Dr. PKS Prakash,
Achyutuni Sri Krishna Rao

R Deep Learning Cookbook

Solve complex neural net problems with TensorFlow, H2O and MXNet

Packt>

R Deep Learning Cookbook

Solve complex neural net problems with TensorFlow, H2O and MXNet

Dr. PKS Prakash
Achyutuni Sri Krishna Rao

BIRMINGHAM - MUMBAI

R Deep Learning Cookbook

First published: August 2017

Production reference: 1030817

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78712-108-9

www.packtpub.com

Credits

Authors
Dr. PKS Prakash
Achyutuni Sri Krishna Rao

Reviewers
Vahid Mirjalili

Commissioning Editor
Veena Pagare

Acquisition Editor
Aman Singh

Content Development Editor
Tejas Limkar

Technical Editor
Sagar Sawant

Copy Editor
Manisha Sinha

Project Coordinator
Manthan Patel

Proofreader
Safis Editing

Indexer
Tejal Daruwale Soni

Graphics
Tania Dutta

Production Coordinator
Deepika Naik

About the Authors

Dr. PKS Prakash is a data scientist and an author. He has spent the last 12 years developing many data science solutions to problems from leading companies in the healthcare, manufacturing, pharmaceutical, and e-commerce domain. He is working as a Data Science Manager at ZS Associates.

ZS is one of the world's largest business service firms, helping clients with commercial success by creating data-driven strategies using advanced analytics, which they can implement within their sales and marketing operations to make them more competitive, and by helping them deliver impact where it matters.

Prakash obtained a PhD in Industrial and System Engineering from Wisconsin-Madison, US. He defended his second PhD in Engineering from University of Warwick, UK. His educational background also includes a master's degree from the University of Wisconsin-Madison, US, and a bachelor's degree from **National Institute of Foundry and Forge Technology** (**NIFFT**), India. He is the co-founder of Warwick Analytics, which is based on his PhD work from the University of Warwick, UK.

Prakash is published widely in research areas of operational research and management, soft computing tools, and advance algorithms in leading journals such as IEEE-Trans, EJOR, and IJPR, among others. He has edited an issue of *Intelligent Approaches to Complex Systems* and contributed to *Evolutionary Computing in Advanced Manufacturing*, published by Wiley, and *Algorithms and Data Structures Using R*, published by Packt.

This book would not have been possible without the support and love from my wife, Dr. **Ritika Singh**, *and my daughter,* **Nishidha Singh**. *Also, I would like to extend my special thanks to so many people from the Packt team whose names may not all be mentioned, but their contribution is sincerely appreciated and gratefully acknowledged. The book started with an early discussion with Aman Singh (Acquisition Editor), so I want to extend special thanks to him as without his input, this book would have never happened. Also, I want to thank Tejas Limkar (Content Development Editor) for continuously pushing us and getting the book delivered on time. I would like to extend thanks to all the reviewers, whose feedback has helped us tremendously in improving the book.*

Achyutuni Sri Krishna Rao is a data scientist, a civil engineer, and an author. He has spent the last four years developing many data science solutions to problems from leading companies in the healthcare, pharmaceuticals, and manufacturing. He is working as a Data Science Consultant at ZS Associates.

Sri Krishna's background includes a master's degree in Enterprise Business Analytics and Machine Learning from National University of Singapore, Singapore. His educational background also includes a bachelor's degree from National Institute of Technology Warangal, India.

Sri Krishna is published widely in research areas of civil engineering. He has contributed to a book titled *Algorithms and Data Structures Using R*, published by Packt.

The journey of this book has been quite memorable, and I would like to give the credit to my loving wife and my baby (on the way). I like to extend special thanks to my caring parents and my adorable sister. Also, I would gratefully acknowledge the support from the entire Packt team, especially Aman Singh (Acquisition Editor) and Tejas Limkar (Content Development Editor), for striving to get the book delivered on time. I would like to extend thanks to all the reviewers, whose feedback has helped us tremendously in improving the book.

About the Reviewer

Vahid Mirjalili is a software engineer/data scientist, currently working toward his PhD in Computer Science at Michigan State University. His research at the i-PRoBE (integrated pattern recognition and biometrics) involves attribute classification of face images from large image datasets. He also teaches Python programming as well as computing concepts for data analysis and databases. With his specialty in data mining, he is very interested in predictive modeling and getting insights from data. He is also a Python developer and likes to contribute to the open source community. He enjoys making tutorials for different areas of data science and computer algorithms, which can be found in his GitHub repository, at `http://github.com/mirjalil/DataScience`.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.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.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1787121089`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

Deep learning is one of the most commonly discussed areas in machine learning due to its ability to model complex functions and learn through a variety of data sources and structures, such as cross-sectional data, sequential data, images, text, audio, and video. Also, R is one of the most popular languages used in the data science community. With the growth of deep learning, the relationship between R and deep learning is growing tremendously. Thus, *Deep Learning Cookbook in R* aims to provide a crash course in building different deep learning models. The application of deep learning is demonstrated through structured, unstructured, image, and audio case studies. The book will also cover transfer learning and how to utilize the power of GPU to enhance the computation efficiency of the deep learning model.

What this book covers

`Chapter 1`, *Getting Started*, introduces different packages that are available for building deep learning models, such as TensorFlow, MXNet, and H2O. and how to set them up to be utilized later in the book.

`Chapter 2`, *Deep Learning with R*, introduces the basics of neural network and deep learning. This chapter covers multiple recipes for building a neural network models using multiple toolboxes in R.

`Chapter 3`, *Convolution Neural Network*, covers recipes on **Convolution Neural Networks** (**CNN**) through applications in image processing and classification.

`Chapter 4`, *Data Representation Using Autoencoders*, builds the foundation of autoencoder using multiple recipes and also covers the application in data compression and denoising.

`Chapter 5`, *Generative Models in Deep learning*, extends the concept of autoencoders to generative models and covers recipes such as Boltzman machines, **restricted Boltzman machines** (**RBMs**), and deep belief networks.

`Chapter 6`, *Recurrent Neural Networks*, sets up the foundation for building machine learning models on a sequential datasets using multiple **recurrent neural networks** (**RNNs**).

`Chapter 7`, *Reinforcement Leaning*, provides the fundamentals for building reinforcement learning using **Markov Decision Process** (**MDP**) and covers both model-based learning and model-free learning.

`Chapter 8`, *Application of Deep Learning in Text-Mining*, provides an end-to-end implementation of the deep learning text mining domain.

`Chapter 9`, *Application of Deep Learning to Signal processing*, covers a detailed case study of deep learning in the signal processing domain.

`Chapter 10`, *Transfer Learning*, covers recipes for using pretrained models such as VGG16 and Inception and explains how to deploy a deep learning model using GPU.

What you need for this book

A lot of inquisitiveness, perseverance, and passion is required to build a strong background in data science. The scope of deep learning is quite broad; thus, the following backgrounds is required to effectively utilize this cookbook:

- Basics of machine learning and data analysis
- Proficiency in R programming
- Basics of Python and Docker

Lastly, you need to appreciate deep learning algorithms and know how they solve complex problems in multiple domains.

Who this book is for

This book is for data science professionals or analysts who have performed machine learning tasks and now want to explore deep learning and want a quick reference that address the pain points that crop up while implementing deep learning. Those who wish to have an edge over other deep learning professionals will find this book quite useful.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the include directive."

A block of code is set as follows:

```
[default]
 exten => s,1,Dial(Zap/1|30)
 exten => s,2,Voicemail(u100)
 exten => s,102,Voicemail(b100)
 exten => i,1,Voicemail(s0)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
 exten => s,1,Dial(Zap/1|30)
 exten => s,2,Voicemail(u100)
 exten => s,102,Voicemail(b100)
 exten => i,1,Voicemail(s0)
```

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

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
/etc/asterisk/cdr_mysql.conf
```

New terms and **important** words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book--what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply email `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

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

You can download the code files by following these steps:

1. Log in or register to our website using your email address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads** & **Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the Code Files button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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/R-Deep-Learning-Cookbook`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/RDeepLearningCookbook_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books--maybe a mistake in the text or the code--we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting Started

In this chapter, we will cover the following topics:

- Installing R with an IDE
- Installing a Jupyter Notebook application
- Starting with the basics of machine learning in R
- Setting up deep learning tools/packages in R
- Installing MXNet in R
- Installing TensorFlow in R
- Installing H2O in R
- Installing all three packages at once using Docker

Introduction

This chapter will get you started with deep learning and help you set up your systems to develop deep learning models. The chapter is more focused on giving the audience a heads-up on what is expected from the book and the prerequisites required to go through the book. The current book is intended for students or professionals who want to quickly build a background in the applications of deep learning. The book will be more practical and application-focused using R as a tool to build deep learning models.

For a detailed theory on deep learning, refer to *Deep Learning* by *Goodfellow et al. 2016.* For a machine learning background refer *Python Machine Learning* by S. Raschka, 2015.

We will use the R programming language to demonstrate applications of deep learning. You are expected to have the following prerequisites throughout the book:

- Basic R programming knowledge
- Basic understanding of Linux; we will use the Ubuntu (16.04) operating system
- Basic understanding of machine learning concepts
- For Windows or macOS, a basic understanding of Docker

Installing R with an IDE

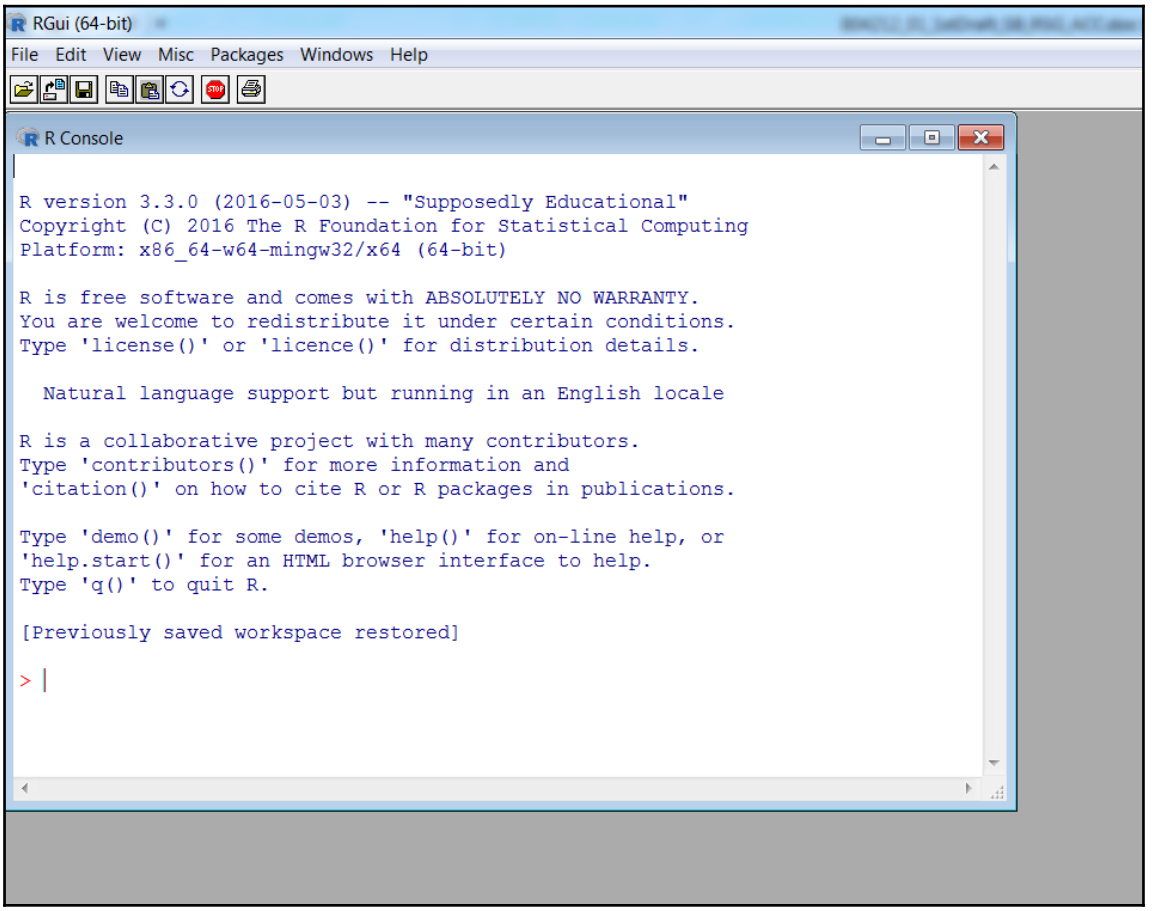
Before we begin, let's install an IDE for R. For R the most popular IDEs are Rstudio and Jupyter. Rstudio is dedicated to R whereas Jupyter provide multi-language support including R. Jupyter also provides an interactive environment and allow you to combine code, text, and graphics into a single notebook.

Getting ready

R supports multiple operating systems such as Windows, macOS X, and Linux. The installation files for R can be downloaded from any one of the mirror sites at **Comprehensive R Archive Network** (**CRAN**) at `https://cran.r-project.org/`. The CRAN is also a major repository for packages in R. The programming language R is available under both 32-bit and 64-bit architectures.

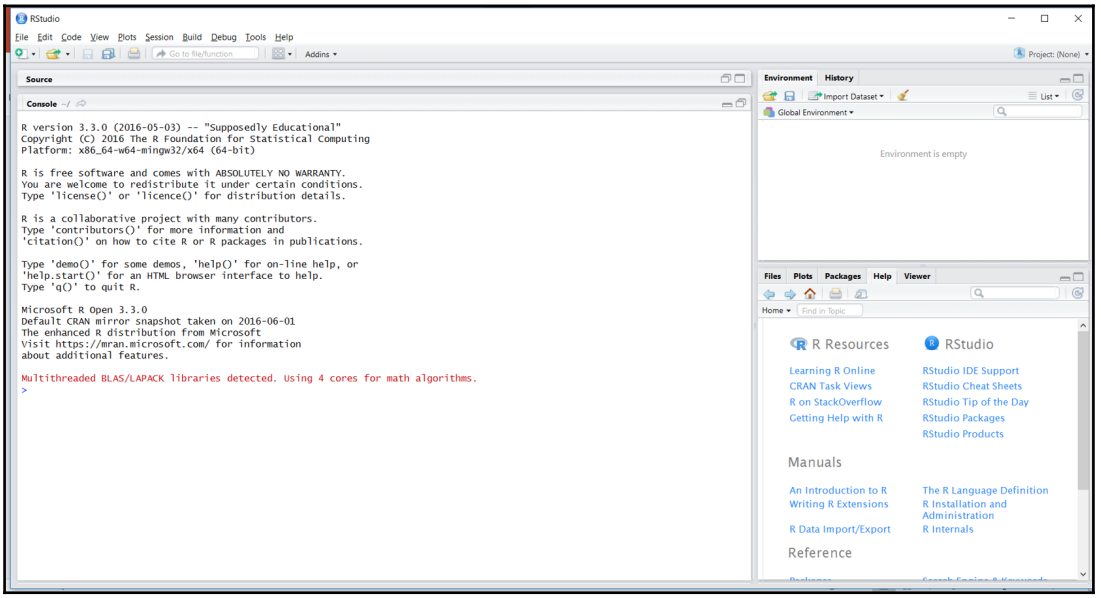
How to do it...

1. Of r-base-dev is also highly recommended as it has many inbuilt functions. It also enables the `install.packages()` command, which is used to compile and install new R packages directly from the CRAN using the **R console**. The default **R console** looks as follows:



Default R console

2. For programming purposes, an **Integrated Development Environment** (**IDE**) is recommended as it helps enhance productivity. One of the most popular open source IDEs for R is Rstudio. Rstudio also provides you with an Rstudio server, which facilitates a web-based environment to program in R. The interface for the Rstudio IDE is shown in the following screenshot:



Rstudio Integrated Development Environment for R

Installing a Jupyter Notebook application

Another famous editor these days is the Jupyter Notebook app. This app produces notebook documents that integrate documentation, code, and analysis together. It supports many computational kernels including R. It is a server, client-side, web-based application that can be accessed using a browser.

How to do it...

Jupyter Notebook can be installed using the following steps:

1. Jupyter Notebook can be installed using `pip`:

```
pip3 install --upgrade pip
pip3 install jupyter
```

2. If you have installed Anaconda, then the default computational kernel installed is Python. To install an R computation kernel in Jupyter within the same environment, type the following command in a terminal:

```
conda install -c r r-essentials
```

3. To install the R computational kernel in a new environment named `new-env` within conda, type as follows:

```
conda create -n new-env -c r r-essentials
```

4. Another way to include the R computational kernel in Jupyter Notebook uses the `IRkernel` package. To install through this process, start the R IDE. The first step is to install dependencies required for the `IRkernal` installation:

```
chooseCRANmirror(ind=55) # choose mirror for installation
install.packages(c('repr', 'IRdisplay', 'crayon', 'pbdZMQ',
'devtools'), dependencies=TRUE)
```

5. Once all the dependencies are installed from CRAN, install the `IRkernal` package from GitHub:

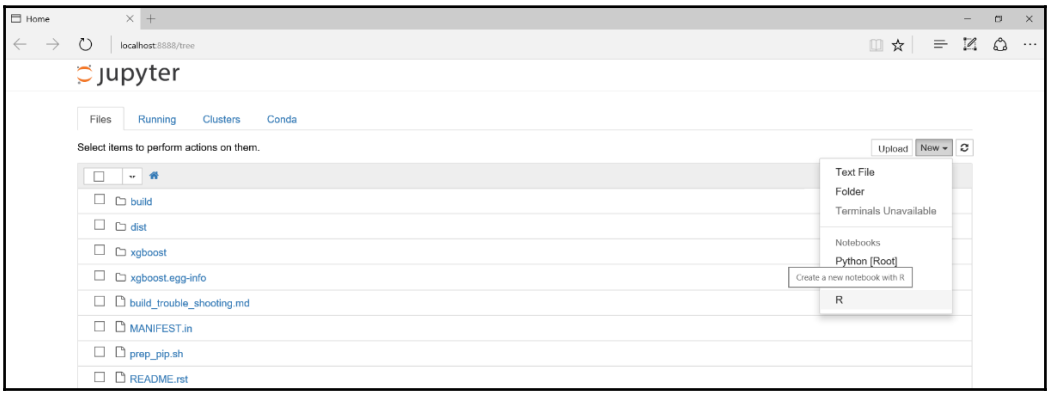
```
library(devtools)
library(methods)
options(repos=c(CRAN='https://cran.rstudio.com'))
devtools::install_github('IRkernel/IRkernel')
```

6. Once all the requirements are satisfied, the R computation kernel can be set up in Jupyter Notebook using the following script:

```
library(IRkernel)
IRkernel::installspec(name = 'ir32', displayname = 'R 3.2')
```

7. Jupyter Notebook can be started by opening a shell/terminal. Run the following command to start the Jupyter Notebook interface in the browser, as shown in the screenshot following this code:

