12. We can now start training our model. We loop through the data in randomly chosen batches. Every so often, we choose to evaluate the model on the train and test batches and record the accuracy and loss. We can see that, after 500 generations, we quickly achieve 96%-97% accuracy on the test data:

```
train_loss = []
train_acc = []
test_acc = []
for i in range(generations):
    rand_index = np.random.choice(len(train_xdata), size=batch_size)
    rand_x = train_xdata[rand_index]
    rand_x = np.expand_dims(rand_x, 3)
    rand_y = train_labels[rand_index]
    train_dict = {x_input: rand_x, y_target: rand_y}
    sess.run(train_step, feed_dict=train_dict)
    temp_train_loss, temp_train_preds = sess.run([loss, prediction], feed_dict=temp_train_dict)
    temp_train_acc = get_accuracy(temp_train_preds, rand_y)
    if (i+1) % eval_every == 0:
        eval_index = np.random.choice(len(test_xdata), size=evaluation_size)
        eval_x = test_xdata[eval_index]
```

```

eval_x = np.expand_dims(eval_x, 3)
eval_y = test_labels[eval_index]
test_dict = {eval_input: eval_x, eval_target: eval_y}
test_preds = sess.run(test_prediction, feed_dict=test_dict)
temp_test_acc = get_accuracy(test_preds, eval_y)
# Record and print results
train_loss.append(temp_train_loss)
train_acc.append(temp_train_acc)
test_acc.append(temp_test_acc)
acc_and_loss = [(i+1), temp_train_loss, temp_train_acc, temp_test_acc]
acc_and_loss = [np.round(x,2) for x in acc_and_loss]
print('Generation # {}. Train Loss: {:.2f}. Train Acc (Test Acc): {:.2f}')

```

13. This results in the following output:

```

Generation # 5. Train Loss: 2.37. Train Acc (Test Acc): 7.00 (9.80)
Generation # 10. Train Loss: 2.16. Train Acc (Test Acc): 31.00 (22.00)
Generation # 15. Train Loss: 2.11. Train Acc (Test Acc): 36.00 (35.20)
...
Generation # 490. Train Loss: 0.06. Train Acc (Test Acc): 98.00 (97.40)
Generation # 495. Train Loss: 0.10. Train Acc (Test Acc): 98.00 (95.40)
Generation # 500. Train Loss: 0.14. Train Acc (Test Acc): 98.00 (96.00)

```

14. The following is the code to plot the loss and accuracies using Matplotlib:

```

eval_indices = range(0, generations, eval_every)
# Plot loss over time
plt.plot(eval_indices, train_loss, 'k-')
plt.title('Softmax Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Softmax Loss')
plt.show()

# Plot train and test accuracy
plt.plot(eval_indices, train_acc, 'k-', label='Train Set Accuracy')
plt.plot(eval_indices, test_acc, 'r--', label='Test Set Accuracy')
plt.title('Train and Test Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

```

We then get the following plots:

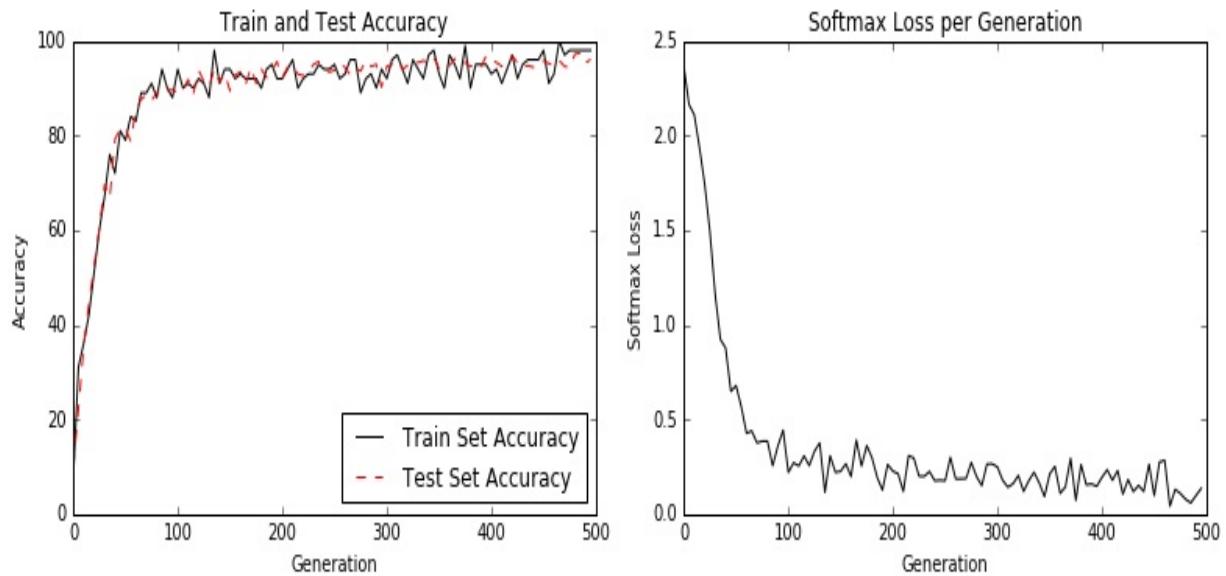


Figure 3: The left plot is the train and test set accuracy across our 500 training generations. The right plot is the softmax loss value over 500 generations.

15. If we want to plot a sample of the latest batch results, here is the code to plot a sample consisting of six of the latest results:

```
# Plot the 6 of the last batch results:
actuals = rand_y[0:6]
predictions = np.argmax(temp_train_preds, axis=1)[0:6]
images = np.squeeze(rand_x[0:6])
Nrows = 2
Ncols = 3
for i in range(6):
    plt.subplot(Nrows, Ncols, i+1)
    plt.imshow(np.reshape(images[i], [28,28]), cmap='Greys_r')
    plt.title('Actual: ' + str(actuals[i]) + ' Pred: ' + str(predictions[i]), fontweight='bold')
    frame = plt.gca()
    frame.axes.get_xaxis().set_visible(False)
    frame.axes.get_yaxis().set_visible(False)
```

We get the following output for the preceding code:

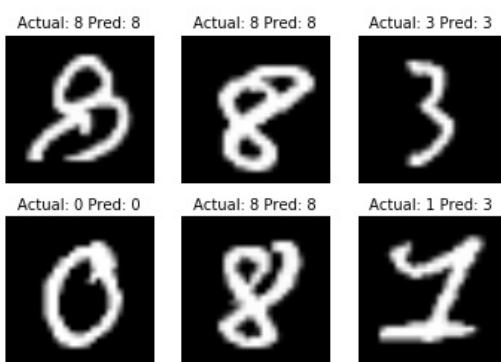


Figure 4: A plot of six random images with the actual and predicted values in the title. The lower-right picture was predicted to be a 3, when in fact it is a 1



How it works...

We increased our performance on the MNIST dataset and built a model that quickly achieves about 97% accuracy while training from scratch. Our first two layers are a combination of convolutions, ReLU, and Max Pooling. The second two layers are fully connected layers. We trained in batches of size 100, and looked at the accuracy and loss across the generations we trained. Finally, we also plotted six random digits and the predictions/actuals for each.

A CNN does very well with image recognition. Part of the reason for this is that the convolutional layer creates its own low-level features that are activated when they come across part of the image that is important. This type of model creates features on its own and uses them for prediction.

There's more...

In the past few years, CNN models have made vast strides in image recognition. Many novel ideas are being explored and new architectures are discovered very frequently. A great repository of papers in this field is a repository website called Arxiv.org (<https://arxiv.org/>), which is created and maintained by Cornell University. Arxiv.org includes some very recent papers in many fields, including computer science and computer science subfields such as computer vision and image recognition (<https://arxiv.org/list/cs.CV/recent>).

See also

Here is a list of some great resources you can use to learn about CNNs:

- Stanford University has a great wiki here: http://scarlet.stanford.edu/teach/index.php/An_Introduction_to_Convolutional_Neural_Networks
- *Deep Learning* by Michael Nielsen, found here: <http://neuralnetworksanddeeplearning.com/chap6.html>
- *An Introduction to Convolutional Neural Networks* by Jianxin Wu, found here: <https://pdfs.semanticscholar.org/450c/a19932fce1ca6d0442cbf52fec38fb9d1e5.pdf>

Implementing an advanced CNN

It is important to be able to extend CNN models for image recognition so that we understand how to increase the depth of the network. This may increase the accuracy of our predictions if we have enough data. Extending the depth of CNN networks is done in a standard fashion: we just repeat the convolution, max pool, and ReLU in series until we are satisfied with the depth. Many of the more accurate image recognition networks operate in this fashion.

Getting ready

In this recipe, we will implement a more advanced method of reading image data and use a larger CNN to do image recognition on the CIFAR10 dataset (<https://www.cs.toronto.edu/~kriz/cifar.html>). This dataset has 60,000 32x32 images that fall into exactly one of ten possible classes. The potential classes for the images are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Please also refer to the first bullet point in the *See also* section.

Most image datasets will be too large to fit into memory. What we can do with TensorFlow is set up an image pipeline to read in one batch at a time from a file. We do this by essentially setting up an image reader, and then creating a batch queue that operates on the image reader.

Also, with image recognition data, it is common to randomly perturb the image before sending it through for training. Here, we will randomly crop, flip, and change the brightness.

This recipe is an adapted version of the official TensorFlow CIFAR-10 tutorial, which is available under the *See also* section at the end of this chapter. We have condensed the tutorial into one script, and we will go through it line by line and explain all the code that is necessary. We also revert some constants and parameters to the original cited paper values; we'll flag this in the appropriate steps.

How to do it...

Perform the following steps:

1. To start with, we load the necessary libraries and start a graph session:

```
import os
import sys
import tarfile
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from six.moves import urllib
sess = tf.Session()
```

2. Now we'll declare some model parameters. Our batch size will be 128 (for train and test). We will output a status every 50 generations and run for a total of 20,000 generations. Every 500 generations, we'll evaluate on a batch of the test data. We'll then declare some image parameters, height and width, and what size the random cropped images will take. There are three channels (red, green, and blue), and we have ten different targets. Then we'll declare where we will store the data and image batches from the queue:

```
batch_size = 128
output_every = 50
generations = 20000
eval_every = 500
image_height = 32
image_width = 32
crop_height = 24
crop_width = 24
num_channels = 3
num_targets = 10
data_dir = 'temp'
extract_folder = 'cifar-10-batches-bin'
```

3. It is recommended you to lower the learning rate as we progress towards a good model, so we will exponentially decrease the learning rate: The initial learning rate will be set at 0.1, and we will exponentially decrease it by a factor of 10% every 250 generations. The exact formula will be given by

$0.1 \cdot 0.9^{\frac{x}{250}}$, where x is the current generation number. The default is for this to continually decrease, but TensorFlow does accept a staircase argument that only updates the learning rate. Here we set some

parameters for future use:

```
learning_rate = 0.1
lr_decay = 0.9
num_gens_to_wait = 250.
```

4. Now we'll set up parameters so that we can read in binary CIFAR-10 images:

```
image_vec_length = image_height * image_width * num_channels
record_length = 1 + image_vec_length
```

5. Next, we'll set up the data directory and the URL to download the CIFAR-10 images, if we don't have them already:

```
data_dir = 'temp'
if not os.path.exists(data_dir):
    os.makedirs(data_dir)
cifar10_url = 'http://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz'
data_file = os.path.join(data_dir, 'cifar-10-binary.tar.gz')
if not os.path.isfile(data_file):
    # Download file
    filepath, _ = urllib.request.urlretrieve(cifar10_url, data_file)
    # Extract file
    tarfile.open(filepath, 'r:gz').extractall(data_dir)
```

6. We'll set up the record reader and return a randomly distorted image with the following `read_cifar_files()` function. First, we need to declare a record reader object that will read in a fixed length of bytes. After we read the image queue, we'll split apart the image and label. Finally, we will randomly distort the image with TensorFlow's built-in image modification functions:

```
def read_cifar_files(filename_queue, distort_images = True):
    reader = tf.FixedLengthRecordReader(record_bytes=record_length)
    key, record_string = reader.read(filename_queue)
    record_bytes = tf.decode_raw(record_string, tf.uint8)
    # Extract label
    image_label = tf.cast(tf.slice(record_bytes, [0], [1]), tf.int32)
    # Extract image
    image_extracted = tf.reshape(tf.slice(record_bytes, [1], [image_vec_length]))
    # Reshape image
    image_uint8image = tf.transpose(image_extracted, [1, 2, 0])
    reshaped_image = tf.cast(image_uint8image, tf.float32)
    # Randomly Crop image
    final_image = tf.image.resize_image_with_crop_or_pad(reshaped_image, crop_wi
if distort_images:
    # Randomly flip the image horizontally, change the brightness and contra
    final_image = tf.image.random_flip_left_right(final_image)
    final_image = tf.image.random_brightness(final_image,max_delta=63)
    final_image = tf.image.random_contrast(final_image,lower=0.2, upper=1.8)
# Normalize whitening
final_image = tf.image.per_image_standardization(final_image)
return final_image, image_label
```

- Now we'll declare a function that will populate our image pipeline for the batch processor to use. We first need to set up a file list of images we want to read through, and to define how to read them with an input producer object, created through prebuilt TensorFlow functions. The input producer can be passed into the reading function that we created in the preceding step: `read_cifar_files()`. We'll then set a batch reader on the queue: `shuffle_batch()`:

```
def input_pipeline(batch_size, train_logical=True):
    if train_logical:
        files = [os.path.join(data_dir, extract_folder, 'data_batch_{}.bin'.format(i)) for i in range(5)]
    else:
        files = [os.path.join(data_dir, extract_folder, 'test_batch.bin')]
    filename_queue = tf.train.string_input_producer(files)
    image, label = read_cifar_files(filename_queue)

    min_after_dequeue = 1000
    capacity = min_after_dequeue + 3 * batch_size
    example_batch, label_batch = tf.train.shuffle_batch([image, label], batch_size=batch_size,
                                                       capacity=capacity,
                                                       min_after_dequeue=min_after_dequeue)
    return example_batch, label_batch
```

i It is important to set `min_after_dequeue` properly. This parameter is responsible for setting the minimum size of an image buffer for sampling. The official TensorFlow documentation recommends setting it to `(#threads + error margin)*batch_size`. Note that setting it to a larger size results in more uniform shuffling, as it is shuffling from a larger set of data in the queue, but more memory will also be used in the process.

- Next, we can declare our model function. The model we will use has two convolutional layers followed by three fully connected layers. To make variable declaration easier, we'll start by declaring two variable functions. The two convolutional layers will create 64 features each. The first fully connected layer will connect the second convolutional layer with 384 hidden nodes. The second fully connected operation will connect those 384 hidden nodes to 192 hidden nodes. The final hidden layer operation will then connect the 192 nodes to the 10 output classes we are trying to predict. See the following inline comments preceded by #:

```
def cifar_cnn_model(input_images, batch_size, train_logical=True):
    def truncated_normal_var(name, shape, dtype):
        return tf.get_variable(name=name, shape=shape, dtype=dtype, initializer=tf.truncated_normal_initializer())
    def zero_var(name, shape, dtype):
        return tf.get_variable(name=name, shape=shape, dtype=dtype, initializer=tf.zeros_initializer())
    # First Convolutional Layer
    with tf.variable_scope('conv1') as scope:
        # Conv_kernel is 5x5 for all 3 colors and we will create 64 features
        conv1_kernel = truncated_normal_var(name='conv_kernel1', shape=[5, 5, 3, 64])
        # We convolve across the image with a stride size of 1
        conv1 = tf.nn.conv2d(input_images, conv1_kernel, [1, 1, 1, 1], padding='VALID')
        # Initialize and add the bias term
        conv1_bias = zero_var(name='conv_bias1', shape=[64], dtype=tf.float32)
        conv1_add_bias = tf.nn.bias_add(conv1, conv1_bias)
        # ReLU element wise
```

```

        relu_conv1 = tf.nn.relu(conv1_add_bias)
# Max Pooling
pool1 = tf.nn.max_pool(relu_conv1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],

# Local Response Normalization
norm1 = tf.nn.lrn(pool1, depth_radius=5, bias=2.0, alpha=1e-3, beta=0.75, na
# Second Convolutional Layer
with tf.variable_scope('conv2') as scope:
    # Conv kernel is 5x5, across all prior 64 features and we create 64 more
    conv2_kernel = truncated_normal_var(name='conv_kernel2', shape=[5, 5, 64
    # Convolve filter across prior output with stride size of 1
    conv2 = tf.nn.conv2d(norm1, conv2_kernel, [1, 1, 1, 1], padding='SAME')
    # Initialize and add the bias
    conv2_bias = zero_var(name='conv_bias2', shape=[64], dtype=tf.float32)
    conv2_add_bias = tf.nn.bias_add(conv2, conv2_bias)
    # ReLU element wise
    relu_conv2 = tf.nn.relu(conv2_add_bias)
# Max Pooling
pool2 = tf.nn.max_pool(relu_conv2, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
    # Local Response Normalization (parameters from paper)
norm2 = tf.nn.lrn(pool2, depth_radius=5, bias=2.0, alpha=1e-3, beta=0.75, na
    # Reshape output into a single matrix for multiplication for the fully conne
reshaped_output = tf.reshape(norm2, [batch_size, -1])
reshaped_dim = reshaped_output.get_shape()[1].value

# First Fully Connected Layer
with tf.variable_scope('full1') as scope:
    # Fully connected layer will have 384 outputs.
    full_weight1 = truncated_normal_var(name='full_mult1', shape=[reshaped_d
    full_bias1 = zero_var(name='full_bias1', shape=[384], dtype=tf.float32)
    full_layer1 = tf.nn.relu(tf.add(tf.matmul(reshaped_output, full_weight1)
# Second Fully Connected Layer
with tf.variable_scope('full2') as scope:
    # Second fully connected layer has 192 outputs.
    full_weight2 = truncated_normal_var(name='full_mult2', shape=[384, 192]
    full_bias2 = zero_var(name='full_bias2', shape=[192], dtype=tf.float32)
    full_layer2 = tf.nn.relu(tf.add(tf.matmul(full_layer1, full_weight2), fu
# Final Fully Connected Layer -> 10 categories for output (num_targets)
with tf.variable_scope('full3') as scope:
    # Final fully connected layer has 10 (num_targets) outputs.
    full_weight3 = truncated_normal_var(name='full_mult3', shape=[192, num_t
    full_bias3 = zero_var(name='full_bias3', shape=[num_targets], dtype=tf.
    final_output = tf.add(tf.matmul(full_layer2, full_weight3), full_bias3)

return final_output

```



Our local response normalization parameters are taken from the paper and are referenced in the 'See also' section of this recipe.

- Now we'll create the loss function. We will use the softmax function because a picture can only take on exactly one category, so the output should be a probability distribution over the ten targets:

```

def cifar_loss(logits, targets):
    # Get rid of extra dimensions and cast targets into integers
    targets = tf.squeeze(tf.cast(targets, tf.int32))
    # Calculate cross entropy from logits and targets
    cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits
    # Take the average loss across batch size
    cross_entropy_mean = tf.reduce_mean(cross_entropy)
    return cross_entropy_mean

```

10. Next, we declare our training step. The learning rate will decrease in an exponential step function:

```
def train_step(loss_value, generation_num):  
    # Our learning rate is an exponential decay (stepped down)  
    model_learning_rate = tf.train.exponential_decay(learning_rate, generation_n  
    # Create optimizer  
    my_optimizer = tf.train.GradientDescentOptimizer(model_learning_rate)  
    # Initialize train step  
    train_step = my_optimizer.minimize(loss_value)  
    return train_step
```

11. We must also have an accuracy function that calculates the accuracy across a batch of images. We'll input the logits and target vectors, and output an averaged accuracy. We can then use this for both the train and test batches:

```
def accuracy_of_batch(logits, targets):  
    # Make sure targets are integers and drop extra dimensions  
    targets = tf.squeeze(tf.cast(targets, tf.int32))  
    # Get predicted values by finding which logit is the greatest  
    batch_predictions = tf.cast(tf.argmax(logits, 1), tf.int32)  
    # Check if they are equal across the batch  
    predicted_correctly = tf.equal(batch_predictions, targets)  
    # Average the 1's and 0's (True's and False's) across the batch size  
    accuracy = tf.reduce_mean(tf.cast(predicted_correctly, tf.float32))  
    return accuracy
```

12. Now that we have an image pipeline function, we can initialize both the training image pipeline and the test image pipeline:

```
images, targets = input_pipeline(batch_size, train_logical=True)  
test_images, test_targets = input_pipeline(batch_size, train_logical=False)
```

13. Next, we'll initialize the model for the training output and the test output. It is important to note that we must declare `scope.reuse_variables()` after we create the training model, so that, when we declare the model for the test network, it will use the same model parameters:

```
with tf.variable_scope('model_definition') as scope:  
    # Declare the training network model  
    model_output = cifar_cnn_model(images, batch_size)  
    # Use same variables within scope  
    scope.reuse_variables()  
    # Declare test model output  
    test_output = cifar_cnn_model(test_images, batch_size)
```

14. We can now initialize our loss and test accuracy functions. Then we'll declare the `generation` variable. This variable needs to be declared as non-trainable, and passed to our training function, which uses it in the learning rate exponential decay calculation:

```
loss = cifar_loss(model_output, targets)
accuracy = accuracy_of_batch(test_output, test_targets)
generation_num = tf.Variable(0, trainable=False)
train_op = train_step(loss, generation_num)
```

15. We'll now initialize all of the model's variables and then start the image pipeline by running the TensorFlow function, `start_queue_runners()`. When we start the train or test model output, the pipeline will feed in a batch of images in place of a feed dictionary:

```
init = tf.global_variables_initializer()
sess.run(init)
tf.train.start_queue_runners(sess=sess)
```

16. We now loop through our training generations and save the training loss and the test accuracy:

```
train_loss = []
test_accuracy = []
for i in range(generations):
    _, loss_value = sess.run([train_op, loss])
    if (i+1) % output_every == 0:
        train_loss.append(loss_value)
        output = 'Generation {}: Loss = {:.5f}'.format((i+1), loss_value)
        print(output)
    if (i+1) % eval_every == 0:
        [temp_accuracy] = sess.run([accuracy])
        test_accuracy.append(temp_accuracy)
        acc_output = ' --- Test Accuracy= {:.2f}%.format(100. * temp_accuracy)
        print(acc_output)
```

17. This results in the following output:

```
...
Generation 19500: Loss = 0.04461
--- Test Accuracy = 80.47%.
Generation 19550: Loss = 0.01171
Generation 19600: Loss = 0.06911
Generation 19650: Loss = 0.08629
Generation 19700: Loss = 0.05296
Generation 19750: Loss = 0.03462
Generation 19800: Loss = 0.03182
Generation 19850: Loss = 0.07092
Generation 19900: Loss = 0.11342
Generation 19950: Loss = 0.08751
Generation 20000: Loss = 0.02228
--- Test Accuracy = 83.59%.
```

18. Finally, here is some `matplotlib` code that will plot the loss and test accuracy over the course of the training:

```
eval_indices = range(0, generations, eval_every)
output_indices = range(0, generations, output_every)
# Plot loss over time
plt.plot(output_indices, train_loss, 'k-')
```

```

plt.title('Softmax Loss per Generation')
plt.xlabel('Generation')
plt.ylabel('Softmax Loss')
plt.show()

# Plot accuracy over time
plt.plot(eval_indices, test_accuracy, 'k-')
plt.title('Test Accuracy')
plt.xlabel('Generation')
plt.ylabel('Accuracy')
plt.show()

```

We get the following plots for the preceding recipe:

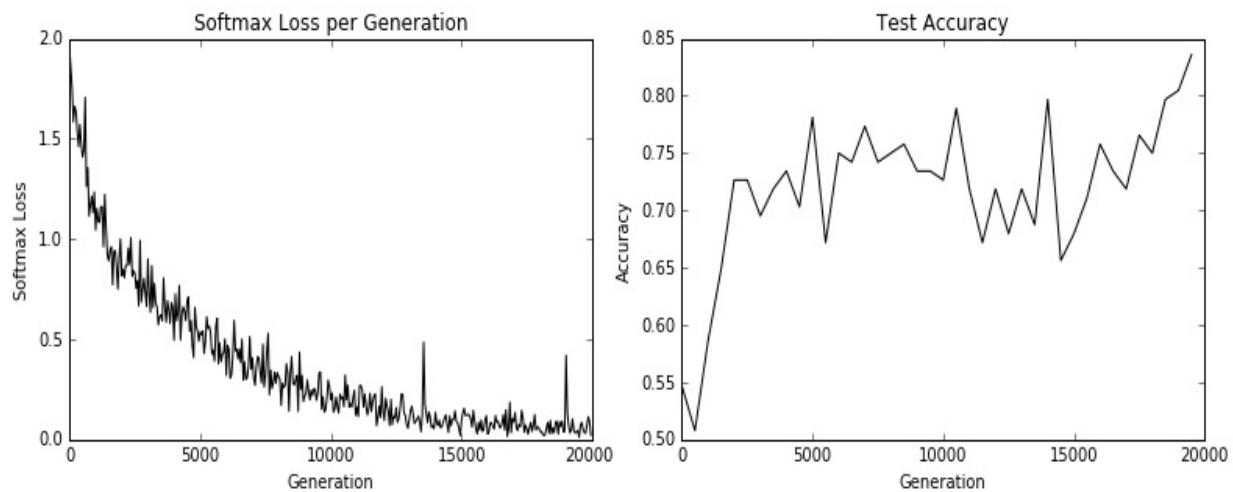


Figure 5: The training loss is on the left and the test accuracy is on the right. For the CIFAR-10 image recognition CNN, we were able to achieve a model that reaches around 75% accuracy on the test set

How it works...

After we downloaded the CIFAR-10 data, we established an image pipeline instead of using a feed dictionary. For more information on the image pipeline, please see the official TensorFlow CIFAR-10 tutorials. We used this train and test pipeline to try to predict the correct category of the images. By the end, the model had achieved around 75% accuracy on the test set.

See also

- For more information about the CIFAR-10 dataset, please see *Learning Multiple Layers of Features from Tiny Images*, Alex Krizhevsky, 2009: <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>
- To view the original TensorFlow code, please see: <https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10>
- For more on local response normalization, please see *ImageNet Classification with Deep Convolutional Neural Networks*, Krizhevsky, A., et. al. 2012: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>

Retraining existing CNN models

Training a new image recognition from scratch requires a lot of time and computational power. If we can take a prior-trained network and retrain it with our images, it may save us computational time. For this recipe, we will show how to use a pre-trained TensorFlow image recognition model and fine-tune it to work on a different set of images.

Getting ready

The idea is to reuse the weights and structure of a prior model from the convolutional layers and retrain the fully connected layers at the top of the network.

TensorFlow has created a tutorial about training on top of existing CNN models (refer to the first bullet point in the next *See also* section). In this recipe, we will illustrate how to use the same methodology for CIFAR-10. The CNN network we are going to employ uses a very popular architecture called **Inception**. The Inception CNN model was created by Google and has performed very well on many image recognition benchmarks. For details, see the paper reference in the second bullet point of the *See also* section.

The main Python script we will cover shows how to download CIFAR-10 image data and automatically separate, label, and save the images into the ten classes in each of the train and test folders. After that, we will reiterate how to train the network on our images.

How to do it...

Perform the following steps:

1. We'll start by loading the necessary libraries to download, unzip, and save CIFAR-10 images:

```
import os
import tarfile
import _pickle as cPickle
import numpy as np
import urllib.request
import scipy.misc
from imageio import imwrite
```

2. We now declare the CIFAR-10 data link and create the temporary directory we will store the data in. We'll also declare the ten categories to reference when saving the images later on:

```
cifar_link = 'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz'
data_dir = 'temp'
if not os.path.isdir(data_dir):
    os.makedirs(data_dir)
objects = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse'
```

3. Now we'll download the CIFAR-10 .tar data file, and un-tar the file:

```
target_file = os.path.join(data_dir, 'cifar-10-python.tar.gz')
if not os.path.isfile(target_file):
    print('CIFAR-10 file not found. Downloading CIFAR data (Size = 163MB)')
    print('This may take a few minutes, please wait.')
    filename, headers = urllib.request.urlretrieve(cifar_link, target_file)
# Extract into memory
tar = tarfile.open(target_file)
tar.extractall(path=data_dir)
tar.close()
```

4. We now create the necessary folder structure for training. The temporary directory will have two folders, `train_dir` and `validation_dir`. In each of these folders, we will create ten sub-folders for each category:

```
# Create train image folders
train_folder = 'train_dir'
if not os.path.isdir(os.path.join(data_dir, train_folder)):
    for i in range(10):
        folder = os.path.join(data_dir, train_folder, objects[i])
        os.makedirs(folder)
# Create test image folders
test_folder = 'validation_dir'
```

```

    if not os.path.isdir(os.path.join(data_dir, test_folder)):
        for i in range(10):
            folder = os.path.join(data_dir, test_folder, objects[i])
            os.makedirs(folder)

```

5. In order to save the images, we will create a function that will load them from memory and store them in an image dictionary:

```

def load_batch_from_file(file):
    file_conn = open(file, 'rb')
    image_dictionary = cPickle.load(file_conn, encoding='latin1')
    file_conn.close()
    return(image_dictionary)

```

6. With the preceding dictionary, we will save each of the files in the correct location, with the following function:

```

def save_images_from_dict(image_dict, folder='data_dir'):

    for ix, label in enumerate(image_dict['labels']):
        folder_path = os.path.join(data_dir, folder, objects[label])
        filename = image_dict['filenames'][ix]
        # Transform image data
        image_array = image_dict['data'][ix]
        image_array.resize([3, 32, 32])
        # Save image
        output_location = os.path.join(folder_path, filename)
        imwrite(output_location, image_array.transpose())

```

7. With the preceding functions, we can loop through the downloaded data files and save each image to the correct location:

```

data_location = os.path.join(data_dir, 'cifar-10-batches-py')
train_names = ['data_batch_' + str(x) for x in range(1,6)]
test_names = ['test_batch']
# Sort train images
for file in train_names:
    print('Saving images from file: {}'.format(file))
    file_location = os.path.join(data_dir, 'cifar-10-batches-py', file)
    image_dict = load_batch_from_file(file_location)
    save_images_from_dict(image_dict, folder=train_folder)
# Sort test images
for file in test_names:
    print('Saving images from file: {}'.format(file))
    file_location = os.path.join(data_dir, 'cifar-10-batches-py', file)
    image_dict = load_batch_from_file(file_location)
    save_images_from_dict(image_dict, folder=test_folder)

```

8. The last part of our script creates the image label file, and this is the last piece of information that we will need. This file will let us interpret the outputs as labels instead of as numerical indices:

```

cifar_labels_file = os.path.join(data_dir, 'cifar10_labels.txt')
print('Writing labels file, {}'.format(cifar_labels_file))
with open(cifar_labels_file, 'w') as labels_file:

```

```
|     for item in objects:  
|         labels_file.write("{}\n".format(item))
```

9. When the preceding script is run, it will download the images and sort them into the correct folder structure that the TensorFlow retraining tutorial expects. Once we have done this, we just follow the tutorial accordingly. First, we should clone the tutorial repository:

```
|     git clone https://github.com/tensorflow/models/tree/master/research/inception
```

10. In order to use a prior-trained model, we must download the network weights and apply them to our model. In order to do this, you must visit the site: <https://github.com/tensorflow/models/tree/master/research/slim> , and follow the directions to download and install the cifar10 model architecture and weights. You will also end up downloading a data directory that contains the build, train, and test scripts described below.



For this step, we navigate to the research/inception/inception directory and then execute the below command, with the paths for --train_directory, --validation_directory, --output_directory and --labels_file to point to to the relative or full paths to the directory structure created.

11. Now that we have the images in the correct folder structure, we have to turn them into a `TFRecords` object. We do this by running the following commands:

```
me@computer:~$ python3 data/build_image_data.py  
--train_directory="temp/train_dir/"  
--validation_directory="temp/validation_dir"  
--output_directory="temp/" --labels_file="temp/cifar10_labels.txt"
```

12. Now we'll train the model using `bazel`, setting the parameter to `true`. This script outputs the loss every 10 generations. We can kill this process at any time and the model output will be in the `temp/training_results` folder. We can load the model from this folder for evaluation:

```
me@computer:~$ bazel-bin/inception/flowers_train  
--train_dir="temp/training_results" --data_dir="temp/data_dir"  
--pretrained_model_checkpoint_path="model.ckpt-157585"  
--fine_tune=True --initial_learning_rate=0.001  
--input_queue_memory_factor=1
```

13. This should result in output similar to the following:

```
2018-06-02 11:10:10.557012: step 1290, loss = 2.02 (1.2 examples/sec; 23.771 se  
...  
|
```

How it works...

The official TensorFlow tutorial on training on top of a pre-trained CNN requires setting up a folder; setup that we created this from the CIFAR-10 data. We then converted the data into the required `TFRecords` format and began to train the model. Remember that we are fine-tuning the model and retraining the fully connected layers at the top to fit our 10-category data.

See also

- Official Tensorflow Inception-v3 tutorial: https://www.tensorflow.org/tutorials/images/image_recognition
- Googlenet Inception-v3 paper: <https://arxiv.org/abs/1512.00567>

Applying stylenet and the neural-style project

Once we have an image recognition CNN trained, we can use the network itself for some interesting data and image processing. Stylenet is a procedure that attempts to learn an image style from one picture and apply it to a second picture while keeping the second image structure (or content) intact. This may be possible if we can find intermediate CNN nodes that correlate strongly with a style, separately from the content of the image.

Getting ready

StyleNet is a procedure that takes two images and applies the style of one image to the content of the second image. It is based on a famous paper in 2015, *A Neural Algorithm of Artistic Style* (refer to the first bullet point under the next *See also* section). The authors found a property in some CNNs where intermediate layers exist that seem to encode the style of a picture, and some encode the content of the picture. To this end, if we train the style layers on the style picture and the content layers on the original image, and back-propagate those calculated losses, we can change the original image to be more like the style image.

In order to accomplish this, we will download the network recommended by the paper; called the **imagenet-vgg-19**. There is also an imangenet-vgg-16 network that works as well, but the paper recommends imangenet-vgg-19.

How to do it...

Perform the following steps:

1. First, we'll download the pre-trained network in `mat` format. The `mat` format is a `matlab` object, and the `scipy` package in Python has a method that can read it. The link to download the `mat` object is here. We save this model in the same folder our Python script is in, for reference:

```
| http://www.vlfeat.org/matconvnet/models/beta16/imagenet-vgg-verydeep-19.mat
```

2. We'll start our Python script by loading the necessary libraries:

```
import os
import scipy.io
import scipy.misc
import imageio
from skimage.transform import resize
from operator import mul
from functools import reduce
import numpy as np
import tensorflow as tf
from tensorflow.python.framework import ops
ops.reset_default_graph()
```

3. Then we can declare the locations of our two images: the original image and the style image. For our purposes, we will use the cover image of this book for the original image; for the style image, we will use *Starry Night* by Vincent van Gough. Feel free to use any two pictures you want here. If you choose to use these pictures, they are available on the book's GitHub site, https://github.com/nfmccleure/tensorflow_cookbook (navigate to the Styelnet section):

```
| original_image_file = 'temp/book_cover.jpg'
| style_image_file = 'temp/starry_night.jpg'
```

4. We'll set some parameters for our model: the location of the `mat` file, `weights`, the learning rate, the number of `generations`, and how frequently we should output the intermediate image. For the weights, it helps to highly weight the style image over the original image. These hyperparameters should be tuned for changes in the desired result:

```
vgg_path = 'imagenet-vgg-verydeep-19.mat'
original_image_weight = 5.0
style_image_weight = 500.0
```

```

regularization_weight = 100
learning_rate = 10
generations = 100
output_generations = 25
beta1 = 0.9
beta2 = 0.999

```

- Now we'll load the two images with `scipy` and change the style image to fit the original image dimensions:

```

original_image = imageio.imread(original_image_file)
style_image = imageio.imread(style_image_file)

# Get shape of target and make the style image the same
target_shape = original_image.shape
style_image = resize(style_image, target_shape)

```

- From the paper, we can define the layers in the order in which they appeared. We'll use the author's naming convention:

```

vgg_layers = ['conv1_1', 'relu1_1',
              'conv1_2', 'relu1_2', 'pool1',
              'conv2_1', 'relu2_1',
              'conv2_2', 'relu2_2', 'pool2',
              'conv3_1', 'relu3_1',
              'conv3_2', 'relu3_2',
              'conv3_3', 'relu3_3',
              'conv3_4', 'relu3_4', 'pool3',
              'conv4_1', 'relu4_1',
              'conv4_2', 'relu4_2',
              'conv4_3', 'relu4_3',
              'conv4_4', 'relu4_4', 'pool4',
              'conv5_1', 'relu5_1',
              'conv5_2', 'relu5_2',
              'conv5_3', 'relu5_3',
              'conv5_4', 'relu5_4']

```

- Now we'll define a function that will extract the parameters from the `mat` file:

```

def extract_net_info(path_to_params):
    vgg_data = scipy.io.loadmat(path_to_params)
    normalization_matrix = vgg_data['normalization'][0][0][0]
    mat_mean = np.mean(normalization_matrix, axis=(0,1))
    network_weights = vgg_data['layers'][0]
    return mat_mean, network_weights

```

- From the loaded weights and the `layer` definitions, we can recreate the network in TensorFlow with the following function. We'll loop through each layer and assign the corresponding function with appropriate `weights` and `biases`, where applicable:

```

def vgg_network(network_weights, init_image):
    network = {}

```

```

image = init_image
for i, layer in enumerate(vgg_layers):
    if layer[1] == 'c':
        weights, bias = network_weights[i][0][0][0][0]
        weights = np.transpose(weights, (1, 0, 2, 3))
        bias = bias.reshape(-1)
        conv_layer = tf.nn.conv2d(image, tf.constant(weights), (1, 1, 1, 1),
                                image = tf.nn.bias_add(conv_layer, bias)
    elif layer[1] == 'r':
        image = tf.nn.relu(image)
    else:
        image = tf.nn.max_pool(image, (1, 2, 2, 1), (1, 2, 2, 1), 'SAME')
    network[layer] = image
return(network)

```

- The paper recommends a few strategies for assigning intermediate layers to the original and style images. While we should keep `relu4_2` for the original image, we can try different combinations of the other `reluX_1` layer outputs for the style image:

```

original_layer = ['relu4_2']
style_layers = ['relu1_1', 'relu2_1', 'relu3_1', 'relu4_1', 'relu5_1']

```

- Next, we'll run the preceding function to get the weights and mean. We also need to evenly set the VGG19 style layer weights. You may experiment by changing the weights if you wish. For now, we assume that they are 0.5 for both layers:

```

# Get network parameters
normalization_mean, network_weights = extract_net_info(vgg_path)
shape = (1,) + original_image.shape
style_shape = (1,) + style_image.shape
original_features = {}
style_features = {}

# Set style weights
style_weights = {l: 1./len(style_layers)} for l in style_layers}

```

- In order to stay faithful to the original image look, we want to add a loss value that compares the content/original features to the original content features. To do so, we load up the VGG19 model and compute the content/original features for the original content features:

```

g_original = tf.Graph()
with g_original.as_default(), tf.Session() as sess1:
    image = tf.placeholder('float', shape=shape)
    vgg_net = vgg_network(network_weights, image)
    original_minus_mean = original_image - normalization_mean
    original_norm = np.array([original_minus_mean])
    for layer in original_layers:
        original_features[layer] = vgg_net[layer].eval(feed_dict={image: origina

```

12. Similar to Step 11, we want to change the style features of the original image toward the style features of the style-picture. To do this, we will add a style loss value to our loss function. This loss value will need to look at the values of the style image at the style layers we have predetermined. We will also run this through a separate graph. We compute these style features as follows:

```
# Get style image network
g_style = tf.Graph()
with g_style.as_default(), tf.Session() as sess2:
    image = tf.placeholder('float', shape=style_shape)
    vgg_net = vgg_network(network_weights, image)
    style_minus_mean = style_image - normalization_mean
    style_norm = np.array([style_minus_mean])
    for layer in style_layers:
        features = vgg_net[layer].eval(feed_dict={image: style_norm})
        features = np.reshape(features, (-1, features.shape[3]))
        gram = np.matmul(features.T, features) / features.size
        style_features[layer] = gram
```

13. We start a default graph to compute the loss and the training step. To start this, we first initialize a random image as a TensorFlow variable:

```
# Make Combined Image via loss function
with tf.Graph().as_default():
    # Get network parameters
    initial = tf.random_normal(shape) * 0.256
    init_image = tf.Variable(initial)
    vgg_net = vgg_network(network_weights, init_image)
```

14. Next, we compute the original content loss(keeping it indented under the default graph). This loss part will keep the structure of the original image intact as much as possible:

```
# Loss from Original Image
original_layers_w = {'relu4_2': 0.5, 'relu5_2': 0.5}
original_loss = 0
for o_layer in original_layers:
    temp_original_loss = original_layers_w[o_layer] * original_image_weight * \
        (2 * tf.nn.l2_loss(vgg_net[o_layer] - original_features[o_layer])
    original_loss += (temp_original_loss / original_features[o_layer].size)
```

15. Still under the default graph indent, we create the second loss term, the style loss. This loss will compare the style features we have pre-computed with the style features from the input image (initialized randomly):

```
# Loss from Style Image
style_loss = 0
style_losses = []
for style_layer in style_layers:
    layer = vgg_net[style_layer]
    feats, height, width, channels = [x.value for x in layer.get_shape()]
```

```

size = height * width * channels
features = tf.reshape(layer, (-1, channels))
style_gram_matrix = tf.matmul(tf.transpose(features), features) / size
style_expected = style_features[style_layer]
style_losses.append(style_weights[style_layer] * 2 *
                     tf.nn.l2_loss(style_gram_matrix - style_expected) /
                     style_expected.size)
style_loss += style_image_weight * tf.reduce_sum(style_losses)

```

- The third and final loss term will help smooth out the image. We use total variation loss here to penalize dramatic changes in neighboring pixels, as follows:

```

total_var_x = reduce(mul, init_image[:, 1:, :, :].get_shape().as_list(), 1)
total_var_y = reduce(mul, init_image[:, :, 1:, :].get_shape().as_list(), 1)
first_term = regularization_weight * 2
second_term_numerator = tf.nn.l2_loss(init_image[:, 1:, :, :] - init_image[:, :, 1:, :])
second_term = second_term_numerator / total_var_y
third_term = (tf.nn.l2_loss(init_image[:, :, 1:, :] - init_image[:, :, :, 1:]) * shape[2])
total_variation_loss = first_term * (second_term + third_term)

```

- Next, we combine the loss terms and create an optimizing function and training step, as follows:

```

# Combined Loss
loss = original_loss + style_loss + total_variation_loss

# Declare Optimization Algorithm
optimizer = tf.train.AdamOptimizer(learning_rate, beta1, beta2)
train_step = optimizer.minimize(loss)

```

- Now we run the training step, save intermediate images, and save the final output image, as follows:

```

# Initialize variables and start training
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for i in range(generations):
        train_step.run()

        # Print update and save temporary output
        if (i+1) % output_generations == 0:
            print('Generation {} out of {}, loss: {}'.format(i + 1, generations,
                image_eval = init_image.eval()
                best_image_add_mean = image_eval.reshape(shape[1:]) + normalization_
                output_file = 'temp_output_{}.jpg'.format(i)
                imageio.imwrite(output_file, best_image_add_mean.astype(np.uint8))

# Save final image
image_eval = init_image.eval()
best_image_add_mean = image_eval.reshape(shape[1:]) + normalization_mean
output_file = 'final_output.jpg'
scipy.misc.imsave(output_file, best_image_add_mean)

```

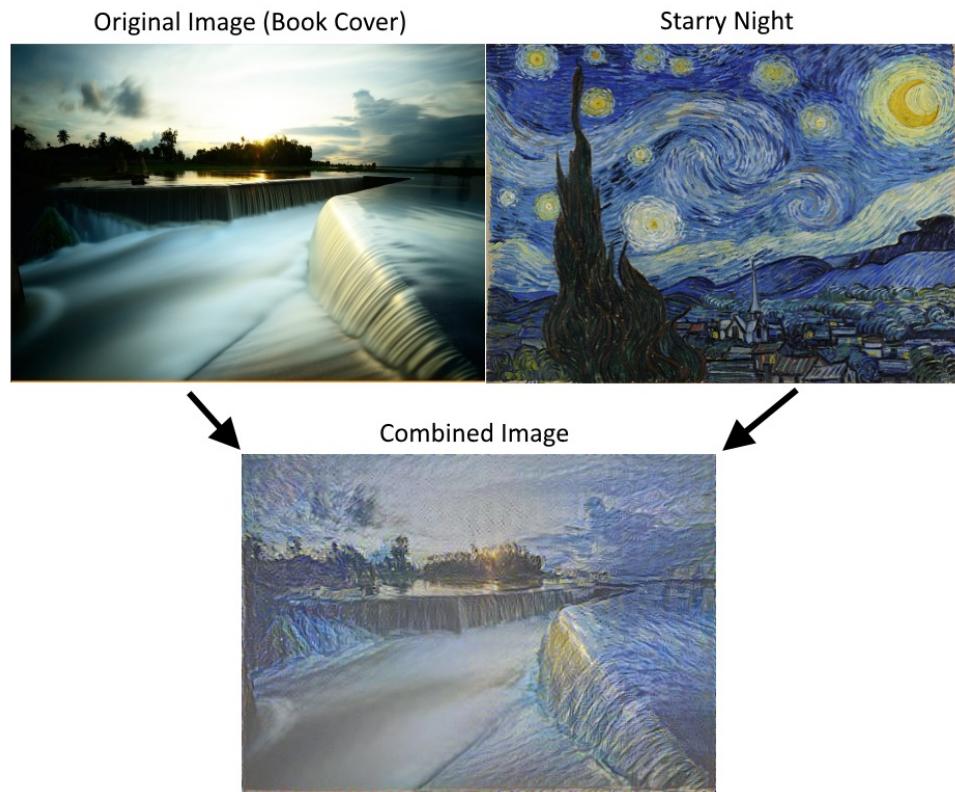


Figure 6: Using the StyleNet algorithm to combine the book cover image with Starry Night. Note that different style emphases can be used by changing the weighting at the beginning of the script

How it works...

We first loaded the two images, then loaded the pre-trained network weights and assigned layers to the original and style images. We calculated three loss functions: an original image loss, a style loss, and a total variation loss. Then we trained random noise pictures to use the style of the style image and the content of the original image.

The loss functions were heavily influenced by the GitHub neural style project: <https://github.com/anishathalye/neural-style>. We also strongly recommend that readers look at the code in those projects for improvements, more details, and a generally more robust algorithm that can give better results.

See also

- *A Neural Algorithm of Artistic Style* by Gatys, Ecker, Bethge. 2015: <https://arxiv.org/abs/1508.06576>
- A good recommended video of a presentation by Leon Gatys at CVPR 2016 (Computer Vision and Pattern Recognition) is here: <https://www.youtube.com/watch?v=UFffxcCQMPQ> .

Implementing DeepDream

Another use for trained CNNs is exploiting the fact that some intermediate nodes detect features of labels (for instance, a cat's ear, or a bird's feather). Using this fact, we can find ways to transform any image to reflect those node features for any node we choose. For this recipe, we will go through the DeepDream tutorial on TensorFlow's website, but we will cover the essential parts in much more detail. The hope is that we can prepare the reader to use the DeepDream algorithm to explore CNNs, and features created in them.

Getting ready

TensorFlow's official tutorials show how to implement DeepDream through a script (refer to the first bullet point in the next *See also* section). The purpose of this recipe is to go through the script they provide and explain each line. While the tutorial is great, there are some parts that are skippable and some parts that could use more explanation. We hope to provide a more detailed line-by-line explanation. We will also make the code to be Python 3-compliant, where necessary.

How to do it...

Perform the following steps:

1. In order to get started with DeepDream, we need to download GoogleNet, which is CNN-trained on CIFAR-1000:

```
| me@computer:~$ wget https://storage.googleapis.com/download.tensorflow.org/model  
| me@computer:~$ unzip inception5h.zip
```

2. We'll start with by loading the necessary libraries and starting a graph session:

```
| import os  
| import matplotlib.pyplot as plt  
| import numpy as np  
| import PIL.Image  
| import tensorflow as tf  
| from io import BytesIO  
| graph = tf.Graph()  
| sess = tf.InteractiveSession(graph=graph)
```

3. We now declare the location of the unzipped model parameters (from *Step 1*) and load the parameters into a TensorFlow graph:

```
| # Model location  
| model_fn = 'tensorflow_inception_graph.pb'  
| # Load graph parameters  
| with tf.gfile.FastGFile(model_fn, 'rb') as f:  
|     graph_def = tf.GraphDef()  
|     graph_def.ParseFromString(f.read())
```

4. We create a placeholder for the input, save the imagenet mean value of 117.0, and then import the graph definition with the normalized placeholder:

```
| # Create placeholder for input  
| t_input = tf.placeholder(np.float32, name='input')  
| # Imagenet average bias to subtract off images  
| imagenet_mean = 117.0  
| t_preprocessed = tf.expand_dims(t_input-imagenet_mean, 0)  
| tf.import_graph_def(graph_def, {'input':t_preprocessed})
```

5. Next, we will import the convolutional layers, to visualize and use them for DeepDream processing later:

```

# Create a list of layers that we can refer to later
layers = [op.name for op in graph.get_operations() if op.type=='Conv2D' and 'imp'
# Count how many outputs for each layer
feature_nums = [int(graph.get_tensor_by_name(name+':0').get_shape()[-1]) for nam

```

- Now we will pick a layer to visualize. We can pick others by name as well. We choose to look at feature number 139. The image starts with random noise:

```

layer = 'mixed4d_3x3_bottleneck_pre_relu'
channel = 139
img_noise = np.random.uniform(size=(224, 224, 3)) + 100.0

```

- We declare a function that will plot an image array:

```

def showarray(a, fmt='jpeg'):
    # First make sure everything is between 0 and 255
    a = np.uint8(np.clip(a, 0, 1)*255)
    # Pick an in-memory format for image display
    f = BytesIO()
    # Create the in memory image
    PIL.Image.fromarray(a).save(f, fmt)
    # Show image
    plt.imshow(a)

```

- We'll shorten some repetitive code by creating a function that retrieves a layer by name from the graph:

```
| def T(layer): #Helper for getting layer output tensor return graph.get_tensor_by
```

- The next function we will create is a wrapper function to create placeholders according to the arguments we specify:

```

# The following function returns a function wrapper that will create the placeho
# inputs of a specified dtype
def tffunc(*argtypes):
    '''Helper that transforms TF-graph generating function into a regular one.
    See "resize" function below.
    '''
    placeholders = list(map(tf.placeholder, argtypes))
    def wrap(f):
        out = f(*placeholders)
        def wrapper(*args, **kw):
            return out.eval(dict(zip(placeholders, args)), session=kw.get('sessi
            return wrapper
    return wrap

```

- We also need a function that resizes an image to a size specification. We do this with TensorFlow's built-in image linear interpolation function: `tf.image.resize.bilinear()`

```
| # Helper function that uses TF to resize an image
```

```

def resize(img, size):
    img = tf.expand_dims(img, 0)
    # Change 'img' size by linear interpolation
    return tf.image.resize_bilinear(img, size)[0,:,:,:]

```

- Now we need a way to update the source image to be more of a feature we use. We do this by specifying how the gradient on the image is calculated. We define a function that will calculate gradients on sub-regions (tiles) over the image to speed up calculations. In order to prevent tiled output, we will randomly shift, or roll, the image in the x and y directions, which will smooth out the tiling effect:

```

def calc_grad_tiled(img, t_grad, tile_size=512):
    '''Compute the value of tensor t_grad over the image in a tiled way.
    Random shifts are applied to the image to blur tile boundaries over
    multiple iterations.'''
    # Pick a subregion square size
    sz = tile_size
    # Get the image height and width
    h, w = img.shape[:2]
    # Get a random shift amount in the x and y direction
    sx, sy = np.random.randint(sz, size=2)
    # Randomly shift the image (roll image) in the x and y directions
    img_shift = np.roll(np.roll(img, sx, 1), sy, 0)
    # Initialize the while image gradient as zeros
    grad = np.zeros_like(img)
    # Now we loop through all the sub-tiles in the image
    for y in range(0, max(h-sz//2, sz),sz):
        for x in range(0, max(w-sz//2, sz),sz):
            # Select the sub image tile
            sub = img_shift[y:y+sz,x:x+sz]
            # Calculate the gradient for the tile
            g = sess.run(t_grad, {t_input:sub})
            # Apply the gradient of the tile to the whole image gradient
            grad[y:y+sz,x:x+sz] = g
    # Return the gradient, undoing the roll operation
    return np.roll(np.roll(grad, -sx, 1), -sy, 0)

```

- Now we can declare our DeepDream function. The objective of our algorithm is the mean of the feature we select. The loss operates on gradients, which will depend on the distance between the input image and the selected feature. The strategy is to separate the image into high and low frequencies, and calculate gradients on the low part. The resulting high-frequency image is split up again and the process is repeated. The set of the original image and the low-frequency images are called `octaves`. For each pass, we calculate gradients and apply them to the images:

```

def render_deepdream(t_obj, img0=img_noise,
                     iter_n=10, step=1.5, octave_n=4, octave_scale=1.4):
    # defining the optimization objective, the objective is the mean of the feature
    t_score = tf.reduce_mean(t_obj)
    # Our gradients will be defined as changing the t_input to get closer to the
    # t_input will be the image octave (starting with the last)

```

```

t_grad = tf.gradients(t_score, t_input)[0] # behold the power of automatic differentiation
# Store the image
img = img0
# Initialize the image octave list
octaves = []
# Since we stored the image, we need to only calculate n-1 octaves
for i in range(octave_n-1):
    # Extract the image shape
    hw = img.shape[:2]
    # Resize the image, scale by the octave_scale (resize by linear interpolation)
    lo = resize(img, np.int32(np.float32(hw)/octave_scale))
    # Residual is hi. Where residual = image - (Resize lo to be hw-shape)
    hi = img-resize(lo, hw)
    # Save the lo image for re-iterating
    img = lo
    # Save the extracted hi-image
    octaves.append(hi)

# generate details octave by octave
for octave in range(octave_n):
    if octave>0:
        # Start with the last octave
        hi = octaves[-octave]
        #
        img = resize(img, hi.shape[:2])+hi
    for i in range(iter_n):
        # Calculate gradient of the image.
        g = calc_grad_tiled(img, t_grad)
        # Ideally, we would just add the gradient, g, but
        # we want do a forward step size of it ('step'),
        # and divide it by the avg. norm of the gradient, so
        # we are adding a gradient of a certain size each step.
        # Also, to make sure we aren't dividing by zero, we add 1e-7.
        img += g*(step / (np.abs(g).mean()+1e-7))
        print('.',end = ' ')
    showarray(img/255.0)

```

13. With all the function setup we have done, we now can run the DeepDream algorithm:

```

# Run Deep Dream
if __name__=="__main__":
    # Create resize function that has a wrapper that creates specified placeholder
    resize = tffunc(np.float32, np.int32)(resize)

    # Open image
    img0 = PIL.Image.open('book_cover.jpg')
    img0 = np.float32(img0)
    # Show Original Image
    showarray(img0/255.0)
    # Create deep dream
    render_deeppdream(T(layer)[:,:,:139], img0, iter_n=15)
    sess.close()

```

The output is as follows:

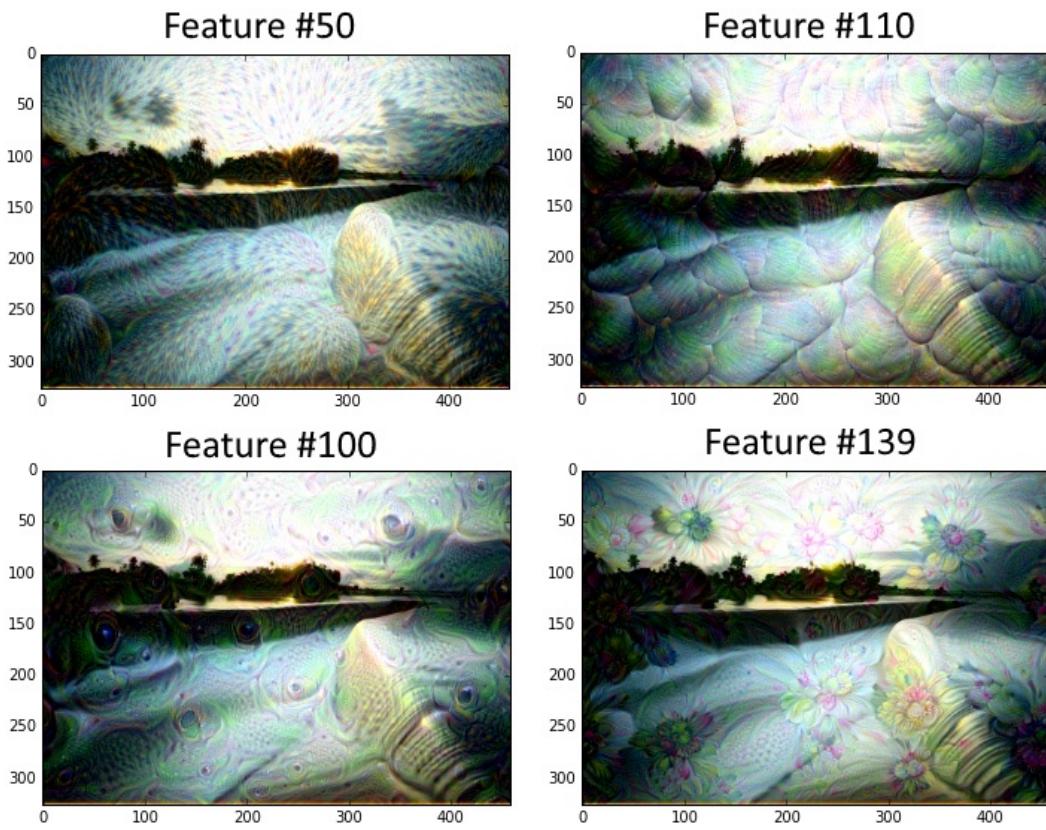


Figure 7: The cover of the book, run through the DeepDream algorithm with feature layer numbers 50, 110, 100, and 139

There's more...