```
jupyter notebook
```



Jupyter Notebook with the R computation engine

There's more...

R, as with most of the packages utilized in this book, is supported by most operating systems. However, you can make use of Docker or VirtualBox to set up a working environment similar to the one used in this book.

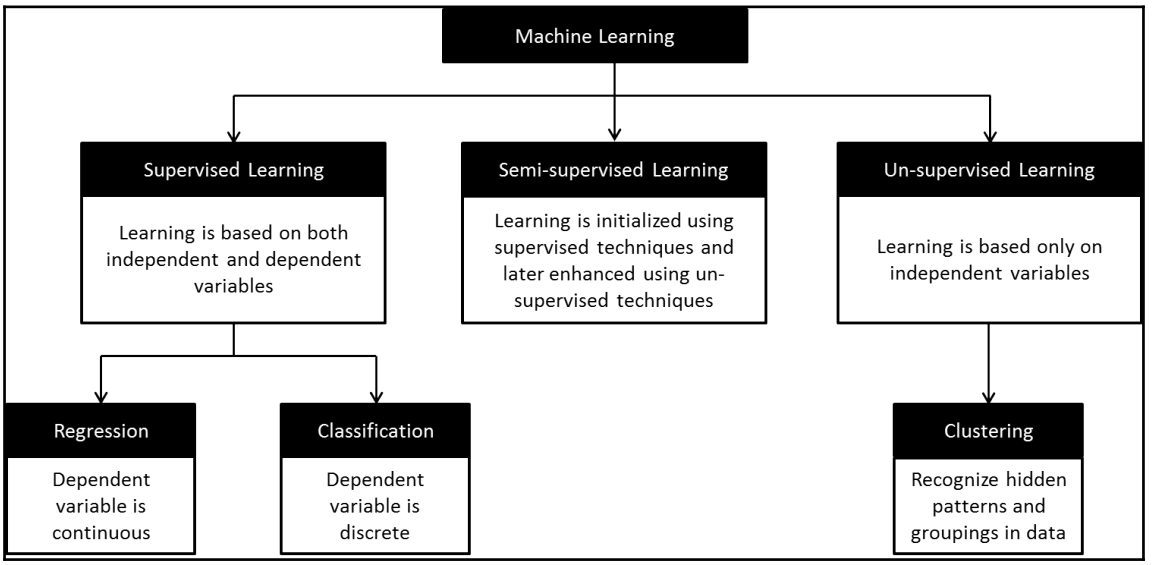
For Docker installation and setup information, refer to `https://docs.docker.com/` and select the Docker image appropriate to your operating system. Similarly, VirtualBox binaries can be downloaded and installed at `https://www.virtualbox.org/wiki/Downloads`.

Starting with the basics of machine learning in R

Deep learning is a subcategory of machine learning inspired by the structure and functioning of a human brain. In recent times, deep learning has gained a lot of traction primarily because of higher computational power, bigger datasets, and better algorithms with (artificial) intelligent learning abilities and more inquisitiveness for data-driven insights. Before we get into the details of deep learning, let's understand some basic concepts of machine learning that form the basis for most analytical solutions.

Machine learning is an arena of developing algorithms with the ability to mine natural patterns from the data such that better decisions are made using predictive insights. These insights are pertinent across a horizon of real-world applications, from medical diagnosis (using computational biology) to real-time stock trading (using computational finance), from weather forecasting to natural language processing, from predictive maintenance (in automation and manufacturing) to prescriptive recommendations (in e-commerce and e-retail), and so on.

The following figure elucidates two primary techniques of machine learning; namely, supervised learning and unsupervised learning:



Classification of different techniques in machine learning

Supervised learning: Supervised learning is a form of evidence-based learning. The evidence is the known outcome for a given input and is in turn used to train the predictive model. Models are further classified into regression and classification, based on the outcome data type. In the former, the outcome is continuous, and in the latter the outcome is discrete. Stock trading and weather forecasting are some widely used applications of regression models, and span detection, speech recognition, and image classification are some widely used applications of classification models.

Some algorithms for regression are linear regression, **Generalized Linear Models** (**GLM**), **Support Vector Regression** (**SVR**), neural networks, decision trees, and so on; in classification, we have logistic regression, **Support Vector Machines** (**SVM**), **Linear discriminant analysis** (**LDA**), Naive Bayes, nearest neighbors, and so on.

Semi-supervised learning: Semi-supervised learning is a class of supervised learning using unsupervised techniques. This technique is very useful in scenarios where the cost of labeling an entire dataset is highly impractical against the cost of acquiring and analyzing unlabeled data.

Unsupervised learning: As the name suggests, learning from data with no outcome (or supervision) is called unsupervised learning. It is a form of inferential learning based on hidden patterns and intrinsic groups in the given data. Its applications include market pattern recognition, genetic clustering, and so on.

Some widely used clustering algorithms are *k*-means, hierarchical, *k*-medoids, Fuzzy C-means, hidden markov, neural networks, and many more.

How to do it...

Let's take a look at linear regression in supervised learning:

1. Let's begin with a simple example of linear regression where we need to determine the relationship between men's height (in cms) and weight (in kgs). The following sample data represents the height and weight of 10 random men:

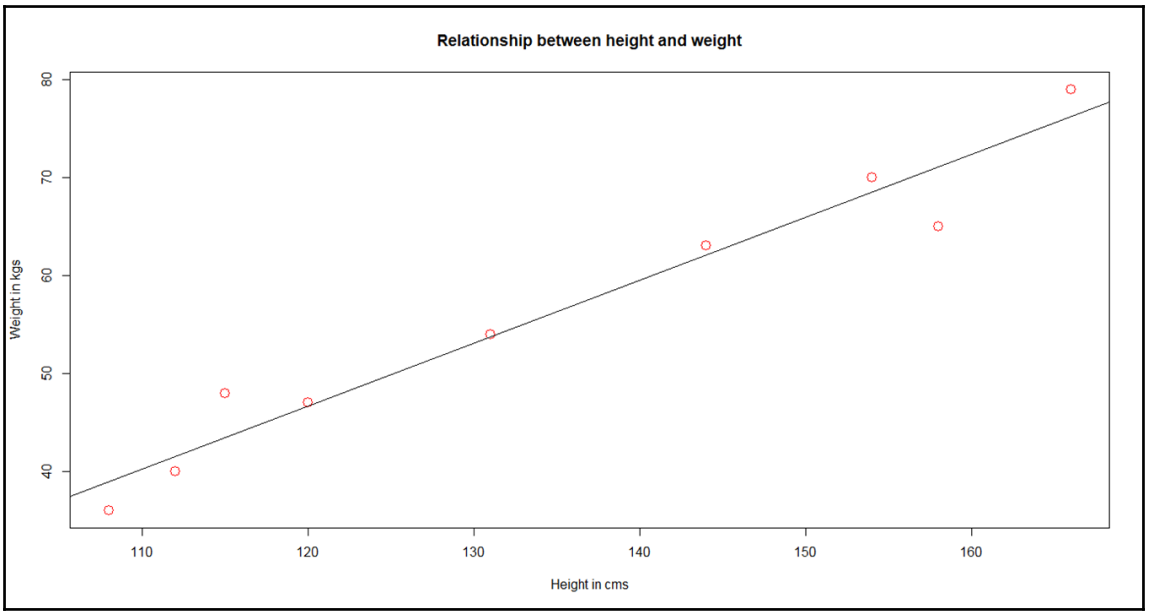
```
data <- data.frame("height" = c(131, 154, 120, 166, 108, 115,
158, 144, 131, 112),
"weight" = c(54, 70, 47, 79, 36, 48, 65,
63, 54, 40))
```

2. Now, generate a linear regression model as follows:

```
lm_model <- lm(weight ~ height, data)
```

3. The following plot shows the relationship between men's height and weight along with the fitted line:

```
plot(data, col = "red", main = "Relationship between height and
weight",cex = 1.7, pch = 1, xlab = "Height in cms", ylab = "Weight
in kgs")
abline(lm(weight ~ height, data))
```



Linear relationship between weight and height

4. In semi-supervised models, the learning is primarily initiated using labeled data (a smaller quantity in general) and then augmented using unlabeled data (a larger quantity in general).

Let's perform K-means clustering (unsupervised learning) on a widely used dataset, iris.

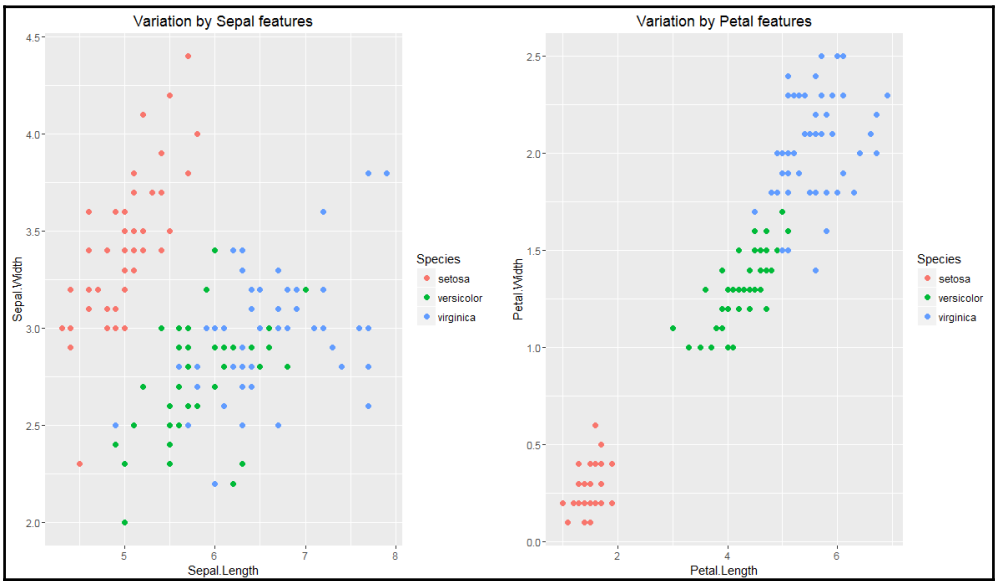
1. This dataset consists of three different species of iris (**Setosa**, **Versicolor**, and **Virginica**) along with their distinct features such as sepal length, sepal width, petal length, and petal width:

```
data(iris)
head(iris)
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1 5.1 3.5 1.4 0.2 setosa
2 4.9 3.0 1.4 0.2 setosa
```

```
3 4.7 3.2 1.3 0.2 setosa
4 4.6 3.1 1.5 0.2 setosa
5 5.0 3.6 1.4 0.2 setosa
6 5.4 3.9 1.7 0.4 setosa
```

2. The following plots show the variation of features across irises. Petal features show a distinct variation as against sepal features:

```
library(ggplot2)
library(gridExtra)
plot1 <- ggplot(iris, aes(Sepal.Length, Sepal.Width, color =
Species))
  geom_point(size = 2)
  ggtitle("Variation by Sepal features")
plot2 <- ggplot(iris, aes(Petal.Length, Petal.Width, color =
  Species))
   geom_point(size = 2)
   ggtitle("Variation by Petal features")
grid.arrange(plot1, plot2, ncol=2)
```



Variation of sepal and petal features by length and width

3. As petal features show a good variation across irises, let's perform K-means clustering using the petal length and petal width:

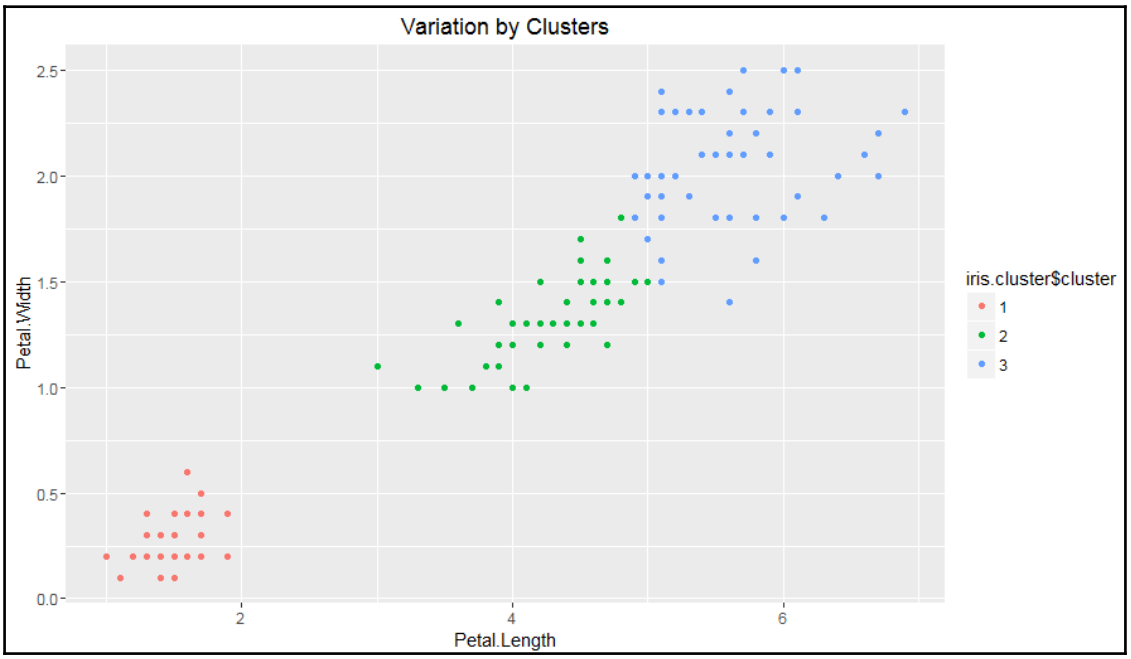
```
set.seed(1234567)
```

```
iris.cluster <- kmeans(iris[, c("Petal.Length","Petal.Width")],
 3, nstart = 10)
iris.cluster$cluster <- as.factor(iris.cluster$cluster)
```

4. The following code snippet shows a cross-tab between clusters and species (irises). We can see that cluster 1 is primarily attributed with setosa, cluster 2 with versicolor, and cluster 3 with virginica:

```
> table(cluster=iris.cluster$cluster,species= iris$Species)
species
cluster setosa versicolor virginica
1 50 0 0
2 0 48 4
3 0 2 46
ggplot(iris, aes(Petal.Length, Petal.Width, color =
iris.cluster$cluster)) + geom_point() + ggtitle("Variation by
Clusters")
```

5. The following plot shows the distribution of clusters:



Variation of irises across three clusters

How it works...

Model evaluation is a key step in any machine learning process. It is different for supervised and unsupervised models. In supervised models, predictions play a major role; whereas in unsupervised models, homogeneity within clusters and heterogeneity across clusters play a major role.

Some widely used model evaluation parameters for regression models (including cross validation) are as follows:

- Coefficient of determination
- Root mean squared error
- Mean absolute error
- Akaike or Bayesian information criterion

Some widely used model evaluation parameters for classification models (including cross validation) are as follows:

- Confusion matrix (accuracy, precision, recall, and F1-score)
- Gain or lift charts
- Area under ROC (receiver operating characteristic) curve
- Concordant and discordant ratio

Some of the widely used evaluation parameters of unsupervised models (clustering) are as follows:

- Contingency tables
- Sum of squared errors between clustering objects and cluster centers or centroids
- Silhouette value
- Rand index
- Matching index
- Pairwise and adjusted pairwise precision and recall (primarily used in NLP)

Bias and variance are two key error components of any supervised model; their trade-off plays a vital role in model tuning and selection. Bias is due to incorrect assumptions made by a predictive model while learning outcomes, whereas variance is due to model rigidity toward the training dataset. In other words, higher bias leads to underfitting and higher variance leads to overfitting of models.

In bias, the assumptions are on target functional forms. Hence, this is dominant in parametric models such as linear regression, logistic regression, and linear discriminant analysis as their outcomes are a functional form of input variables.

Variance, on the other hand, shows how susceptible models are to change in datasets. Generally, target functional forms control variance. Hence, this is dominant in non-parametric models such as decision trees, support vector machines, and K-nearest neighbors as their outcomes are not directly a functional form of input variables. In other words, the hyperparameters of non-parametric models can lead to overfitting of predictive models.

Setting up deep learning tools/packages in R

The major deep learning packages are developed in C/C++ for efficiency purposes and wrappers are developed in R to efficiently develop, extend, and execute deep learning models.

A lot of open source deep learning libraries are available. The prominent libraries in this area are as follows:

- Theano
- TensorFlow
- Torch
- Caffe

There are other prominent packages available on the market such as H2O, CNTK (Microsoft Cognitive Toolkit), darch, Mocha, and ConvNetJS. There are a lot of wrappers that are developed around these packages to support the easy development of deep learning models, such as Keras and Lasagne in Python and MXNet, both supporting multiple languages.

How to do it...

1. This chapter will cover the MXNet and TensorFlow packages (developed in C++ and CUDA for a highly optimized performance in GPU).
2. Additionally, the `h2o` package will be used to develop some deep learning models. The `h2o` package in R is implemented as a REST API, which connects to the H2O server (it runs as **Java Virtual Machines** (**JVM**)). We will provide quick setup instructions for these packages in the following sections

Installing MXNet in R

This section will cover the installation of MXNet in R.

Getting ready

The MXNet package is a lightweight deep learning architecture supporting multiple programming languages such as R, Python, and Julia. From a programming perspective, it is a combination of symbolic and imperative programming with support for CPU and GPU.

The CPU-based MXNet in R can be installed using the prebuilt binary package or the source code where the libraries need to be built. In Windows/mac, prebuilt binary packages can be download and installed directly from the R console. MXNet requires the R version to be 3.2.0 and higher. The installation requires the `drat` package from CRAN. The `drat` package helps maintain R repositories and can be installed using the `install.packages()` command.

To install MXNet on Linux (13.10 or later), the following are some dependencies:

- Git (to get the code from GitHub)
- libatlas-base-dev (to perform linear algebraic operations)
- libopencv-dev (to perform computer vision operations)

To install MXNet with a GPU processor, the following are some dependencies:

- Microsoft Visual Studio 2013
- The NVIDIA CUDA Toolkit
- The MXNet package
- cuDNN (to provide a deep neural network library)

Another quick way to install `mxnet` with all the dependencies is to use the prebuilt Docker image from the `chstone` repository. The `chstone/mxnet-gpu` Docker image will be installed using the following tools:

- MXNet for R and Python
- Ubuntu 16.04
- CUDA (Optional for GPU)
- cuDNN (Optional for GPU)

How to do it...

1. The following R command installs MXNet using prebuilt binary packages, and is hassle-free. The `drat` package is then used to add the `dlmc` repository from git followed by the `mxnet` installation:

```
install.packages("drat", repos="https://cran.rstudio.com")
drat:::addRepo("dmlc")
install.packages("mxnet")
```

2. The following code helps install MXNet in Ubuntu (V16.04). The first two lines are used to install dependencies and the remaining lines are used to install MXNet, subject to the satisfaction of all the dependencies:

```
sudo apt-get update
sudo apt-get install -y build-essential git libatlas-base-dev
libopencv-dev
git clone https://github.com/dmlc/mxnet.git ~/mxnet --recursive
cd ~/mxnet
cp make/config.mk .
echo "USE_BLAS=openblas" >>config.mk
make -j$(nproc)
```

3. If MXNet is to be built for GPU, the following `config` needs to be updated before the `make` command:

```
echo "USE_CUDA=1" >>config.mk
echo "USE_CUDA_PATH=/usr/local/cuda" >>config.mk
echo "USE_CUDNN=1" >>config.mk
```

A detailed installation of MXNet for other operating systems can be found at `http://mxnet.io/get_started/setup.html`.

4. The following command is used to install MXNet (GPU-based) using Docker with all the dependencies:

```
docker pull chstone/mxnet-gpu
```

Installing TensorFlow in R

This section will cover another very popular open source machine learning package, TensorFlow, which is very effective in building deep learning models.

Getting ready

TensorFlow is another open source library developed by the Google Brain Team to build numerical computation models using data flow graphs. The core of TensorFlow was developed in C++ with the wrapper in Python. The `tensorflow` package in R gives you access to the TensorFlow API composed of Python modules to execute computation models. TensorFlow supports both CPU- and GPU-based computations.

The `tensorflow` package in R calls the Python tensorflow API for execution, which is essential to install the `tensorflow` package in both R and Python to make R work. The following are the dependencies for `tensorflow`:

- Python 2.7 / 3.x
- R (>3.2)
- devtools package in R for installing TensorFlow from GitHub
- TensorFlow in Python
- pip

How to do it...

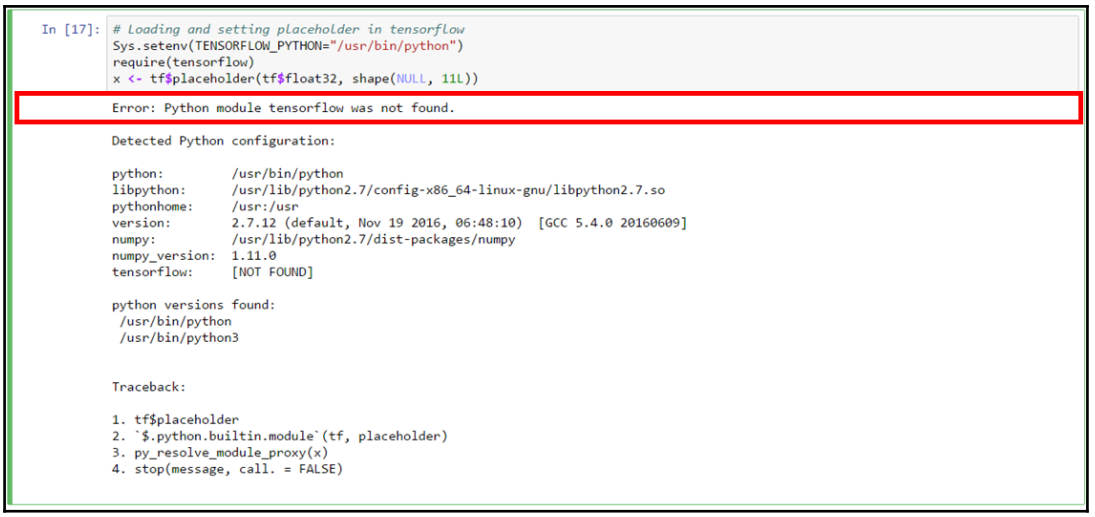
1. Once all the mentioned dependencies are installed, `tensorflow` can be installed from `devtools` directly using the `install_github` command as follows:

```
devtools::install_github("rstudio/tensorflow")
```

2. Before loading `tensorflow` in R, you need to set up the path for Python as the system environment variable. This can be done directly from the R environment, as shown in the following command:

```
Sys.setenv(TENSORFLOW_PYTHON="/usr/bin/python")
library(tensorflow)
```

If the Python `tensorflow` module is not installed, R will give the following error:



Error raised by R if tensorflow in Python is not installed

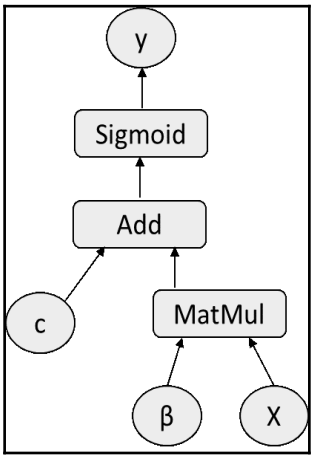
`tensorflow` in Python can be installed using `pip`:

```
pip install tensorflow # Python 2.7 with no GPU support
pip3 install tensorflow # Python 3.x with no GPU support
pip install tensorflow-gpu # Python 2.7 with GPU support
pip3 install tensorflow-gpu # Python 3.x with GPU support
```

How it works...

TensorFlow follows directed graph philosophy to set up computational models where mathematical operations are represented as nodes with each node supporting multiple input and output while the edges represent the communication of data between nodes. There are also edges known as **control dependencies** in TensorFlow that do not represent the data flow; rather the provide information related to control dependence such as node for the control dependence must finish processing before the destination node of control dependence starts executing.

An example TensorFlow graph for logistic regression scoring is shown in the following diagram:



TensorFlow graph for logistic regression

The preceding figure illustrates the TensorFlow graph to score logistic regression with optimized weights:

$$y = \frac{1}{1 + e^{-(\beta X + C)}}$$

The *MatMul* node performs matrix multiplication between input feature matrix *X* and optimized weight β. The constant *C* is then added to the output from the *MatMul* node. The output from *Add* is then transformed using the *Sigmoid* function to output *Pr(y=1|X)*.

See also

Get started with TensorFlow in R using resources at `https://rstudio.github.io/tensorflow/`.

Installing H2O in R

H2O is another very popular open source library to build machine learning models. It is produced by H2O.ai and supports multiple languages including R and Python. The H2O package is a multipurpose machine learning library developed for a distributed environment to run algorithms on big data.

Getting ready

To set up H2O, the following systems are required:

- 64-bit Java Runtime Environment (version 1.6 or later)
- Minimum 2 GB RAM

H2O from R can be called using the `h2o` package. The `h2o` package has the following dependencies:

- RCurl
- rjson
- statmod
- survival
- stats
- tools
- utils
- methods

For machines that do not have curl-config installed, the RCurl dependency installation will fail in R and curl-config needs to be installed outside R.

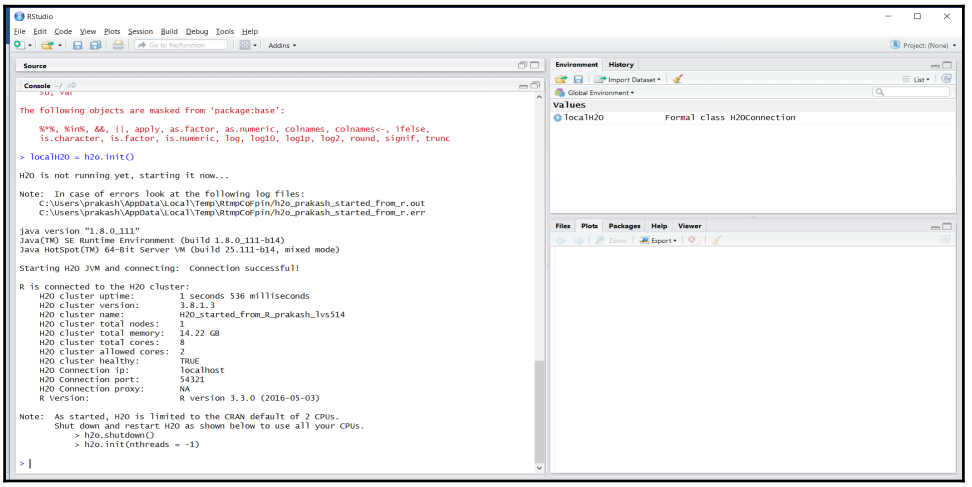
How to do it...

1. H2O can be installed directly from CRAN with the dependency parameter TRUE to install all CRAN-related `h2o` dependencies. This command will install all the R dependencies required for the `h2o` package:

```
install.packages("h2o", dependencies = T)
```

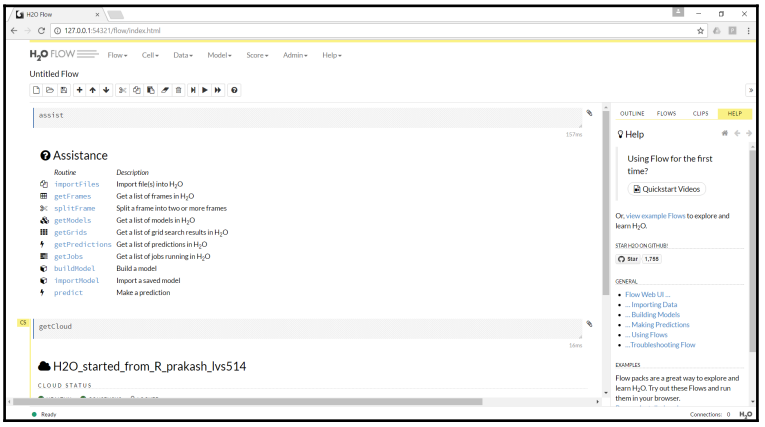
2. The following command is used to call the `h2o` package in the current R environment. The first-time execution of the `h2o` package will automatically download the JAR file before launching H2O, as shown in the following figure:

```
library(h2o)
localH2O = h2o.init()
```



Starting H2O cluster

3. The H2O cluster can be accessed using **cluster ip** and **port information**. The current H2O cluster is running on localhost at port `54321`, as shown in the following screenshot:



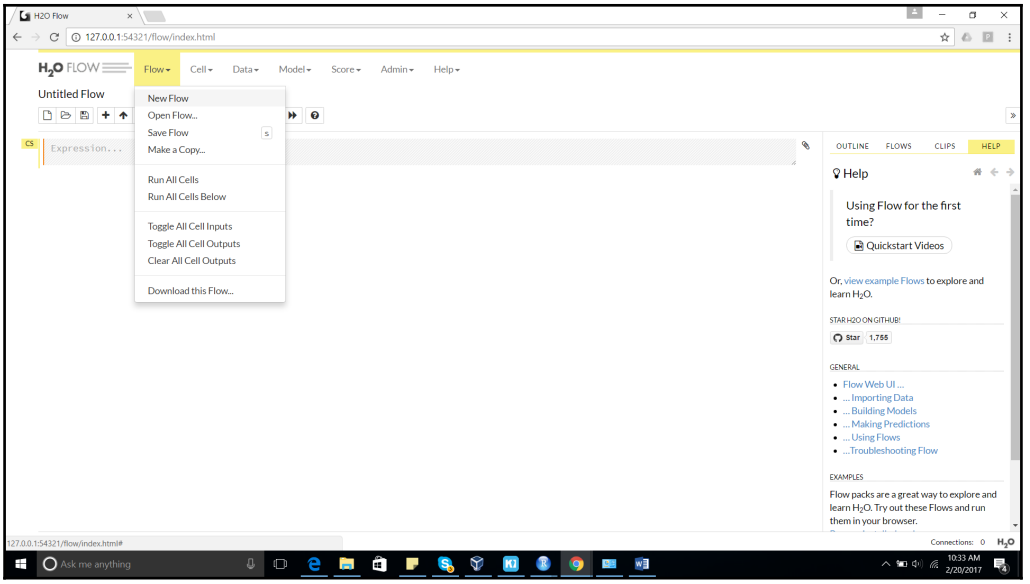
H2O cluster running in the browser

Models in H2O can be developed interactively using a browser or scripting from R. H2O modeling is like creating a Jupyter Notebook but you create a flow with different operations such as importing data, splitting data, setting up a model, and scoring.

How it works...

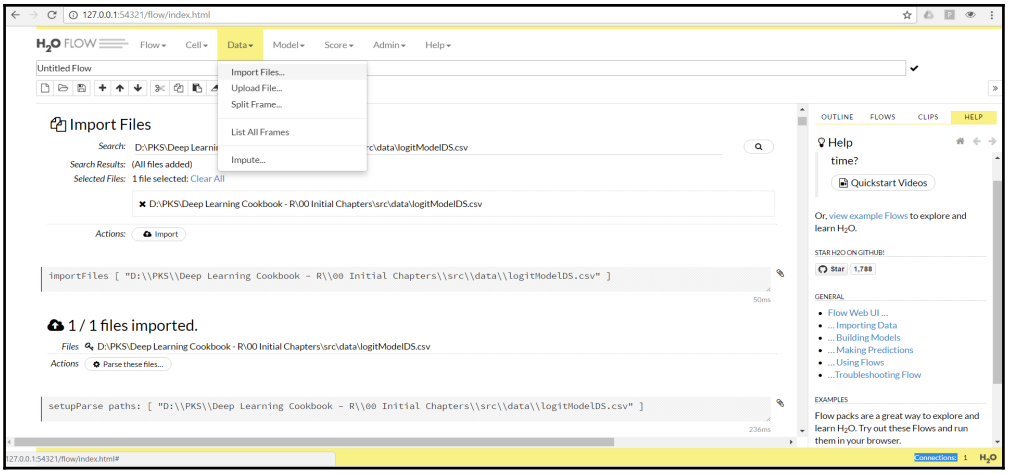
Let's build a logistic regression interactively using the H2O browser.

1. Start a new flow, as shown in the following screenshot:



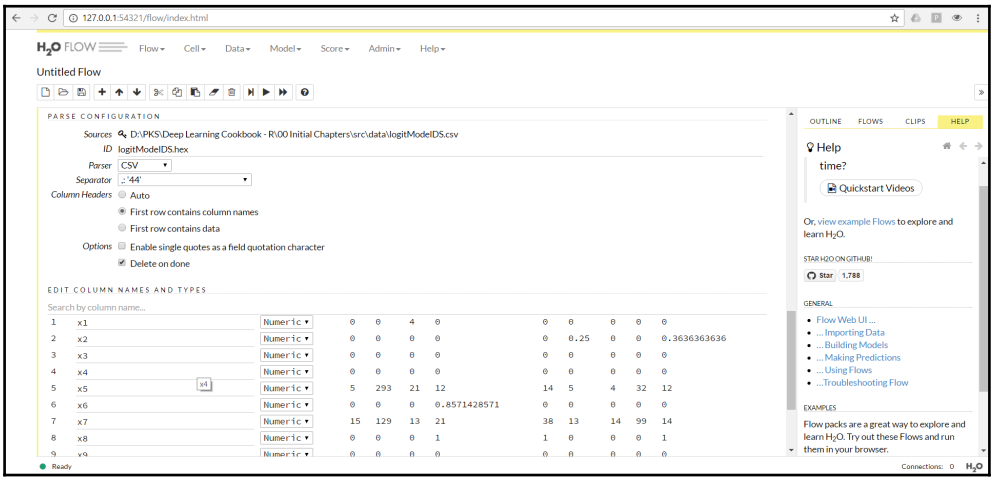
Creating a new flow in H2O

2. Import a dataset using the Data menu, as shown in the following screenshot:



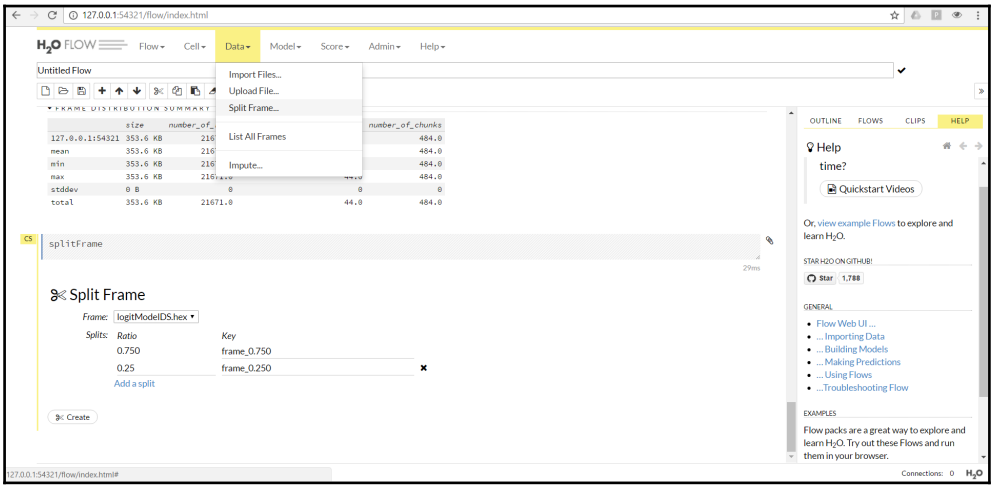
Importing files to the H2O environment

3. The imported file in H2O can be parsed into the hex format (the native file format for H2O) using the **Parse these files** action, which will appear once the file is imported to the H2O environment:



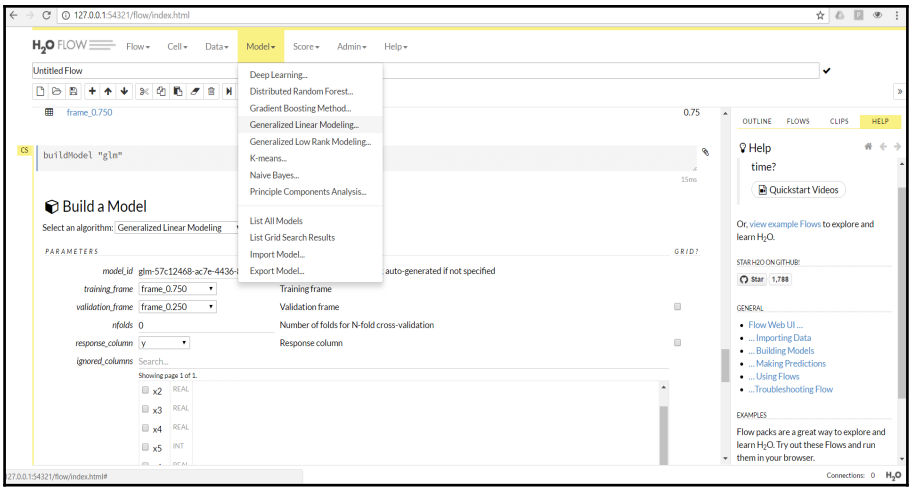
Parsing the file to the hex format

4. The parsed data frame in H2O can be split into training and validation using the **Data | Split Frame** action, as shown in the following screenshot:



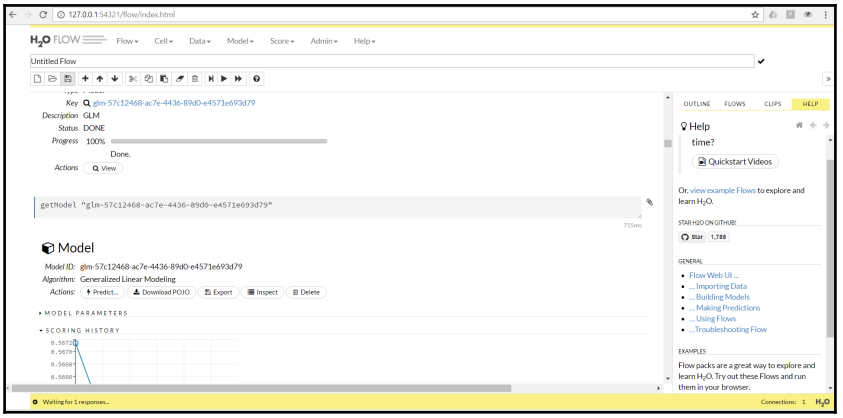
Splitting the dataset into training and validation

5. Select the model from the **Model** menu and set up the model-related parameters. An example for a glm model is seen in the following screenshot:



Building a model in H2O

6. The **Score** | **predict** action can be used to score another hex data frame in H2O:



Scoring in H2O

There's more...

For more complicated scenarios that involve a lot of preprocessing, H2O can be called from R directly. This book will focus more on building models using H2O from R directly. If H2O is set up at a different location instead of localhost, then it can be connected within R by defining the correct `ip` and `port` at which the cluster is running:

```
localH2O = h2o.init(ip = "localhost", port = 54321, nthreads = -1)
```

Another critical parameter is the number of threads to be used to build the model; by default, *n* threads are set to -2, which means that two cores will be used. The value of -1 for *n* threads will make use of all available cores.

`http://docs.h2o.ai/h2o/latest-stable/index.html#gettingstarted` is very good using H2O in interactive mode.

Installing all three packages at once using Docker

Docker is a software-contained platform that is used to host multiple software or apps side by side in isolated containers to get better computing density. Unlike virtual machines, containers are built only using libraries and the settings required by any software but do not bundle the entire operating system, thus making it lightweight and efficient.

Getting ready

Setting up all three packages could be quite cumbersome depending on the operating system utilized. The following dockerfile code can be used to set up an environment with `tensorflow`, `mxnet` with GPU, and `h2o` installed with all the dependencies:

```
FROM chstone/mxnet-gpu:latest
MAINTAINER PKS Prakash <prakash5801>


# Install dependencies
RUN apt-get update && apt-get install -y
 python2.7
 python-pip
 python-dev
 ipython
 ipython-notebook
 python-pip
 default-jre


# Install pip and Jupyter notebook
RUN pip install --upgrade pip &&
 pip install jupyter

# Add R to Jupyter kernel
RUN Rscript -e "install.packages(c('repr', 'IRdisplay', 'crayon',
'pbdZMQ'), dependencies=TRUE, repos='https://cran.rstudio.com')" &&
 Rscript -e "library(devtools); library(methods);
options(repos=c(CRAN='https://cran.rstudio.com'));
devtools::install_github('IRkernel/IRkernel')" &&
 Rscript -e "library(IRkernel); IRkernel::installspec(name = 'ir32',
displayname = 'R 3.2')"

# Install H2O
RUN Rscript -e "install.packages('h2o', dependencies=TRUE,
```

```
repos='http://cran.rstudio.com')"

# Install tensorflow fixing the proxy port
RUN pip install tensorflow-gpu
RUN Rscript -e "library(devtools);
devtools::install_github('rstudio/tensorflow')"
```

The current image is created on top of the `chstone/mxnet-gpu` Docker image.

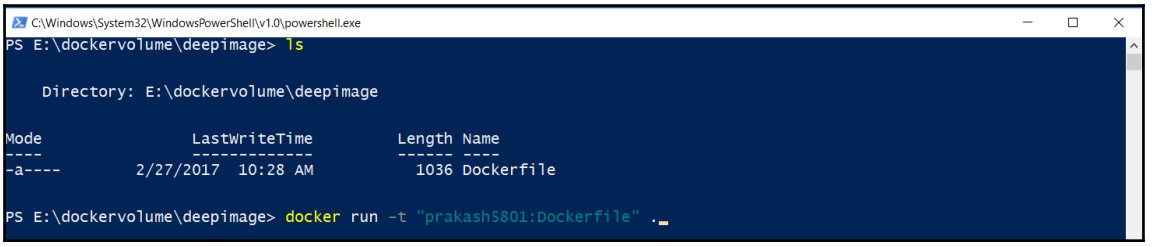
The chstone/mxnet-gpu is a docker hub repository at `https://hub.docker.com/r/chstone/mxnet-gpu/`.

How to do it...

Docker will all dependencies can be installed using following steps:

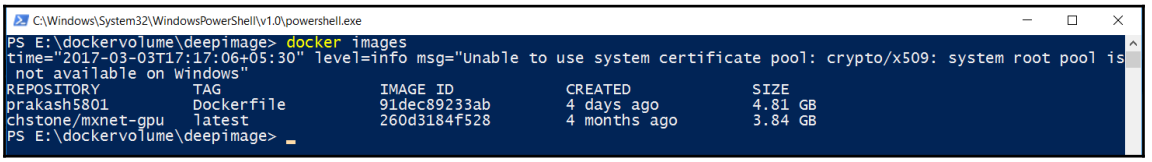
1. Save the preceding code to a location with a name, say, `Dockerfile`.
2. Using the command line, go to the file location and use the following command and it is also shown in the screenshot after the command:

```
docker run -t "TagName:FILENAME"
```



Building the docker image

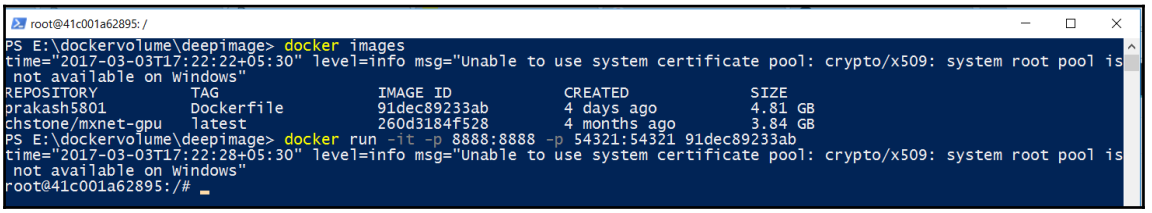
3. Access the image using the `docker images` command as follows:



View docker images

4. Docker images can be executed using the following command:

```
docker run -it -p 8888:8888 -p 54321:54321 <<IMAGE ID>>
```



Running a Docker image

Here, the option *-i* is for interactive mode and *-t* is to allocate *--tty*. The option *-p* is used to forward the port. As we will be running Jupyter on port `8888` and H2O on `54321`, we have forwarded both ports to accessible from the local browser.

There's more...

More options for Docker can be checked out using `docker run --help`.

2
Deep Learning with R

This chapter will build a foundation for neural networks followed by deep learning foundation and trends. We will cover the following topics:

- Starting with logistic regression
- Introducing the dataset
- Performing logistic regression using H2O
- Performing logistic regression using TensorFlow
- Visualizing TensorFlow graphs
- Starting with multilayer perceptrons
- Setting up a neural network using H2O
- Tuning hyper-parameters using grid searches in H2O
- Setting up a neural network using MXNet
- Setting up a neural network using TensorFlow

Starting with logistic regression

Before we delve into neural networks and deep learning models, let's take a look at logistic regression, which can be viewed as a single layer neural network. Even the **sigmoid** function commonly used in logistic regression is used as an activation function in neural networks.

Getting ready

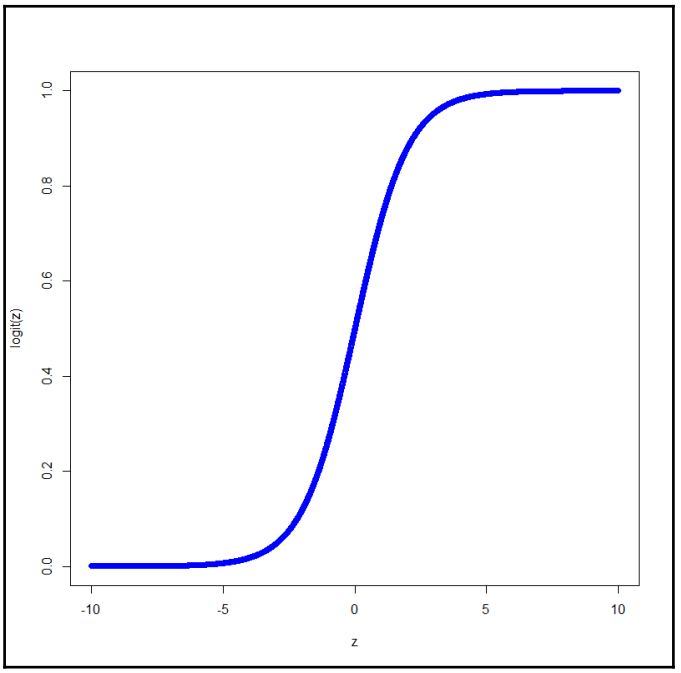
Logistic regression is a supervised machine learning approach for the classification of dichotomous/ordinal (order discrete) categories.

How to do it...

Logistic regression serves as a building block for complex neural network models using sigmoid as an activation function. The logistic function (or sigmoid) can be represented as follows:

$$y = \frac{1}{1 + e^{-z}}$$

The preceding sigmoid function forms a continuous curve with a value bound between [0, 1], as illustrated in the following screenshot:

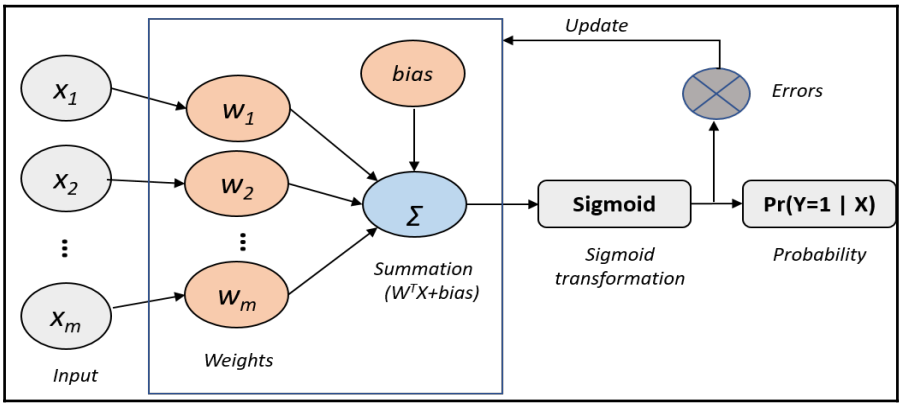


Sigmoid functional form

The formulation of a logistic regression model can be written as follows:

$$\Pr(y = 1|\mathbf{X}) = \frac{1}{1 + e^{-(W^T X + b)}}$$

Here, W is the weight associated with features $X = [x_1, x_2, ..., x_m]$ and b is the model intercept, also known as the model bias. The whole objective is to optimize W for a given loss function such as cross entropy. Another view of the logistic regression model to attain $Pr(y=1|\mathbf{X})$ is shown in the following figure:



Logistic regression architecture with the sigmoid activation function

Introducing the dataset

This recipe shows how to prepare a dataset to be used to demonstrate different models.

Getting ready

As logistic regression is a linear classifier, it assumes linearity in independent variables and log odds. Thus, in scenarios where independent features are linear-dependent on log odds, the model performs very well. Higher-order features can be included in the model to capture nonlinear behavior. Let's see how to build logistic regression models using major deep learning packages as discussed in the previous chapter. Internet connectivity will be required to download the dataset from the UCI repository.

How to do it...

In this chapter, the Occupancy Detection dataset from the **UC Irivine ML repository** is used to build models on logistic regression and neural networks. It is an experimental dataset primarily used for binary classification to determine whether a room is occupied (1) or not occupied (0) based on multivariate predictors as described in the following table. The contributor of the dataset is *Luis Candanedo* from UMONS.

Download the dataset at `https://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+`.

There are three datasets tobe downloaded; however, we will use `datatraining.txt` for training/cross validation purposes and `datatest.txt` for testing purposes.

The dataset has seven attributes (including response occupancy) with 20,560 instances. The following table summarizes the attribute information:

Attribute	**Description**	**Characteristic**
Date time	Year-month-day hour:minute:second format	Date
Temperature	In Celsius	Real
Relative Humidity	In %	Real
Light	In Lux	Real
CO2	In ppm	Real
Humidity ratio	Derived quantity from temperature and relative humidity, in kg water-vapor/kg-air	Real
Occupancy	0 for not occupied; 1 for occupied	Binary class

Performing logistic regression using H2O

Generalized linear models (**GLM**) are widely used in both regression- and classification-based predictive analysis. These models optimize using maximum likelihood and scale well with larger datasets. In H2O, GLM has the flexibility to handle both L1 and L2 penalties (including elastic net). It supports Gaussian, Binomial, Poisson, and Gamma distributions of dependent variables. It is efficient in handling categorical variables, computing full regularizations, and performing distributed *n-fold* cross validations to control for model overfitting. It has a feature to optimize hyperparameters such as elastic net (α) using distributed grid searches along with handling upper and lower bounds for predictor attribute coefficients. It can also handle automatic missing value imputation. It uses the Hogwild method for optimization, a parallel version of stochastic gradient descent.

Getting ready

The previous chapter provided the details for the installation of H2O in R along with a working example using its web interface. To start modeling, load the `h2o` package in the R environment:

```
require(h2o)
```

Then, initialize a single-node H2O instance using the `h2o.init()` function on eight cores and instantiate the corresponding client module on the IP address `localhost` and port number `54321`:

```
localH2O = h2o.init(ip = "localhost", port = 54321, startH2O =
TRUE,min_mem_size = "20G",nthreads = 8)
```

The H2O package has dependency on the Java JRE. Thus, it should be pre-installed before executing the initialization command.

How to do it...

The section will demonstrate steps to build the GLM model using H2O.

1. Now, load the occupancy train and test datasets in R:

```
# Load the occupancy data
occupancy_train <-
read.csv("C:/occupation_detection/datatraining.txt",stringsAsFactor
s = T)
occupancy_test <-
read.csv("C:/occupation_detection/datatest.txt",stringsAsFactors =
T)
```

2. The following independent (`x`) and dependent (`y`) variables will be used to model GLM:

```
# Define input (x) and output (y) variables"
x = c("Temperature", "Humidity", "Light", "CO2", "HumidityRatio")
y = "Occupancy"
```

3. Based on the requirement for H2O, convert the dependent variables into factors as follows:

```
# Convert the outcome variable into factor
occupancy_train$Occupancy <- as.factor(occupancy_train$Occupancy)
occupancy_test$Occupancy <- as.factor(occupancy_test$Occupancy)
```

4. Then, convert the datasets to H2OParsedData objects:

```
occupancy_train.hex <- as.h2o(x = occupancy_train,
destination_frame = "occupancy_train.hex")
occupancy_test.hex <- as.h2o(x = occupancy_test, destination_frame
= "occupancy_test.hex")
```

5. Once the data is loaded and converted to H2OParsedData objects, run a GLM model using the `h2o.glm` function. In the current setup, we intend to train for parameters such as five-fold cross validation, elastic net regularization ($\alpha = 5$), and optimal regularization strength (with `lamda_search = TRUE`):

```
# Train the model
occupancy_train.glm <- h2o.glm(x = x, # Vector of predictor
variable names
                               y = y, # Name of response/dependent
variable
                               training_frame =
occupancy_train.hex, # Training data
                               seed = 1234567,        # Seed for
random numbers
                               family = "binomial",   # Outcome
variable
                               lambda_search = TRUE,  # Optimum
regularisation lambda
                               alpha = 0.5,           # Elastic net
regularisation
                               nfolds = 5             # N-fold
cross validation
                               )
```

6. In addition to the preceding command, you can also define other parameters to fine-tune the model performance. The following list does not cover all the functional parameters, but covers some based on importance. The complete list of parameters can be found in the documentation of the `h2o` package.

    - Specify the strategy of generating cross-validation samples such as random sampling, stratified sampling, modulo sampling, and auto (select) using fold_assignment. The sampling can also be performed on a particular attribute by specifying the column name (fold_column).
    - Option to handle skewed outcomes (imbalanced data) by specifying weights to each observation using weights_column or performing over/under sampling using balance_classes.
    - Option to handle missing values by mean imputation or observation skip using missing_values_handling.
    - Option to restrict the coefficients to be non-negative using non_negative and constrain their values using beta_constraints.

- Option to provide prior probability for y==1(logistic regression) in the case of sampled data if its mean of response does not reflect the reality (prior).
- Specify the variables to be considered for interactions (interactions).

How it works...

The performance of the model can be assessed using many metrics such as accuracy, **Area under curve** (**AUC**), misclassification error (%), misclassification error count, F1-score, precision, recall, specificity, and so on. However, in this chapter, the assessment of model performance is based on AUC.

The following is the training and cross validation accuracy of the trained model:

```
# Training accuracy (AUC)
> occupancy_train.glm@model$training_metrics@metrics$AUC
[1] 0.994583

# Cross validation accuracy (AUC)
> occupancy_train.glm@model$cross_validation_metrics@metrics$AUC
[1] 0.9945057
```

Now, let's assess the performance of the model on test data. The following code helps in predicting the outcome of the test data:

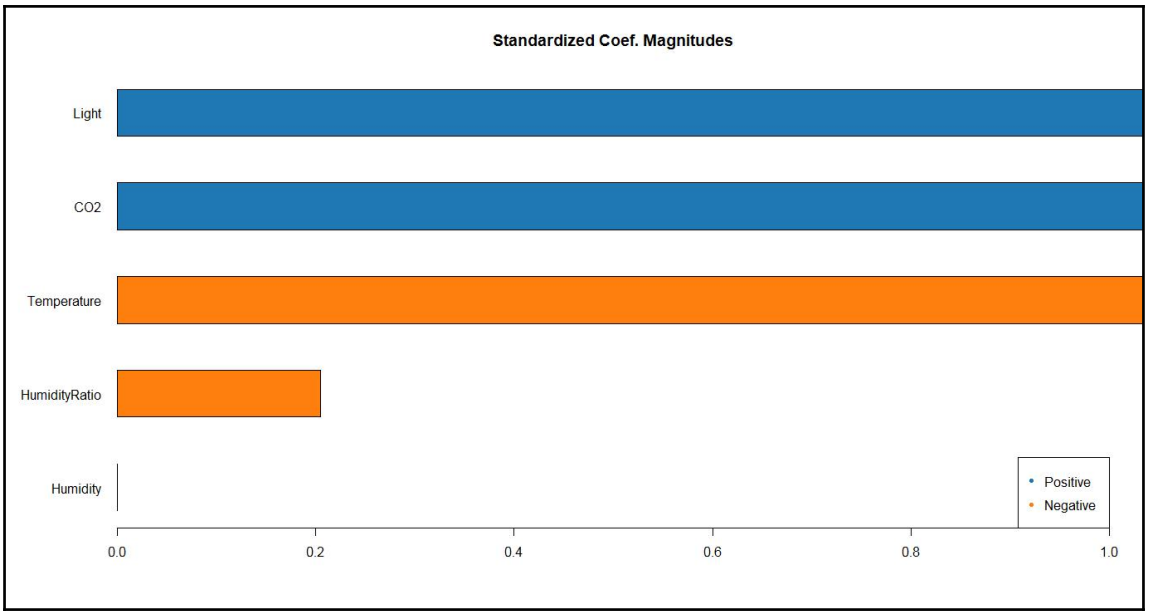
```
# Predict on test data
yhat <- h2o.predict(occupancy_train.glm, occupancy_test.hex)
```

Then, evaluate the `AUC` value based on the actual test outcome as follows:

```
# Test accuracy (AUC)
> yhat$pmax <- pmax(yhat$p0, yhat$p1, na.rm = TRUE)
> roc_obj <- pROC::roc(c(as.matrix(occupancy_test.hex$Occupancy)),
                       c(as.matrix(yhat$pmax)))
> auc(roc_obj)
Area under the curve: 0.9915
```

In H2O, one can also compute variable importance from the GLM model, as shown in the figure following this command:

```
#compute variable importance and performance
h2o.varimp_plot(occupancy_train.glm, num_of_features = 5)
```



Variable importance using H2O

See also

More functional parameters for `h2o.glm` can be found at `https://www.rdocumentation.org/packages/h2o/versions/3.10.3.6/topics/h2o.gbm`.

Performing logistic regression using TensorFlow

In this section, we will cover the application of TensorFlow in setting up a logistic regression model. The example will use a similar dataset to that used in the H2O model setup.

Getting ready

The previous chapter provided details for the installation of TensorFlow. The code for this section is created on Linux but can be run on any operating system. To start modeling, load the `tensorflow` package in the environment. R loads the default TensorFlow environment variable and also the NumPy library from Python in the `np` variable:

```
library("tensorflow") # Load TensorFlow
np <- import("numpy") # Load numpy library
```

How to do it...

The data is imported using a standard function from R, as shown in the following code.

1. The data is imported using the `read.csv` file and transformed into the matrix format followed by selecting the features used to model as defined in `xFeatures` and `yFeatures`. The next step in TensorFlow is to set up a graph to run optimization:

```
# Loading input and test data
xFeatures = c("Temperature", "Humidity", "Light", "CO2",
"HumidityRatio")
yFeatures = "Occupancy"
occupancy_train <-
as.matrix(read.csv("datatraining.txt",stringsAsFactors = T))
occupancy_test <-
as.matrix(read.csv("datatest.txt",stringsAsFactors = T))

# subset features for modeling and transform to numeric values
occupancy_train<-apply(occupancy_train[, c(xFeatures, yFeatures)],
2, FUN=as.numeric)
```

```
occupancy_test<-apply(occupancy_test[, c(xFeatures, yFeatures)], 2,
FUN=as.numeric)

# Data dimensions
nFeatures<-length(xFeatures)
nRow<-nrow(occupancy_train)
```

2. Before setting up the graph, let's reset the graph using the following command:

```
# Reset the graph
tf$reset_default_graph()
```

3. Additionally, let's start an interactive session as it will allow us to execute variables without referring to the session-to-session object:

```
# Starting session as interactive session
sess<-tf$InteractiveSession()
```

4. Define the logistic regression model in TensorFlow:

```
# Setting-up Logistic regression graph
x <- tf$constant(unlist(occupancy_train[, xFeatures]),
shape=c(nRow, nFeatures), dtype=np$float32) #
W <- tf$Variable(tf$random_uniform(shape(nFeatures, 1L)))
b <- tf$Variable(tf$zeros(shape(1L)))
y <- tf$matmul(x, W) + b
```

6. The input feature `x` is defined as a constant as it will be an input to the system. The weight `W` and bias `b` are defined as variables that will be optimized during the optimization process. The y is set up as a symbolic representation between `x`, `W`, and `b`. The weight `W` is set up to initialize random uniform distribution and `b` is assigned the value zero.
7. The next step is to set up the cost function for logistic regression:

```
# Setting-up cost function and optimizer
y_ <- tf$constant(unlist(occupancy_train[, yFeatures]),
dtype="float32", shape=c(nRow, 1L))
cross_entropy<-
tf$reduce_mean(tf$nn$sigmoid_cross_entropy_with_logits(labels=y_,
logits=y, name="cross_entropy"))
optimizer <-
tf$train$GradientDescentOptimizer(0.15)$minimize(cross_entropy)
```

The variable `y_` is the response variable. Logistic regression is set up using cross entropy as the loss function. The loss function is passed to the gradient descent optimizer with a learning rate of 0.15. Before running the optimization, initialize the global variables:

```
# Start a session
init <- tf$global_variables_initializer()
sess$run(init)
```

8. Execute the gradient descent algorithm for the optimization of weights using cross entropy as the loss function:

```
# Running optimization
for (step in 1:5000) {
  sess$run(optimizer)
  if (step %% 20== 0)
    cat(step, "-", sess$run(W), sess$run(b), "==>",
sess$run(cross_entropy), "n")
}
```

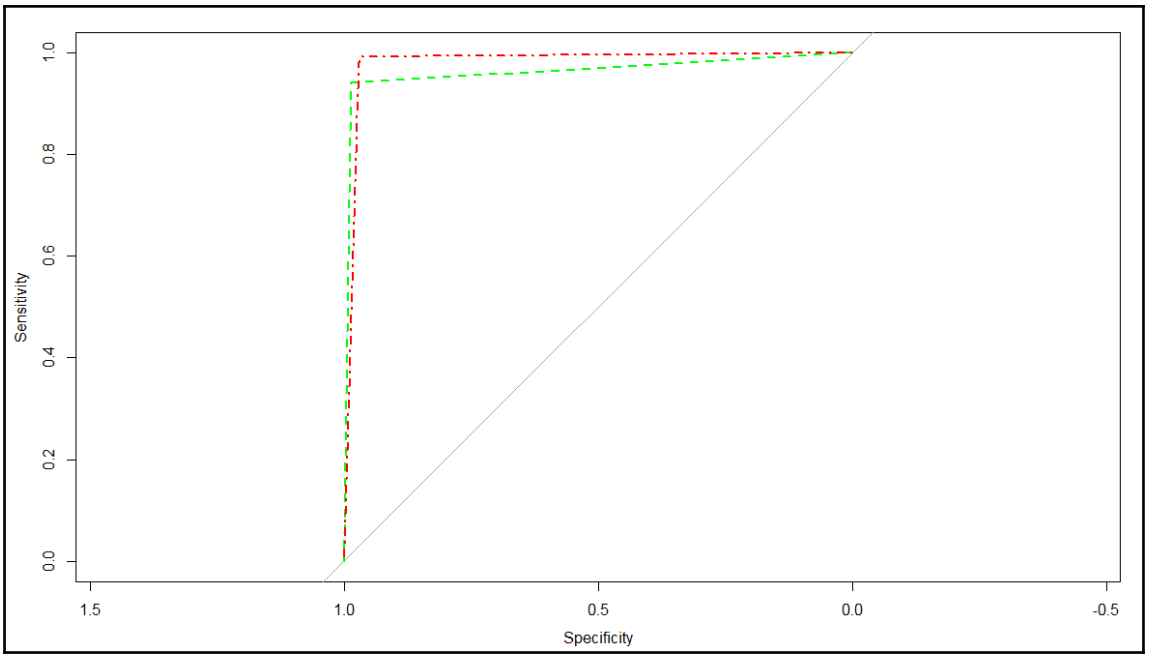
How it works...

The performance of the model can be evaluated using AUC:

```
# Performance on Train
library(pROC)
ypred <- sess$run(tf$nn$sigmoid(tf$matmul(x, W) + b))
roc_obj <- roc(occupancy_train[, yFeatures], as.numeric(ypred))


# Performance on test
nRowt<-nrow(occupancy_test)
xt <- tf$constant(unlist(occupancy_test[, xFeatures]), shape=c(nRowt,
nFeatures), dtype=np$float32)
ypredt <- sess$run(tf$nn$sigmoid(tf$matmul(xt, W) + b))
roc_objt <- roc(occupancy_test[, yFeatures], as.numeric(ypredt)).
```

AUC can be visualized using the `plot.auc` function from the `pROC` package, as shown in the screenshot following this command. The performance for training and testing (hold-out) is very similar.

```
plot.roc(roc_obj, col = "green", lty=2, lwd=2)
plot.roc(roc_objt, add=T, col="red", lty=4, lwd=2)
```



Performance of logistic regression using TensorFlow

Visualizing TensorFlow graphs

TensorFlow graphs can be visualized using TensorBoard. It is a service that utilizes TensorFlow event files to visualize TensorFlow models as graphs. Graph model visualization in TensorBoard is also used to debug TensorFlow models.

Getting ready

TensorBoard can be started using the following command in the terminal:

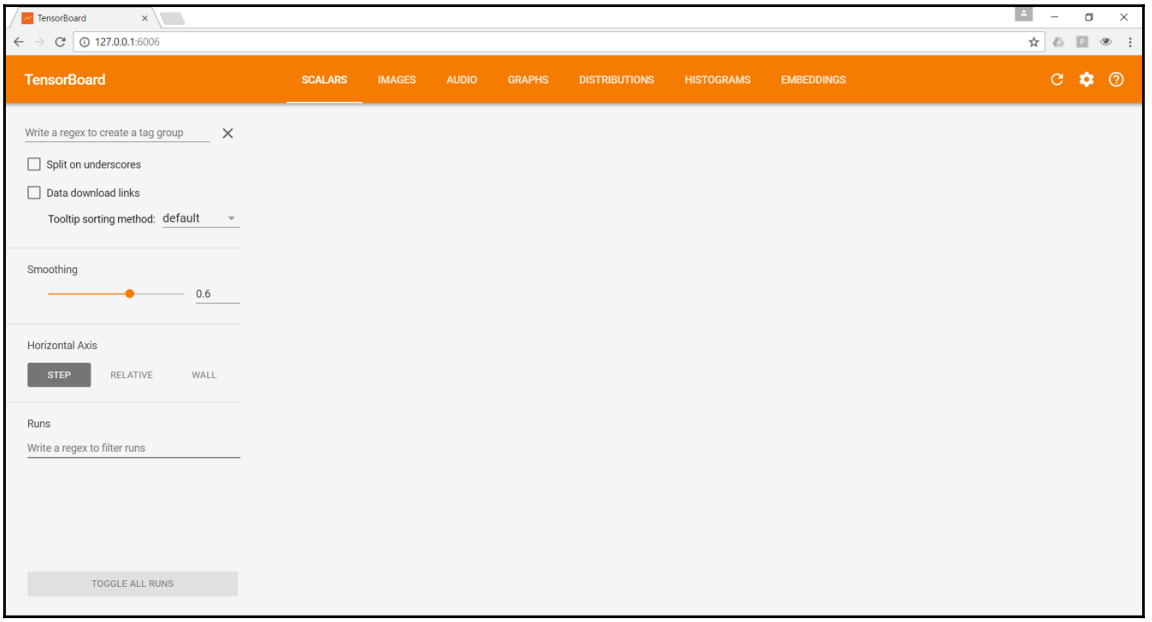
```
$ tensorboard --logdir home/log --port 6006
```

The following are the major parameters for TensorBoard:

- `--logdir` : To map to the directory to load TensorFlow events
- `--debug`: To increase log verbosity

- `--host`: To define the host to listen to its localhost (`127.0.0.1`) by default
- `--port`: To define the port to which TensorBoard will serve

The preceding command will launch the TensorFlow service on localhost at port `6006`, as shown in the following screenshot:



TensorBoard

The tabs on the TensorBoard capture relevant data generated during graph execution.

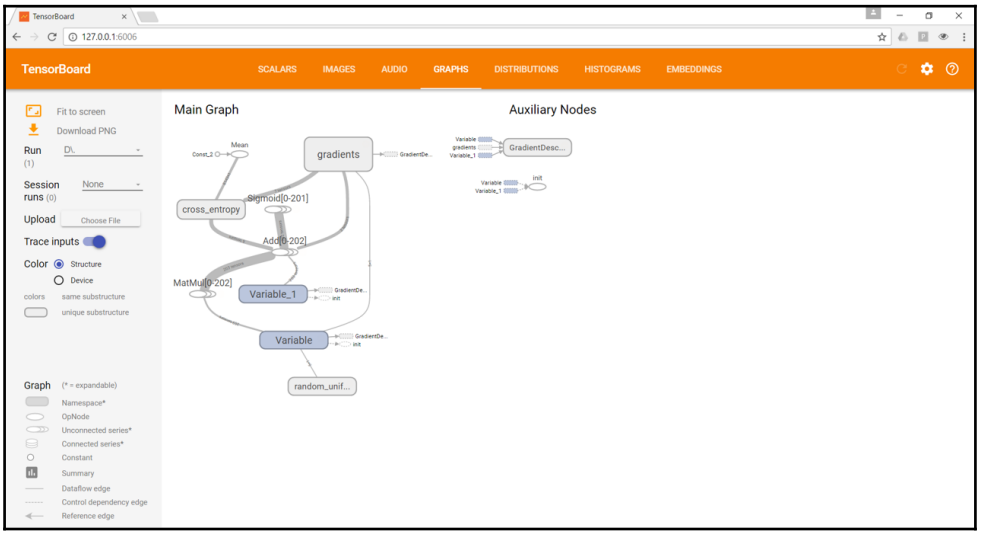
How to do it...

The section covers how to visualize TensorFlow models and output in TernsorBoard.

1. To visualize summaries and graphs, data from TensorFlow can be exported using the `FileWriter` command from the summary module. A default session graph can be added using the following command:

```
# Create Writer Obj for log
log_writer = tf$summary$FileWriter('c:/log', sess$graph)
```

The graph for logistic regression developed using the preceding code is shown in the following screenshot:



Visualization of the logistic regression graph in TensorBoard

Details about symbol descriptions on TensorBoard can be found at `https://www.tensorflow.org/get_started/graph_viz`.

2. Similarly, other variable summaries can be added to the TensorBoard using correct summaries, as shown in the following code:

```
# Adding histogram summary to weight and bias variable
w_hist = tf$histogram_summary("weights", W)
b_hist = tf$histogram_summary("biases", b)
```

The summaries can be a very useful way to determine how the model is performing. For example, for the preceding case, the cost function for test and train can be studied to understand optimization performance and convergence.

3. Create a cross entropy evaluation for test. An example script to generate the cross entropy cost function for test and train is shown in the following command:

```
# Set-up cross entropy for test
nRowt<-nrow(occupancy_test)
xt <- tf$constant(unlist(occupancy_test[, xFeatures]),
```

```
shape=c(nRowt, nFeatures), dtype=np$float32)
ypredt <- tf$nn$sigmoid(tf$matmul(xt, W) + b)
yt_ <- tf$constant(unlist(occupancy_test[, yFeatures]),
dtype="float32", shape=c(nRowt, 1L))
cross_entropy_tst<-
tf$reduce_mean(tf$nn$sigmoid_cross_entropy_with_logits(labels=yt_,
logits=ypredt, name="cross_entropy_tst"))
```

The preceding code is similar to training cross entropy calculations with a different dataset. The effort can be minimized by setting up a function to return tensor objects.

4. Add summary variables to be collected:

```
# Add summary ops to collect data
w_hist = tf$summary$histogram("weights", W)
b_hist = tf$summary$histogram("biases", b)
crossEntropySummary<-tf$summary$scalar("costFunction",
cross_entropy)
crossEntropyTstSummary<-tf$summary$scalar("costFunction_test",
cross_entropy_tst)
```

The script defines the summary events to be logged in the file.

5. Open the writing object, `log_writer`. It writes the default graph to the location, `c:/log`:

```
# Create Writer Obj for log
log_writer = tf$summary$FileWriter('c:/log', sess$graph)
```

6. Run the optimization and collect the summaries:

```
for (step in 1:2500) {
  sess$run(optimizer)

  # Evaluate performance on training and test data after 50
Iteration
  if (step %% 50== 0){
   ### Performance on Train
   ypred <- sess$run(tf$nn$sigmoid(tf$matmul(x, W) + b))
   roc_obj <- roc(occupancy_train[, yFeatures], as.numeric(ypred))

   ### Performance on Test
   ypredt <- sess$run(tf$nn$sigmoid(tf$matmul(xt, W) + b))
   roc_objt <- roc(occupancy_test[, yFeatures], as.numeric(ypredt))
   cat("train AUC: ", auc(roc_obj), " Test AUC: ", auc(roc_objt),
"n")
```

```
    # Save summary of Bias and weights
    log_writer$add_summary(sess$run(b_hist), global_step=step)
    log_writer$add_summary(sess$run(w_hist), global_step=step)
    log_writer$add_summary(sess$run(crossEntropySummary),
global_step=step)
    log_writer$add_summary(sess$run(crossEntropyTstSummary),
global_step=step)
} }
```

7. Collect all the summaries to a single tensor using the `merge_all` command from the summary module:

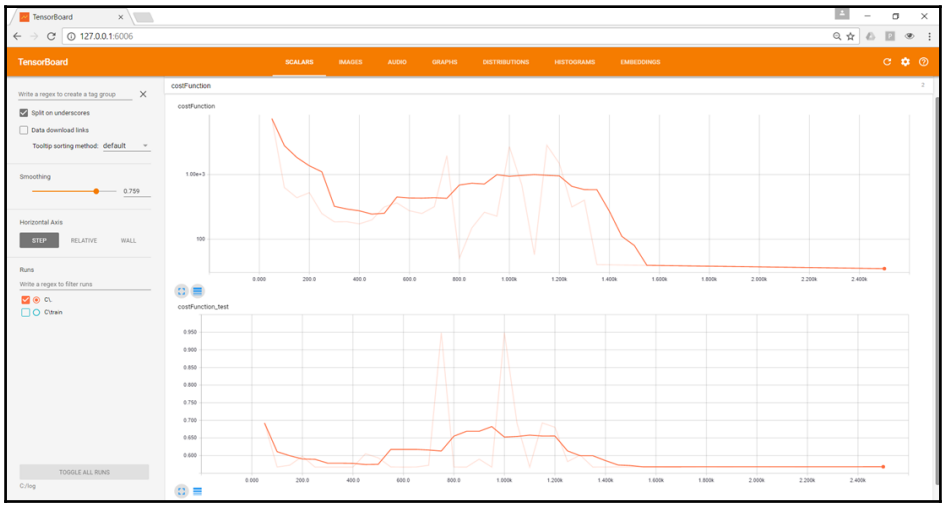
```
summary = tf$summary$merge_all()
```

8. Write the summaries to the log file using the `log_writer` object:

```
log_writer = tf$summary$FileWriter('c:/log', sess$graph)
summary_str = sess$run(summary)
log_writer$add_summary(summary_str, step)
log_writer$close()
```

How it works...

The section covers model performance visualization using TensorBoard. The cross entropy for train and test are recorded in the SCALARS tab, as shown in the following screenshot:



Cross entropy for train and test data

The objective function shows similar behaviors for train and test cost function; thus, the model seems to be stable for the given case with convergence attaining around 1,600 iterations.

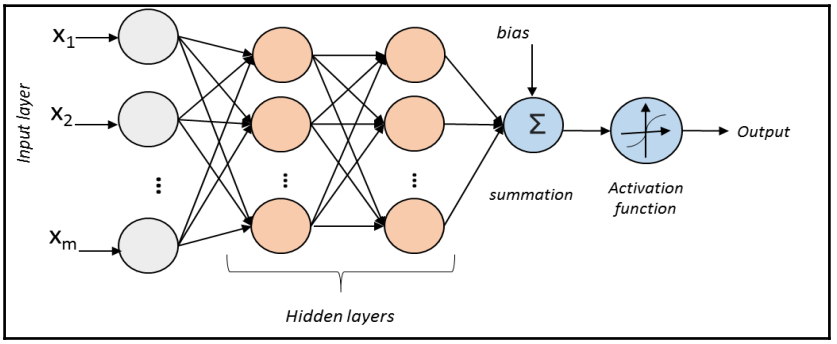
Starting with multilayer perceptrons

This section will focus on extending the logistic regression concept to neural networks.

The neural network, also known as **Artificial neural network** (**ANN**), is a computational paradigm that is inspired by the neuronal structure of the biological brain.

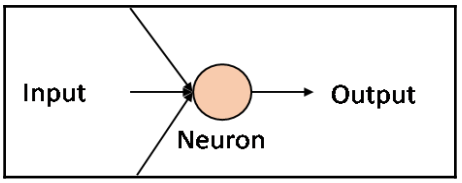
Getting ready

The ANN is a collection of artificial neurons that perform simple operations on the data; the output from this is passed to another neuron. The output generated at each neuron is called its **activation function**. An example of a multilayer perceptron model can be seen in the following screenshot:



An example of a multilayer neural network

Each link in the preceding figure is associated to weights processed by a neuron. Each neuron can be looked at as a processing unit that takes input processing and the output is passed to the next layer, as shown in the following screenshot:



An example of a neuron getting three inputs and one output

The preceding figure demonstrates three inputs combined at neuron to give an output that may be further passed to another neuron. The processing conducted at the neuron could be a very simple operation such as the input multiplied by weights followed by summation or a transformation operation such as the sigmoid activation function.

How to do it...

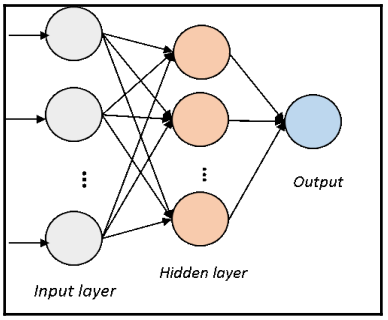
This section covers type activation functions in multilayer perceptrons. Activation is one of the critical component of ANN as it defines the output of that node based on the given input. There are many different activation functions used while building a neural network:

- **Sigmoid**: The sigmoid activation function is a continuous function also known as a logistic function and has the form, *1/(1+exp(-x))*. The sigmoid function has a tendency to zero out the backpropagation terms during training leading to saturation in response. In TensorFlow, the sigmoid activation function is defined using the `tf.nn.sigmoid` function.
- **ReLU**: Rectified linear unit (ReLU) is one of the most famous continuous, but not smooth, activation functions used in neural networks to capture non-linearity. The ReLU function is defined as *max(0,x)*. In TensorFlow, the ReLU activation function is defined as `tf.nn.relu`.
- **ReLU6**: It caps the ReLU function at 6 and is defined as *min(max(0,x), 6)*, thus the value does not become very small or large. The function is defined in TensorFlow as `tf.nn.relu6`.
- **tanh**: Hypertangent is another smooth function used as an activation function in neural networks and is bound [-1 to 1] and implemented as `tf.nn.tanh`.
- **softplus**: It is a continuous version of ReLU, so the differential exists and is defined as *log(exp(x)+1)*. In TensorFlow the softplus is defined as `tf.nn.softplus`.

There's more...

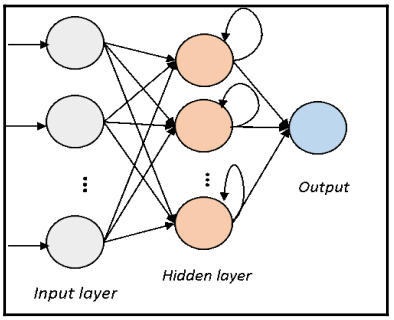
There are three main neural network architectures in neural networks:

- **Feedforward ANN**: This is a class of neural network models where the flow of information is unidirectional from input to output; thus, the architecture does not form any cycle. An example of a Feedforward network is shown in the following image:



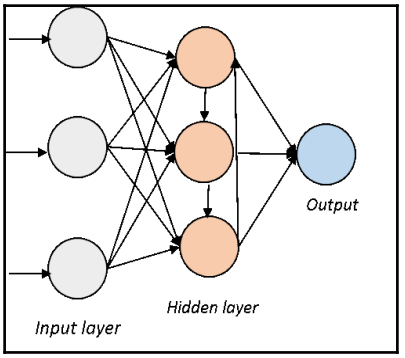
Feedforward architecture of neural networks

- **Feedback ANN**: This is also known as the Elman recurrent network and is a class of neural networks where the error at the output node used as feedback to update iteratively to minimize errors. An example of a one layer Feedback neural network architecture is shown in the following image:



xFeedback architecture of neural networks

- **Lateral ANN**: This is a class of neural networks between Feedback and Feedforward neural networks with neurons interacting within layers. An example lateral neural network architecture is shown in the following image:



Lateral neural network architecture

See also

More activation functions supported in TensorFlow can be found at `https://www.tensorflow.org/versions/r0.10/api_docs/python/nn/activation_functions_`.

Setting up a neural network using H2O

In this section, we will cover the application of H2O in setting up a neural network. The example will use a similar dataset as used in logistic regression.

Getting ready

We first load all the required packages with the following code:

```
# Load the required packages
require(h2o)
```

Then, initialize a single-node H2O instance using the `h2o.init()` function on eight cores and instantiate the corresponding client module on the IP address `localhost` and port number `54321`:

```
# Initialize H2O instance (single node)
localH2O = h2o.init(ip = "localhost", port = 54321, startH2O =
TRUE,min_mem_size = "20G",nthreads = 8)
```

How to do it...

The section shows how to build neural network using H20.

1. Load the occupancy train and test datasets in R:

```
# Load the occupancy data
occupancy_train <-
read.csv("C:/occupation_detection/datatraining.txt",stringsAsFactor
s = T)
occupancy_test <-
read.csv("C:/occupation_detection/datatest.txt",stringsAsFactors =
T)
```

2. The following independent (`x`) and dependent (`y`) variables will be used to model GLM:

```
# Define input (x) and output (y) variables
x = c("Temperature", "Humidity", "Light", "CO2", "HumidityRatio")
y = "Occupancy"
```

3. Based on the requirement by H2O, convert dependent variables to factors as follows:

```
# Convert the outcome variable into factor
occupancy_train$Occupancy <- as.factor(occupancy_train$Occupancy)
occupancy_test$Occupancy <- as.factor(occupancy_test$Occupancy)
```

4. Then convert the datasets to H2OParsedData objects:

```
# Convert Train and Test datasets into H2O objects
occupancy_train.hex <- as.h2o(x = occupancy_train,
```

```
destination_frame = "occupancy_train.hex")
occupancy_test.hex <- as.h2o(x = occupancy_test, destination_frame
= "occupancy_test.hex")
```

5. Once the data is loaded and converted to H2OParsedData objects, build a multilayer Feedforward neural network using the `h2o.deeplearning` function. In the current setup, the following parameters are used to build the NN model:

    - Single hidden layer with five neurons using `hidden`
    - 50 iterations using `epochs`
    - Adaptive learning rate (`adaptive_rate`) instead of a fixed learning rate (rate)
    - `Rectifier` activation function based on ReLU
    - Five-fold cross validation using `nfold`

```
# H2O based neural network to Train the model
occupancy.deepmodel <- h2o.deeplearning(x = x,
                                        y = y,
                                        training_frame =
occupancy_train.hex,
                                        validation_frame =
occupancy_test.hex,
                                        standardize = F,
                                        activation = "Rectifier",
                                        epochs = 50,
                                        seed = 1234567,
                                        hidden = 5,
                                        variable_importances = T,
                                        nfolds = 5,
                                        adpative_rate = TRUE)
```

6. In addition to the command described in the recipe *Performing logistic regression using H2O*, you can also define other parameters to fine-tune the model performance. The following list does not cover all the functional parameters, but covers some based on importance. The complete list of parameters is available in the documentation of the `h2o` package.

    - Option to initialize a model using a pretrained autoencoder model.
    - Provision to fine-tune the adaptive learning rate via an option to modify the time decay factor (*rho*) and smoothing factor (*epsilon*). In the case of a fixed learning rate (*rate*), an option to modify the annealing rate (*rate_annealing*) and decay factor between layers (*rate_decay*).

- Option to initialize weights and biases along with weight distribution and scaling.
- Stopping criteria based on the error fraction in the case of classification and mean squared errors with regard to regression (*classification_stop* and *regression_stop*). An option to also perform early stopping.
- Option to improve distributed model convergence using the elastic averaging method with parameters such as moving rate and regularization strength.

How it works...

The performance of the model can be assessed using many metrics such as accuracy, AUC, misclassification error (%), misclassification error count, F1-score, precision, recall, specificity, and so on. However, in this chapter, the assessment of the model performance is based on AUC.

The following is the training and cross validation accuracy for the trained model. The training and cross validation AUC is `0.984` and `0.982` respectively:

```
# Get the training accuracy (AUC)
> train_performance <- h2o.performance(occupancy.deepmodel,train = T)
> train_performance@metrics$AUC
[1] 0.9848667

# Get the cross-validation accuracy (AUC)
> xval_performance <- h2o.performance(occupancy.deepmodel,xval = T)
> xval_performance@metrics$AUC
[1] 0.9821723
```

As we have already provided test data in the model (as a validation dataset), the following is its performance. The AUC on the test data is `0.991`.

```
# Get the testing accuracy(AUC)
> test_performance <- h2o.performance(occupancy.deepmodel,valid = T)
> test_performance@metrics$AUC
[1] 0.9905056
```

Tuning hyper-parameters using grid searches in H2O

H2O packages also allow you to perform hyper-parameter tuning using grid search (`h2o.grid`).

Getting ready

We first load and initialize the H2O package with the following code:

```
# Load the required packages
require(h2o)

# Initialize H2O instance (single node)
localH2O = h2o.init(ip = "localhost", port = 54321, startH2O =
TRUE,min_mem_size = "20G",nthreads = 8)
```

The occupancy dataset is loaded, converted to hex format, and named *occupancy_train.hex*.

How to do it...

The section will focus on optimizing hyper parameters in H2O using grid searches.

1. In our case, we will optimize for the activation function, the number of hidden layers (along with the number of neurons in each layer), `epochs`, and regularization lambda (`l1` and `l2`):

```
# Perform hyper parameter tuning
activation_opt <- c("Rectifier","RectifierWithDropout",
"Maxout","MaxoutWithDropout")
hidden_opt <- list(5, c(5,5))
epoch_opt <- c(10,50,100)
l1_opt <- c(0,1e-3,1e-4)
l2_opt <- c(0,1e-3,1e-4)

hyper_params <- list(activation = activation_opt,
                     hidden = hidden_opt,
                     epochs = epoch_opt,
                     l1 = l1_opt,
                     l2 = l2_opt)
```

2. The following search criteria have been set to perform a grid search. Adding to the following list, one can also specify the type of stopping metric, the minimum tolerance for stopping, and the maximum number of rounds for stopping:

```
#set search criteria
search_criteria <- list(strategy = "RandomDiscrete",
max_models=300)
```

3. Now, let's perform a grid search on the training data as follows:

```
# Perform grid search on training data
dl_grid <- h2o.grid(x = x,
                    y = y,
                    algorithm = "deeplearning",
                    grid_id = "deep_learn",
                    hyper_params = hyper_params,
                    search_criteria = search_criteria,
                    training_frame = occupancy_train.hex,
                    nfolds = 5)
```

4. Once the grid search is complete (here, there are 216 different models), the best model can be selected based on multiple metrics such as logloss, residual deviance, mean squared error, AUC, accuracy, precision, recall, f1, and so on. In our scenario, let's select the best model with the highest AUC:

```
#Select best model based on auc
d_grid <- h2o.getGrid("deep_learn",sort_by = "auc", decreasing = T)
best_dl_model <- h2o.getModel(d_grid@model_ids[[1]])
```

How it works...

The following is the performance of the grid-searched model on both the training and cross-validation datasets. We can observe that the AUC has increased by one unit in both training and cross-validation scenarios, after performing a grid search. The training and cross validation AUC after the grid search is `0.996` and `0.997` respectively.

```
# Performance on Training data after grid search
> train_performance.grid <- h2o.performance(best_dl_model,train = T)
> train_performance.grid@metrics$AUC
[1] 0.9965881

# Performance on Cross validation data after grid search
> xval_performance.grid <- h2o.performance(best_dl_model,xval = T)
> xval_performance.grid@metrics$AUC
[1] 0.9979131
```

Now, let's assess the performance of the best grid-searched model on the test dataset. We can observe that the AUC has increased by 0.25 units after performing the grid search. The AUC on the test data is `0.993`.

```
# Predict the outcome on test dataset
yhat <- h2o.predict(best_dl_model, occupancy_test.hex)

# Performance of the best grid-searched model on the Test dataset
> yhat$pmax <- pmax(yhat$p0, yhat$p1, na.rm = TRUE)
> roc_obj <- pROC::roc(c(as.matrix(occupancy_test.hex$Occupancy)),
c(as.matrix(yhat$pmax)))
> pROC::auc(roc_obj)
Area under the curve: 0.9932
```

Setting up a neural network using MXNet

The previous chapter provided the details for the installation of MXNet in R along with a working example using its web interface. To start modeling, load the MXNet package in the R environment.

Getting ready

Load the required packages:

```
# Load the required packages
require(mxnet)
```

How to do it...

1. Load the occupancy train and test datasets in R:

```
# Load the occupancy data
occupancy_train <-
read.csv("C:/occupation_detection/datatraining.txt",stringsAsFactor
s = T)
occupancy_test <-
read.csv("C:/occupation_detection/datatest.txt",stringsAsFactors =
T)
```

2. The following independent (`x`) and dependent (`y`) variables will be used to model GLM:

```
# Define input (x) and output (y) variables
x = c("Temperature", "Humidity", "Light", "CO2", "HumidityRatio")
y = "Occupancy"
```

3. Based on the requirement by MXNet, convert the train and test datasets to a matrix and ensure that the class of the outcome variable is numeric (instead of factor as in the case of H2O):

```
# convert the train data into matrix
occupancy_train.x <- data.matrix(occupancy_train[,x])
occupancy_train.y <- occupancy_train$Occupancy

# convert the test data into matrix
occupancy_test.x <- data.matrix(occupancy_test[,x])
occupancy_test.y <- occupancy_test$Occupancy
```

4. Now, let's configure a neural network manually. First, configure a symbolic variable with a specific name. Then configure a symbolic fully connected network with five neurons in a single hidden layer followed with the softmax activation function with logit loss (or cross entropy loss). One can also create additional (fully connected) hidden layers with different activation functions.

```
# Configure Neural Network structure
smb.data <- mx.symbol.Variable("data")
smb.fc <- mx.symbol.FullyConnected(smb.data, num_hidden=5)
smb.soft <- mx.symbol.SoftmaxOutput(smb.fc)
```

5. Once the neural network is configured, let's create (or train) the (Feedforward) neural network model using the `mx.model.FeedForward.create` function. The model is fine-tuned for parameters such as the number of iterations or epochs (*100*), the metric for evaluation (classification accuracy), the size of each iteration or epoch (100 observations), and the learning rate (*0.01*):

```
# Train the network
model.nn <- mx.model.FeedForward.create(symbol = smb.soft,
                                        X = occupancy_train.x,
                                        y = occupancy_train.y,
                                        ctx = mx.cpu(),
                                        num.round = 100,
                                        eval.metric =
```

```
            mx.metric.accuracy,
                                                    array.batch.size = 100,
                                                    learning.rate = 0.01)
```

How it works...

Now, let's assess the performance of the model on train and test datasets. The AUC on the train data is `0.978` and on the test data is `0.982`:

```
# Train accuracy (AUC)
> train_pred <- predict(model.nn,occupancy_train.x)
> train_yhat <- max.col(t(train_pred))-1
> roc_obj <- pROC::roc(c(occupancy_train.y), c(train_yhat))
> pROC::auc(roc_obj)
Area under the curve: 0.9786

#Test accuracy (AUC)
> test_pred <- predict(nnmodel,occupancy_test.x)
> test_yhat <- max.col(t(test_pred))-1
> roc_obj <- pROC::roc(c(occupancy_test.y), c(test_yhat))
> pROC::auc(roc_obj)
Area under the curve: 0.9824
```

Setting up a neural network using TensorFlow

In this section, we will cover an application of TensorFlow in setting up a two-layer neural network model.

Getting ready

To start modeling, load the `tensorflow` package in the environment. R loads the default tf environment variable and also the NumPy library from Python in the `np` variable:

```
library("tensorflow") # Load Tensorflow
np <- import("numpy") # Load numpy library
```

How to do it...

1. The data is imported using the standard function from R, as shown in the following code. The data is imported using the `read.csv` file and transformed into the matrix format followed by selecting the features used for the modeling as defined in `xFeatures` and `yFeatures`:

```
# Loading input and test data
xFeatures = c("Temperature", "Humidity", "Light", "CO2",
"HumidityRatio")
yFeatures = "Occupancy"
occupancy_train <-
as.matrix(read.csv("datatraining.txt",stringsAsFactors = T))
occupancy_test <-
as.matrix(read.csv("datatest.txt",stringsAsFactors = T))

# subset features for modeling and transform to numeric values
occupancy_train<-apply(occupancy_train[, c(xFeatures, yFeatures)],
2, FUN=as.numeric)
occupancy_test<-apply(occupancy_test[, c(xFeatures, yFeatures)], 2,
FUN=as.numeric)

# Data dimensions
nFeatures<-length(xFeatures)
nRow<-nrow(occupancy_train)
```

2. Now load both the network and model parameters. The network parameters define the structure of the neural network and the model parameters define its tuning criteria. As stated earlier, the neural network is built using two hidden layers, each with five neurons. The `n_input` parameter defines the number of independent variables and `n_classes` defines one fewer than the number of output classes. In cases where the output variable is one-hot encoded (one attribute with yes occupancy and a second attribute with no occupancy), then `n_classes` will be 2L (equal to the number of one-hot encoded attributes). Among model parameters, the learning rate is `0.001` and the number of epochs (or iterations) for model building is `10000`:

```
# Network Parameters
n_hidden_1 = 5L # 1st layer number of features
n_hidden_2 = 5L # 2nd layer number of features
n_input = 5L    # 5 attributes
n_classes = 1L  # Binary class

# Model Parameters
```

```
learning_rate = 0.001
training_epochs = 10000
```

3. The next step in TensorFlow is to set up a graph to run the optimization. Before setting up the graph, let's reset the graph using the following command:

```
# Reset the graph
tf$reset_default_graph()
```

4. Additionally, let's start an interactive session as it will allow us to execute variables without referring to the session-to-session object:

```
# Starting session as interactive session
sess<-tf$InteractiveSession()
```

5. The following script defines the graph input (x for independent variables and y for dependent variable). The input feature `x` is defined as a constant as it will be input to the system. Similarly, the output feature `y` is also defined as a constant with the `float32` type:

```
# Graph input
x = tf$constant(unlist(occupancy_train[,xFeatures]), shape=c(nRow,
n_input), dtype=np$float32)
y = tf$constant(unlist(occupancy_train[,yFeatures]),
dtype="float32", shape=c(nRow, 1L))
```

6. Now, let's create a multilayer perceptron with two hidden layers. Both the hidden layers are built using the ReLU activation function and the output layer is built using the linear activation function. The weights and biases are defined as variables that will be optimized during the optimization process. The initial values are randomly selected from a normal distribution. The following script is used to initialize and store a hidden layer's weights and biases along with a mulitilayer perceptron model:

```
# Initializes and store hidden layer's weight & bias
weights = list(
 "h1" = tf$Variable(tf$random_normal(c(n_input, n_hidden_1))),
 "h2" = tf$Variable(tf$random_normal(c(n_hidden_1, n_hidden_2))),
 "out" = tf$Variable(tf$random_normal(c(n_hidden_2, n_classes)))
)
biases = list(
 "b1" = tf$Variable(tf$random_normal(c(1L,n_hidden_1))),
 "b2" = tf$Variable(tf$random_normal(c(1L,n_hidden_2))),
 "out" = tf$Variable(tf$random_normal(c(1L,n_classes)))
```

```
)

# Create model
multilayer_perceptron <- function(x, weights, biases){
 # Hidden layer with RELU activation
 layer_1 = tf$add(tf$matmul(x, weights[["h1"]]), biases[["b1"]])
 layer_1 = tf$nn$relu(layer_1)
 # Hidden layer with RELU activation
 layer_2 = tf$add(tf$matmul(layer_1, weights[["h2"]]),
biases[["b2"]])
 layer_2 = tf$nn$relu(layer_2)
 # Output layer with linear activation
 out_layer = tf$matmul(layer_2, weights[["out"]]) + biases[["out"]]
 return(out_layer)
}
```

7. Now, construct the model using the initialized `weights` and `biases`:

```
pred = multilayer_perceptron(x, weights, biases)
```

8. The next step is to define the `cost` and `optimizer` functions of the neural network:

```
# Define cost and optimizer
cost =
tf$reduce_mean(tf$nn$sigmoid_cross_entropy_with_logits(logits=pred,
labels=y))
optimizer =
tf$train$AdamOptimizer(learning_rate=learning_rate)$minimize(cost)
```

9. The neural network is set up using sigmoid cross entropy as the cost function. The cost function is then passed to a gradient descent optimizer (Adam) with a learning rate of 0.001. Before running the optimization, initialize the global variables as follows:

```
# Initializing the global variables
init = tf$global_variables_initializer()
sess$run(init)
```

10. Once the global variables are initialized along with the cost and optimizer functions, let's begin training on the train dataset:

```
# Training cycle
for(epoch in 1:training_epochs){
    sess$run(optimizer)
   if (epoch %% 20== 0)
    cat(epoch, "-", sess$run(cost), "n")
}
```

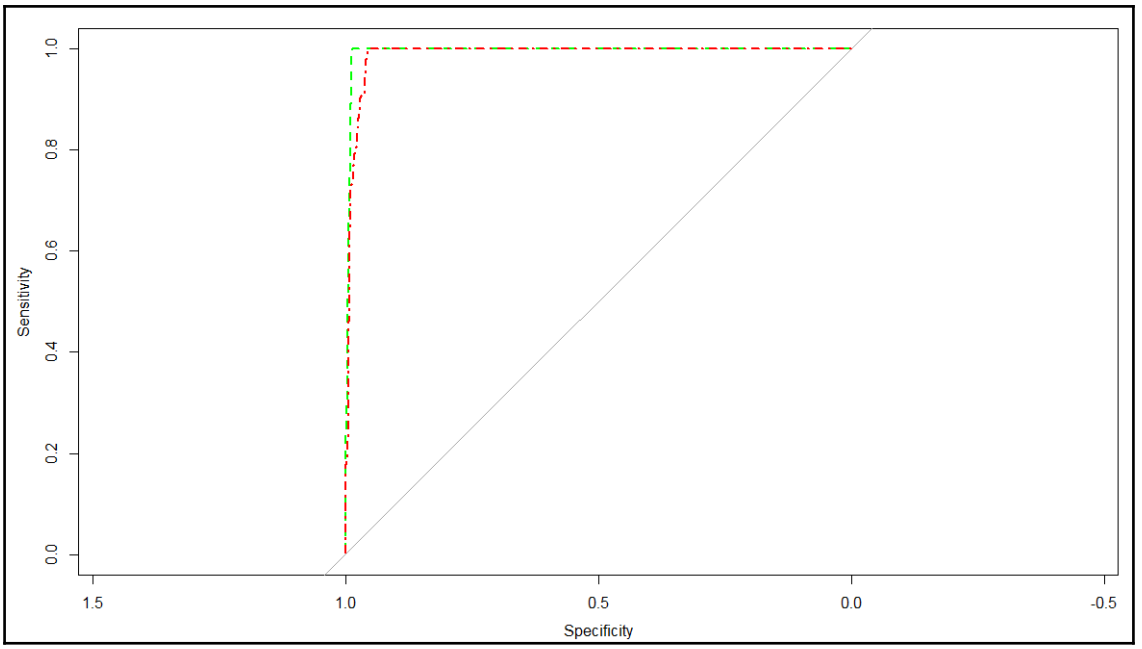
How it works...

The performance of the model can be evaluated using AUC:

```
# Performance on Train
library(pROC)
ypred <- sess$run(tf$nn$sigmoid(multilayer_perceptron(x, weights, biases)))
roc_obj <- roc(occupancy_train[, yFeatures], as.numeric(ypred))

# Performance on Test
nRowt<-nrow(occupancy_test)
xt <- tf$constant(unlist(occupancy_test[, xFeatures]), shape=c(nRowt,
nFeatures), dtype=np$float32) #
ypredt <- sess$run(tf$nn$sigmoid(multilayer_perceptron(xt, weights,
biases)))
roc_objt <- roc(occupancy_test[, yFeatures], as.numeric(ypredt))
```

AUC can be visualized using the `plot.auc` function from the `pROC` package, as shown in the image following the next command. The performance of train and test (hold out) is very similar.

```
plot.roc(roc_obj, col = "green", lty=2, lwd=2)
plot.roc(roc_objt, add=T, col="red", lty=4, lwd=2)
```



Performance of multilayer perceptron using TensorFlow

There's more...

Neural networks are based on the philosophy from the brain; however, the brain consists of around 100 billion neurons with each neuron connected to 10,000 other neurons. Neural networks developed in the early 90s faced a lot of challenges in building deeper neural networks due to computation and algorithmic limitations.

With advances in big data, computational resources (such as GPUs), and better algorithms, the concept of deep learning has emerged and allows us to capture a deeper representation from all kinds of data such as text, image, audio, and so on.

- **Trends in Deep Learning**: Deep learning is an advance on neural networks, which are very much driven by technology enhancement. The main factors that have impacted the development of deep learning as a dominant area in artificial intelligence are as follows:

- **Computational power**: The consistency of Moore's law, which states that the acceleration power of hardware will double every two years, helped in training more layers and bigger data within time limitations
- **Storage and better compression algorithms**: The ability to store big models due to cheaper storage and better compression algorithms have pushed this area with practitioners focusing on capturing real-time data feeds in the form of image, text, audio, and video formats
- **Scalability**: The ability to scale from a simple computer to a farm or using GPU devices has given a great boost to training deep learning models
- **Deep learning architectures**: With new architectures such as Convolution Neural network, re-enforcement learning has provided a boost to what we can learn and also helped expedite learning rates
- **Cross-platform programming:** The ability to program and build models in a cross-platform architecture significantly helped increase the user base and in drastic development in the domain
- **Transfer learning:** This allows reusing pretrained models and further helps in significantly reducing training times

3
Convolution Neural Network

In this chapter, we will cover the following topics:

- Downloading and configuring an image dataset
- Learning the architecture of a CNN classifier
- Using functions to initialize weights and biases
- Using functions to create a new convolution layer
- Using functions to flatten the densely connected layer
- Defining placeholder variables
- Creating the first convolution layer
- Creating the second convolution layer
- Flattening the second convolution layer
- Creating the first fully connected layer
- Applying dropout to the first fully connected layer
- Creating the second fully connected layer with dropout
- Applying softmax activation to obtain a predicted class
- Defining the cost function used for optimization
- Performing gradient descent cost optimization
- Executing the graph in a TensorFlow session
- Evaluating the performance on test data

Introduction

Convolution neural networks (**CNN**) are a category of deep learning neural networks with a prominent role in building image recognition- and natural language processing-based classification models.

The CNN follows a similar architecture to LeNet, which was primarily designed to recognize characters such as numbers, zip codes, and so on. As against artificial neural networks, CNN have layers of neurons arranged in three-dimensional space (width, depth, and height). Each layer transforms a two-dimensional image into a three-dimensional input volume, which is then transformed into a three-dimensional output volume using neuron activation.

Primarily, CNNs are built using three main types of activation layers: convolution layer ReLU, pooling layer, and fully connected layer. The convolution layer is used to extract features (spatial relationship between pixels) from the input vector (of images) and stores them for further processing after computing a dot product with weights (and biases).

Rectified Linear Unit (**ReLU**) is then applied after convolution to introduce non-linearity in the operation.

This is an element-wise operation (such as a threshold function, sigmoid, and tanh) applied to each convolved feature map. Then, the pooling layer (operations such as max, mean, and sum) is used to downsize the dimensionality of each feature map ensuring minimum information loss. This operation of spatial size reduction is used to control overfitting and increase the robustness of the network to small distortions or transformations. The output of the pooling layer is then connected to a traditional multilayer perceptron (also called the fully connected layer). This perceptron uses activation functions such as softmax or SVM to build classifier-based CNN models.

The recipes in this chapter will focus on building a convolution neural network for image classification using Tensorflow in R. While the recipes will provide you with an overview of a typical CNN, we encourage you to adapt and modify the parameters according to your needs.

Downloading and configuring an image dataset

In this chapter, we will use the CIFAR-10 dataset to build a convolution neural network for image classification. The CIFAR-10 dataset consists of 60,000 32 x 32 color images of 10 classes, with 6,000 images per class. These are further divided into five training batches and one test batch, each with 10,000 images.

The test batch contains exactly 1,000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5,000 images from each class. The ten outcome classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The classes are completely mutually exclusive. In addition, the format of the dataset is as follows:

- The first column: The label with 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck
- The next 1,024 columns: Red pixels in the range of 0 to 255
- The next 1,024 columns: Green pixels in the range of 0 to 255
- The next 1,024 columns: Blue pixels in the range of 0 to 255

Getting ready

For this recipe, you will require R with some packages installed such as `data.table` and `imager`.

How to do it...

1. Start R (using Rstudio or Docker) and load the required packages.
2. Download the dataset (binary version) from `http://www.cs.toronto.edu/~kriz/cifar.html`manually or use the following function to download the data in the R environment. The function takes the working directory or the downloaded dataset's location path as an input parameter (`data_dir`):

```
# Function to download the binary file
download.cifar.data <- function(data_dir) {
dir.create(data_dir, showWarnings = FALSE)
setwd(data_dir)
```

```
if (!file.exists('cifar-10-binary.tar.gz')){
download.file(url='http://www.cs.toronto.edu/~kriz/cifar-10-binary.
tar.gz', destfile='cifar-10-binary.tar.gz', method='wget')
untar("cifar-10-binary.tar.gz") # Unzip files
file.remove("cifar-10-binary.tar.gz") # remove zip file
}
setwd("..")
}
# Download the data
download.cifar.data(data_dir="Cifar_10/")
```

3. Once the dataset is downloaded and untarred (or unzipped), read it in the R environment as train and test datasets. The function takes the filenames of the train and test batch datasets (`filenames`) and the number of images to retrieve per batch file (`num.images`) as input parameters:

```
# Function to read cifar data
read.cifar.data <- function(filenames,num.images){
images.rgb <- list()
images.lab <- list()
for (f in 1:length(filenames)) {
to.read <- file(paste("Cifar_10/",filenames[f], sep=""), "rb")
for(i in 1:num.images) {
l <- readBin(to.read, integer(), size=1, n=1, endian="big")
r <- as.integer(readBin(to.read, raw(), size=1, n=1024,
endian="big"))
g <- as.integer(readBin(to.read, raw(), size=1, n=1024,
endian="big"))
b <- as.integer(readBin(to.read, raw(), size=1, n=1024,
endian="big"))
index <- num.images * (f-1) + i
images.rgb[[index]] = data.frame(r, g, b)
images.lab[[index]] = l+1
}
close(to.read)
cat("completed :", filenames[f], "\n")
remove(l,r,g,b,f,i,index, to.read)
}
return(list("images.rgb"=images.rgb,"images.lab"=images.lab))
}
# Train dataset
cifar_train <- read.cifar.data(filenames =
c("data_batch_1.bin","data_batch_2.bin","data_batch_3.bin","data_ba
tch_4.bin", "data_batch_5.bin"))
images.rgb.train <- cifar_train$images.rgb
images.lab.train <- cifar_train$images.lab
rm(cifar_train)
```

```
# Test dataset
cifar_test <- read.cifar.data(filenames = c("test_batch.bin"))
images.rgb.test <- cifar_test$images.rgb
images.lab.test <- cifar_test$images.lab
rm(cifar_test)
```

4. The outcome of the earlier function is a list of red, green, and blue pixel dataframes for each image along with their labels. Then, flatten the data into a list of two dataframes (one for input and the other for output) using the following function, which takes two parameters--a list of input variables (`x_listdata`) and a list of output variables (`y_listdata`):

```
# Function to flatten the data
flat_data <- function(x_listdata,y_listdata){
# Flatten input x variables
x_listdata <- lapply(x_listdata,function(x){unlist(x)})
x_listdata <- do.call(rbind,x_listdata)
# Flatten outcome y variables
y_listdata <- lapply(y_listdata,function(x){a=c(rep(0,10)); a[x]=1;
return(a)})
y_listdata <- do.call(rbind,y_listdata)
# Return flattened x and y variables
return(list("images"=x_listdata, "labels"=y_listdata))
}
# Generate flattened train and test datasets
train_data <- flat_data(x_listdata = images.rgb.train, y_listdata =
images.lab.train)
test_data <- flat_data(x_listdata = images.rgb.test, y_listdata =
images.lab.test)
```

5. Once the list of input and output train and test dataframes is ready, perform sanity checks by plotting the images along with their labels. The function requires two mandatory parameters (`index`: image row number and `images.rgb`: flattened input dataset) and one optional parameter (`images.lab`: flattened output dataset):

```
labels <- read.table("Cifar_10/batches.meta.txt")
# function to run sanity check on photos & labels import
drawImage <- function(index, images.rgb, images.lab=NULL) {
require(imager)
# Testing the parsing: Convert each color layer into a matrix,
# combine into an rgb object, and display as a plot
img <- images.rgb[[index]]
img.r.mat <- as.cimg(matrix(img$r, ncol=32, byrow = FALSE))
img.g.mat <- as.cimg(matrix(img$g, ncol=32, byrow = FALSE)
img.b.mat <- as.cimg(matrix(img$b, ncol=32, byrow = FALSE))
```

```
img.col.mat <- imappend(list(img.r.mat,img.g.mat,img.b.mat),"c")
#Bind the three channels into one image
# Extract the label
if(!is.null(images.lab)){
lab = labels[[1]][images.lab[[index]]]
}
# Plot and output label
plot(img.col.mat,main=paste0(lab,":32x32 size",sep=" "),xaxt="n")
axis(side=1, xaxp=c(10, 50, 4), las=1)
return(list("Image label" =lab,"Image description" =img.col.mat))
}
# Draw a random image along with its label and description from
train dataset
drawImage(sample(1:50000, size=1), images.rgb.train,
images.lab.train)
```

6. Now transform the input data using the min-max standardization technique. The `preProcess` function from the package can be used for normalization. The `"range"` option of the method performs min-max normalization as follows:

```
# Function to normalize data
Require(caret)
normalizeObj<-preProcess(train_data$images, method="range")
train_data$images<-predict(normalizeObj, train_data$images)
test_data$images <- predict(normalizeObj, test_data$images)
```

How it works...

Let's take a look at what we did in the earlier recipe. In step 2, we downloaded the CIFAR-10 dataset from the link mentioned in case it is not present in the given link or working directory. In step 3, the unzipped files are loaded in the R environment as train and test datasets. The train dataset has a list of 50,000 images and the test dataset has a list of 10,000 images along with their labels. Then, in step 4, the train and test datasets are flattened into a list of two dataframes: one with input variables (or images) of length 3,072 (1,024 of red, 1,024 of green, and 1,024 of blue) and the other with output variables (or labels) of length 10 (binary for each class). In step 5, we perform sanity checks for the created train and test datasets by generating plots. The following figure shows a set of six train images along with their labels. Finally, in step 6, the input data is transformed using the min-max standardization technique. An example of categories from the CIFAR-10 dataset is shown in the following figure:

See also

Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009 (`http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`). This is also the reference for this section.

Learning the architecture of a CNN classifier

The CNN classifier covered in this chapter has two convolution layers followed by two fully connected layers in the end, in which the last layer acts as a classifier using the softmax `activation` function.

Getting ready

The recipe requires the CIFAR-10 dataset. Thus, the CIFAR-10 dataset should be downloaded and loaded into the R environment. Also, images are of size 32 x 32 pixels.

How to do it...

Let's define the configuration of the CNN classifier as follows:

1. Each input image (CIFAR-10) is of size 32 x 32 pixels and can be labeled one among 10 classes:

```
# CIFAR images are 32 x 32 pixels.
img_width  = 32L
img_height = 32L

# Tuple with height and width of images used to reshape arrays.
img_shape = c(img_width, img_height)
# Number of classes, one class for each of 10 images
num_classes = 10L
```

2. The images of the CIFAR-10 dataset have three channels (red, green, and blue):

```
# Number of color channels for the images: 3 channel for red, blue,
green scales.
num_channels = 3L
```

3. The images are stored in one-dimensional arrays of the following length (`img_size_flat`):

```
# Images are stored in one-dimensional arrays of length.
img_size_flat = img_width * img_height * num_channels
```

4. In the first convolution layer, the size (width x height) of the convolution filter is 5 x 5 pixels (`filter_size1`) and the depth (or number) of convolution filter is `64` (`num_filters1`):

```
# Convolutional Layer 1.
filter_size1 = 5L
num_filters1 = 64L
```

5. In the second convolution layer, the size and depth of the convolution filter is the same as the first convolution layer:

```
# Convolutional Layer 2.
filter_size2 = 5L
num_filters2 = 64L
```

6. Similarly, the output of the first fully connected layer is the same as the input of the second fully connected layer:

```
# Fully-connected layer.
fc_size = 1024L
```

How it works...

The dimensions and characteristics of an input image are shown in steps 1 and 2, respectively. Every input image is further processed in a convolution layer using a set of filters as defined in steps 4 and 5. The first convolution layer results in a set of 64 images (one for each set filter). In addition, the resolution of these images are also reduced to half (because of 2 x 2 max pooling); namely, from 32 x 32 pixels to 16 x 16 pixels.

The second convolution layer will input these 64 images and provide an output of new 64 images with further reduced resolutions. The updated resolution is now 8 x 8 pixels (again due to 2 x 2 max pooling). In the second convolution layer, a total of 64 x 64 = 4,096 filters are created, which are then further convoluted into 64 output images (or channels). Remember that these 64 images of 8 x 8 resolution correspond to a single input image.

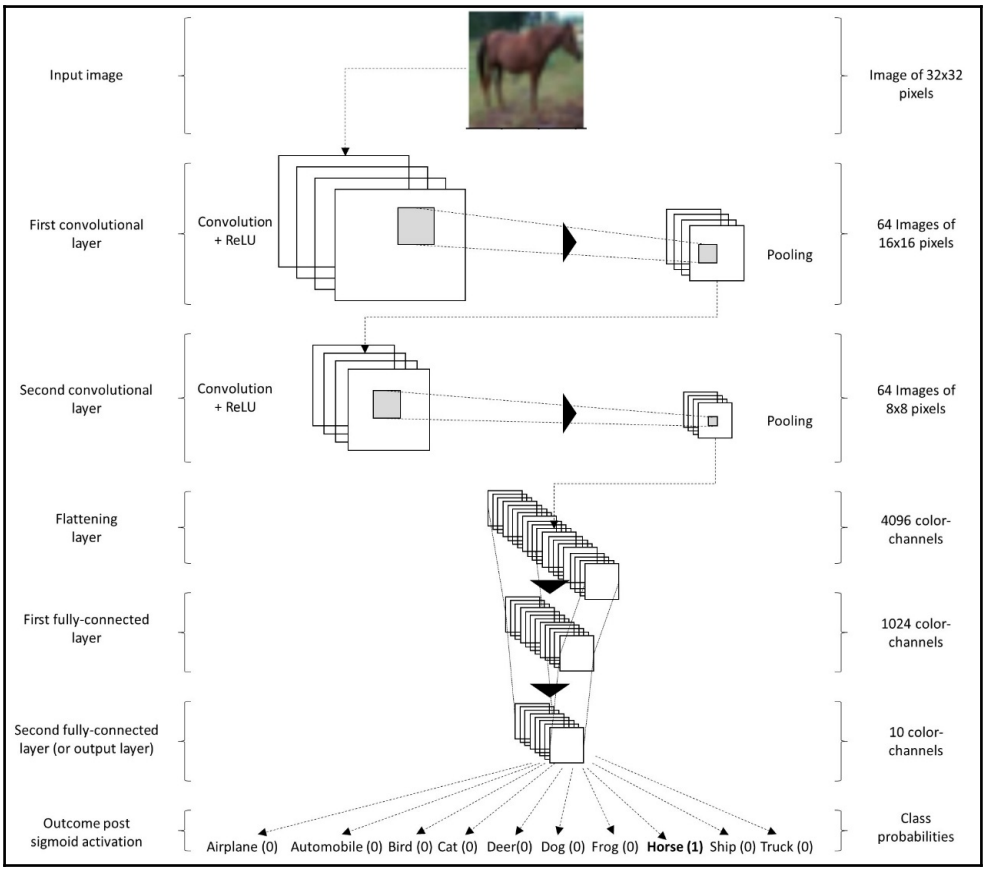
Further, these 64 output images of 8 x 8 pixels are flattened into a single vector of length 4,096 (8 x 8 x 64), as defined in step 3, and are used as an input to a fully connected layer of a given set of neurons, as defined in step 6. The vector of 4,096 elements is then fed into the first fully connected layer of 1,024 neurons. The output neurons are again fed into a second fully connected layer of 10 neurons (equal to `num_classes`). These 10 neurons represent each of the class labels, which are then used to determine the (final) class of the image.

First, the weights of the convolution and fully connected layers are randomly initialized till the classification stage (the end of CNN graph). Here, the classification error is computed based on the true class and the predicted class (also called cross entropy).

Then, the optimizer backpropagates the error through the convolution network using the chain rule of differentiation, post which the weights of the layers (or filters) are updated such that the error is minimized. This entire cycle of one forward and backward propagation is called one iteration. Thousands of such iterations are performed till the classification error is reduced to a sufficiently low value.

Generally, these iterations are performed using a batch of images instead of a single image to increase the efficiency of computation.

The following image represents the convolution network designed in this chapter:



Using functions to initialize weights and biases

Weights and biases form an integral part of any deep neural network optimization and here we define a couple of functions to automate these initializations. It is a good practice to initialize weights with small noise to break symmetry and prevent zero gradients. Additionally, a small positive initial bias would avoid inactivated neurons, suitable for ReLU activation neurons.

Getting ready

Weights and biases are model coefficients which need to be initialized before model compilation. This steps require the `shape` parameter to be determined based on input dataset.

How to do it...

1. The following function is used to return randomly initialized weights:

```
# Weight Initialization
weight_variable <- function(shape) {
initial <- tf$truncated_normal(shape, stddev=0.1)
tf$Variable(initial)
}
```

2. The following function is used to return constant biases:

```
bias_variable <- function(shape) {
initial <- tf$constant(0.1, shape=shape)
tf$Variable(initial)
}
```

How it works...

These functions return TensorFlow variables that are later used as part of a Tensorflow graph. The `shape` is defined as a list of attributes defining a filter in the convolution layer, which is covered in the next recipe. The weights are randomly initialized with a standard deviation equal to `0.1` and biases are initialized with a constant value of `0.1`.

Using functions to create a new convolution layer

Creating a convolution layer is the primary step in a CNN TensorFlow computational graph. This function is primarily used to define the mathematical formulas in the TensorFlow graph, which is later used in actual computation during optimization.

Getting ready

The input dataset is defined and loaded. The `create_conv_layer` function presented in the recipe takes the following five input parameters and needs to be defined while setting-up a convolution layer:

1. `Input`: This is a four-dimensional tensor (or a list) that comprises a number of (input) images, the height of each image (here 32L), the width of each image (here 32L), and the number of channels of each image (here 3L : red, blue, and green).
2. `Num_input_channels`: This is defined as the number of color channels in the case of the first convolution layer or the number of filter channels in the case of subsequent convolution layers.
3. `Filter_size`: This is defined as the width and height of each filter in the convolution layer. Here, the filter is assumed to be a square.
4. `Num_filters`: This is defined as the number of filters in a given convolution layer.
5. `Use_pooling`: This is a binary variable that is used perform 2 x 2 max pooling.

How to do it...

1. Run the following function to create a new convolution layer:

```
# Create a new convolution layer
create_conv_layer <- function(input,
num_input_channels,
filter_size,
num_filters,
use_pooling=True)
{
# Shape of the filter-weights for the convolution.
shape1 = shape(filter_size, filter_size, num_input_channels,
num_filters)
# Create new weights
weights = weight_variable(shape=shape1)
# Create new biases
biases = bias_variable(shape=shape(num_filters))
# Create the TensorFlow operation for convolution.
layer = tf$nn$conv2d(input=input,
filter=weights,
strides=shape(1L, 1L, 1L ,1L),
padding="SAME")
# Add the biases to the results of the convolution.
```

```
layer = layer + biases
# Use pooling (binary flag) to reduce the image resolution
if(use_pooling){
layer = tf$nn$max_pool(value=layer,
ksize=shape(1L, 2L, 2L, 1L),
strides=shape(1L, 2L, 2L, 1L),
padding='SAME')
}
# Add non-linearity using Rectified Linear Unit (ReLU).
layer = tf$nn$relu(layer)
# Retrun resulting layer and updated weights
return(list("layer" = layer, "weights" = weights))
}
```

2. Run the following function to generate plots of convolution layers:

```
drawImage_conv <- function(index, images.bw,
images.lab=NULL,par_imgs=8) {
require(imager)
img <- images.bw[index,,,]
n_images <- dim(img)[3]
par(mfrow=c(par_imgs,par_imgs), oma=c(0,0,0,0),
mai=c(0.05,0.05,0.05,0.05),ann=FALSE,ask=FALSE)
for(i in 1:n_images){
img.bwmat <- as.cimg(img[,,i])
# Extract the label
if(!is.null(images.lab)){
lab = labels[[1]][images.lab[[index]]]
}
# Plot and output label
plot(img.bwmat,axes=FALSE,ann=FALSE)
}
par(mfrow=c(1,1))
}
```

3. Run the following function to generate plots of convolution layer weights:

```
drawImage_conv_weights <- function(weights_conv, par_imgs=8) {
require(imager)
n_images <- dim(weights_conv)[4]
par(mfrow=c(par_imgs,par_imgs), oma=c(0,0,0,0),
mai=c(0.05,0.05,0.05,0.05),ann=FALSE,ask=FALSE)
for(i in 1:n_images){
img.r.mat <- as.cimg(weights_conv[,,1,i])
img.g.mat <- as.cimg(weights_conv[,,2,i])
img.b.mat <- as.cimg(weights_conv[,,3,i])
img.col.mat <- imappend(list(img.r.mat,img.g.mat,img.b.mat),"c")
#Bind the three channels into one image
```

```
# Plot and output label
plot(img.col.mat,axes=FALSE,ann=FALSE)
}
par(mfrow=c(1,1))
}
```

How it works...

The function begins with creating a shape tensor; namely, the list of four integers that are the width of a filter, the height of a filter, the number of input channels, and the number of given filters. Using this shape tensor, initialize a tensor of new weights with the defined shape and create a tensor a new (constant) biases, one for each filter.

Once the necessary weights and biases are initialized, create a TensorFlow operation for convolution using the `tf$nn$conv2d` function. In our current setup, the strides are set to 1 in all four dimensions and padding is set to `SAME`. The first and last are set to 1 by default, but the middle two can factor in higher strides. A stride is the number of pixels by which we allow the filter matrix to slide over the input (image) matrix.

A stride of 3 would mean three pixel jumps across the *x* or *y* axis for each filter slide. Smaller strides would produce larger feature maps, thereby requiring higher computation for convergence. As the padding is set to `SAME`, the input (image) matrix is padded with zeros around the border so that we can apply the filter to border elements of the input matrix. Using this feature, we can control the size of the output matrix (or feature maps) to be the same as the input matrix.

On convolution, the bias values are added to each filter channel followed by pooling to prevent overfitting. In the current setup, 2 x 2 max-pooling (using `tf$nn$max_pool`) is performed to downsize the image resolution. Here, we consider 2 x 2 (`ksize`)-sized windows and select the largest value in each window. These windows stride by two pixels (`strides`) either in the x or y direction.

On pooling, we add non-linearity to the layer using the ReLU activation function (`tf$nn$relu`). In ReLU, each pixel is triggered in the filter and all negative pixel values are replaced with zero using the `max(x,0)` function, where *x* is a pixel value. Generally, ReLU activation is performed before pooling. However, as we are using max-pooling, it doesn't necessarily impact the outcome as such because `relu(max_pool(x))` is equivalent to `max_pool(relu(x))`. Thus, by applying ReLU post pooling, we can save a lot of ReLU operations (~75%).

Finally, the function returns a list of convoluted layers and their corresponding weights. The convoluted layer is a four-dimensional tensor with the following attributes:

- Number of (input) images, the same as `input`
- Height of each image (reduced to half in the case of 2 x 2 max-pooling)
- Width of each image (reduced to half in the case of 2 x 2 max-pooling)
- Number of channels produced, one for each convolution filter

Using functions to create a new convolution layer

The four-dimensional outcome of a newly created convolution layer is flattened to a two-dimensional layer such that it can be used as an input to a fully connected multilayered perceptron.

Getting ready

The recipe explains how to flatten a convolution layer before building the deep learning model. The input to the given function (`flatten_conv_layer`) is a four-dimensional convolution layer that is defined based on previous layer.

How to do it...

1. Run the following function to flatten the convolution layer:

```
flatten_conv_layer <- function(layer){
# Extract the shape of the input layer
layer_shape = layer$get_shape()
# Calculate the number of features as img_height * img_width *
num_channels
num_features =
prod(c(layer_shape$as_list()[[2]],layer_shape$as_list()[[3]],layer_
shape$as_list()[[4]]))
# Reshape the layer to [num_images, num_features].
layer_flat = tf$reshape(layer, shape(-1, num_features))
# Return both the flattened layer and the number of features.
return(list("layer_flat"=layer_flat, "num_features"=num_features))
}
```

How it works...

The function begins with extracting the shape of the given input layer. As stated in previous recipes, the shape of the input layer comprises four integers: image number, image height, image width, and the number of color channels in the image. The number of features (`num_features`) is then evaluated using a dot-product of image height, image weight, and number of color channels.

Then, the layer is flattened or reshaped into a two-dimensional tensor (using `tf$reshape`). The first dimension is set to -1 (which is equal to the total number of images) and the second dimension is the number of features.

Finally, the function returns a list of flattened layers along with the total number of (input) features.

Using functions to flatten the densely connected layer

The CNN generally ends with a fully connected multilayered perceptron using softmax activation in the output layer. Here, each neuron in the previous convoluted-flattened layer is connected to every neuron in the next (fully connected) layer.

The key purpose of the fully convoluted layer is to use the features generated in the convolution and pooling stage to classify the given input image into various outcome classes (here, 10L). It also helps in learning the non-linear combinations of these features to define the outcome classes.

In this chapter, we use two fully connected layers for optimization. This function is primarily used to define the mathematical formulas in the TensorFlow graph, which is later used in actual computation during optimization.

Getting ready

The (`create_fc_layer`) function takes four input parameters, which are as follows:

- `Input`: This is similar to the input of the new convolution layer function

- `Num_inputs`: This is the number of input features generated post flattening the convoluted layer
- `Num_outputs`: This is the number of output neurons fully connected with the input neurons
- `Use_relu`: This takes the binary flag that is set to `FALSE` only in the case of the final fully connected layer

How to do it...

1. Run the following function to create a new fully connected layer:

```
# Create a new fully connected layer
create_fc_layer <- function(input,
num_inputs,
num_outputs,
use_relu=True)
{
# Create new weights and biases.
weights = weight_variable(shape=shape(num_inputs, num_outputs))
biases = bias_variable(shape=shape(num_outputs))
# Perform matrix multiplication of input layer with weights and
then add biases
layer = tf$matmul(input, weights) + biases
# Use ReLU?
if(use_relu){
layer = tf$nn$relu(layer)
}
return(layer)
}
```

How it works...

The function begins with initialing new weights and biases. Then, perform matrix multiplication of the input layer with initialized weights and add relevant biases.

If, the fully connected layer is not the final layer of the CNN TensorFlow graph, ReLU non-linear activation can be performed. Finally, the fully connected layer is returned.

Defining placeholder variables

In this recipe, let's define the placeholder variables that serve as input to the modules in a TensorFlow computational graph. These are typically multidimensional arrays or matrices in the form of tensors.

Getting ready

The data type of placeholder variables is set to float32 (`tf$float32`) and the shape is set to a two-dimensional tensor.

How to do it...

1. Create an input placeholder variable:

```
x = tf$placeholder(tf$float32, shape=shape(NULL, img_size_flat),
name='x')
```

   The NULL value in the placeholder allows us to pass non-deterministic arrays size.

2. Reshape the input placeholder `x` into a four-dimensional tensor:

```
x_image = tf$reshape(x, shape(-1L, img_size, img_size,
num_channels))
```

3. Create an output placeholder variable:

```
y_true = tf$placeholder(tf$float32, shape=shape(NULL, num_classes),
name='y_true')
```

4. Get the (`true`) classes of the output using argmax:

```
y_true_cls = tf$argmax(y_true, dimension=1L)
```

How it works...

In step 1, we define an input placeholder variable. The dimensions of the shape tensor are `NULL` and `img_size_flat`. The former is set to hold any number of images (as rows) and the latter defines the length of input features for each image (as columns). In step 2, the input two-dimensional tensor is reshaped into a four-dimensional tensor, which can be served as input convolution layers. The four dimensions are as follows:

- The first defines the number of input images (currently set to -1)
- The second defines the height of each image (equivalent to image size 32L)
- The third defines the width of each image (equivalent to image size, again 32L)
- The fourth defines the number of color channels in each image (here 3L)

In step 3, we define an output placeholder variable to hold true classes or labels of the images in `x`. The dimensions of the shape tensor are `NULL` and `num_classes`. The former is set to hold any number of images (as rows) and the latter defines the true class of each image as a binary vector of length `num_classes` (as columns). In our scenario, we have 10 classes. In step 4, we compress the two-dimensional output placeholder into a one-dimensional tensor of class numbers ranging from 1 to 10.

Creating the first convolution layer

In this recipe, let's create the first convolution layer.

Getting ready

The following are the inputs to the function `create_conv_layer` defined in the recipe *Using functions to create a new convolution layer*.

- `Input`: This is a four-dimensional reshaped input placeholder variable: `x_image`
- `Num_input_channels`: This is the number of color channels, namely `num_channels`
- `Filter_size`: This is the height and width of the filter layer `filter_size1`
- `Num_filters`: This is the depth of the filter layer, namely `num_filters1`
- `Use_pooling`: This is the binary flag set to `TRUE`

How to do it...

1. Run the `create_conv_layer` function with the preceding input parameters:

```
# Convolutional Layer 1
conv1 <- create_conv_layer(input=x_image,
num_input_channels=num_channels,
filter_size=filter_size1,
num_filters=num_filters1,
use_pooling=TRUE)
```

2. Extract the `layers` of the first convolution layer:

```
layer_conv1 <- conv1$layer
conv1_images <- conv1$layer$eval(feed_dict = dict(x =
train_data$images, y_true = train_data$labels))
```

3. Extract the final `weights` of the first convolution layer:

```
weights_conv1 <- conv1$weights
weights_conv1 <- weights_conv1$eval(session=sess)
```

4. Generate the first convolution layer plots:

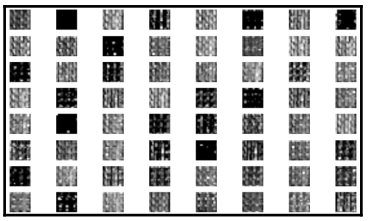
```
drawImage_conv(sample(1:50000, size=1), images.bw = conv1_images,
images.lab=images.lab.train)
```

5. Generate the first convolution layer weight plots:

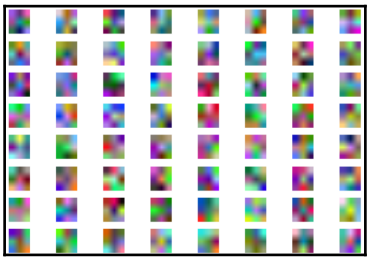
```
drawImage_conv_weights(weights_conv1)
```

How it works...

In steps 1 and 2, we create a first convolution layer of four-dimensions. The first dimension (?) represents any number of input images, the second and third dimensions represent the height (16 pixels) and width (16 pixels) of each convoluted image, and the fourth dimension represents the number of channels (64) produced--one for each convoluted filter. In steps 3 and 5, we extract the final weights of the convolution layer, as shown in the following screenshot:



In step 4, we plot the output of the first convolution layer, as shown in the following screenshot:



Creating the second convolution layer

In this recipe, let's create the second convolution layer.

Getting ready

The following are the inputs to the function `create_conv_layer` defined in the recipe *Using functions to create a new convolution layer*.

- `Input`: This is the four-dimensional output of the first convoluted layer; that is, `layer_conv1`
- `Num_input_channels`: This is the number of filters (or depth) in the first convoluted layer, `num_filters1`
- `Filter_size`: This is the height and width of the filter layer; namely, `filter_size2`
- `Num_filters`: This is the depth of the filter layer, `num_filters2`
- `Use_pooling`: This is the binary flag set to `TRUE`

How to do it...

1. Run the `create_conv_layer` function with the preceding input parameters:

```
# Convolutional Layer 2
conv2 <- create_conv_layer(input=layer_conv1,
num_input_channels=num_filters1,
filter_size=filter_size2,
num_filters=num_filters2,
use_pooling=TRUE)
```

2. Extract the layers of the second convolution layer:

```
layer_conv2 <- conv2$layer
conv2_images <- conv2$layer$eval(feed_dict = dict(x =
train_data$images, y_true = train_data$labels))
```

3. Extract the final weights of the second convolution layer:

```
weights_conv2 <- conv2$weights
weights_conv2 <- weights_conv2$eval(session=sess)
```

4. Generate the second convolution layer plots:

```
drawImage_conv(sample(1:50000, size=1), images.bw = conv2_images,
images.lab=images.lab.train)
```

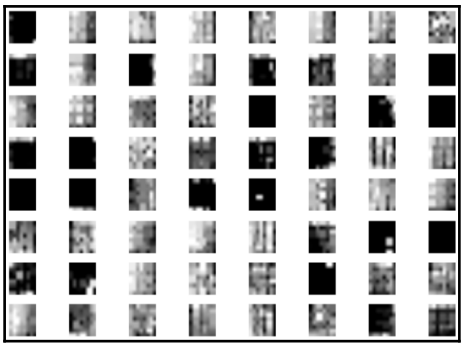
5. Generate the second convolution layer weight plots:

```
drawImage_conv_weights(weights_conv2)
```

How it works...

In steps 1 and 2, we create a second convolution layer of four dimensions. The first dimension (?) represents any number of input images, the second and third dimensions represent the height (8 pixels) and width (8 pixels) of each convoluted image, and the fourth dimension represents the number of channels (64) produced, one for each convoluted filter.

In steps 3 and 5, we extract the final weights of the convolution layer, as shown in the following screenshot:



In step 4, we plot the output of the second convolution layer, as shown in the following screenshot:

	predicted									
actual	1	2	3	4	5	6	7	8	9	10
1	582	36	50	28	55	22	22	29	117	59
2	46	598	9	30	24	24	13	24	38	194
3	62	18	348	100	177	96	89	63	20	27
4	