We urge the reader to use the official DeepDream tutorials as a source of further information, and also to visit the original Google research blog post on DeepDream (refer to the second bullet point in the following *See also* section).

See also

- The TensorFlow tutorial on DeepDream: <https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/tutorials/deepdream>
- The original Google research blog post on DeepDream: <https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html>

Recurrent Neural Networks

In this chapter, we will cover recurrent neural networks (**RNNs**) and how to implement them in TensorFlow. We will start by demonstrating how to use an RNN to predict spam. We will then introduce a variant on RNNs for creating Shakespearean text. We will finish by creating an RNN sequence-to-sequence model to translate from English to German:

- Implementing RNNs for spam prediction
- Implementing an LSTM model
- Stacking multiple LSTM layers
- Creating sequence-to-sequence models
- Training a Siamese similarity measure

All the code to this chapter can be found online at https://github.com/nfmccclure/tensorflow_cookbook and at the Packt online repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

Introduction

Of all the machine learning algorithms we have considered thus far, none have considered data as a sequence. To take sequence data into account, we extend neural networks that store outputs from prior iterations. This type of neural network is called an **RNN**. Consider the fully connected network formulation:

$$y = \sigma(Ax)$$

Here, the weights are given by A multiplied by the input layer, x , and then run through an activation function, σ , which gives the output layer, y .

If we have a sequence of input data, x_1, x_2, x_3, \dots , we can adapt the fully connected layer to take prior inputs into account, as follows:

$$y_t = \sigma(By_{t-1} + Ax_t)$$

On top of this recurrent iteration to get the next input, we want to get the probability distribution output, as follows:

$$S_t = \text{softmax}(Cy_t)$$

Once we have a full sequence output, $\{s_1, s_2, s_3, \dots\}$, we can consider the target as a number or category by just considering the last output. See the following diagram for how a general architecture might work:

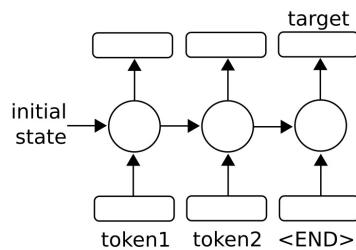


Figure 1: To predict a single number, or a category, we take a sequence of inputs (tokens) and consider the final output as the predicted output

We can also consider the sequence output as an input in a sequence-to-sequence

model:

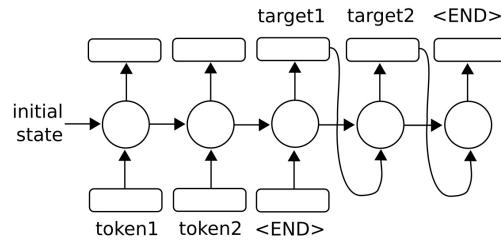


Figure 2: To predict a sequence, we may also feed the outputs back into the model to generate multiple outputs

For arbitrarily long sequences, training with the back-propagation algorithm creates long time-dependent gradients. Because of this, there exists a *vanishing or exploding gradient problem*. Later in this chapter, we will explore a solution to this problem by expanding the RNN cell into what is called a **Long Short Term Memory (LSTM)** cell. The main idea is that the LSTM cell introduces another operation, called a **gate**, which controls the flow of information through the sequence. We will go over the details in later chapters.



When dealing with RNN models for NLP, **encoding** is a term used to describe the process of converting data (words or characters in NLP) into numerical RNN features. The term **decoding** is the process of converting RNN numerical features into output words or characters.

Implementing RNN for spam prediction

To start, we will apply the standard RNN unit to predict a singular numerical output, a probability of being spam.

Getting ready

In this recipe, we will implement a standard RNN in TensorFlow to predict whether or not a text message is spam or ham. We will use the SMS spam-collection dataset from the ML repository at UCI. The architecture we will use for prediction will be an input RNN sequence from embedded text, and we will take the last RNN output as a prediction of spam or ham (1 or 0).

How to do it...

1. We start by loading the libraries required for this script:

```
import os
import re
import io
import requests
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from zipfile import ZipFile
```

2. Next, we start a graph session and set the RNN model parameters. We will run the data through 20 epochs, in batch sizes of 250. The maximum length of each text we will consider is 25 words; we will cut longer texts to 25 or zero-pad shorter texts. The RNN will be size 10 units. We will only consider words that appear at least 10 times in our vocabulary, and every word will be embedded in a trainable vector of size 50. The dropout rate will be a placeholder that we can set at 0.5 during training time or 1.0 during evaluation:

```
sess = tf.Session()
epochs = 20
batch_size = 250
max_sequence_length = 25
rnn_size = 10
embedding_size = 50
min_word_frequency = 10
learning_rate = 0.0005
dropout_keep_prob = tf.placeholder(tf.float32)
```

3. Now we get the SMS text data. First, we check whether it has already been downloaded and, if so, read in the file. Otherwise, we download the data and save it:

```
data_dir = 'temp'
data_file = 'text_data.txt'
if not os.path.exists(data_dir):
    os.makedirs(data_dir)
if not os.path.isfile(os.path.join(data_dir, data_file)):
    zip_url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00228/smsspamcollection.zip'
    r = requests.get(zip_url)
    z = ZipFile(io.BytesIO(r.content))
    file = z.read('SMSSpamCollection')
    # Format Data
    text_data = file.decode()
    text_data = text_data.encode('ascii', errors='ignore')
    text_data = text_data.decode().split('\n')
```

```

# Save data to text file
    with open(os.path.join(data_dir, data_file), 'w') as file_conn:
        for text in text_data:
            file_conn.write("{}\n".format(text))
else:
    # Open data from text file
    text_data = []
    with open(os.path.join(data_dir, data_file), 'r') as file_conn:
        for row in file_conn:
            text_data.append(row)
    text_data = text_data[:-1]
text_data = [x.split('\t') for x in text_data if len(x)>=1]
[text_data_target, text_data_train] = [list(x) for x in zip(*text_data)]

```

4. To reduce our vocabulary, we will clean the input text by removing special characters, and extra white space, and putting everything in lowercase:

```

def clean_text(text_string):
    text_string = re.sub(r'([^\w\s]|_|[0-9])+', ' ', text_string)
    text_string = " ".join(text_string.split())
    text_string = text_string.lower()
    return text_string

# Clean texts
text_data_train = [clean_text(x) for x in text_data_train]

```

 **TIP** Note that our cleaning step removes special characters. As an alternative, we could also have replaced them with a space. Ideally, this depends on the formatting of the dataset.

5. Now we process the text with a built-in vocabulary processor function from TensorFlow. This will convert the text to an appropriate list of indices:

```

vocab_processor = tf.contrib.learn.preprocessing.VocabularyProcessor(max_sequence_length)
text_processed = np.array(list(vocab_processor.fit_transform(text_data_train)))

```

 **i** Note that the functions in `contrib.learn.preprocessing` are currently deprecated, (using the current TensorFlow version, 1.10). The current replacement suggestion, the TensorFlow-preprocessing package, only runs in Python 2 as of now. The effort to move TensorFlow-preprocessing to Python 3 is currently underway and will replace the previous two lines. Remember that all current and up-to-date code can be found on this GitHub page: https://www.github.com/nfmcculler/tensorflow_cookbook and at the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

6. Next, we shuffle the data to randomize it:

```

text_processed = np.array(text_processed)
text_data_target = np.array([1 if x=='ham' else 0 for x in text_data_target])
shuffled_ix = np.random.permutation(np.arange(len(text_data_target)))
x_shuffled = text_processed[shuffled_ix]
y_shuffled = text_data_target[shuffled_ix]

```

7. We also split the data into an 80-20 train-test dataset:

```

ix_cutoff = int(len(y_shuffled)*0.80)
x_train, x_test = x_shuffled[:ix_cutoff], x_shuffled[ix_cutoff:]
y_train, y_test = y_shuffled[:ix_cutoff], y_shuffled[ix_cutoff:]

```

```
vocab_size = len(vocab_processor.vocabulary_)
print("Vocabulary Size: {:d}".format(vocab_size))
print("80-20 Train Test split: {:d} -- {:d}".format(len(y_train), len(y_test)))
```



For this recipe, we will not be doing any hyperparameter tuning. If the reader goes in this direction, remember to split up the dataset into train-test-validation sets before proceeding. A good option for this is the Scikit-learn function `model_selection.train_test_split()`.

8. Next, we declare the graph placeholders. The `x`-input will be a placeholder of size `[None, max_sequence_length]`, which will be the batch size according to the maximum allowed word length for text messages. The `y`-output placeholder is just an integer, 0 or 1, for ham or spam:

```
x_data = tf.placeholder(tf.int32, [None, max_sequence_length])
y_output = tf.placeholder(tf.int32, [None])
```

9. We now create our embedding matrix and embedding lookup operation for the `x`-input data:

```
embedding_mat = tf.Variable(tf.random_uniform([vocab_size, embedding_size], -1.0
                                             embedding_output = tf.nn.embedding_lookup(embedding_mat, x_data)
```

10. We declare our model as follows. First, we initialize a type of RNN cell to use (RNN of size 10). Then we create the RNN sequence by making it a dynamic RNN. We then add dropout to the RNN:

```
cell = tf.nn.rnn_cell.BasicRNNCell(num_units = rnn_size)
output, state = tf.nn.dynamic_rnn(cell, embedding_output, dtype=tf.float32)
output = tf.nn.dropout(output, dropout_keep_prob)
```



Note that dynamic RNN allows for variable length sequences. Even though we are using a fixed sequence length in this example, it is usually preferred to use `dynamic_rnn` in TensorFlow for two main reasons. One reason is that, in practice, the dynamic RNN actually runs computationally faster; the second is that, if we choose, we can run sequences of different lengths through the RNN.

11. Now to get our predictions, we have to rearrange the RNN and slice off the last output:

```
output = tf.transpose(output, [1, 0, 2])
last = tf.gather(output, int(output.get_shape()[0]) - 1)
```

12. To finish the RNN prediction, we convert from the `rnn_size` output to the two category outputs via a fully connected network layer:

```
weight = tf.Variable(tf.truncated_normal([rnn_size, 2], stddev=0.1))
bias = tf.Variable(tf.constant(0.1, shape=[2]))
logits_out = tf.nn.softmax(tf.matmul(last, weight) + bias)
```

13. We declare our loss function next. Remember that, when using the

`sparse_softmax` function from TensorFlow, the targets have to be integer indices (of type `int`) and the logits have to be floats:

```
losses = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits_out, labels)
loss = tf.reduce_mean(losses)
```

14. We also need an accuracy function so that we can compare the algorithm on the test and training sets:

```
accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(logits_out, 1), tf.cast(y_0,
```

15. Next, we create the optimization function and initialize model variables:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate)
train_step = optimizer.minimize(loss)
init = tf.global_variables_initializer()
sess.run(init)
```

16. Now we can start looping through our data and training the model. When looping through the data multiple times, it is good practice to shuffle the data every epoch to prevent over-training:

```
train_loss = []
test_loss = []
train_accuracy = []
test_accuracy = []
# Start training
for epoch in range(epochs):
    # Shuffle training data
    shuffled_ix = np.random.permutation(np.arange(len(x_train)))
    x_train = x_train[shuffled_ix]
    y_train = y_train[shuffled_ix]
    num_batches = int(len(x_train)/batch_size) + 1
    for i in range(num_batches):
        # Select train data
        min_ix = i * batch_size
        max_ix = np.min([len(x_train), ((i+1) * batch_size)])
        x_train_batch = x_train[min_ix:max_ix]
        y_train_batch = y_train[min_ix:max_ix]

        # Run train step
        train_dict = {x_data: x_train_batch, y_output: y_train_batch, dropout_keep_prob: 0.5}
        sess.run(train_step, feed_dict=train_dict)

    # Run loss and accuracy for training
    temp_train_loss, temp_train_acc = sess.run([loss, accuracy], feed_dict=train_dict)
    train_loss.append(temp_train_loss)
    train_accuracy.append(temp_train_acc)

    # Run Eval Step
    test_dict = {x_data: x_test, y_output: y_test, dropout_keep_prob: 1.0}
    temp_test_loss, temp_test_acc = sess.run([loss, accuracy], feed_dict=test_dict)
    test_loss.append(temp_test_loss)
    test_accuracy.append(temp_test_acc)
    print('Epoch: {}, Test Loss: {:.2}, Test Acc: {:.2}'.format(epoch+1, temp_te
```

17. This results in the following output:

```
Vocabulary Size: 933
80-20 Train Test split: 4459 -- 1115
Epoch: 1, Test Loss: 0.59, Test Acc: 0.83
Epoch: 2, Test Loss: 0.58, Test Acc: 0.83
...
Epoch: 19, Test Loss: 0.46, Test Acc: 0.86
Epoch: 20, Test Loss: 0.46, Test Acc: 0.86
```

18. The following is the code to plot the train/test loss and accuracy:

```
epoch_seq = np.arange(1, epochs+1)
plt.plot(epoch_seq, train_loss, 'k--', label='Train Set')
plt.plot(epoch_seq, test_loss, 'r-', label='Test Set')
plt.title('Softmax Loss')
plt.xlabel('Epochs')
plt.ylabel('Softmax Loss')
plt.legend(loc='upper left')
plt.show()
# Plot accuracy over time
plt.plot(epoch_seq, train_accuracy, 'k--', label='Train Set')
plt.plot(epoch_seq, test_accuracy, 'r-', label='Test Set')
plt.title('Test Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.show()
```

How it works...

In this recipe, we created an RNN-to-category model to predict whether SMS text is spam or ham. We achieved about 86% accuracy on the test set. The following are accuracy and loss plots on both the test and training sets:

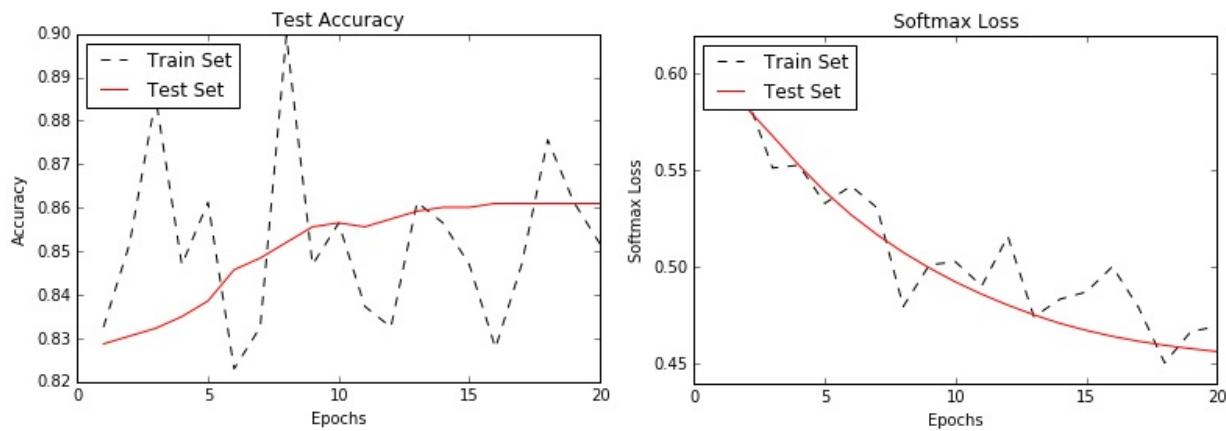


Figure 3: The accuracy (left) and loss (right) of the training and test sets

There's more...

It is highly recommended that you go through the training dataset multiple times for sequential data (and this also recommended for non-sequential data). Each pass through the data is called an **epoch**. Also, it is very common (and highly recommended) to shuffle the data before each epoch, to minimize the effect of the data order on the training.

Implementing an LSTM model

We will extend our RNN model to be able to use longer sequences by introducing the LSTM unit in this recipe.

Getting ready

Long Short Term Memory (LSTM) is a variant of the traditional RNN. LSTM is a way to address the vanishing/exploding gradient problem that variable length RNNs have. To address this issue, LSTM cells introduce an internal forget gate, which can modify a flow of information from one cell to the next. To conceptualize how this works, we will walk through an unbiased version of LSTM one equation at a time. The first step is the same as for the regular RNN:

$$i_t = \sigma(B_i h_{t-1} + A_i x_t)$$

In order to figure out which values we want to forget or pass through, we will evaluate candidate values as follows. These values are often referred to as memory cells:

$$C_t = \tanh(B_c h_{t-1} + A_c x_t)$$

Now we modify the candidate memory cells with a forget matrix, which is calculated as follows:

$$f_t = \sigma(B_f h_{t-1} + A_f x_t)$$

We now combine the forget memory with the prior memory step and add it to the candidate memory cells to arrive at the new memory values:

$$N_t = i_t \cdot C_t + f_t N_{t-1}$$

Now we combine everything to get the output of the cell:

$$O_t = \sigma(B_o h_{t-1} + A_o x_t + D_o N_t)$$

Then for the next iteration, we update h as follows:

$$h_t = O_t \cdot \tanh(N_t)$$

The idea with LSTM is to have a self-regulating flow of information through the cells that can be forgotten or modified based on the information inputted to the cell.



One great thing about using TensorFlow here is that we do not have to keep track of these operations and their corresponding back propagation attributes. TensorFlow will keep track of these and update the model variables here automatically, according to the gradients specified by our loss function, optimizer, and learning rates.

For this recipe, we will use a sequence RNN with LSTM cells to try to predict the next words, trained on the works of Shakespeare. To test how we are doing, we will feed the model candidate phrases, such as, *thou art more*, and see whether the model can figure out what words should follow the phrase.

How to do it...

1. To start, we load the necessary libraries for the script:

```
import os
import re
import string
import requests
import numpy as np
import collections
import random
import pickle
import matplotlib.pyplot as plt
import tensorflow as tf
```

2. Next, we start a graph session and set the RNN parameters:

```
sess = tf.Session()

# Set RNN Parameters
min_word_freq = 5
rnn_size = 128
epochs = 10
batch_size = 100
learning_rate = 0.001
training_seq_len = 50
embedding_size = rnn_size
save_every = 500
eval_every = 50
prime_texts = ['thou art more', 'to be or not to', 'wherefore art thou']
```

3. We set up the data and model folders and filenames, along with declaring punctuation to be removed. We will want to keep hyphens and apostrophes because Shakespeare uses them frequently to combine words and syllables:

```
data_dir = 'temp'
data_file = 'shakespeare.txt'
model_path = 'shakespeare_model'
full_model_dir = os.path.join(data_dir, model_path)
# Declare punctuation to remove, everything except hyphens and apostrophe's
punctuation = string.punctuation
punctuation = ''.join([x for x in punctuation if x not in ['-', "'"]])
```

4. Next, we get the data. If the data file doesn't exist, we download and save the Shakespearean text. If it does exist, we load the data:

```
if not os.path.exists(full_model_dir):
    os.makedirs(full_model_dir)
# Make data directory
if not os.path.exists(data_dir):
    os.makedirs(data_dir)
```

```

print('Loading Shakespeare Data')
# Check if file is downloaded.
if not os.path.isfile(os.path.join(data_dir, data_file)):
    print('Not found, downloading Shakespeare texts from www.gutenberg.org')
    shakespeare_url = 'http://www.gutenberg.org/cache/epub/100/pg100.txt'
    # Get Shakespeare text
    response = requests.get(shakespeare_url)
    shakespeare_file = response.content
    # Decode binary into string
    s_text = shakespeare_file.decode('utf-8')
    # Drop first few descriptive paragraphs.
    s_text = s_text[7675:]
    # Remove newlines
    s_text = s_text.replace('\r\n', '')
    s_text = s_text.replace('\n', '')

    # Write to file
    with open(os.path.join(data_dir, data_file), 'w') as out_conn:
        out_conn.write(s_text)
else:
    # If file has been saved, load from that file
    with open(os.path.join(data_dir, data_file), 'r') as file_conn:
        s_text = file_conn.read().replace('\n', '')

```

5. We clean the Shakespearean text by removing punctuation and extra whitespace:

```

s_text = re.sub(r'[\{}]'.format(punctuation), ' ', s_text)
s_text = re.sub('s+', ' ', s_text).strip().lower()

```

6. We now deal with creating the Shakespearean vocabulary to use. We create a function that will return two dictionaries (word to index and index to word) with words that appear with more than the specified frequency:

```

def build_vocab(text, min_word_freq):
    word_counts = collections.Counter(text.split(' '))
    # limit word counts to those more frequent than cutoff
    word_counts = {key:val for key, val in word_counts.items() if val>min_word_f
    # Create vocab --> index mapping
    words = word_counts.keys()
    vocab_to_ix_dict = {key:(ix+1) for ix, key in enumerate(words)}
    # Add unknown key --> 0 index
    vocab_to_ix_dict['unknown']=0
    # Create index --> vocab mapping
    ix_to_vocab_dict = {val:key for key, val in vocab_to_ix_dict.items()}

    return ix_to_vocab_dict, vocab_to_ix_dict
ix2vocab, vocab2ix = build_vocab(s_text, min_word_freq)
vocab_size = len(ix2vocab) + 1

```



Note that, when dealing with text, we have to be careful about indexing words with the value of zero. We should save the zero value for padding, and potentially for unknown words as well.

7. Now that we have our vocabulary, we turn the Shakespearean text into an array of indices:

```

s_text_words = s_text.split(' ')
s_text_ix = []
for ix, x in enumerate(s_text_words):
    try:
        s_text_ix.append(vocab2ix[x])
    except:
        s_text_ix.append(0)
s_text_ix = np.array(s_text_ix)

```

8. In this recipe, we will show how to create a model in a class object. This will be helpful for us, because we would like to use the same model (with the same weights) to train on batches and to generate text from sample text. This will prove hard to do without a class with an internal sampling method. Ideally, this class code should sit in a separate Python file, which we can import at the beginning of this script:

```

class LSTM_Model():
    def __init__(self, rnn_size, batch_size, learning_rate,
                 training_seq_len, vocab_size, infer=False):
        self.rnn_size = rnn_size
        self.vocab_size = vocab_size
        self.infer = infer
        self.learning_rate = learning_rate

        if infer:
            self.batch_size = 1
            self.training_seq_len = 1
        else:
            self.batch_size = batch_size
            self.training_seq_len = training_seq_len

        self.lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(rnn_size)
        self.initial_state = self.lstm_cell.zero_state(self.batch_size, tf.float32)

        self.x_data = tf.placeholder(tf.int32, [self.batch_size, self.training_seq_len])
        self.y_output = tf.placeholder(tf.int32, [self.batch_size, self.training_seq_len])

        with tf.variable_scope('lstm_vars'):
            # Softmax Output Weights
            W = tf.get_variable('W', [self.rnn_size, self.vocab_size], tf.float32)
            b = tf.get_variable('b', [self.vocab_size], tf.float32, tf.constant_initializer(0))

            # Define Embedding
            embedding_mat = tf.get_variable('embedding_mat', [self.vocab_size, self.rnn_size])

            embedding_output = tf.nn.embedding_lookup(embedding_mat, self.x_data)
            rnn_inputs = tf.split(embedding_output, num_or_size_splits=self.training_seq_len, axis=1)
            rnn_inputs_trimmed = [tf.squeeze(x, [1]) for x in rnn_inputs]

            # If we are inferring (generating text), we add a 'loop' function
            # Define how to get the i+1 th input from the i th output
            def inferred_loop(prev, count):
                prev_transformed = tf.matmul(prev, W) + b
                prev_symbol = tf.stop_gradient(tf.argmax(prev_transformed, 1))
                output = tf.nn.embedding_lookup(embedding_mat, prev_symbol)
                return output

            decoder = tf.nn.seq2seq.rnn_decoder
            outputs, last_state = decoder(rnn_inputs_trimmed,

```

```

        self.initial_state,
        self.lstm_cell,
        loop_function=inferred_loop if infer else
    # Non inferred outputs
    output = tf.reshape(tf.concat(1, outputs), [-1, self.rnn_size])
    # Logits and output
    self.logit_output = tf.matmul(output, W) + b
    self.model_output = tf.nn.softmax(self.logit_output)
    loss_fun = tf.contrib.legacy_seq2seq.sequence_loss_by_example
    loss = loss_fun([self.logit_output],[tf.reshape(self.y_output, [-1])],
                   [tf.ones([self.batch_size * self.training_seq_len])],
                   self.vocab_size)
    self.cost = tf.reduce_sum(loss) / (self.batch_size * self.training_seq_1
    self.final_state = last_state
    gradients, _ = tf.clip_by_global_norm(tf.gradients(self.cost, tf.trainable_variables()),
                                          self.optimizer = tf.train.AdamOptimizer(self.learning_rate)
                                          self.train_op = optimizer.apply_gradients(zip(gradients, tf.trainable_variables()))
                                          self.train_op = optimizer.apply_gradients(zip(gradients, tf.trainable_variables()))

def sample(self, sess, words=ix2vocab, vocab=vocab2ix, num=10, prime_text='t
state = sess.run(self.lstm_cell.zero_state(1, tf.float32))
word_list = prime_text.split()
for word in word_list[:-1]:
    x = np.zeros((1, 1))
    x[0, 0] = vocab[word]
    feed_dict = {self.x_data: x, self.initial_state:state}
    [state] = sess.run([self.final_state], feed_dict=feed_dict)
    out_sentence = prime_text
    word = word_list[-1]
    for n in range(num):
        x = np.zeros((1, 1))
        x[0, 0] = vocab[word]
        feed_dict = {self.x_data: x, self.initial_state:state}
        [model_output, state] = sess.run([self.model_output, self.final_state])
        sample = np.argmax(model_output[0])
        if sample == 0:
            break
        word = words[sample]
        out_sentence = out_sentence + ' ' + word
    return out_sentence

```

- Now we will declare the LSTM model as well as the test model. We will do this within a variable scope and tell the scope that we will reuse the variables for the test LSTM model:

```

with tf.variable_scope('lstm_model', reuse=tf.AUTO_REUSE) as scope:
    # Define LSTM Model
    lstm_model = LSTM_Model(rnn_size, batch_size, learning_rate,
                           training_seq_len, vocab_size)
    scope.reuse_variables()
    test_lstm_model = LSTM_Model(rnn_size, batch_size, learning_rate,
                                 training_seq_len, vocab_size, infer=True)

```

- We create a saving operation, as well as splitting up the input text into equal batch-sized chunks. Then we initialize the variables of the model:

```

saver = tf.train.Saver()
# Create batches for each epoch
num_batches = int(len(s_text_ix)/(batch_size * training_seq_len)) + 1
# Split up text indices into subarrays, of equal size

```

```

batches = np.array_split(s_text_ix, num_batches)
# Reshape each split into [batch_size, training_seq_len]
batches = [np.resize(x, [batch_size, training_seq_len]) for x in batches]
# Initialize all variables
init = tf.global_variables_initializer()
sess.run(init)

```

11. We can now iterate through our epochs, shuffling the data before each epoch starts. The target for our data is just the same data, but shifted by a value of 1 (using the `numpy.roll()` function):

```

train_loss = []
iteration_count = 1
for epoch in range(epochs):
    # Shuffle word indices
    random.shuffle(batches)
    # Create targets from shuffled batches
    targets = [np.roll(x, -1, axis=1) for x in batches]
    # Run a through one epoch
    print('Starting Epoch #{} of {}'.format(epoch+1, epochs))
    # Reset initial LSTM state every epoch
    state = sess.run(lstm_model.initial_state)
    for ix, batch in enumerate(batches):
        training_dict = {lstm_model.x_data: batch, lstm_model.y_output: targets[
            c, h = lstm_model.initial_state
            training_dict[c] = state.c
            training_dict[h] = state.h

        temp_loss, state, _ = sess.run([lstm_model.cost, lstm_model.final_state,
        train_loss.append(temp_loss)

        # Print status every 10 gens
        if iteration_count % 10 == 0:
            summary_nums = (iteration_count, epoch+1, ix+1, num_batches+1, temp_
            print('Iteration: {}, Epoch: {}, Batch: {} out of {}, Loss: {:.2f}'.

        # Save the model and the vocab
        if iteration_count % save_every == 0:
            # Save model
            model_file_name = os.path.join(full_model_dir, 'model')
            saver.save(sess, model_file_name, global_step = iteration_count)
            print('Model Saved To: {}'.format(model_file_name))
            # Save vocabulary
            dictionary_file = os.path.join(full_model_dir, 'vocab.pkl')
            with open(dictionary_file, 'wb') as dict_file_conn:
                pickle.dump([vocab2ix, ix2vocab], dict_file_conn)

        if iteration_count % eval_every == 0:
            for sample in prime_texts:
                print(test_lstm_model.sample(sess, ix2vocab, vocab2ix, num=10, p

        iteration_count += 1

```

12. This results in the following output:

```

Loading Shakespeare Data
Cleaning Text
Building Shakespeare Vocab
Vocabulary Length = 8009
Starting Epoch #1 of 10.

```

```

Iteration: 10, Epoch: 1, Batch: 10 out of 182, Loss: 10.37
Iteration: 20, Epoch: 1, Batch: 20 out of 182, Loss: 9.54
...
Iteration: 1790, Epoch: 10, Batch: 161 out of 182, Loss: 5.68
Iteration: 1800, Epoch: 10, Batch: 171 out of 182, Loss: 6.05
thou art more than i am a
to be or not to the man i have
wherefore art thou art of the long
Iteration: 1810, Epoch: 10, Batch: 181 out of 182, Loss: 5.99

```

13. And finally, the following is how we plot the training loss over the epochs:

```

plt.plot(train_loss, 'k-')
plt.title('Sequence to Sequence Loss')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()

```

This results in the following plot of our loss values:

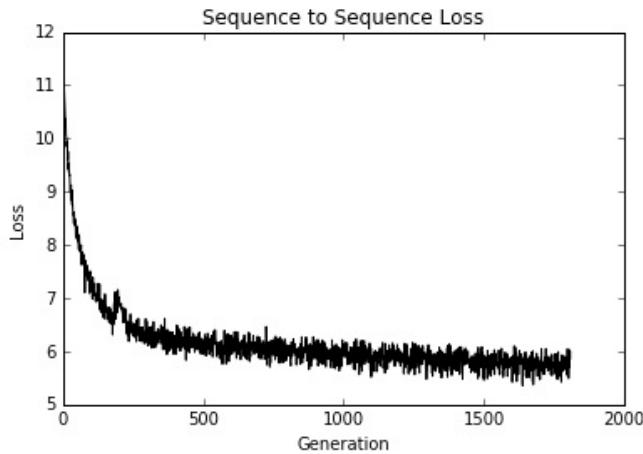


Figure 4: The sequence-to-sequence loss over all generations of the model

How it works...

In this example, we built an RNN model with LSTM units to predict the next word, based on Shakespearean vocabulary. There are a few things that could be done to improve the model, maybe increasing the sequence size, having a decaying learning rate, or training the model for more epochs.

There's more...

For sampling, we implemented a greedy sampler. Greedy samplers can get stuck repeating the same phrases over and over; for example, they may get stuck saying *for the for the for the....* To prevent this, we could also implement a more random way of sampling words, maybe by making a weighted sampler based on the logits or probability distribution of the output.

Stacking multiple LSTM layers

Just as we can increase the depth of neural networks or CNNs, we can increase the depth of RNN networks. In this recipe we apply a three-layer-deep LSTM to improve our Shakespearean language generation.

Getting ready

We can increase the depth of recurrent neural networks by stacking them on top of each other. Essentially, we will be taking the target outputs and feeding them into another network.

To get an idea of how this might work for just two layers, see the following diagram:

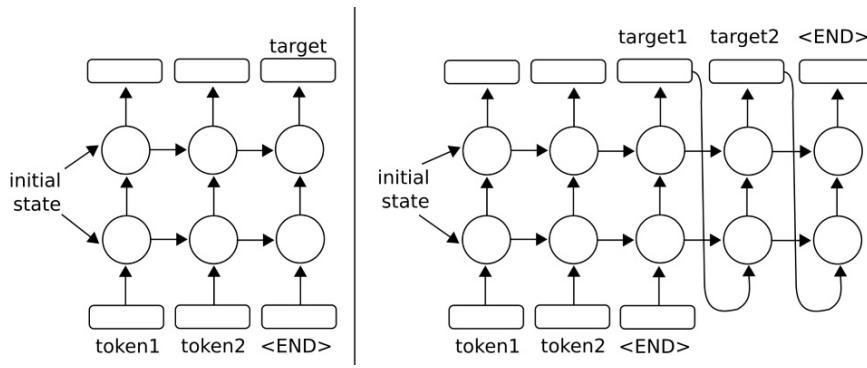


Figure 5: In the preceding diagram, we have extended one-layer RNNs so that they have two layers. For the original one-layer versions, see the diagrams in the introduction to the previous chapter. The left architecture illustrates a way of using a multi-layered RNN to predict one output from a sequence of outputs. The right architecture shows a way to use a multi-layered RNN to predict a sequence of outputs, which uses the outputs as inputs

TensorFlow allows easy implementation of multiple layers with a `MultiRNNCell()` function that accepts a list of RNN cells. With this behavior, it is easy to create a multi-layer RNN from one cell in Python with `MultiRNNCell([rnn_cell(num_units) for n in num_layers])` cells.

For this recipe, we will perform the same Shakespearean prediction that we performed in the previous recipe. There will be two changes: the first change will be having a three-stacked LSTM model instead of only one layer, and the second change will be doing character-level predictions instead of words. Doing character-level predictions will greatly decrease our potential vocabulary to only 40 characters (26 letters, 10 numbers, 1 whitespace, and 3 special characters).

How to do it...

Instead of relisting all the same code, we will illustrate where our code in this section will differ from the previous section. For the full code, please see the GitHub repository at https://github.com/nfmccclure/tensorflow_cookbook or the Packt repository at <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

1. We first need to set the number of layers for the model. We put this as a parameter at the beginning of our script, with the other model parameters:

```
| num_layers = 3  
| min_word_freq = 5  
  
| rnn_size = 128  
| epochs = 10
```

2. The first major change is that we will load, process, and feed the text by characters, not by words. In order to accomplish this, after cleaning the text, we can separate the whole text by character with Python's `list()` command:

```
| s_text = re.sub(r'[\{\}].format(punctuation), ' ', s_text)  
| s_text = re.sub('s+', ' ', s_text).strip().lower()  
| # Split up by characters  
| char_list = list(s_text)
```

3. We now need to change our LSTM model so that it has multiple layers. We take in the `num_layers` variable and create a multiple-layer RNN model with TensorFlow's `MultiRNNCell()` function, as follows:

```
class LSTM_Model():  
    def __init__(self, rnn_size, num_layers, batch_size, learning_rate,  
                 training_seq_len, vocab_size, infer_sample=False):  
        self.rnn_size = rnn_size  
        self.num_layers = num_layers  
        self.vocab_size = vocab_size  
        self.infer_sample = infer_sample  
        self.learning_rate = learning_rate  
        ...  
  
        self.lstm_cell = tf.contrib.rnn.BasicLSTMCell(rnn_size)  
        self.lstm_cell = tf.contrib.rnn.MultiRNNCell([self.lstm_cell for _ in ra  
        self.initial_state = self.lstm_cell.zero_state(self.batch_size, tf.float  
  
        self.x_data = tf.placeholder(tf.int32, [self.batch_size, self.training_s  
        self.y_output = tf.placeholder(tf.int32, [self.batch_size, self.training
```

Note that TensorFlow's `MultiRNNCell()` function accepts a list of RNN cells. In this project, the



RNN layers are all the same, but you can make a list of whichever RNN layers you would like to stack on top of each other.

4. Everything else is essentially the same. Here, we can see some of the training output:

```
Building Shakespeare Vocab by Characters
Vocabulary Length = 40
Starting Epoch #1 of 10
Iteration: 9430, Epoch: 10, Batch: 889 out of 950, Loss: 1.54
Iteration: 9440, Epoch: 10, Batch: 899 out of 950, Loss: 1.46
Iteration: 9450, Epoch: 10, Batch: 909 out of 950, Loss: 1.49
thou art more than the
to be or not to the serva
wherefore art thou dost thou
Iteration: 9460, Epoch: 10, Batch: 919 out of 950, Loss: 1.41
Iteration: 9470, Epoch: 10, Batch: 929 out of 950, Loss: 1.45
Iteration: 9480, Epoch: 10, Batch: 939 out of 950, Loss: 1.59
Iteration: 9490, Epoch: 10, Batch: 949 out of 950, Loss: 1.42
```

5. The following is a sample of the final text output:

```
| thou art more fancy with to be or not to be for be wherefore art thou art thou
```

6. Finally, the following is how we plot the training loss over the generations:

```
plt.plot(train_loss, 'k-')
plt.title('Sequence to Sequence Loss')
plt.xlabel('Generation')
plt.ylabel('Loss')
plt.show()
```

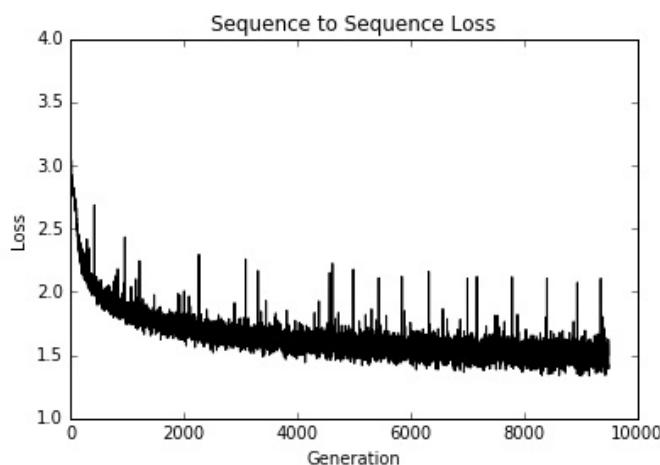


Figure 6: A plot of the training loss versus generations for the multiple-layer LSTM Shakespearean model

How it works...

TensorFlow makes it easy to extend an RNN layer to multiple layers with just a list of RNN cells. For this recipe, we used the same Shakespearean data as the previous recipe, but processed it by characters instead of words. We fed this through a triple-layer LSTM model to generate Shakespearean text. We can see that after only 10 epochs, we were able to generate archaic English in the form of words.

Creating sequence-to-sequence models

Since every RNN unit we use also has an output, we can train RNN sequences to predict other sequences of variable length. For this recipe, we will take advantage of this fact to create an English-to-German translation model.

Getting ready

For this recipe, we will attempt to build a language translation model to translate from English to German.

TensorFlow has a built-in model class for sequence-to-sequence training. We will illustrate how to train and use it on downloaded English-German sentences. The data we will use comes from a compiled zip file at <http://www.manythings.org/>, which compiles the data from the Tatoeba Project (<http://tatoeba.org/home>). This data is a tab-delimited English-German sentence translation; for example, a row might contain the sentence, `hello. /t hallo.` The data contains thousands of sentences of various lengths.

The code for this section has been upgraded to use the Neural Machine Translation models provided by the official TensorFlow repositories here: <https://github.com/tensorflow/nmt>.

This project will show you how to download data, use, modify, and add to the hyperparameters, and configure your own data to use the project files.

While official tutorials show you how to do this via the command line, this tutorial will show you how to use the internal code provided to train your own model from scratch.

How to do it...

1. We start by loading the necessary libraries:

```
import os
import re
import sys
import json
import math
import time
import string
import requests
import io
import numpy as np
import collections
import random
import pickle
import string
import matplotlib.pyplot as plt
import tensorflow as tf
from zipfile import ZipFile
from collections import Counter
from tensorflow.python.ops import lookup_ops
from tensorflow.python.framework import ops
ops.reset_default_graph()

local_repository = 'temp/seq2seq'
```

2. The following block of code will import the whole NMT model repository into the `temp` folder:

```
if not os.path.exists(local_repository):
    from git import Repo
    tf_model_repository = 'https://github.com/tensorflow/nmt/'
    Repo.clone_from(tf_model_repository, local_repository)
    sys.path.insert(0, 'temp/seq2seq/nmt/')

# May also try to use 'attention model' by importing the attention model:
# from temp.seq2seq.nmt import attention_model as attention_model
from temp.seq2seq.nmt import model as model
from temp.seq2seq.nmt.utils import vocab_utils as vocab_utils
import temp.seq2seq.nmt.model_helper as model_helper
import temp.seq2seq.nmt.utils.iterator_utils as iterator_utils
import temp.seq2seq.nmt.utils.misc_utils as utils
import temp.seq2seq.nmt.train as train
```

3. Next, we set some parameters about the vocabulary size, what punctuation we'll remove, and where the data will be stored:

```
# Model Parameters
vocab_size = 10000
punct = string.punctuation
```

```

# Data Parameters
data_dir = 'temp'
data_file = 'eng_ger.txt'
model_path = 'seq2seq_model'
full_model_dir = os.path.join(data_dir, model_path)

```

4. We will use the hyperparameter format that TensorFlow provides. This type of parameter storage (in external json or xml files) allows us to iterate through different types of architecture (in different files) programmatically. For this demonstration, we will use the `wmt16.json` provided to us and make a few changes:

```

# Load hyper-parameters for translation model. (Good defaults are provided in Re
hparams = tf.contrib.training.HParams()
param_file = 'temp/seq2seq/nmt/standard_hparams/wmt16.json'
# Can also try: (For different architectures)
# 'temp/seq2seq/nmt/standard_hparams/iwslt15.json'
# 'temp/seq2seq/nmt/standard_hparams/wmt16_gnmt_4_layer.json',
# 'temp/seq2seq/nmt/standard_hparams/wmt16_gnmt_8_layer.json',

with open(param_file, "r") as f:
    params_json = json.loads(f.read())

for key, value in params_json.items():
    hparams.add_hparam(key, value)
hparams.add_hparam('num_gpus', 0)
hparams.add_hparam('num_encoder_layers', hparams.num_layers)
hparams.add_hparam('num_decoder_layers', hparams.num_layers)
hparams.add_hparam('num_encoder_residual_layers', 0)
hparams.add_hparam('num_decoder_residual_layers', 0)
hparams.add_hparam('init_op', 'uniform')
hparams.add_hparam('random_seed', None)
hparams.add_hparam('num_embeddings_partitions', 0)
hparams.add_hparam('warmup_steps', 0)
hparams.add_hparam('length_penalty_weight', 0)
hparams.add_hparam('sampling_temperature', 0.0)
hparams.add_hparam('num_translations_per_input', 1)
hparams.add_hparam('warmup_scheme', 't2t')
hparams.add_hparam('epoch_step', 0)
hparams.num_train_steps = 5000

# Not use any pretrained embeddings
hparams.add_hparam('src_embed_file', '')
hparams.add_hparam('tgt_embed_file', '')
hparams.add_hparam('num_keep_ckpts', 5)
hparams.add_hparam('avg_ckpts', False)

# Remove attention
hparams.attention = None

```

5. Create the model and data directories if they do not already exist:

```

# Make Model Directory
if not os.path.exists(full_model_dir):
    os.makedirs(full_model_dir)

# Make data directory
if not os.path.exists(data_dir):

```

```
|     os.makedirs(data_dir)
```

6. Now we remove punctuation and split up the translation data into lists of words for both the English and German sentences:

```
print('Loading English-German Data')
# Check for data, if it doesn't exist, download it and save it
if not os.path.isfile(os.path.join(data_dir, data_file)):
    print('Data not found, downloading Eng-Ger sentences from www.manythings.org')
    sentence_url = 'http://www.manythings.org/anki/deu-eng.zip'
    r = requests.get(sentence_url)
    z = ZipFile(io.BytesIO(r.content))
    file = z.read('deu.txt')
    # Format Data
    eng_ger_data = file.decode('utf-8')
    eng_ger_data = eng_ger_data.encode('ascii', errors='ignore')
    eng_ger_data = eng_ger_data.decode().split('\n')
    # Write to file
    with open(os.path.join(data_dir, data_file), 'w') as out_conn:
        for sentence in eng_ger_data:
            out_conn.write(sentence + '\n')
else:
    eng_ger_data = []
    with open(os.path.join(data_dir, data_file), 'r') as in_conn:
        for row in in_conn:
            eng_ger_data.append(row[:-1])
print('Done!')
```

7. Now we remove punctuation for both the English and German sentences:

```
# Remove punctuation
eng_ger_data = [''.join(char for char in sent if char not in punct) for sent in
# Split each sentence by tabs
eng_ger_data = [x.split('\t') for x in eng_ger_data if len(x) >= 1]
[english_sentence, german_sentence] = [list(x) for x in zip(*eng_ger_data)]
english_sentence = [x.lower().split() for x in english_sentence]
german_sentence = [x.lower().split() for x in german_sentence]
```

8. In order to use the faster data-pipeline functions from TensorFlow, we will need to write the formatted data to disk in an appropriate format. The formats that the translation models expect are as follows:

```
train_prefix.source_suffix = train.en
train_prefix.target_suffix = train.de
```

The suffix will determine the language (`en` = English, `de` = deutsch), and the prefix determines the type of dataset (train or test):

```
# We need to write them to separate text files for the text-line-dataset operation
train_prefix = 'train'
src_suffix = 'en' # English
tgt_suffix = 'de' # Deutsch (German)
source_txt_file = train_prefix + '.' + src_suffix
hparams.add_hparam('src_file', source_txt_file)
target_txt_file = train_prefix + '.' + tgt_suffix
```

```

hparams.add_hparam('tgt_file', target_txt_file)
with open(source_txt_file, 'w') as f:
    for sent in english_sentence:
        f.write(' '.join(sent) + '\n')

with open(target_txt_file, 'w') as f:
    for sent in german_sentence:
        f.write(' '.join(sent) + '\n')

```

9. Next, we need to parse some (~100) testing sentence translations. We arbitrarily choose around 100 sentences. Then we also write them to the appropriate files:

```

# Partition some sentences off for testing files
test_prefix = 'test_sent'
hparams.add_hparam('dev_prefix', test_prefix)
hparams.add_hparam('train_prefix', train_prefix)
hparams.add_hparam('test_prefix', test_prefix)
hparams.add_hparam('src', src_suffix)
hparams.add_hparam('tgt', tgt_suffix)

num_sample = 100
total_samples = len(english_sentence)
# Get around 'num_sample's every so often in the src/tgt sentences
ix_sample = [x for x in range(total_samples) if x % (total_samples // num_sample)]
test_src = [' '.join(english_sentence[x]) for x in ix_sample]
test_tgt = [' '.join(german_sentence[x]) for x in ix_sample]

# Write test sentences to file
with open(test_prefix + '.' + src_suffix, 'w') as f:
    for eng_test in test_src:
        f.write(eng_test + '\n')

with open(test_prefix + '.' + tgt_suffix, 'w') as f:
    for ger_test in test_src:
        f.write(ger_test + '\n')

```

10. Next, we process the vocabularies of both the English and German sentences. Then we save vocabulary lists to the appropriate files:

```

print('Processing the vocabularies.')
# Process the English Vocabulary
all_english_words = [word for sentence in english_sentence for word in sentence]
all_english_counts = Counter(all_english_words)
eng_word_keys = [x[0] for x in all_english_counts.most_common(vocab_size-3)] # -
eng_vocab2ix = dict(zip(eng_word_keys, range(1, vocab_size)))
eng_ix2vocab = {val: key for key, val in eng_vocab2ix.items()}
english_processed = []
for sent in english_sentence:
    temp_sentence = []
    for word in sent:
        try:
            temp_sentence.append(eng_vocab2ix[word])
        except KeyError:
            temp_sentence.append(0)
    english_processed.append(temp_sentence)

# Process the German Vocabulary
all_german_words = [word for sentence in german_sentence for word in sentence]

```

```

all_german_counts = Counter(all_german_words)
ger_word_keys = [x[0] for x in all_german_counts.most_common(vocab_size-3)]
# -3 because UNK, S, /S is also in there
ger_vocab2ix = dict(zip(ger_word_keys, range(1, vocab_size)))
ger_ix2vocab = {val: key for key, val in ger_vocab2ix.items()}
german_processed = []
for sent in german_sentence:
    temp_sentence = []
    for word in sent:
        try:
            temp_sentence.append(ger_vocab2ix[word])
        except KeyError:
            temp_sentence.append(0)
    german_processed.append(temp_sentence)

# Save vocab files for data processing
source_vocab_file = 'vocab' + '.' + src_suffix
hparams.add_hparam('src_vocab_file', source_vocab_file)
eng_word_keys = ['<unk>', '<s>', '</s>'] + eng_word_keys

target_vocab_file = 'vocab' + '.' + tgt_suffix
hparams.add_hparam('tgt_vocab_file', target_vocab_file)
ger_word_keys = ['<unk>', '<s>', '</s>'] + ger_word_keys

# Write out all unique english words
with open(source_vocab_file, 'w') as f:
    for eng_word in eng_word_keys:
        f.write(eng_word + '\n')

# Write out all unique german words
with open(target_vocab_file, 'w') as f:
    for ger_word in ger_word_keys:
        f.write(ger_word + '\n')

# Add vocab size to hyper parameters
hparams.add_hparam('src_vocab_size', vocab_size)
hparams.add_hparam('tgt_vocab_size', vocab_size)

# Add out-directory
out_dir = 'temp/seq2seq/nmt_out'
hparams.add_hparam('out_dir', out_dir)
if not tf.gfile.Exists(out_dir):
    tf.gfile.MakeDirs(out_dir)

```

11. Next, we will be create training, inferring, and evaluation graphs separately. First, we create the training graph. We do this with a class and make the arguments a `namedtuple`. This code is from the NMT TensorFlow repository. See the file in the repository named `model_helper.py` for more information:

```

class TrainGraph(collections.namedtuple("TrainGraph", ("graph", "model", "iterator",
pass

def create_train_graph(scope=None):
    graph = tf.Graph()
    with graph.as_default():
        src_vocab_table, tgt_vocab_table = vocab_utils.create_vocab_tables(hpara
src_dataset = tf.data.TextLineDataset(hparams.src_file)
tgt_dataset = tf.data.TextLineDataset(hparams.tgt_file)

```

```

skip_count_placeholder = tf.placeholder(shape=(), dtype=tf.int64)

iterator = iterator_utils.get_iterator(src_dataset, tgt_dataset, src_vocab_t
final_model = model.Model(hparams, iterator=iterator, mode=tf.contrib.learn.Mod
return TrainGraph(graph=graph, model=final_model, iterator=iterator, skip_count
train_graph = create_train_graph()

```

12. We now create the evaluation graph:

```

# Create the evaluation graph
class EvalGraph(collections.namedtuple("EvalGraph", ("graph", "model", "src_file
pass

def create_eval_graph(scope=None):
    graph = tf.Graph()

    with graph.as_default():
        src_vocab_table, tgt_vocab_table = vocab_utils.create_vocab_tables(
            hparams.src_vocab_file, hparams.tgt_vocab_file, hparams.share_vocab)
        src_file_placeholder = tf.placeholder(shape=(), dtype=tf.string)
        tgt_file_placeholder = tf.placeholder(shape=(), dtype=tf.string)
        src_dataset = tf.data.TextLineDataset(src_file_placeholder)
        tgt_dataset = tf.data.TextLineDataset(tgt_file_placeholder)
        iterator = iterator_utils.get_iterator(
            src_dataset,
            tgt_dataset,
            src_vocab_table,
            tgt_vocab_table,
            hparams.batch_size,
            sos=hparams.sos,
            eos=hparams.eos,
            random_seed=hparams.random_seed,
            num_buckets=hparams.num_buckets,
            src_max_len=hparams.src_max_len_infer,
            tgt_max_len=hparams.tgt_max_len_infer)
        final_model = model.Model(hparams,
                                  iterator=iterator,
                                  mode=tf.contrib.learn.ModeKeys.EVAL,
                                  source_vocab_table=src_vocab_table,
                                  target_vocab_table=tgt_vocab_table,
                                  scope=scope)
    return EvalGraph(graph=graph,
                    model=final_model,
                    src_file_placeholder=src_file_placeholder,
                    tgt_file_placeholder=tgt_file_placeholder,
                    iterator=iterator)

eval_graph = create_eval_graph()

```

13. And now we do the same for the inference graph:

```

# Inference graph
class InferGraph(collections.namedtuple("InferGraph", ("graph", "model", "src_plac
pass

def create_infer_graph(scope=None):
    graph = tf.Graph()
    with graph.as_default():

```

```

        src_vocab_table, tgt_vocab_table = vocab_utils.create_vocab_tables(hparams)
        reverse_tgt_vocab_table = lookup_ops.index_to_string_table_from_file(hparams)

        src_placeholder = tf.placeholder(shape=[None], dtype=tf.string)
        batch_size_placeholder = tf.placeholder(shape=[], dtype=tf.int64)
        src_dataset = tf.data.Dataset.from_tensor_slices(src_placeholder)
        iterator = iterator_utils.get_infer_iterator(src_dataset,
                                                     src_vocab_table,
                                                     batch_size=batch_size_placeholder,
                                                     eos=hparams.eos,
                                                     src_max_len=hparams.src_max)

        final_model = model.Model(hparams,
                                  iterator=iterator,
                                  mode=tf.contrib.learn.ModeKeys.INFER,
                                  source_vocab_table=src_vocab_table,
                                  target_vocab_table=tgt_vocab_table,
                                  reverse_target_vocab_table=reverse_tgt_vocab_table,
                                  scope=scope)

        return InferGraph(graph=graph,
                          model=final_model,
                          src_placeholder=src_placeholder,
                          batch_size_placeholder=batch_size_placeholder,
                          iterator=iterator)

infer_graph = create_infer_graph()

```

14. For more illustrative output during training, we provide a short list of arbitrary source/target translations that will be output during training iterations:

```

# Create sample data for evaluation
sample_ix = [25, 125, 240, 450]
sample_src_data = [' '.join(english_sentence[x]) for x in sample_ix]
sample_tgt_data = [' '.join(german_sentence[x]) for x in sample_ix]
print([x for x in zip(sample_src_data, sample_tgt_data)])

```

15. Next, we load the training graph:

```

config_proto = utils.get_config_proto()

train_sess = tf.Session(config=config_proto, graph=train_graph.graph)
eval_sess = tf.Session(config=config_proto, graph=eval_graph.graph)
infer_sess = tf.Session(config=config_proto, graph=infer_graph.graph)

# Load the training graph
with train_graph.graph.as_default():
    loaded_train_model, global_step = model_helper.create_or_load_model(train_graph,
                                                                      hparams,
                                                                      train_se
                                                                      "train")

summary_writer = tf.summary.FileWriter(os.path.join(hparams.out_dir, 'Training'))

```

16. Now we add evaluation operations to the graph:

```

for metric in hparams.metrics:
    hparams.add_hparam("best_" + metric, 0)
    best_metric_dir = os.path.join(hparams.out_dir, "best_" + metric)

```

```

    hparams.add_hparam("best_" + metric + "_dir", best_metric_dir)
    tf.gfile.MakeDirs(best_metric_dir)

    eval_output = train.run_full_eval(hparams.out_dir, infer_graph, infer_sess, eval
    eval_results, _, acc_blue_scores = eval_output

```

17. Now we create initialization operations and initialize graphs; we also initialize some parameters that will update each iteration (time, global step, and epoch step):

```

# Training Initialization
last_stats_step = global_step
last_eval_step = global_step
last_external_eval_step = global_step

steps_per_eval = 10 * hparams.steps_per_stats
steps_per_external_eval = 5 * steps_per_eval

avg_step_time = 0.0
step_time, checkpoint_loss, checkpoint_predict_count = 0.0, 0.0, 0.0
checkpoint_total_count = 0.0
speed, train_ppl = 0.0, 0.0

utils.print_out("# Start step %d, lr %g, %s" %
               (global_step, loaded_train_model.learning_rate.eval(session=trai
               time.ctime())))
skip_count = hparams.batch_size * hparams.epoch_step
utils.print_out("# Init train iterator, skipping %d elements" % skip_count)

train_sess.run(train_graph.iterator.initializer,
              feed_dict={train_graph.skip_count_placeholder: skip_count})

```

 Note that the training will save the model every 1,000 iterations by default. You can change this in the hyperparameters if you want. Currently, training this model and saving the five most recent models take up about 2 GB of hard drive space.

18. The following code will start the training and evaluation of the model. The important part of the training is at the very beginning (the top third) of the loop. The rest of the code is devoted to evaluation, inferring from samples, and saving the models, as follows:

```

# Run training
while global_step < hparams.num_train_steps:
    start_time = time.time()
    try:
        step_result = loaded_train_model.train(train_sess)
        (_, step_loss, step_predict_count, step_summary, global_step, step_word_
         batch_size, __, __) = step_result
        hparams.epoch_step += 1
    except tf.errors.OutOfRangeError:
        # Next Epoch
        hparams.epoch_step = 0
        utils.print_out("# Finished an epoch, step %d. Perform external evaluati
        train.run_sample_decode(infer_graph,
                               infer_sess,
                               hparams.out_dir,
                               hparams,

```

```

                summary_writer,
                sample_src_data,
                sample_tgt_data)
        dev_scores, test_scores, _ = train.run_external_eval(infer_graph,
                                                          infer_sess,
                                                          hparams.out_dir,
                                                          hparams,
                                                          summary_writer)
        train_sess.run(train_graph.iterator.initializer, feed_dict={train_graph.
        continue

        summary_writer.add_summary(step_summary, global_step)

# Statistics
step_time += (time.time() - start_time)
checkpoint_loss += (step_loss * batch_size)
checkpoint_predict_count += step_predict_count
checkpoint_total_count += float(step_word_count)

# print statistics
if global_step - last_stats_step >= hparams.steps_per_stats:
    last_stats_step = global_step
    avg_step_time = step_time / hparams.steps_per_stats
    train_ppl = utils.safe_exp(checkpoint_loss / checkpoint_predict_count)
    speed = checkpoint_total_count / (1000 * step_time)

    utils.print_out(" global step %d lr %g "
                  "step-time %.2fs wps %.2fK ppl %.2f %s" %
                  (global_step,
                   loaded_train_model.learning_rate.eval(session=train_sess),
                   avg_step_time, speed, train_ppl, train._get_best_result

if math.isnan(train_ppl):
    break

# Reset timer and loss.
step_time, checkpoint_loss, checkpoint_predict_count = 0.0, 0.0, 0.0
checkpoint_total_count = 0.0

if global_step - last_eval_step >= steps_per_eval:
    last_eval_step = global_step
    utils.print_out("# Save eval, global step %d" % global_step)
    utils.add_summary(summary_writer, global_step, "train_ppl", train_ppl)

    # Save checkpoint
    loaded_train_model.saver.save(train_sess, os.path.join(hparams.out_dir,

    # Evaluate on dev/test
    train.run_sample_decode(infer_graph,
                           infer_sess,
                           out_dir,
                           hparams,
                           summary_writer,
                           sample_src_data,
                           sample_tgt_data)
    dev_ppl, test_ppl = train.run_internal_eval(eval_graph,
                                                eval_sess,
                                                out_dir,
                                                hparams,
                                                summary_writer)

if global_step - last_external_eval_step >= steps_per_external_eval:
    last_external_eval_step = global_step

```


How it works...

For this recipe, we used TensorFlow's built in sequence-to-sequence model to translate from English to German.

As we did not get perfect translations for our test sentences, there is room for improvement. If we trained for longer, and potentially combined some buckets (with more training data in each bucket), we might be able to improve our translations.

There's more...

There are other similar bilingual sentence datasets hosted on the Many Things website (<http://www.manythings.org/anki/>). Feel free to substitute any language dataset that appeals to you.

Training a Siamese similarity measure

A great property of RNN models, when compared to many other models, is that they can deal with sequences of various lengths. Taking advantage of this, and the fact that they can generalize to sequences not seen before, we can create a way to measure how similar sequences of inputs are to each other. In this recipe, we will train a Siamese similarity RNN to measure the similarity between addresses for record matching.

Getting ready

In this recipe, we will build a bidirectional RNN model that feeds into a fully connected layer that outputs a fixed-length numerical vector. We create a bidirectional RNN layer for both input addresses and feed the outputs into a fully connected layer that outputs a fixed-length numerical vector (length 100). We then compare the two vector outputs with the cosine distance, which is bounded between -1 and 1. We denote input data to be similar with a target of 1, and different with a target of -1. The prediction of the cosine distance is just the sign of the output (negative means dissimilar, positive means similar). We can use this network to do record matching by taking the reference address that scores the highest on the cosine distance from the query address.

See the following network architecture diagram:

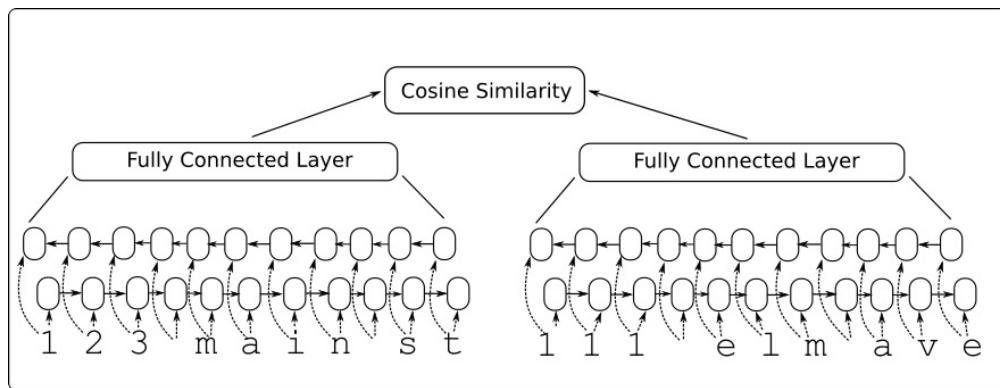


Figure 8: Siamese RNN similarity model architecture

What is also great about this model is that it accepts inputs that it has not seen before and can compare them with an output of -1 to 1. We will show this in the code by picking a test address that the model has not seen before and seeing whether it can match it to a similar address.

How to do it...

1. We start by loading the necessary libraries and starting a graph session:

```
import os
import random
import string
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
sess = tf.Session()
```

2. We now set the model parameters as follows:

```
batch_size = 200
n_batches = 300
max_address_len = 20
margin = 0.25
num_features = 50
dropout_keep_prob = 0.8
```

3. Next, we create the Siamese RNN similarity model class as follows:

```
def snn(address1, address2, dropout_keep_prob,
        vocab_size, num_features, input_length):

    # Define the Siamese double RNN with a fully connected layer at the end
    def Siamese_nn(input_vector, num_hidden):
        cell_unit = tf.nn.rnn_cell.BasicLSTMCell

        # Forward direction cell
        lstm_forward_cell = cell_unit(num_hidden, forget_bias=1.0)
        lstm_forward_cell = tf.nn.rnn_cell.DropoutWrapper(lstm_forward_cell, out

        # Backward direction cell
        lstm_backward_cell = cell_unit(num_hidden, forget_bias=1.0)
        lstm_backward_cell = tf.nn.rnn_cell.DropoutWrapper(lstm_backward_cell, o

        # Split title into a character sequence
        input_embed_split = tf.split(1, input_length, input_vector)
        input_embed_split = [tf.squeeze(x, squeeze_dims=[1]) for x in input_embe

        # Create bidirectional layer
        outputs, _, _ = tf.nn.bidirectional_rnn(lstm_forward_cell,
                                                lstm_backward_cell,
                                                input_embed_split,
                                                dtype=tf.float32)
        # Average The output over the sequence
        temporal_mean = tf.add_n(outputs) / input_length

        # Fully connected layer
        output_size = 10
        A = tf.get_variable(name="A", shape=[2*num_hidden, output_size],
                            dtype=tf.float32,
```

```

        initializer=tf.random_normal_initializer(stddev=0.1)
b = tf.get_variable(name="b", shape=[output_size], dtype=tf.float32,
        initializer=tf.random_normal_initializer(stddev=0.1)

final_output = tf.matmul(temporal_mean, A) + b
final_output = tf.nn.dropout(final_output, dropout_keep_prob)

return(final_output)

with tf.variable_scope("Siamese") as scope:
    output1 = Siamese_nn(address1, num_features)
    # Declare that we will use the same variables on the second string
    scope.reuse_variables()
    output2 = Siamese_nn(address2, num_features)

    # Unit normalize the outputs
    output1 = tf.nn.l2_normalize(output1, 1)
    output2 = tf.nn.l2_normalize(output2, 1)
    # Return cosine distance
    #   in this case, the dot product of the norms is the same.
    dot_prod = tf.reduce_sum(tf.mul(output1, output2), 1)

    return dot_prod

```



Note the use of a variable scope to share parameters between the two parts of the Siamese network for the two address inputs. Also, note that the cosine distance is achieved by the dot product of the normalized vectors.

- Now we will declare our prediction function, which is just the sign of the cosine distance as follows:

```

def get_predictions(scores):
    predictions = tf.sign(scores, name="predictions")
    return predictions

```

- Now we will declare our `loss` function as described before. Remember that we want to leave a margin (similar to an SVM model) for error. We will also have a loss term for true positives and true negatives. Use the following code for the loss:

```

def loss(scores, y_target, margin):
    # Calculate the positive losses
    pos_loss_term = 0.25 * tf.square(tf.sub(1., scores))
    pos_mult = tf.cast(y_target, tf.float32)

    # Make sure positive losses are on similar strings
    positive_loss = tf.mul(pos_mult, pos_loss_term)

    # Calculate negative losses, then make sure on dissimilar strings
    neg_mult = tf.sub(1., tf.cast(y_target, tf.float32))

    negative_loss = neg_mult*tf.square(scores)

    # Combine similar and dissimilar losses
    loss = tf.add(positive_loss, negative_loss)

    # Create the margin term. This is when the targets are 0, and the scores ar

```

```

# Check if target is zero (dissimilar strings)
target_zero = tf.equal(tf.cast(y_target, tf.float32), 0.)
# Check if cosine outputs is smaller than margin
less_than_margin = tf.less(scores, margin)
# Check if both are true
both_logical = tf.logical_and(target_zero, less_than_margin)
both_logical = tf.cast(both_logical, tf.float32)
# If both are true, then multiply by (1-1)=0.
multiplicative_factor = tf.cast(1. - both_logical, tf.float32)
total_loss = tf.mul(loss, multiplicative_factor)

# Average loss over batch
avg_loss = tf.reduce_mean(total_loss)
return avg_loss

```

6. We declare an accuracy function as follows:

```

def accuracy(scores, y_target):
    predictions = get_predictions(scores)
    correct_predictions = tf.equal(predictions, y_target)
    accuracy = tf.reduce_mean(tf.cast(correct_predictions, tf.float32))
    return accuracy

```

7. We will create similar addresses by creating a typo in an address. We will denote these addresses (the reference address and typo address) as similar:

```

def create_typos(s):
    rand_ind = random.choice(range(len(s)))
    s_list = list(s)
    s_list[rand_ind]=random.choice(string.ascii_lowercase + '0123456789')
    s = ''.join(s_list)
    return s

```

8. The data that we will generate will be random combinations of street numbers, street_names, and street suffixes. The names and suffixes are from the following list:

```

street_names = ['abbey', 'baker', 'canal', 'donner', 'elm', 'fifth', 'grandvia',
street_types = ['rd', 'st', 'ln', 'pass', 'ave', 'hwy', 'cir', 'dr', 'jct']

```

9. We generate test queries and references as follows:

```

test_queries = ['111 abbey ln', '271 doner cicle',
                '314 king avenue', 'tensorflow is fun']
test_references = ['123 abbey ln', '217 donner cir', '314 kings ave', '404 holly

```



Note that the last query and reference are not addresses that the model will have seen before, but we hope that they will be what the model ultimately sees as most similar.

10. We will now define how to generate a batch of data. Our batch of data will be 50% similar addresses (reference address and a typo address) and 50% dissimilar addresses. We generate the dissimilar addresses by taking half of

the address list and shifting the targets by one position (with the `numpy.roll()` function):

```
def get_batch(n):
    # Generate a list of reference addresses with similar addresses that have
    # a typo.
    numbers = [random.randint(1, 9999) for i in range(n)]
    streets = [random.choice(street_names) for i in range(n)]
    street_suffs = [random.choice(street_types) for i in range(n)]
    full_streets = [str(w) + ' ' + x + ' ' + y for w,x,y in zip(numbers, streets)]
    typo_streets = [create_typos(x) for x in full_streets]
    reference = [list(x) for x in zip(full_streets, typo_streets)]

    # Shuffle last half of them for training on dissimilar addresses
    half_ix = int(n/2)
    bottom_half = reference[half_ix:]
    true_address = [x[0] for x in bottom_half]
    typo_address = [x[1] for x in bottom_half]
    typo_address = list(np.roll(typo_address, 1))
    bottom_half = [[x,y] for x,y in zip(true_address, typo_address)]
    reference[half_ix:] = bottom_half

    # Get target similarities (1's for similar, -1's for non-similar)
    target = [1]*(n-half_ix) + [-1]*half_ix
    reference = [[x,y] for x,y in zip(reference, target)]
    return reference
```

11. Next, we define our address vocabulary and specify how to one-hot-encode the addresses to indices:

```
vocab_chars = string.ascii_lowercase + '0123456789 '
vocab2ix_dict = {char:(ix+1) for ix, char in enumerate(vocab_chars)}
vocab_length = len(vocab_chars) + 1

# Define vocab one-hot encoding
def address2onehot(address,
                    vocab2ix_dict = vocab2ix_dict,
                    max_address_len = max_address_len):
    # translate address string into indices
    address_ix = [vocab2ix_dict[x] for x in list(address)]

    # Pad or crop to max_address_len
    address_ix = (address_ix + [0]*max_address_len)[0:max_address_len]
    return address_ix
```

12. After dealing with the vocabulary, we will start declaring our model placeholders and the embedding lookup. For the embedding lookup, we will be using one-hot encoded embedding by using an identity matrix as the lookup matrix. Use the following code:

```
address1_ph = tf.placeholder(tf.int32, [None, max_address_len], name="address1_p")
address2_ph = tf.placeholder(tf.int32, [None, max_address_len], name="address2_p")
y_target_ph = tf.placeholder(tf.int32, [None], name="y_target_ph")
dropout_keep_prob_ph = tf.placeholder(tf.float32, name="dropout_keep_prob")

# Create embedding lookup
identity_mat = tf.diag(tf.ones(shape=[vocab_length]))
```

```
| address1_embed = tf.nn.embedding_lookup(identity_mat, address1_ph)
| address2_embed = tf.nn.embedding_lookup(identity_mat, address2_ph)
```

13. We will now declare the `model`, `batch_accuracy`, `batch_loss`, and `predictions` operations as follows:

```
# Define Model
text_snn = model.snn(address1_embed, address2_embed, dropout_keep_prob_ph,
                      vocab_length, num_features, max_address_len)
# Define Accuracy
batch_accuracy = model.accuracy(text_snn, y_target_ph)
# Define Loss
batch_loss = model.loss(text_snn, y_target_ph, margin)
# Define Predictions
predictions = model.get_predictions(text_snn)
```

14. Finally, before we can start training, we add optimization and initialization operations to the graph as follows:

```
# Declare optimizer
optimizer = tf.train.AdamOptimizer(0.01)
# Apply gradients
train_op = optimizer.minimize(batch_loss)
# Initialize Variables
init = tf.global_variables_initializer()
sess.run(init)
```

15. We will now iterate through the training generations and keep track of the loss and accuracy:

```
train_loss_vec = []
train_acc_vec = []
for b in range(n_batches):
    # Get a batch of data
    batch_data = get_batch(batch_size)
    # Shuffle data
    np.random.shuffle(batch_data)
    # Parse addresses and targets
    input_addresses = [x[0] for x in batch_data]
    target_similarity = np.array([x[1] for x in batch_data])
    address1 = np.array([address2onehot(x[0]) for x in input_addresses])
    address2 = np.array([address2onehot(x[1]) for x in input_addresses])

    train_feed_dict = {address1_ph: address1,
                       address2_ph: address2,
                       y_target_ph: target_similarity,
                       dropout_keep_prob_ph: dropout_keep_prob}

    _, train_loss, train_acc = sess.run([train_op, batch_loss, batch_accuracy],
                                         feed_dict=train_feed_dict)
    # Save train loss and accuracy
    train_loss_vec.append(train_loss)
    train_acc_vec.append(train_acc)
```

16. After training, we now process the testing queries and references to see how the model can perform:

```
test_queries_ix = np.array([address2onehot(x) for x in test_queries])
test_references_ix = np.array([address2onehot(x) for x in test_references])
num_refs = test_references_ix.shape[0]
best_fit_refs = []
for query in test_queries_ix:
    test_query = np.repeat(np.array([query]), num_refs, axis=0)
    test_feed_dict = {address1_ph: test_query,
                      address2_ph: test_references_ix,
                      y_target_ph: target_similarity,
                      dropout_keep_prob_ph: 1.0}
    test_out = sess.run(text_snn, feed_dict=test_feed_dict)
    best_fit = test_references[np.argmax(test_out)]
    best_fit_refs.append(best_fit)
print('Query Addresses: {}'.format(test_queries))
print('Model Found Matches: {}'.format(best_fit_refs))
```

17. This results in the following output:

```
| Query Addresses: ['111 abbey ln', '271 doner cicle', '314 king avenue', 'tensorf
| Model Found Matches: ['123 abbey ln', '217 donner cir', '314 kings ave', 'tensor
```

There's more...

We can see from the test queries and references that the model was not only able to identify the correct reference addresses, it was also able to generalize to a non-address phrase. We can also see how the model performed by looking at the loss and accuracy during training:

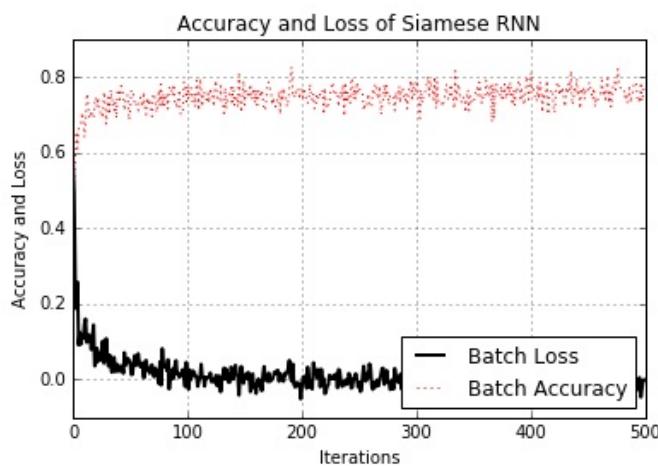


Figure 9: The accuracy and loss for the Siamese RNN similarity model during training

Notice that we did not have a designated test set for this exercise. This is because of how we generated the data. We created a batch function that creates new batch data every time it is called, so the model always sees new data. Because of this, we can use the batch loss and accuracy as proxies to the test loss and accuracy. But, this is never true with a finite set of real data, as we would always have to have training and test sets to judge the performance of the model.

Taking TensorFlow to Production

In this chapter, we will cover the following topics:

- Implementing unit tests
- Using multiple executors
- Parallelizing TensorFlow
- Taking TensorFlow to production
- An example of productionalizing TensorFlow
- Using TensorFlow Serving

Introduction

So far, we have covered how to train and evaluate various models in TensorFlow. So, in this chapter, we will show you how to write code that is ready for production use. There are various definitions of production-ready code, but for us, production code will be defined as code that has unit tests, separates the training and evaluation code, and efficiently saves, and loads the various required parts of the data pipeline and graph session.



The Python scripts provided in this chapter should be run from the command line. This allows tests to be run, and device placements to be logged to the screen.

Implementing unit tests

Testing code results in faster prototyping, more efficient debugging, and faster changing, and makes it easier to share code. There are a number of simple ways to implement unit tests in TensorFlow, and we will cover them in this recipe.

Getting ready

When programming a TensorFlow model, it helps to have unit tests to check the functionality of the program. This helps us because when we want to make changes to a program unit, tests will make sure those changes do not break the model in unknown ways. In this recipe, we will create a simple CNN network that relies on the `MNIST` data. With it, we will implement three different types of unit test to illustrate how to write them in TensorFlow.



Note that Python has a great testing library called `Nose`. TensorFlow also has built-in testing functions, which we will look at, that make it easier to test the value of Tensor objects without having to evaluate the values in a session.

1. First, we need to load the necessary libraries and format the data as follows:

```
import sys
import numpy as np
import tensorflow as tf
from tensorflow.python.framework import ops
ops.reset_default_graph()
# Start a graph session
sess = tf.Session()
# Load data
data_dir = 'temp'
mnist = tf.keras.datasets.mnist
(train_xdata, train_labels), (test_xdata, test_labels) = mnist.load_data()
train_xdata = train_xdata / 255.0
test_xdata = test_xdata / 255.0
# Set model parameters
batch_size = 100
learning_rate = 0.005
evaluation_size = 100
image_width = train_xdata[0].shape[0]
image_height = train_xdata[0].shape[1]
target_size = max(train_labels) + 1
num_channels = 1 # greyscale = 1 channel
generations = 100
eval_every = 5
conv1_features = 25
conv2_features = 50
max_pool_size1 = 2 # NxN window for 1st max pool layer
max_pool_size2 = 2 # NxN window for 2nd max pool layer
fully_connected_size1 = 100
dropout_prob = 0.75
```

2. Then, we need to declare our placeholders, variables, and model formulation, as follows:

```
# Declare model placeholders
```

```

x_input_shape = (batch_size, image_width, image_height, num_channels)
x_input = tf.placeholder(tf.float32, shape=x_input_shape)
y_target = tf.placeholder(tf.int32, shape=(batch_size))
eval_input_shape = (evaluation_size, image_width, image_height, num_channels)
eval_input = tf.placeholder(tf.float32, shape=eval_input_shape)
eval_target = tf.placeholder(tf.int32, shape=(evaluation_size))
dropout = tf.placeholder(tf.float32, shape=())
# Declare model parameters
conv1_weight = tf.Variable(tf.truncated_normal([4, 4, num_channels, conv1_features],
                                              stddev=0.1, dtype=tf.float32))
conv1_bias = tf.Variable(tf.zeros([conv1_features], dtype=tf.float32))
conv2_weight = tf.Variable(tf.truncated_normal([4, 4, conv1_features, conv2_features],
                                              stddev=0.1, dtype=tf.float32))
conv2_bias = tf.Variable(tf.zeros([conv2_features], dtype=tf.float32))
# fully connected variables
resulting_width = image_width // (max_pool_size1 * max_pool_size2)
resulting_height = image_height // (max_pool_size1 * max_pool_size2)
full1_input_size = resulting_width * resulting_height * conv2_features
full1_weight = tf.Variable(tf.truncated_normal([full1_input_size, fully_connected_size1],
                                              stddev=0.1, dtype=tf.float32))
full1_bias = tf.Variable(tf.truncated_normal([fully_connected_size1], stddev=0.1,
                                             dtype=tf.float32))
full2_weight = tf.Variable(tf.truncated_normal([fully_connected_size1, target_size],
                                              stddev=0.1, dtype=tf.float32))
full2_bias = tf.Variable(tf.truncated_normal([target_size], stddev=0.1, dtype=tf.float32))

# Initialize Model Operations
def my_conv_net(input_data):
    # First Conv-ReLU-MaxPool Layer
    conv1 = tf.nn.conv2d(input_data, conv1_weight, strides=[1, 1, 1, 1], padding='SAME')
    relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_bias))
    max_pool1 = tf.nn.max_pool(relu1, ksize=[1, max_pool_size1, max_pool_size1, 1],
                               strides=[1, max_pool_size1, max_pool_size1, 1], padding='SAME')
    # Second Conv-ReLU-MaxPool Layer
    conv2 = tf.nn.conv2d(max_pool1, conv2_weight, strides=[1, 1, 1, 1], padding='SAME')
    relu2 = tf.nn.relu(tf.nn.bias_add(conv2, conv2_bias))
    max_pool2 = tf.nn.max_pool(relu2, ksize=[1, max_pool_size2, max_pool_size2, 1],
                               strides=[1, max_pool_size2, max_pool_size2, 1], padding='SAME')
    # Transform Output into a 1xN layer for next fully connected layer
    final_conv_shape = max_pool2.get_shape().as_list()
    final_shape = final_conv_shape[1] * final_conv_shape[2] * final_conv_shape[3]
    flat_output = tf.reshape(max_pool2, [final_conv_shape[0], final_shape])
    # First Fully Connected Layer
    fully_connected1 = tf.nn.relu(tf.add(tf.matmul(flat_output, full1_weight), full1_bias))
    # Second Fully Connected Layer
    final_model_output = tf.add(tf.matmul(fully_connected1, full2_weight), full2_bias)

    # Add dropout
    final_model_output = tf.nn.dropout(final_model_output, dropout)
    return final_model_output

model_output = my_conv_net(x_input)
test_model_output = my_conv_net(eval_input)

```

3. Next, we create our loss function and our prediction and accuracy operations. Then, we initialize the following model variables:

```

# Declare Loss Function (softmax cross entropy)
loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(model_output,
                                                                    test_target))
# Create a prediction function
prediction = tf.nn.softmax(model_output)
test_prediction = tf.nn.softmax(test_model_output)

```

```

# Create accuracy function
def get_accuracy(logits, targets):
    batch_predictions = np.argmax(logits, axis=1)
    num_correct = np.sum(np.equal(batch_predictions, targets))
    return 100. * num_correct/batch_predictions.shape[0]

# Create an optimizer
my_optimizer = tf.train.MomentumOptimizer(learning_rate, 0.9)
train_step = my_optimizer.minimize(loss)
# Initialize Variables
init = tf.global_variables_initializer()
sess.run(init)

```

- For our first unit test, we use the class `tf.test.TestCase` and create a way to test the values of a placeholder (or variable). For this test case, we make sure that the dropout probability (for keeping) is greater than `0.25`, so that the model is not changed to attempt to train on more than a 75% dropout, shown as follows:

```

# Check values of tensors!
class DropOutTest(tf.test.TestCase):
    # Make sure that we don't drop too much
    def dropout_greaterthan(self):
        with self.test_session():
            self.assertGreater(dropout.eval(), 0.25)

```

- Next, we need to test that our `accuracy` function is behaving as expected. To do this, we create a sample array of probabilities and what we expect, and then make sure that the test accuracy returns 100%, as follows:

```

# Test accuracy function
class AccuracyTest(tf.test.TestCase):
    # Make sure accuracy function behaves correctly
    def accuracy_exact_test(self):
        with self.test_session():
            test_preds = [[0.9, 0.1], [0.01, 0.99]]
            test_targets = [0, 1]
            test_acc = get_accuracy(test_preds, test_targets)
            self.assertEqual(test_acc.eval(), 100.)

```

- We can also make sure that a Tensor object is the shape that we expect. To test that the model output is the expected shape of `batch_size` by `target_size`, input the following code:

```

# Test tensorshape
class ShapeTest(tf.test.TestCase):
    # Make sure our model output is size [batch_size, num_classes]
    def output_shape_test(self):
        with self.test_session():
            numpy_array = np.ones([batch_size, target_size])
            self.assertShapeEqual(numpy_array, model_output)

```

- Now we need a `main()` function in our script to tell TensorFlow which

application we are running. The script is as follows:

```
def main(argv):
    # Start training loop
    train_loss = []
    train_acc = []
    test_acc = []
    for i in range(generations):
        rand_index = np.random.choice(len(train_xdata), size=batch_size)
        rand_x = train_xdata[rand_index]
        rand_x = np.expand_dims(rand_x, 3)
        rand_y = train_labels[rand_index]
        train_dict = {x_input: rand_x, y_target: rand_y, dropout: dropout_prob}

        sess.run(train_step, feed_dict=train_dict)
        temp_train_loss, temp_train_preds = sess.run([loss, prediction], feed_di
temp_train_acc = get_accuracy(temp_train_preds, rand_y)

        if (i + 1) % eval_every == 0:
            eval_index = np.random.choice(len(test_xdata), size=evaluation_size)
            eval_x = test_xdata[eval_index]
            eval_x = np.expand_dims(eval_x, 3)
            eval_y = test_labels[eval_index]
            test_dict = {eval_input: eval_x, eval_target: eval_y, dropout: 1.0}
            test_preds = sess.run(test_prediction, feed_dict=test_dict)
            temp_test_acc = get_accuracy(test_preds, eval_y)

            # Record and print results
            train_loss.append(temp_train_loss)
            train_acc.append(temp_train_acc)
            test_acc.append(temp_test_acc)
            acc_and_loss = [(i + 1), temp_train_loss, temp_train_acc, temp_test_
acc_and_loss = [np.round(x, 2) for x in acc_and_loss]
            print('Generation # {}. Train Loss: {:.2f}. Train Acc (Test Acc): {:.2f}'.format(*acc_and_loss))
```

8. To have our script perform either the testing or the training, we need to call it from the command line in a different way. The following snippet is the main program code. If the program receives the argument `test`, it will perform the testing; otherwise, it will run the training:

```
if __name__ == '__main__':
    cmd_args = sys.argv
    if len(cmd_args) > 1 and cmd_args[1] == 'test':
        # Perform unit-tests
        tf.test.main(argv=cmd_args[1:])
    else:
        # Run the TensorFlow app
        tf.app.run(main=None, argv=cmd_args)
```

9. If we run the program on the command line, we should get the following output:

```
$ python3 implementing_unit_tests.py test
...
-----
Ran 3 tests in 0.001s
```

| **OK**

The full program as described in the preceding steps can be found in the book's GitHub repository at https://github.com/nfmccclure/tensorflow_cookbook/ and at the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

How it works...

In this section, we implemented three types of unit tests: Tensor values, operations outputs, and Tensor shapes. There are more types of unit test functions for TensorFlow, and they can be found here: https://www.tensorflow.org/versions/master/api_docs/python/test.html.

Remember that unit testing helps assure us that code will function as expected, provides confidence in sharing code, and makes reproducibility more accessible.

Using multiple executors

You will be aware that there are many features of TensorFlow, including computational graphs that lend themselves naturally to being computed in parallel. Computational graphs can be split over different processors as well as in processing different batches. We will address how to access different processors on the same machine in this recipe.

Getting ready

For this recipe, we will show you how to access multiple devices on the same system and train on them. This is a very common occurrence: along with a CPU, a machine may have one or more GPUs that can share the computational load. If TensorFlow can access these devices, it will automatically distribute the computations to multiple devices via a greedy process. However, TensorFlow also allows the program to specify which operations will be on which device via a name scope placement.

In order to access GPU devices, the GPU version of TensorFlow must be installed. To install the GPU version of TensorFlow, visit https://www.tensorflow.org/versions/master/get_started/os_setup.html. Download it, set it up, and follow the instructions for your specific system. Be aware that the GPU versions of TensorFlow require CUDA to use the GPU.

In this recipe, we will show you various commands that will allow you to access various devices on your system; we'll also demonstrate how to find out which devices TensorFlow is using.

How to do it...

1. In order to find out which devices TensorFlow is using for which operations, we need to set a `config` in the session parameters that sets `log_device_placement` to `True`. When we run the script from the command line, we will see specific device placements, as shown in the following output:

```
import tensorflow as tf
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
c = tf.matmul(a, b)
# Runs the op.
print(sess.run(c))
```

2. From the terminal, run the following command:

```
$python3 using_multiple_devices.py
Device mapping: no known devices.
I tensorflow/core/common_runtime/direct_session.cc:175] Device mapping:
MatMul: /job:localhost/replica:0/task:0/cpu:0
I tensorflow/core/common_runtime/simple_placer.cc:818] MatMul: /job:localhost/re
b: /job:localhost/replica:0/task:0/cpu:0
I tensorflow/core/common_runtime/simple_placer.cc:818] b: /job:localhost/replica
a: /job:localhost/replica:0/task:0/cpu:0
I tensorflow/core/common_runtime/simple_placer.cc:818] a: /job:localhost/replica
[[ 22.  28.]
 [ 49.  64.]]
```

3. By default, TensorFlow automatically decides how to distribute computations across compute devices (CPUs and GPUs) and sometimes we need to understand what those placements are. This can be useful when loading an earlier, existing model that had hard placements assigned in the graph while our machine has different devices. We can set soft placements in the configuration to address this, as follows:

```
config = tf.ConfigProto()
config.allow_soft_placement = True
sess_soft = tf.Session(config=config)
```

4. When using GPUs, TensorFlow automatically takes up a large portion of the GPU memory. While this is usually desired, we can take steps to be more careful with GPU memory allocation. While TensorFlow never releases GPU memory, we can slowly grow its allocation to the maximum limit (only when needed) by setting a GPU memory growth option as

follows:

```
| config.gpu_options.allow_growth = True  
| sess_grow = tf.Session(config=config)
```

5. If we want to put a hard limit on the percentage of GPU memory used by TensorFlow, we can use the `config` setting `per_process_gpu_memory_fraction` as follows:

```
| config.gpu_options.per_process_gpu_memory_fraction = 0.4  
| sess_limited = tf.Session(config=config)
```

6. Sometimes we may need to write robust code that can determine whether it is running with the GPU available or not. TensorFlow has a built-in function that can test if the GPU is available. This is helpful for when we want to write code that takes advantage of the GPU when it is available and assign specific operations to it. This is done with the following code:

```
| if tf.test.is_built_with_cuda():  
|     <Run GPU specific code here>
```

7. If we need to assign specific operations, say to the GPU, input the following code. This will perform simple calculations and assign operations to the main CPU and the two auxiliary GPUs:

```
with tf.device('/cpu:0'):  
    a = tf.constant([1.0, 3.0, 5.0], shape=[1, 3])  
    b = tf.constant([2.0, 4.0, 6.0], shape=[3, 1])  
  
    with tf.device('/gpu:0'):  
        c = tf.matmul(a,b)  
        c = tf.reshape(c, [-1])  
  
    with tf.device('/gpu:1'):  
        d = tf.matmul(b,a)  
        flat_d = tf.reshape(d, [-1])  
  
    combined = tf.multiply(c, flat_d)  
print(sess.run(combined))
```

How it works...

When we want to specify specific devices on our machine for TensorFlow operations, we need to know how TensorFlow refers to such devices. Device names in TensorFlow follow the following conventions:

Device	Device name
Main CPU	/cpu:0
Second CPU	/cpu:1
Main GPU	/gpu:0
Second GPU	/gpu:1
Third GPU	/gpu:2

There's more...

Fortunately, running TensorFlow in the cloud is now easier than ever. Many cloud computation service providers offer GPU instances that have a main CPU and a powerful GPU alongside it. **Amazon Web Services (AWS)** has G instances and P2 instances that allow the usage of powerful GPUs that offer great speed for TensorFlow processes. There are even **AWS Machine Images (AMIs)** that you can select for free that will boot up a selected instance with the GPU instance of TensorFlow installed.

Parallelizing TensorFlow

To extend our reach for parallelizing TensorFlow, we can also perform separate operations from our graph on entirely different machines in a distributed manner. This recipe will show you how.

Getting ready

A few months after the release of TensorFlow, Google released Distributed TensorFlow, which was a big upgrade to the TensorFlow ecosystem, and one that allowed a TensorFlow cluster to be set up (on separate worker machines) and share the computational task of training and evaluating models. Using Distributed TensorFlow is as easy as setting up parameters for workers and then assigning different jobs to different workers.

In this recipe, we will set up two local workers and assign them to different jobs.

How to do it...

1. To start, we load TensorFlow and define our two local workers with a configuration dictionary file (ports 2222 and 2223) as follows:

```
import tensorflow as tf
# Cluster for 2 local workers (tasks 0 and 1):
cluster = tf.train.ClusterSpec({'local': ['localhost:2222', 'localhost:2223']})
```

2. Now, we join the two workers into the server and label them with the following task numbers:

```
server = tf.train.Server(cluster, job_name="local", task_index=0)
server = tf.train.Server(cluster, job_name="local", task_index=1)
```

3. Now we will have each worker complete a task. The first worker will initialize two matrices (each one will be 25 by 25). The second worker will find the sum of all of the elements. Then, we will auto-assign the sum of the two sums and print the output, as follows:

```
mat_dim = 25
matrix_list = {}
with tf.device('/job:local/task:0'):
    for i in range(0, 2):
        m_label = 'm_{}'.format(i)
        matrix_list[m_label] = tf.random_normal([mat_dim, mat_dim])
# Have each worker calculate the sums
sum_outs = {}
with tf.device('/job:local/task:1'):
    for i in range(0, 2):
        A = matrix_list['m_{}'.format(i)]
        sum_outs['m_{}'.format(i)] = tf.reduce_sum(A)
    # Sum all the sums
    summed_out = tf.add_n(list(sum_outs.values()))
with tf.Session(server.target) as sess:
    result = sess.run(summed_out)
    print('Summed Values:{}\n'.format(result))
```

4. After inputting the preceding code, we can run the following on the command prompt:

```
$ python3 parallelizing_tensorflow.py
I tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:197] Initialize GrpcCh
I tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:206] Started server
I tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:197] Initialize GrpcCh
I tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:206] Started server
I tensorflow/core/distributed_runtime/master_session.cc:928] Start master sessio
Summed Values:-21.12611198425293
```

How it works...

Using Distributed TensorFlow is quite easy. All you have to do is just assign worker IPs to the server with names. Then, operations can be manually or automatically assigned to workers.

Taking TensorFlow to production

If we want to use our machine learning scripts in a production setting, there are some points we first need to consider as a best practice. In this section, we will outline some of them.

Getting ready

In this recipe, we want to summarize and condense various tips for bringing TensorFlow to production. We will cover how best to save and load vocabularies, graphs, variables, and model checkpoints. We will also talk about how to use TensorFlow's command-line argument parser and change TensorFlow's logging verbosity.

How to do it...

- When running a TensorFlow program, we may want to check that no other graph session is already in memory, or that the graph session is cleared after debugging a program. We can accomplish this by using the following command line:

```
|     from tensorflow.python.framework import ops  
|     ops.reset_default_graph()
```

- When dealing with text (or any data pipeline), we need to be sure that we save how we process the data, so that we can process future evaluation data the same way. If we are dealing with text, for example, we need to be sure we can save and load the vocabulary dictionary. The following code is an example of how to save the vocabulary dictionary with the `JSON` library:

```
|     import json  
|     word_list = ['to', 'be', 'or', 'not', 'to', 'be']  
|     vocab_list = list(set(word_list))  
|     vocab2ix_dict = dict(zip(vocab_list, range(len(vocab_list))))  
|     ix2vocab_dict = {val:key for key,val in vocab2ix_dict.items()}  
  
|     # Save vocabulary  
|     import json  
|     with open('vocab2ix_dict.json', 'w') as file_conn:  
|         json.dump(vocab2ix_dict, file_conn)  
  
|     # Load vocabulary  
|     with open('vocab2ix_dict.json', 'r') as file_conn:  
|         vocab2ix_dict = json.load(file_conn)
```



Here, we saved the vocabulary dictionary in `JSON` format, but we can also save it in a `text` file, `csv`, or even a `binary` format. If the vocabulary is large, a `binary` file is preferred. You can also look into using the `pickle` library to create a `pkl` binary file, but be aware that `pickle` files do not translate well between library and Python versions.

- To save the model graph and variables, we create a `saver()` operation and add that to the graph. It is recommended that we save the model on a regular basis during training. To save a model, input the following code:

```
|     After model declaration, add a saving operations  
|     saver = tf.train.Saver()  
|     # Then during training, save every so often, referencing the training generation  
|     for i in range(generations):  
|         ...  
|         if i%save_every == 0:  
|             saver.save(sess, 'my_model', global_step=step)  
|     # Can also save only specific variables:  
|     saver = tf.train.Saver({"my_var": my_variable})
```

 Note that the `saver()` operation also takes other parameters. As shown in the preceding example, it can take a dictionary of variables and tensors to save specific elements. It can also take a checkpoint every `n` hours, which performs the saving operation regularly. By default, the saving operation only keeps the last five model saves (for space considerations). This can be changed with the `maximum_to_keep` option.

4. Before saving a model, be sure to name the important operations of the model. TensorFlow does not have an easy way to load specific placeholders, operations, or variables if they do not have a name. Most operations and functions in TensorFlow accept a `name` argument, such as in the following example:

```
conv_weights = tf.Variable(tf.random_normal(), name='conv_weights')
loss = tf.reduce_mean(..., name='loss')
```

5. TensorFlow also makes it easy to perform arg-parsing on the command line with the `tf.app.flags` library. With these functions, we can define command-line arguments that are strings, floats, integers, or Booleans, as shown in the following snippet. With these flag definitions, we can just run `tf.app.run()`, which will run the `main()` function with the following flag arguments:

```
tf.flags.DEFINE_string("worker_locations", "", "List of worker addresses.")
tf.flags.DEFINE_float('learning_rate', 0.01, 'Initial learning rate.')
tf.flags.DEFINE_integer('generations', 1000, 'Number of training generations.')
tf.flags.DEFINE_boolean('run_unit_tests', False, 'If true, run tests.')
FLAGS = tf.flags.FLAGS
# Need to define a 'main' function for the app to run
def main(_):
    worker_ips = FLAGS.worker_locations.split(",")
    learning_rate = FLAGS.learning_rate
    generations = FLAGS.generations
    run_unit_tests = FLAGS.run_unit_tests

    # Run the Tensorflow app
if __name__ == "__main__":
    # The following is looking for a "main()" function to run and will pass.
    tf.app.run()
    # Can modify this to be more custom:
    tf.app.run(main=my_main_function(), argv=my_arguments)
```

6. TensorFlow has built-in logging that we can set the level parameter for. The levels we can set are `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL`. The default is `WARN`, as follows:

```
tf.logging.set_verbosity(tf.logging.WARN)
# WARN is the default value, but to see more information, you can set it to
#     INFO or DEBUG
tf.logging.set_verbosity(tf.logging.DEBUG)
```

How it works...

In this section, we provided tips for creating production-level code in TensorFlow. We wanted to introduce concepts such as app-flags, model saving, and logging so that users can consistently write code with these tools and understand what it means when they see these tools in other code. There are many other ways to write good production code, but a full example will be shown in the following recipe.

An example of productionalizing TensorFlow

A good practice for productionalizing machine learning models is to separate the training and evaluation programs. In this section, we will illustrate an evaluation script that has been expanded to include a unit test, model saving and loading, and evaluation.

Getting ready

In this recipe, we will show you how to implement an evaluation script using the afore-mentioned criteria. The code actually consists of a training script and an evaluation script, but for this recipe we will only show you the evaluation script. As a reminder, both scripts can be seen in the online GitHub repository at https://github.com/nfmccclure/tensorflow_cookbook/ and at the official Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

For the upcoming example, we will implement the first RNN example from [Chapter 9, Recurrent Neural Networks](#), which attempts to predict whether a text message is spam or ham. We will assume the RNN model is trained and saved, along with the vocabulary.

How to do it...

1. First, we start by loading the necessary libraries and declaring the TensorFlow application flags as follows:

```
import os
import re
import numpy as np
import tensorflow as tf
from tensorflow.python.framework import ops
ops.reset_default_graph()
# Define App Flags
tf.flags.DEFINE_string("storage_folder", "temp", "Where to store model and data.")
tf.flags.DEFINE_float('learning_rate', 0.0005, 'Initial learning rate.')
tf.flags.DEFINE_float('dropout_prob', 0.5, 'Per to keep probability for dropout.')
tf.flags.DEFINE_integer('epochs', 20, 'Number of epochs for training.')
tf.flags.DEFINE_integer('batch_size', 250, 'Batch Size for training.')
tf.flags.DEFINE_integer('rnn_size', 15, 'RNN feature size.')
tf.flags.DEFINE_integer('embedding_size', 25, 'Word embedding size.')
tf.flags.DEFINE_integer('min_word_frequency', 20, 'Word frequency cutoff.')
tf.flags.DEFINE_boolean('run_unit_tests', False, 'If true, run tests.')

FLAGS = tf.flags.FLAGS
```

2. Next, we declare a text cleaning function. This will be the same cleaning function used in the training script and is as follows:

```
def clean_text(text_string):
    text_string = re.sub(r'([^\w\s]|_|[0-9])+', '', text_string)
    text_string = " ".join(text_string.split())
    text_string = text_string.lower()
    return text_string
```

3. Now, we need to load the following vocabulary processing function:

```
def load_vocab():
    vocab_path = os.path.join(FLAGS.storage_folder, "vocab")
    vocab_processor = tf.contrib.learn.preprocessing.VocabularyProcessor.restore
    return vocab_processor
```

4. Now that we have a way to clean the text, and also have a vocab processor, we can combine these functions to create a data-processing pipeline for a given text, shown as follows:

```
def process_data(input_data, vocab_processor):
    input_data = clean_text(input_data)
    input_data = input_data.split()
    processed_input = np.array(list(vocab_processor.transform(input_data)))
    return processed_input
```

5. Next, we need a way to get data to evaluate. For this purpose, we will ask the user to type in text on the screen. We will then process the text and return the following processed text:

```
def get_input_data():
    input_text = input("Please enter a text message to evaluate: ")
    vocab_processor = load_vocab()
    return process_data(input_text, vocab_processor)
```



For this example, we have created evaluation data by asking the user to type it in. While many applications will get data via a supplied file or API request, we can change this input data function accordingly.

6. For a unit test, we need to make sure that our text cleaning function behaves properly with the following code:

```
class clean_test(tf.test.TestCase):
    # Make sure cleaning function behaves correctly
    def clean_string_test(self):
        with self.test_session():
            test_input = '--Tensorflow's so Great! Dont you think so?      '
            test_expected = 'tensorflows so great don you think so'
            test_out = clean_text(test_input)
            self.assertEqual(test_expected, test_out)
```

7. Now that we have our model and data, we can run the `main` function. The `main` function will get the data, set up the graph, load the variables, feed in the processed data, and print the output, as shown in the following snippet:

```
def main(args):
    # Get flags
    storage_folder = FLAGS.storage_folder
    # Get user input text
    x_data = get_input_data()

    # Load model
    graph = tf.Graph()
    with graph.as_default():
        sess = tf.Session()
        with sess.as_default():
            # Load the saved meta graph and restore variables
            saver = tf.train.import_meta_graph("{}{}.meta".format(os.path.join(sto
            saver.restore(sess, os.path.join(storage_folder, "model.ckpt")))
            # Get the placeholders from the graph by name
            x_data_ph = graph.get_operation_by_name("x_data_ph").outputs[0]
            dropout_keep_prob = graph.get_operation_by_name("dropout_keep_prob")
            probability_outputs = graph.get_operation_by_name("probability_outpu
            # Make the prediction
            eval_feed_dict = {x_data_ph: x_data, dropout_keep_prob: 1.0}
            probability_prediction = sess.run(tf.reduce_mean(probability_outputs

            # Print output (Or save to file or DB connection?)
            print('Probability of Spam: {:.4}'.format(probability_prediction[1]))
```

8. Finally, to run the `main()` function or unit test, use the following code:

```
| if __name__ == "__main__":
|     if FLAGS.run_unit_tests:
|         # Perform unit tests
|         tf.test.main()
|     else:
|         # Run evaluation
|         tf.app.run()
```

How it works...

To evaluate the model, we were able to load command-line arguments with TensorFlow's app flags, load the model and vocabulary processor, and then run the processed data through the model and make a prediction.

Remember to run this script through the command line, and check that the training script is run before creating the model and vocabulary dictionary.

Using TensorFlow Serving

In this section, we will show you how to set up your RNN model to predict spam or ham text messages on TensorFlow. We will first illustrate how to save a model in a protobuf format, and will then load the model into a local server, listening on port 9000 for input.

Getting ready

We start this section by encouraging reader to read through the official documentation and the short tutorials on the TensorFlow Serving site available at https://www.tensorflow.org/serving/serving_basic.

For this example, we will reuse most of the RNN code we used in the on *Predicting Spam with RNNs* recipe in [chapter 9, Recurrent Neural Networks](#). We will alter our model saving code to save a protobuf model in the correct folder structure that is necessary to use TensorFlow Serving.



Note that all scripts in this chapter should be executed from the command line bash prompt.

For the updated installation instructions, visit the official installation site at: <https://www.tensorflow.org/serving/setup>. Normal installation is as simple as adding a gpg-key to your Linux sources and running the following installation command:

```
| $ sudo apt install tensorflow-model-server
```

How to do it...

1. Here, we will start in the same way as before, by loading the necessary libraries and setting the TensorFlow flags as follows:

```
import os
import re
import io
import sys
import requests
import numpy as np
import tensorflow as tf
from zipfile import ZipFile
from tensorflow.python.framework import ops

ops.reset_default_graph()

# Define App Flags
tf.flags.DEFINE_string("storage_folder", "temp", "Where to store model and data.")
tf.flags.DEFINE_float('learning_rate', 0.0005, 'Initial learning rate.')
tf.flags.DEFINE_float('dropout_prob', 0.5, 'Per to keep probability for dropout.')
tf.flags.DEFINE_integer('epochs', 20, 'Number of epochs for training.')
tf.flags.DEFINE_integer('batch_size', 250, 'Batch Size for training.')
tf.flags.DEFINE_integer('rnn_size', 15, 'RNN feature size.')
tf.flags.DEFINE_integer('embedding_size', 25, 'Word embedding size.')
tf.flags.DEFINE_integer('min_word_frequency', 20, 'Word frequency cutoff.')
tf.flags.DEFINE_boolean('run_unit_tests', False, 'If true, run tests.')

FLAGS = tf.flags.FLAGS
```

2. We will continue through the script in the exact same way. For brevity, we will only include the differences in the training script, which is how we save the protobuf model. This is done by inserting the following code after training is complete:



Note how similar this code is to the tutorial code. The major differences here are in the model names, version numbers, and the fact that we are saving an RNN instead of a CNN.

```
# Save the finished model for TensorFlow Serving (pb file)
# Here, it's our storage folder / version number
out_path = os.path.join(tf.compat.as_bytes(os.path.join(storage_folder, '1')))
print('Exporting finished model to : {}'.format(out_path))
builder = tf.saved_model.builder.SavedModelBuilder(out_path)

# Build the signature_def_map.
classification_inputs = tf.saved_model.utils.build_tensor_info(x_data_ph)
classification_outputs_classes = tf.saved_model.utils.build_tensor_info(rnn_mode

classification_signature = (tf.saved_model.signature_def_utils.build_signature_d
    inputs={tf.saved_model.signature_constants.CLASSIFY_INPUTS:
        classification_inputs},
    outputs={tf.saved_model.signature_constants.CLASSIFY_OUTPUT_CLAS
```

```

        classification_outputs_classes},
method_name=tf.saved_model.signature_constants.CLASSIFY_METHOD_N

tensor_info_x = tf.saved_model.utils.build_tensor_info(x_data_ph)
tensor_info_y = tf.saved_model.utils.build_tensor_info(y_output_ph)

prediction_signature = (
    tf.saved_model.signature_def_utils.build_signature_def(
        inputs={'texts': tensor_info_x},
        outputs={'scores': tensor_info_y},
        method_name=tf.saved_model.signature_constants.PREDICT_METHOD_NA

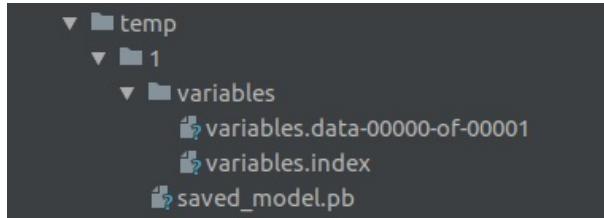
    legacy_init_op = tf.group(tf.tables_initializer(), name='legacy_init_op')
builder.add_meta_graph_and_variables(
    sess, [tf.saved_model.tag_constants.SERVING],
    signature_def_map={
        'predict_spam': prediction_signature,
        tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF
            classification_signature,
    },
    legacy_init_op=legacy_init_op)

builder.save()

print('Done exporting!')

```

3. It is important for us to realize that TensorFlow Serving expects a specific file or folder structure for loading models. The script will install files in the following format:



A screenshot of the directory structure that TensorFlow Serving expects.

The preceding screenshot shows the desired directory structure. In it, we have our defined data directory, `temp`, followed by our model-version number, `1`. In the version number directory, we save our protobuf model and a `variables` folder that contains the desired variables to save.



We should be aware that inside our data directory, TensorFlow Serving will look for integer folders. TensorFlow Serving will automatically boot up and grab the model under the largest integer number. This means that to deploy a new model, we need to label it version 2, and stick it under a new folder that is also labeled 2. TensorFlow Serving will then automatically pick up the model.

4. To start our server, we call the command `tensorflow_model_server` with a port, `model_name`, and `model_base_path` arguments. Then, TensorFlow Serving looks for version number folders and picks up the largest version numbered

model. It then deploys it to the machine the command was run on through the port given as an argument. In the following example, we are running on a local machine (`0.0.0.0`) and the accepted default port is `9000`:

```
$ tensorflow_model_server --port=9000 --model_name=spam_ham --model_base_path=<d  
2018-08-09 12:05:16.206712: I tensorflow_serving/model_servers/main.cc:153] Buil  
2018-08-09 12:05:16.206874: I tensorflow_serving/model_servers/server_core.cc:45  
2018-08-09 12:05:16.206903: I tensorflow_serving/model_servers/server_core.cc:51  
2018-08-09 12:05:16.307681: I tensorflow_serving/core/basic_manager.cc:716] Succ  
2018-08-09 12:05:16.307744: I tensorflow_serving/core/loader_harness.cc:66] Appr  
2018-08-09 12:05:16.307773: I tensorflow_serving/core/loader_harness.cc:74] Load  
2018-08-09 12:05:16.307829: I external/org_tensorflow/tensorflow/contrib/session  
2018-08-09 12:05:16.307867: I external/org_tensorflow/tensorflow/cc/saved_model/  
2018-08-09 12:05:16.313811: I external/org_tensorflow/tensorflow/core/platform/c  
2018-08-09 12:05:16.325866: I external/org_tensorflow/tensorflow/cc/saved_model/  
2018-08-09 12:05:16.329290: I external/org_tensorflow/tensorflow/cc/saved_model/  
2018-08-09 12:05:16.332936: I external/org_tensorflow/tensorflow/cc/saved_model/  
2018-08-09 12:05:16.332972: I tensorflow_serving/servables/tensorflow/saved_mode  
2018-08-09 12:05:16.333335: I tensorflow_serving/core/loader_harness.cc:86] Succ  
2018-08-09 12:05:16.334678: I tensorflow_serving/model_servers/main.cc:323] Runn
```

5. We can now submit binary data to the `<host>:9000` and get back the JSON response showing the results. We can do this via any machine and with any programming language. It is very useful to not have to rely on the client to have a local copy of TensorFlow.

How it works...

If we compare the earlier productionalizing section with the immediately preceding section, the major difference is that we have a model server deployed on a host machine that can respond to incoming requests. The prior section is an example of a great setup for performing batch results or working on a machine that can load TensorFlow, but the recipe is not very good at deploying a model that is available, can do the calculations, and return the results to any client. In this section, we will look at how to address that kind of architecture, which is summarized in the following table:

	Section 5 - Production in Batch	Section 6 - Production via Model Server
Pros	Does not rely on network connections or a host computer	Results are agnostic to client, can accept correctly-formatted binary data
Cons	Clients must have TensorFlow and model files	Relies on host computer to run the code
Ideal Use	Large batches of data	Production service always running

Of course, the advantages and disadvantages of each are debatable, and both can satisfy the requirements of each case. There are also many other architectures available that may be used that can satisfy different requirements, such as Docker, Kubernetes, Luigi, Django/Flask, Celery, AWS, and Azure.

There's more...

Links to tools and resources for architectures not covered in this chapter are as follows:

- **Using TensorFlow Serving in Docker:** <https://www.tensorflow.org/serving/docker>
- **Using TensorFlow Serving in Kubernetes:** https://www.tensorflow.org/serving/serving_inception
- **Luigi, a pipeline tool for batch jobs:** <https://github.com/spotify/luigi>
- **Using TensorFlow in Flask:** <https://guillaumegenthial.github.io/serving.html>
- **A Python framework for distributed task-queuing:** <http://www.celeryproject.org/community/>
- **How to use AWS lambdas with TensorFlow models:** <https://aws.amazon.com/blogs/machine-learning/how-to-deploy-deep-learning-models-with-aws-lambda-and-tensorflow/>

More with TensorFlow

In this chapter, we'll cover the following recipes:

- Visualizing graphs in TensorBoard
- Working with a genetic algorithm
- Clustering using k-means
- Solving a system of ordinary differential equations
- Using random forests
- Using TensorFlow with Keras

All the code that appears in this chapter is available online at https://github.com/nfmccclure/tensorflow_cookbook and at the Packt repository: <https://github.com/PacktPublishing/TensorFlow-Machine-Learning-Cookbook-Second-Edition>.

Introduction

Throughout this book, we have seen that TensorFlow is capable of implementing many models, but there is more that TensorFlow can do. This chapter will show you a few of those things. We'll start by showing how to use the various aspects of TensorBoard, a capability that comes with TensorFlow that allows us to visualize summary metrics, graphs, and images even while our model is training. The remaining recipes in the chapter will show how to use TensorFlow's `group()` function to do step-wise updates. This function will allow us to implement a genetic algorithm, perform k-means clustering, solve a system of ODEs, and even create a gradient boosted random forest.

Visualizing graphs in TensorBoard

Monitoring and troubleshooting machine learning algorithms can be a daunting task, especially if you have to wait a long time for the training to complete before you know the results. To work around this, TensorFlow includes a computational graph visualization tool called **TensorBoard**. With TensorBoard, we can visualize and graph's important values (loss, accuracy, batch training time, and so on) even during training.

Getting ready

To illustrate the various ways we can use TensorBoard, we will reimplement the linear regression model from *The TensorFlow way of linear regression* recipe in [Chapter 3, Linear Regression](#). We'll generate linear data with errors, and use TensorFlow loss and back-propagation to fit a line to the data. We will show how to monitor numerical values, histograms of sets of values, and how to create an image in TensorBoard.

How to do it...

1. First, we'll load the libraries necessary for the script:

```
import os
import io
import time
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

2. We'll now initialize a session and create a summary writer that can write TensorBoard summaries to a `tensorboard` folder:

```
sess = tf.Session()
# Create a visualizer object
summary_writer = tf.summary.FileWriter('tensorboard', sess.graph)
```

3. We need to make sure that the `tensorboard` folder exists for the summary writer to write the `tensorboard` logs:

```
if not os.path.exists('tensorboard'):
    os.makedirs('tensorboard')
```

4. We'll now set the model parameters and generate the linear data for the model. Note that, because of the way we generated the data, our true slope value is 2. We will visualize the changing slope over time and see it approach the true value:

```
batch_size = 50
generations = 100
# Create sample input data
x_data = np.arange(1000)/10.
true_slope = 2.
y_data = x_data * true_slope + np.random.normal(loc=0.0, scale=25, size=1000)
```

5. Next, we'll split the dataset into a train and test set:

```
train_ix = np.random.choice(len(x_data), size=int(len(x_data)*0.9), replace=False)
test_ix = np.setdiffid(np.arange(1000), train_ix)
x_data_train, y_data_train = x_data[train_ix], y_data[train_ix]
x_data_test, y_data_test = x_data[test_ix], y_data[test_ix]
```

6. Now we can create the placeholders, variables, model operations, loss, and optimizing operations:

```

x_graph_input = tf.placeholder(tf.float32, [None])
y_graph_input = tf.placeholder(tf.float32, [None])
# Declare model variables
m = tf.Variable(tf.random_normal([1], dtype=tf.float32), name='Slope')
# Declare model
output = tf.multiply(m, x_graph_input, name='Batch_Multiplication')
# Declare loss function (L1)
residuals = output - y_graph_input
l1_loss = tf.reduce_mean(tf.abs(residuals), name="L1_Loss")
# Declare optimization function
my_optim = tf.train.GradientDescentOptimizer(0.01)
train_step = my_optim.minimize(l1_loss)

```

7. We can now create a TensorBoard operation that will summarize a scalar value. The scalar value that we will summarize is the slope estimate of the model:

```

with tf.name_scope('Slope_Estimate'):
    tf.summary.scalar('Slope_Estimate', tf.squeeze(m))

```

8. Another summary we can add to TensorBoard is a histogram summary, which inputs multiple values in a tensor and outputs graphs and histograms:

```

with tf.name_scope('Loss_and_Residuals'):
    tf.summary.histogram('Histogram_Errors', tf.squeeze(l1_loss))
    tf.summary.histogram('Histogram_Residuals', tf.squeeze(residuals))

```

9. After creating these summary operations, we need to create a summary merging operation that will combine all the summaries. We can then initialize the model variables:

```

summary_op = tf.summary.merge_all()
# Initialize Variables
init = tf.global_variables_initializer()
sess.run(init)

```

10. Now, we can train the linear model and write summaries every generation:

```

for i in range(generations):
    batch_indices = np.random.choice(len(x_data_train), size=batch_size)
    x_batch = x_data_train[batch_indices]
    y_batch = y_data_train[batch_indices]
    _, train_loss, summary = sess.run([train_step, l1_loss, summary_op],
                                      feed_dict={x_graph_input: x_batch,
                                                 y_graph_input: y_batch})

    test_loss, test_resids = sess.run([l2_loss, residuals], feed_dict={x_graph_i

if (i+1)%10==0:
    print('Generation {} of {}. Train Loss: {:.3}, Test Loss: {:.3}'.format
log_writer = tf.train.SummaryWriter('tensorboard')
log_writer.add_summary(summary, i)

```

- In order to put the final graph of the linear fit with the data points in TensorBoard, we have to create the graph image in `protobuf` format. To start this, we will create a function that outputs a `protobuf` graph:

```
def gen_linear_plot(slope):
    linear_prediction = x_data * slope
    plt.plot(x_data, y_data, 'b.', label='data')
    plt.plot(x_data, linear_prediction, 'r-', linewidth=3, label='predicted line')
    plt.legend(loc='upper left')
    buf = io.BytesIO()
    plt.savefig(buf, format='png')
    buf.seek(0)
    return(buf)
```

- Now, we can create and add the `protobuf` image to TensorBoard:

```
# Get slope value
slope = sess.run(m)

# Generate the linear plot in buffer
plot_buf = gen_linear_plot(slope[0])

# Convert PNG buffer to TF image
image = tf.image.decode_png(plot_buf.getvalue(), channels=4)

# Add the batch dimension
image = tf.expand_dims(image, 0)

# Add image summary
image_summary_op = tf.summary.image("Linear_Plot", image)
image_summary = sess.run(image_summary_op)
log_writer.add_summary(image_summary, i)
log_writer.close()
```

 **TIP** Be careful writing image summaries too often to TensorBoard. For example, if we were to write an image summary every generation for 10,000 generations, that would generate 10,000 images worth of summary data. This tends to eat up disk space very quickly.

There's more...

- As we want to run the described python script from the command line, we open a command prompt and run the following command:

```
$ python3 using_tensorboard.py  
Run the command: $tensorboard --logdir="tensorboard" Then navigate to http://1  
Generation 10 of 100. Train Loss: 20.4, Test Loss: 20.5.  
Generation 20 of 100. Train Loss: 17.6, Test Loss: 20.5.  
Generation 90 of 100. Train Loss: 20.1, Test Loss: 20.5.  
Generation 100 of 100. Train Loss: 19.4, Test Loss: 20.5.
```

- We'll then run the preceding specified command to start tensorboard:

```
$ tensorboard --logdir="tensorboard" Starting tensorboard b'29' on port 6006 (Yo
```

Here is a sample of what we can see in TensorBoard:

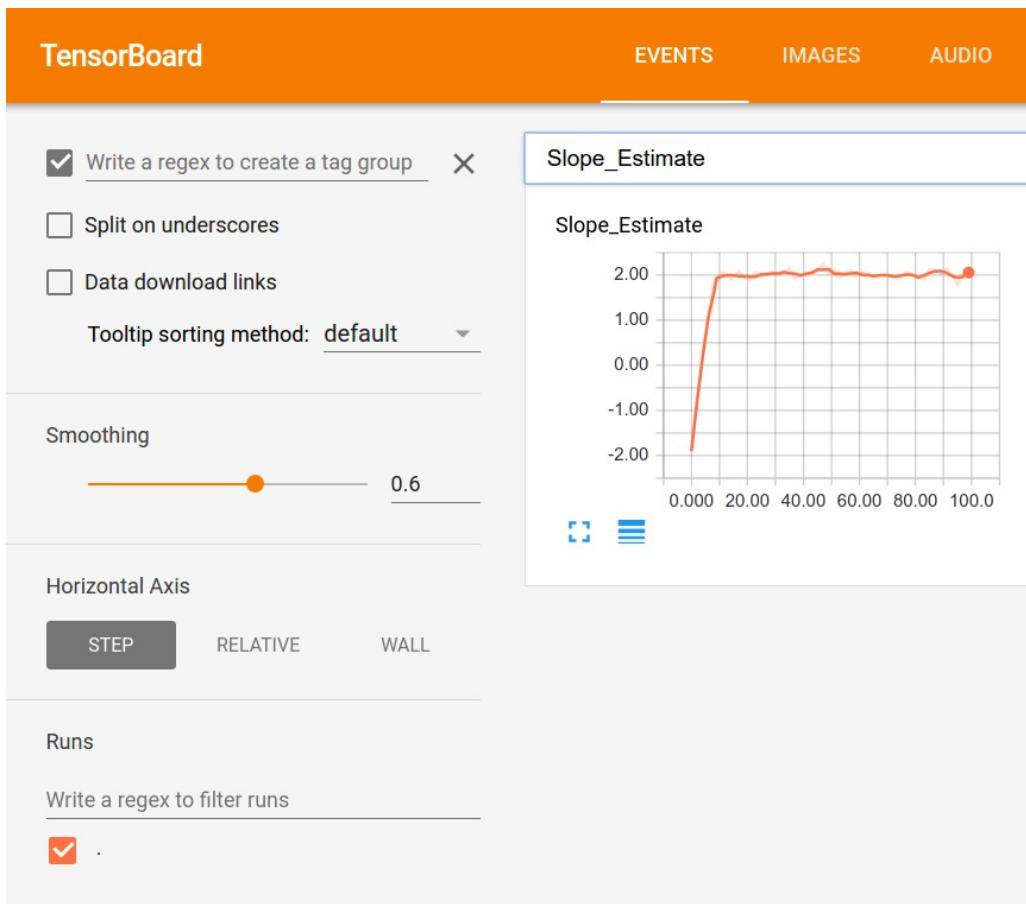


Figure 1: The scalar value, our slope estimate, visualized in tensorboard

Here, we can see a plot over the 100 generations of our scalar summary, the slope estimate. We can see that it does, in fact, approach the true value of 2:

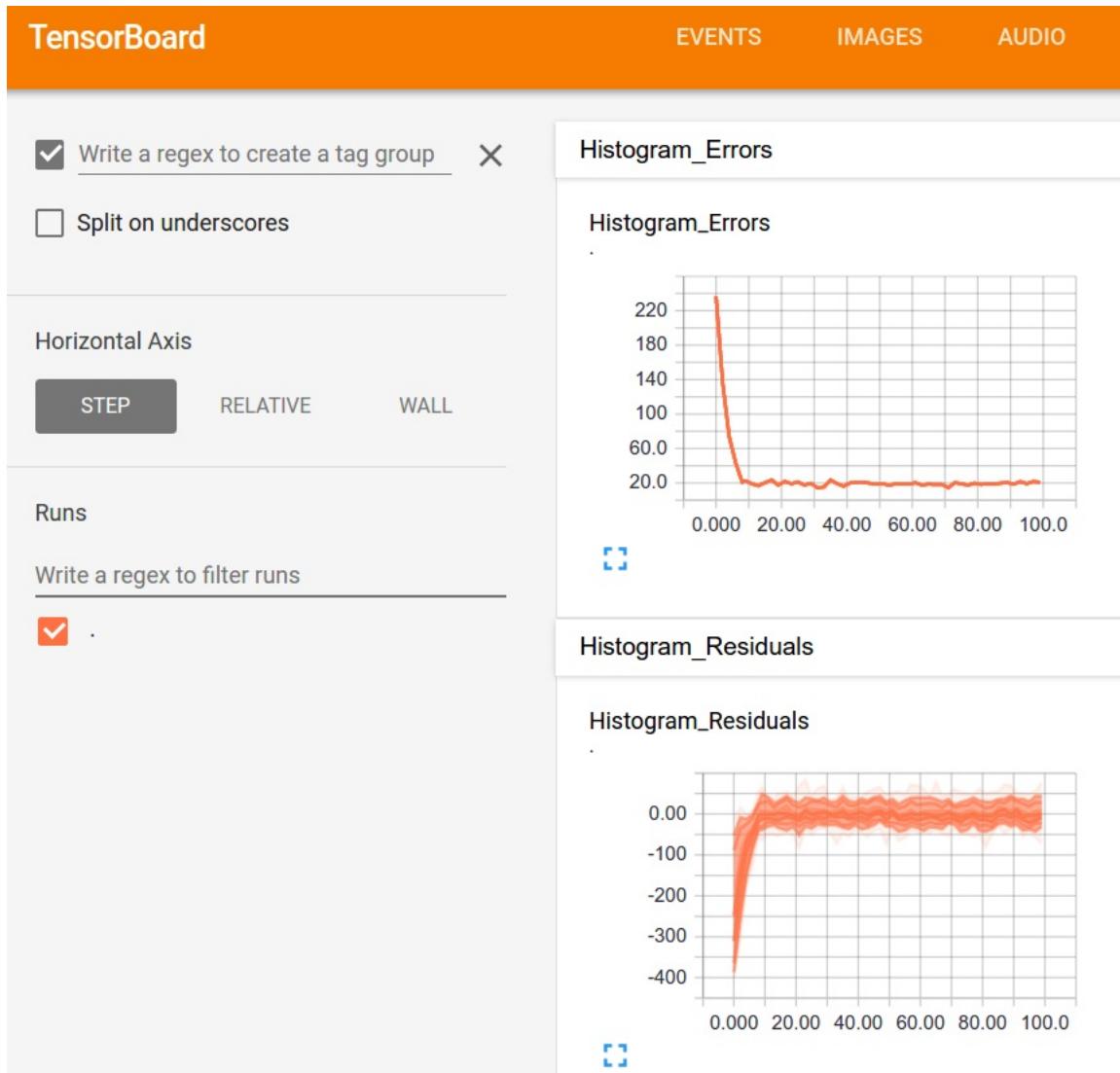


Figure 2: Here we visualize histograms of the errors and residuals for our model

The preceding graph shows one way to view histogram summaries, which can be viewed as multiple line graphs:

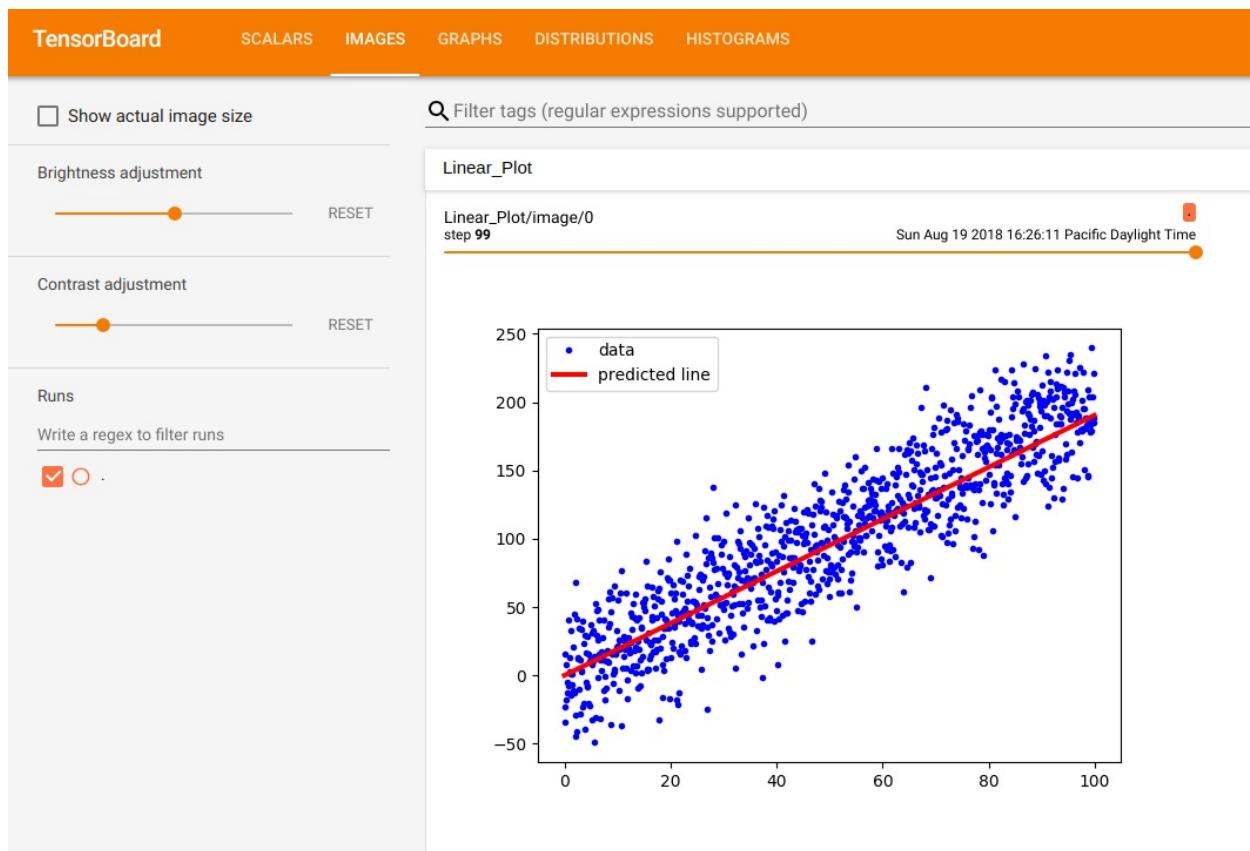


Figure 3: An inserted picture in tensorboard

The preceding is the final fit and data point graph that we put in `protobuf` format, and inserted into an image summary in TensorBoard.

Working with a genetic algorithm

TensorFlow can also be used to update any iterative algorithm that we can express in a computational graph. One such iterative algorithm is the genetic algorithm, an optimization procedure.

Getting ready

In this recipe, we will illustrate how to implement a simple genetic algorithm. Genetic algorithms are a way to optimize over any parameter space (discrete, continuous, smooth, non-smooth, and so on). The idea is to create a population of randomly initialized solutions, and apply selection, recombination, and mutation to generate new (and potentially better) child solutions. The whole idea rests on the fact that we can calculate the fitness of an individual solution by seeing how well that individual solves the problem.

Generally, the outline for a genetic algorithm is to start with a randomly initialized population, rank them in terms of their fitness, and then select the top fit individuals to randomly recombine (or cross over) to create new child solutions. These child solutions are then mutated slightly, to generate new and unseen improvements, and then added back into the parent population. After we have combined the children and parents, we repeat the whole process again.

Stopping criteria for genetic algorithms vary, but for our purposes we will just iterate them for a fixed amount of generations. We could also stop when our best fit individual achieves a desired level of fitness or when the maximum fitness does not change after so many generations.

For this recipe, we will keep it simple to illustrate how to do this in Tensorflow. The problem we will be solving is to generate an individual (an array of 50 floating point numbers) that is the closest to the ground truth function,

$f(x) = \sin\left(\frac{2\pi x}{50}\right)$. The fitness will be the negative of the mean squared error (higher is better) between the individual and the ground truth.

How to do it...

1. We'll start by loading the necessary libraries for the script:

```
import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

2. Next, we'll set the parameters of the genetic algorithm. Here, we will have 100 individuals, each with a length of 50. The selection percentage will be 20% (keeping the top 20 individuals). The mutation will be set to be the inverse of the number of features, a common place to start for the mutation. This means that we expect one feature per child solution to change. We will run the genetic algorithm for 200 generations:

```
pop_size = 100
features = 50
selection = 0.2
mutation = 1./ features
generations = 200
num_parents = int(pop_size*selection)
num_children = pop_size - num_parents
```

3. We'll initialize the graph session and create the ground truth function, which we will use to quickly calculate the fitness:

```
sess = tf.Session()
# Create ground truth
truth = np.sin(2*np.pi*(np.arange(features, dtype=np.float32))/features)
```

4. Next, we'll initialize the population as a TensorFlow variable with a random normal input:

```
population = tf.Variable(np.random.randn(pop_size, features), dtype=tf.float32)
```

5. We can now create the placeholders for the genetic algorithm. The placeholders are for the ground truth and also for data that will change for every generation. Since we want the crossover places between parents to change and the mutation probabilities/values to change, those will be placeholders in our model:

```
truth_ph = tf.placeholder(tf.float32, [1, features])
crossover_mat_ph = tf.placeholder(tf.float32, [num_children, features])
```

```
| mutation_val_ph = tf.placeholder(tf.float32, [num_children, features])
```

6. Now, we will calculate the population `fitness` (negative mean squared error), and find the top performing individuals:

```
| fitness = -tf.reduce_mean(tf.square(tf.subtract(population, truth_ph)), 1)
| top_vals, top_ind = tf.nn.top_k(fitness, k=pop_size)
```

7. For results and plotting purposes, we also want to retrieve the best fit individual in the population:

```
| best_val = tf.reduce_min(top_vals)
| best_ind = tf.argmin(top_vals, 0)
| best_individual = tf.gather(population, best_ind)
```

8. Next, we sort the parent population and slice off the top performing individuals to make them parents for the next generation:

```
| population_sorted = tf.gather(population, top_ind)
| parents = tf.slice(population_sorted, [0, 0], [num_parents, features])
```

9. Now, we'll create the children by creating two parent matrices that are randomly shuffled. Then, we multiply and add the parent matrices by the crossover matrix of ones and zeros that we will generate each generation for the placeholders:

```
| # Indices to shuffle-gather parents
| rand_parent1_ix = np.random.choice(num_parents, num_children)
| rand_parent2_ix = np.random.choice(num_parents, num_children)
| # Gather parents by shuffled indices, expand back out to pop_size too
| rand_parent1 = tf.gather(parents, rand_parent1_ix)
| rand_parent2 = tf.gather(parents, rand_parent2_ix)
| rand_parent1_sel = tf.multiply(rand_parent1, crossover_mat_ph)
| rand_parent2_sel = tf.multiply(rand_parent2, tf.subtract(1., crossover_mat_ph))
| children_after_sel = tf.add(rand_parent1_sel, rand_parent2_sel)
```

10. The last steps are to mutate the children, which we will do by adding a random normal amount to small amount of entries in the children matrix and to concatenate this matrix back into the parent population:

```
| mutated_children = tf.add(children_after_sel, mutation_val_ph)
| # Combine children and parents into new population
| new_population = tf.concat([parents, mutated_children])
```

11. The final step in our model is to use TensorFlow's `group()` operation to assign the new population to the old population's variable:

```
| step = tf.group(population.assign(new_population))
```

12. We can now initialize the model variables, as follows:

```
|     init = tf.global_variables_initializer()  
|     sess.run(init)
```

13. Finally, we loop through the generations, recreating the random crossover and mutation matrices and updating the population each generation:

```
| for i in range(generations):  
|     # Create cross-over matrices for plugging in.  
|     crossover_mat = np.ones(shape=[num_children, features])  
|     crossover_point = np.random.choice(np.arange(1, features-1, step=1), num_chi  
|     for pop_ix in range(num_children):  
|         crossover_mat[pop_ix, 0:crossover_point[pop_ix]] = 0.  
|     # Generate mutation probability matrices  
|     mutation_prob_mat = np.random.uniform(size=[num_children, features])  
|     mutation_values = np.random.normal(size=[num_children, features])  
|     mutation_values[mutation_prob_mat >= mutation] = 0  
|  
|     # Run GA step  
|     feed_dict = {truth_ph: truth.reshape([1, features]),  
|                   crossover_mat_ph: crossover_mat,  
|                   mutation_val_ph: mutation_values}  
|     step.run(feed_dict, session=sess)  
|     best_individual_val = sess.run(best_individual, feed_dict=feed_dict)  
|  
|     if i % 5 == 0:  
|         best_fit = sess.run(best_val, feed_dict = feed_dict)  
|         print('Generation: {}, Best Fitness (lowest MSE): {:.2}'.format(i, -best_
```

14. This results in the following output:

```
| Generation: 0, Best Fitness (lowest MSE): 1.5  
| Generation: 5, Best Fitness (lowest MSE): 0.83  
| Generation: 10, Best Fitness (lowest MSE): 0.55  
| Generation: 185, Best Fitness (lowest MSE): 0.085  
| Generation: 190, Best Fitness (lowest MSE): 0.15  
| Generation: 195, Best Fitness (lowest MSE): 0.083
```

How it works...

In this recipe, we have shown you how to use TensorFlow to run a simple genetic algorithm. To verify that it is working, we can also look at the most fit individual solution and the ground truth on a plot:

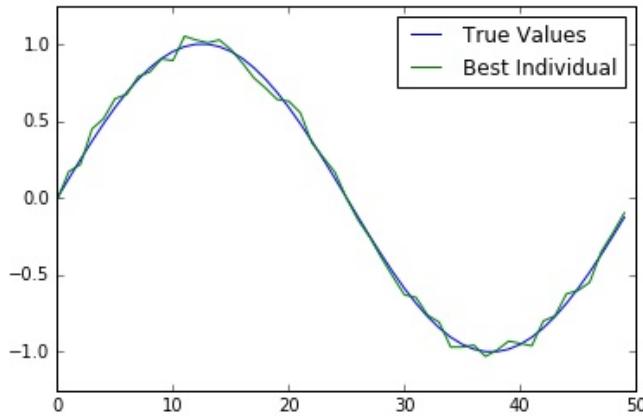


Figure 4: A plot of the ground truth and the best fit individual after 200 generations. We can see that the best fit individual is very close to the ground truth

There's more...

There are many variations to genetic algorithms. We can have two parent populations with two different fitness criteria (for example, lowest MSE and smoothness). We could impose restrictions on the mutation values to not be greater than 1 or less than -1. There are many different changes we could make, and these changes vary greatly, depending on the problem we are trying to optimize. For this contrived problem, the fitness was easily calculated, but for most genetic algorithms, calculating the fitness is an arduous task. For example, if we wanted to use a genetic algorithm to optimize the architecture of a convolutional neural network, we could have an individual be an array of parameters. The parameters could stand for the filter sizes, stride sizes, and so on of each convolutional layer. The fitness of such an individual would be the accuracy of classification after a fixed amount of iterations through a dataset. If we have 100 individuals in this population, we would have to evaluate 100 different CNN models for each generation. This is very computationally intense.

Before using a genetic algorithm to solve your problem, it is wise to figure out how long it takes to calculate the `fitness` of an individual. If this operation is time-consuming, genetic algorithms may not be the best tool to use.

Clustering using k-means

TensorFlow can also be used to implement iterative clustering algorithms, such as k-means. In this recipe, we show an example of using k-means on the `iris` dataset.

Getting ready

Almost all of the machine learning models we have explored in this book have been supervised models. TensorFlow is ideal for these types of problems. But, we can also implement unsupervised models if we wish. As an example, this recipe will implement k-means clustering.

The dataset we will implement clustering on is the `iris` dataset. One of the reasons this is a good dataset is because we already know there are three different targets (three types of iris flowers). This gives us a leg up on knowing that we are looking for three different clusters in the data.

We will cluster the `iris` dataset into three groups, and then compare the accuracy of these clusters against the real labels.

How to do it...

1. To start, we load the necessary libraries. We are also loading some PCA tools from `sklearn` so that we can change the resulting data from four dimensions to two dimensions for visualization purposes:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn import datasets
from scipy.spatial import cKDTree
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale
```

2. We start a graph session, and load the `iris` dataset:

```
sess = tf.Session()
iris = datasets.load_iris()
num_pts = len(iris.data)
num_feats = len(iris.data[0])
```

3. We'll now set the groups, generations, and create the variables we need for the graph:

```
k=3
generations = 25
data_points = tf.Variable(iris.data)
cluster_labels = tf.Variable(tf.zeros([num_pts], dtype=tf.int64))
```

4. The next variables we need are the centroids for each group. We will initialize the centroids for the k-means algorithm by randomly choosing three different points of the `iris` dataset:

```
rand_starts = np.array([iris.data[np.random.choice(len(iris.data))]] for _ in ran
centroids = tf.Variable(rand_starts)
```

5. Now, we need to calculate the distances between each of the data points and each of the centroids. We do this by expanding the `centroids` into a matrix, and the same for the data points. We'll then calculate the Euclidean distances between the two matrices:

```
centroid_matrix = tf.reshape(tf.tile(centroids, [num_pts, 1]), [num_pts, k, num_
point_matrix = tf.reshape(tf.tile(data_points, [1, k]), [num_pts, k, num_feats])
distances = tf.reduce_sum(tf.square(point_matrix - centroid_matrix), reduction_i
```

6. The `centroids` assignment is then the closest `centroids` (smallest distance) to each data point:

```
| centroid_group = tf.argmin(distances, 1)
```

7. Now, we have to calculate the group average to get the new centroid:

```
def data_group_avg(group_ids, data):
    # Sum each group
    sum_total = tf.unsorted_segment_sum(data, group_ids, 3)
    # Count each group
    num_total = tf.unsorted_segment_sum(tf.ones_like(data), group_ids, 3)
    # Calculate average
    avg_by_group = sum_total/num_total
    return(avg_by_group)
means = data_group_avg(centroid_group, data_points)
update = tf.group(centroids.assign(means), cluster_labels.assign(centroid_group))
```

8. Next, we initialize the model variables:

```
| init = tf.global_variables_initializer()
| sess.run(init)
```

9. We'll iterate through the generations and update the centroids for each group accordingly:

```
for i in range(generations):
    print('Calculating gen {}, out of {}'.format(i, generations))
    _, centroid_group_count = sess.run([update, centroid_group])
    group_count = []
    for ix in range(k):
        group_count.append(np.sum(centroid_group_count==ix))
    print('Group counts: {}'.format(group_count))
```

10. This results in the following output:

```
| Calculating gen 0, out of 25. Group counts: [50, 28, 72] Calculating gen 1, out
```

11. To verify our clustering, we can use the clusters to make predictions. We now see how many data points are in a similar cluster of the same iris species:

```
[centers, assignments] = sess.run([centroids, cluster_labels])
def most_common(my_list):
    return(max(set(my_list), key=my_list.count))
label0 = most_common(list(assignments[0:50]))
label1 = most_common(list(assignments[50:100]))
label2 = most_common(list(assignments[100:150]))
group0_count = np.sum(assignments[0:50]==label0)
group1_count = np.sum(assignments[50:100]==label1)
group2_count = np.sum(assignments[100:150]==label2)
accuracy = (group0_count + group1_count + group2_count)/150.
print('Accuracy: {:.2}'.format(accuracy))
```

12. This results in the following output:

```
| Accuracy: 0.89
```

13. To visually see our groupings, and if they have indeed separated out the `iris` species, we will transform the four dimensions to two dimensions using PCA, and plot the data points and groups. After the PCA decomposition, we create predictions on a grid of x-y values for plotting a color graph:

```
pca_model = PCA(n_components=2)
reduced_data = pca_model.fit_transform(iris.data)
# Transform centers
reduced_centers = pca_model.transform(centers)
# Step size of mesh for plotting
h = .02
x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
# Get k-means classifications for the grid points
xx_pt = list(xx.ravel())
yy_pt = list(yy.ravel())
xy_pts = np.array([[x,y] for x,y in zip(xx_pt, yy_pt)])
mytree = cKDTree(reduced_centers)
dist, indexes = mytree.query(xy_pts)
indexes = indexes.reshape(xx.shape)
```

14. And, here is `matplotlib` code to combine our findings on one graph. This plotting section of code is heavily adapted from a demo on the scikit-learn (<http://scikit-learn.org/>) documentation website (http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_digits.html):

```
plt.clf()
plt.imshow(indexes, interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()),
           cmap=plt.cm.Paired,
           aspect='auto', origin='lower')
# Plot each of the true iris data groups
symbols = ['o', '^', 'D']
label_name = ['Setosa', 'Versicolour', 'Virginica']
for i in range(3):
    temp_group = reduced_data[(i*50):(50)*(i+1)]
    plt.plot(temp_group[:, 0], temp_group[:, 1], symbols[i], markersize=10, label=label_name[i])
# Plot the centroids as a white X
plt.scatter(reduced_centers[:, 0], reduced_centers[:, 1],
            marker='x', s=169, linewidths=3,
            color='w', zorder=10)
plt.title('K-means clustering on Iris Datasets'
          '\nCentroids are marked with white cross')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.legend(loc='lower right')
plt.show()
```

This plotting code will show us the three classes, predicted spaces of the three

classes, and the centroids of each group:

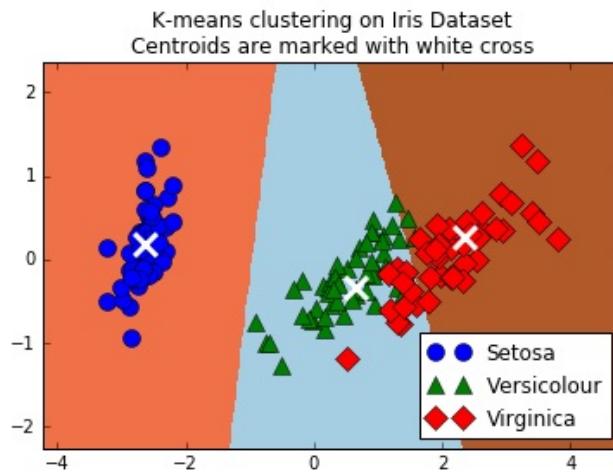


Figure 5: Screenshot showing how the unsupervised classification algorithm of k-means; can be used to group together the three iris flower species. The three k-means groups are the three shaded regions, and the three different points (circles, triangles, and diamonds) are the three true species classifications

There's more...

For this recipe, we clustered the `iris` dataset into three groups using TensorFlow. Then, we calculated the percentage of data points that fell into similar groups (89%) and plotted a graph of the resulting k-means groups. Since k-means as a classification algorithm is locally linear (linear separator up close), it will be hard to learn the naturally non-linear boundary between Iris versicolor and Iris virginica. But, one advantage is that the k-means algorithm did not need labeled data at all to perform.

Solving a system of ordinary differential equations

TensorFlow can be used for many algorithmic implementations and procedures. A great example of TensorFlow's versatility is implementing an ODE solver. Solving an ODE numerically is an iterative procedure that can be easily described in a computational graph. For this recipe, we will solve the Lotka-Volterra predator-prey system.

Getting ready

This recipe will illustrate how to solve a system of **ordinary differential equations (ODEs)**. We can use similar methods to the previous two sections to update values as we iterate through and solve an ODE system.

The ODE system we will consider is the famous Lotka-Volterra predator-prey system. This system shows how a predator-prey system can be oscillating, given specific parameters.

The Lotka-Volterra system was published in a paper in 1920 (see also *Figure 1, The scalar value, our slope estimate, visualized in tensorboard*). We will use similar parameters to show that an oscillating system can occur. Here is the system represented in a mathematically discrete way:

$$\begin{aligned} X_{t+1} &= X_t + (aX_t + bX_t Y_t) \Delta t \\ Y_{t+1} &= Y_t + (cY_t + dX_t Y_t) \Delta t \end{aligned}$$

Here, X is the prey and Y will be the predator. We determine which is the prey and which is the predator by the values of a , b , c , and d : for the prey, $a>0$, $b<0$, and for the predator, $c<0$, $d>0$. We will implement this discrete version in the TensorFlow solution to the system.

How to do it...

1. We'll start by loading libraries and starting a graph session:

```
| import matplotlib.pyplot as plt  
| import tensorflow as tf  
| sess = tf.Session()
```

2. We then declare our constants and variables in the graph:

```
| x_initial = tf.constant(1.0)  
| y_initial = tf.constant(1.0)  
| X_t1 = tf.Variable(x_initial)  
| Y_t1 = tf.Variable(y_initial)  
| # Make the placeholders  
| t_delta = tf.placeholder(tf.float32, shape=())  
| a = tf.placeholder(tf.float32, shape=())  
| b = tf.placeholder(tf.float32, shape=())  
| c = tf.placeholder(tf.float32, shape=())  
| d = tf.placeholder(tf.float32, shape=())
```

3. Next, we will implement the prior introduced discrete system, and then update the x and y populations:

```
| X_t2 = X_t1 + (a * X_t1 + b * X_t1 * Y_t1) * t_delta  
| Y_t2 = Y_t1 + (c * Y_t1 + d * X_t1 * Y_t1) * t_delta  
| # Update to New Population  
| step = tf.group(  
|     X_t1.assign(X_t2),  
|     Y_t1.assign(Y_t2))
```

4. We now initialize the graph and run the discrete ODE system, with specific parameters to illustrate a cyclic behavior:

|  **TIP** init = tf.global_variables_initializer() sess.run(init) # Run the ODE prey_value
A steady state (and cyclic) solution to this specific system, the Lotka-Volterra equations,
very much depends on specific parameters and population values. We encourage the reader
to try different parameters and values to see what can happen.

5. Now, we can plot the predator and prey values:

```
| plt.plot(prey_values, label="Prey")  
| plt.plot(predator_values, label="Predator")  
| plt.legend(loc='upper right')  
| plt.show()
```

This plotting code will result in a screenshot that shows the oscillating

populations of predators and prey:

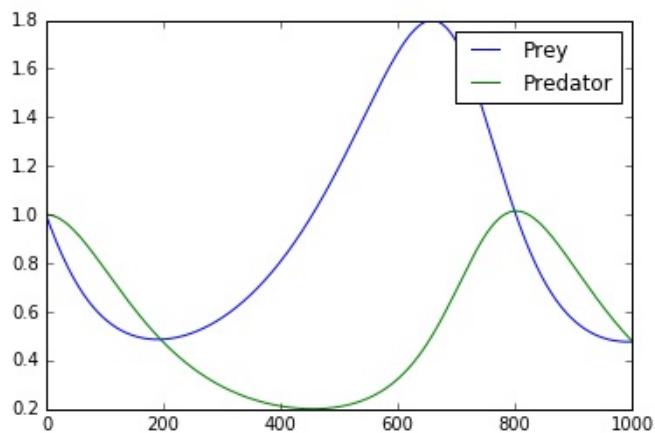


Figure 6: Here, we plot the predator and prey values for the ODE solution. We can see that, indeed, cycles do occur

How it works...

We used TensorFlow to incrementally solve a discrete version of an ODE system. For specific parameters, we saw that the predator-prey system can indeed have cyclic solutions. This would make sense in our system biologically, because if there are too many predators, the prey start to die off, and then there is less food for the predators so they will die off, and so on.

See also

Lotka, A. J., *Analytical note on certain rhythmic relations in organic systems.* Proc. Nat. Acad. 6 (1920) (<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1084562/>).

Using a random forest

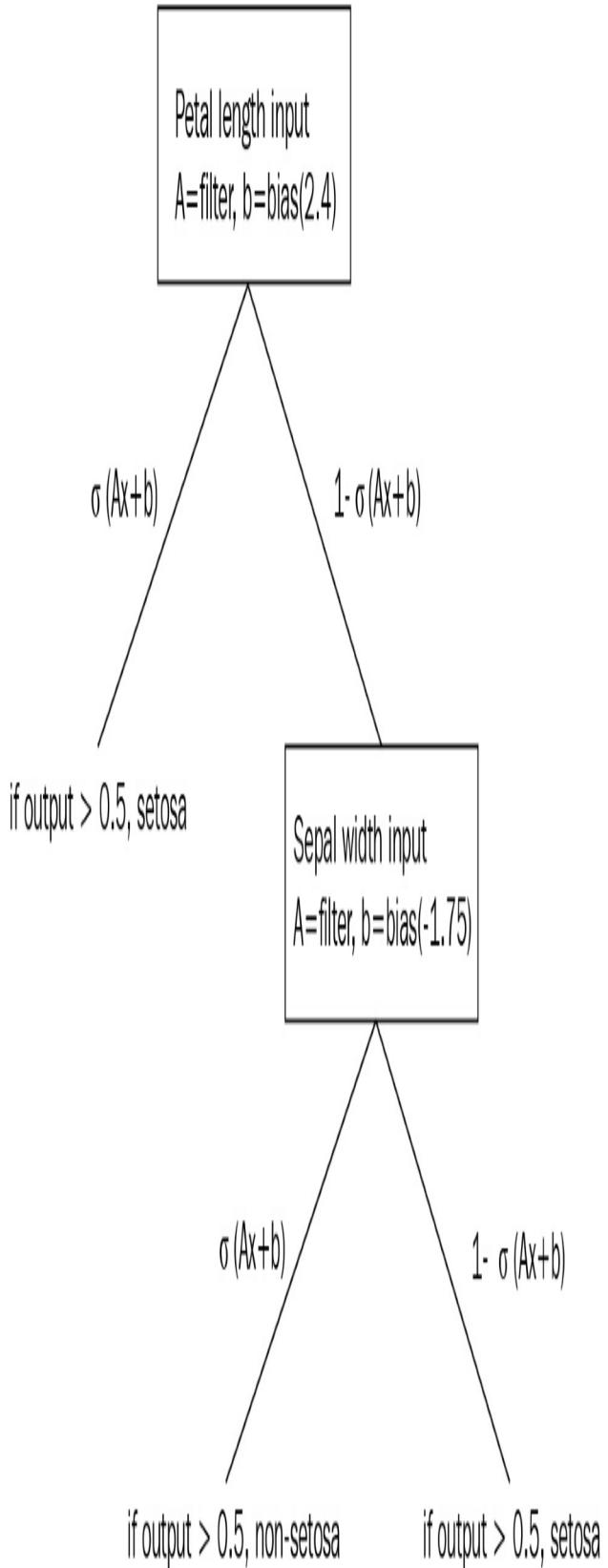
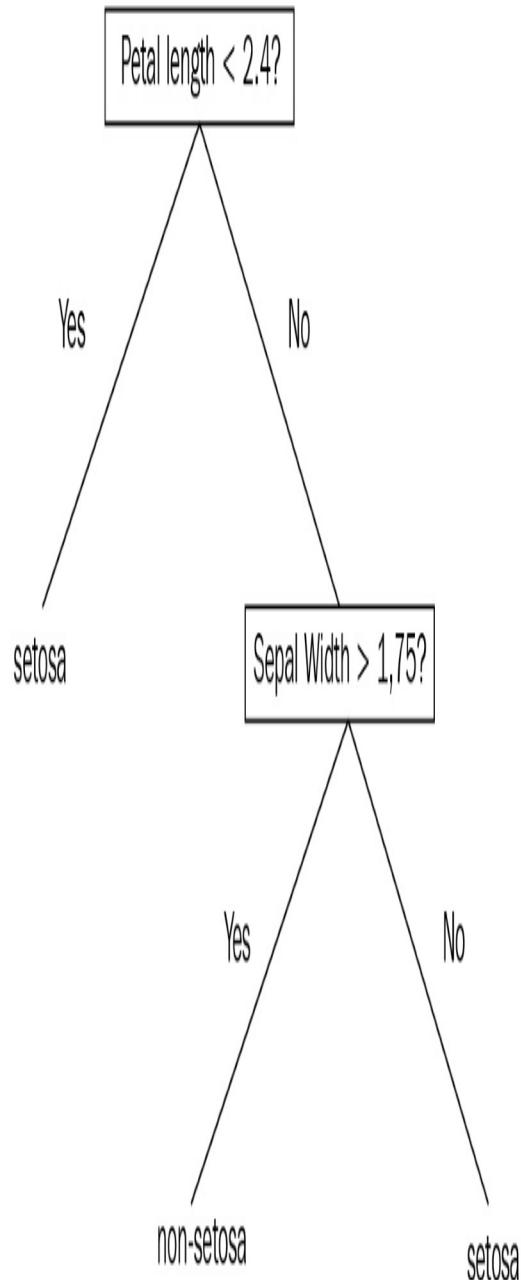
Random forest algorithms are built on aggregating decision trees on randomly selected observations and/or randomly selected features. We will not cover how decision trees are trained, but will show that there are types of random forests that can be trained using gradient boosting, which TensorFlow can calculate for us.

Getting ready

Tree based algorithms are traditionally non-smooth, as they are based on partitioning the data to minimize the variance in the target outputs. Non-smooth methods do not lend themselves well to gradient based methods. TensorFlow relies on the fact that the functions used in the model are smooth and that it automatically calculates how to change the model parameters to minimize the function loss. The way that TensorFlow gets around this obstacle is to do a smooth approximation to a decision boundary. One can approximate a decision boundary with a softmax function or similar shaped function.

Decision trees will provide a hard split on a dataset through generating rules, for example, If $x > 0.5$, then move to this branch of the tree.... This tells us that a whole subset of the data will group together or split up, depending on the values of x . The smooth approximation of this deals with probabilities and not whole splits. This means that each observation of a dataset will have a probability of being in every end-node of the tree. The differences are better illustrated in the following diagram comparing a traditional decision tree and a probabilistic decision tree.

The following diagram illustrates the difference between two example decision trees:



This diagram illustrates a standard decision tree (left) which is non-differentiable, and a smooth decision tree (right), which illustrates the usage of sigmoid functions to develop probabilities of an observation appearing in a labeled leaf or end-node.

We choose not to go into detail about differentiability, continuity, and smoothness of functions.

 *The purpose of this section is to provide an intuitive description on how a non-differentiable model can be approximated by a differentiable one. For more mathematical details, we encourage the reader to look at the See also section at the end of this recipe.*

How to do it...

TensorFlow has included some default model estimator functions that we will rely on for this recipe. There are two main gradient-boosted models, a regression-tree and a classification-tree. For this example, we will be using a regression tree to predict the `boston house price` dataset (<https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html>).

1. To start we load the necessary libraries:

```
| import numpy as np  
| import tensorflow as tf  
| from keras.datasets import boston_housing  
| from tensorflow.python.framework import ops  
| ops.reset_default_graph()
```

2. Next, we set the model we are going to use from the TensorFlow estimator library. Here, we will use the `BoostedTreesRegressor` model, which is used for regression with gradient boosted trees:

```
| regression_classifier = tf.estimator.BoostedTreesRegressor
```



Alternatively, for binary classification problems the reader may use the estimator, `BoostedTreesClassifier`. Multiclass classification is not supported at this time, although it is on the roadmap in the future.

3. Now, we can use the Keras data import function to load the Boston housing price dataset in one line as follows:

```
| (x_train, y_train), (x_test, y_test) = boston_housing.load_data()
```

4. Here, we can set some of the model parameters; batch size is the number of training observations to train on at a time, we'll train for 500 iterations and the gradient boosted forest will have 100 trees with a maximum depth of each tree (number of splits) to be 6.

```
# Batch size  
batch_size = 32  
# Number of training steps  
train_steps = 500  
# Number of trees in our 'forest'  
n_trees = 100  
# Maximum depth of any tree in forest  
max_depth = 6
```

- The model estimators provided by TensorFlow want an input function. We will create a data input function for the estimator function. But first, we need to put the data in a dictionary of `numpy` arrays format that are correctly labeled. These are called **feature columns** in TensorFlow. Purely numerical columns are not yet supported, so we will put the numerical columns into automatic buckets as follows, (a) the binary features will have two buckets, and (b) other continuous numerical features will be bucketed into 5 buckets.

```

binary_split_cols = ['CHAS', 'RAD']
col_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'T
X_dtrain = {col: x_train[:, ix] for ix, col in enumerate(col_names)}
X_dtest = {col: x_test[:, ix] for ix, col in enumerate(col_names)}

# Create feature columns!
feature_cols = []
for ix, column in enumerate(x_train.T):
    col_name = col_names[ix]

    # Create binary split feature
    if col_name in binary_split_cols:
        # To create 2 buckets, need 1 boundary - the mean
        bucket_boundaries = [column.mean()]
        numeric_feature = tf.feature_column.numeric_column(col_name)
        final_feature = tf.feature_column.bucketized_column(source_column=numeric
    # Create bucketed feature
    else:
        # To create 5 buckets, need 4 boundaries
        bucket_boundaries = list(np.linspace(column.min() * 1.1, column.max() *
        numeric_feature = tf.feature_column.numeric_column(col_name)
        final_feature = tf.feature_column.bucketized_column(source_column=numeric

    # Add feature to feature_col list
    feature_cols.append(final_feature)

```



It is a good idea to have the shuffle option of the input function set to `true` for training and `false` for testing. We would like to shuffle the x and y training sets every epoch, but not during testing.

- We now declare our data input function with the `numpy` input function of the inputs library in the TensorFlow estimators. We will specify the training observation dictionary we created and an array of `y` targets.

```
|     input_fun = tf.estimator.inputs.numpy_input_fn(X_dtrain, y=y_train, batch_size=b
```

- Now, we declare our model and start the training:

```

model = regression_classifier(feature_columns=feature_cols,
                               n_trees=n_trees,
                               max_depth=max_depth,
                               learning_rate=0.25,
                               n_batches_per_layer=batch_size)
model.train(input_fn=input_fun, steps=train_steps)

```

- During training, we should see similar output as follows:

```
INFO:tensorflow:Using default config.
WARNING:tensorflow:Using temporary folder as model directory: /tmp/tmpqxyd62cu
INFO:tensorflow:Using config: {'_model_dir': '/tmp/tmpqxyd62cu', '_tf_random_see
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow>Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 0 into /tmp/tmpqxyd62cu/model.ckpt.
INFO:tensorflow:loss = 691.09814, step = 1
INFO:tensorflow:global_step/sec: 587.923
INFO:tensorflow:loss = 178.62021, step = 101 (0.171 sec)
INFO:tensorflow:Saving checkpoints for 127 into /tmp/tmpqxyd62cu/model.ckpt.
INFO:tensorflow:Loss for final step: 37.436565.
Out[190]: <tensorflow.python.estimator.canned.boosted_trees.BoostedTreesRegresso
```

9. To evaluate our model, we create another input function for the test data and get the predictions for each of the test data points. The following is code to get the predictions and print the **mean absolute error (MAE)**:

```
p_input_fun = tf.estimator.inputs.numpy_input_fn(X_dtest, y=y_test, batch_size=b
# Get predictions
predictions = list(model.predict(input_fn=p_input_fun))
final_preds = [pred['predictions'][0] for pred in predictions]

# Get accuracy (mean absolute error, MAE)
mae = np.mean([np.abs((actual - predicted) / predicted) for actual, predicted in
print('Mean Abs Err on test set: {}'.format(acc))
```

10. Which prints off around 0.71 as the error. Note that, due to the randomness of the shuffling, the reader may get slightly different results. To increase accuracy, we may consider increasing the number epochs or introducing a lower learning rate or even a decaying learning rate of some type (exponential or linear):

```
| Mean Abs Err on test set: 0.7111111111111111
```

How it works...

In this recipe, we illustrate how to use a TensorFlow estimator and the data-input functions provided by TensorFlow. These are very powerful tools that not only make our TensorFlow code shorter and more readable, but they also increase the efficiency of the algorithm and decrease the development time required for creating and testing algorithms.

See also

For more references on decision trees, random forests, gradient boosted forests, and the mathematics behind differentiability, smoothness, and continuity, we encourage the reader to read the following references.

1. Tutorial on decision trees. From a machine learning crash course by Berkeley. <https://ml.berkeley.edu/blog/2017/12/26/tutorial-5/>
2. Random forest python tutorial. By Chris Albon. https://chrisalbon.com/machine_learning/trees_and_forests/random_forest_classifier_example/
3. A nice PDF presentation on convex functions, how they are used in machine learning, and the differences between smoothness, differentiability, and continuity. By Francis Bach. Also has ~6 pages of useful references at the end, which the reader may find helpful. https://www.di.ens.fr/~fbach/gradsto_allerton.pdf
4. Article on Soft Decision Trees: *Distilling a Neural Network into a Soft Decision Tree*. Frosst and Hinton. 2017. https://cex.inf.unibz.it/resources/Frosst+Hinton-CExAIIA_2017.pdf
5. TensorFlow implementation of a Neural Tree. By Benoit Deschamps. <https://github.com/benoitdescamps/Neural-Tree>

Using TensorFlow with Keras

TensorFlow is great for the flexibility and power it provides to the programmer. A drawback of this is that prototyping models, and iterating through various tests can be cumbersome for the programmer. Keras is a wrapper for deep learning libraries that makes it simpler to deal with various aspects of the model and make the programming easier. Here, we choose to use Keras on top of TensorFlow. In fact, using Keras with the TensorFlow backend is so popular, that there is a Keras library within TensorFlow. For this recipe, we will be using that TensorFlow library to do a fully connected neural network and a simple CNN image network on the MNIST dataset.

Getting ready

For this recipe, we will use the Keras functions that reside inside TensorFlow already. Keras (<https://keras.io/>) is already a separate python library which you can install. If you choose to go for the pure Keras route, you will have to choose a backend for Keras (like TensorFlow).

In this recipe we will perform two separate models on the MNIST image recognition dataset. The first is a straightforward fully connected neural network, while the second replicates our CNN network from Chapter 8, Section 2, "Implementing a Simple CNN".

How to do it...

1. We will start by loading the necessary libraries for our script.

```
import tensorflow as tf
from sklearn.preprocessing import MultiLabelBinarizer
from keras.utils import to_categorical
from tensorflow import keras
from tensorflow.python.framework import ops
ops.reset_default_graph()

# Load MNIST data
from tensorflow.examples.tutorials.mnist import input_data
```

2. We can load the library with the provided MNIST data import function in TensorFlow. Although the original MNIST images are 28 pixels by 28 pixels, the imported data is a flattened version of them, where each observation is a row of 784 greyscale points between 0 and 1. The y-labels are imported as integers between 0 and 9.

```
mnist = input_data.read_data_sets("MNIST_data/")
x_train = mnist.train.images
x_test = mnist.test.images
y_train = mnist.train.labels
y_test = mnist.test.labels
y_train = [[i] for i in y_train]
y_test = [[i] for i in y_test]
```

3. We will now convert the target integers into a one-hot encoded vector by using scikit-learn's `MultiLabelBinarizer()` function, as follows:

```
one_hot = MultiLabelBinarizer()
y_train = one_hot.fit_transform(y_train)
y_test = one_hot.transform(y_test)
```

4. We will create a three layer fully connected neural network with 32, 16, and 10 respective hidden nodes. Then the final output has size ten (one for each digit). We create this network with the following code:

```
# We start with a 'sequential' model type (connecting layers together)
model = keras.Sequential()

# Adds a densely-connected layer with 32 units to the model, followed by an ReLU
model.add(keras.layers.Dense(32, activation='relu'))

# Adds a densely-connected layer with 16 units to the model, followed by an ReLU
model.add(keras.layers.Dense(16, activation='relu'))
```

```
|     # Add a softmax layer with 10 output units:  
|     model.add(keras.layers.Dense(10, activation='softmax'))
```

5. To train the model, all we have to do next is call the `compile()` method with the appropriate arguments. The arguments we will need are the optimization function and the loss type. But we also want to record the model accuracy, so the list of metrics includes the `accuracy` argument.

```
|     model.compile(optimizer=tf.train.AdamOptimizer(0.001),  
|                     loss='categorical_crossentropy',  
|                     metrics=['accuracy'])
```

6. This will result in output that should similar to the following:

```
Epoch 1/5  
64/55000 [...........................] - ETA: 1:44 - loss: 2.3504 - acc: 0  
3776/55000 [=>.....] - ETA: 2s - loss: 1.7904 - acc: 0.3  
...  
47104/55000 [=====>.....] - ETA: 0s - loss: 0.1337 - acc: 0.9  
50880/55000 [=====>....] - ETA: 0s - loss: 0.1336 - acc: 0.9  
55000/55000 [=====] - 1s 13us/step - loss: 0.1335 - acc  
Out[]: <tensorflow.python.keras.callbacks.History at 0x7f5768a40da0>
```



To configure a regression model for mean squared error loss, we would use a model compilation as follows: `model.compile(optimizer=tf.train.AdamOptimizer(0.01), loss='mse', metrics=['mae'])`

7. Next, we will see how to implement a CNN model with two convolutional layers, with max pooling, all followed by a fully connected layer. To start we will have to reshape our flattened images to 2D images and make the y-targets into numpy arrays, as follows:

```
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)  
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)  
input_shape = (28, 28, 1)  
num_classes = 10  
  
# Categorize y targets  
y_test = to_categorical(mnist.test.labels)  
y_train = to_categorical(mnist.train.labels)
```

8. We will create our CNN in a similar sequential layer method as before. This time we will use the `conv2D()`, `MaxPooling2D()`, and `Dense()` Keras functions to create our CNN model as follows:

```
cnn_model = keras.Sequential()  
# First convolution layer  
cnn_model.add(keras.layers.Conv2D(25,  
                                 kernel_size=(4, 4),  
                                 strides=(1, 1),  
                                 activation='relu',  
                                 input_shape=input_shape))  
  
# Max pooling
```

```

cnn_model.add(keras.layers.MaxPooling2D(pool_size=(2, 2),
                                         strides=(2, 2)))
# Second convolution layer
cnn_model.add(keras.layers.Conv2D(50,
                                 kernel_size=(5, 5),
                                 strides=(1, 1),
                                 activation='relu'))
# Max pooling
cnn_model.add(keras.layers.MaxPooling2D(pool_size=(2, 2),
                                         strides=(2, 2)))
# Flatten for dense (fully connected) layer
cnn_model.add(keras.layers.Flatten())
# Add dense (fully connected) layer
cnn_model.add(keras.layers.Dense(num_classes, activation='softmax'))

```

9. Next, we will compile our model with choosing an optimization and loss function.

```

cnn_model.compile(optimizer=tf.train.AdamOptimizer(0.001),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

```

10. Keras also allows us to insert functions into our training code called **Callbacks**. Callbacks are functions that are executed at certain times in the code and can be used to perform various functions. There are many premade callbacks, that can save the model, stop training under specific criteria, record values, and many others. See <https://keras.io/callbacks/> for more on the various options. To illustrate how to make our own custom callback and to show how they work in general, we will create a callback called `RecordAccuracy()` that is a Keras Callback class and will store the accuracy at the end of every epoch as follows:

```

class RecordAccuracy(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.acc = []

    def on_epoch_end(self, batch, logs={}):
        self.acc.append(logs.get('acc'))

accuracy = RecordAccuracy()

```

11. Next, we will train our CNN model with the `fit()` method. Here we will provide `validation_data` and `callbacks` as follows:

```

cnn_model.fit(x_train,
               y_train,
               batch_size=64,
               epochs=3,
               validation_data=(x_test, y_test),
               callbacks=[accuracy])

print(accuracy.acc)

```

12. This training will result in similar output as shown below:

```
Train on 55000 samples, validate on 64 samples
Epoch 1/3
  64/55000 [...........................] - ETA: 2:59 - loss: 2.2805 - acc: 0
  192/55000 [>.....] - ETA: 1:14 - loss: 2.2729 - acc: 0
...
54848/55000 [=====>.] - ETA: 0s - loss: 0.0603 - acc: 0.9
54976/55000 [=====>.] - ETA: 0s - loss: 0.0603 - acc: 0.9
55000/55000 [=====] - 26s 469us/step - loss: 0.0604 - a
Out[]: <tensorflow.python.keras.callbacks.History at 0x7f69494c7780>
[0.9414363636450334, 0.9815818181731484, 0.9998980778226293]
```

How it works...

In this recipe we have shown how succinct creating and training models with Keras can be. Many of the intricate details of variable type, dimensions, and data ingestion are automatically handled for you. While this can be reassuring, we should be aware that if we gloss over too many model details, we may be unintentionally implementing a wrong model.

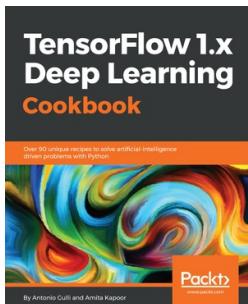
See also

For further Keras information, the reader is encouraged to look at the following resources:

- Official Keras documentation: <https://keras.io/>
- TensorFlow Keras Tutorial: <https://www.tensorflow.org/guide/keras>
- "An Introduction to Keras", a guest lecture by Francois Chollet at Standford (slides in a PDF): <https://web.stanford.edu/class/cs20si/lectures/march9guestlecture.pdf>

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

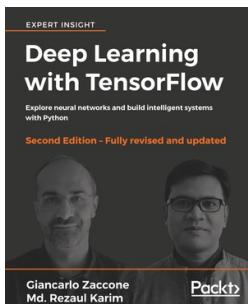


TensorFlow 1.x Deep Learning Cookbook

Antonio Gulli, Amita Kapoor

ISBN: 978-1-78829-359-4

- Install TensorFlow and use it for CPU and GPU operations
- Implement DNNs and apply them to solve different AI-driven problems.
- Leverage different data sets such as MNIST, CIFAR-10, and YouTube8m with TensorFlow and learn how to access and use them in your code.
- Use TensorBoard to understand neural network architectures, optimize the learning process, and peek inside the neural network black box.
- Use different regression techniques for prediction and classification problems
- Build single and multilayer perceptrons in TensorFlow



Deep Learning with TensorFlow - Second Edition

Giancarlo Zaccone, Md. Rezaul Karim

ISBN: 978-1-78883-110-9

- Apply deep machine intelligence and GPU computing with TensorFlow
- Access public datasets and use TensorFlow to load, process, and transform the data
- Discover how to use the high-level TensorFlow API to build more powerful applications
- Use deep learning for scalable object detection and mobile computing
- Train machines quickly to learn from data by exploring reinforcement learning techniques
- Explore active areas of deep learning research and applications

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!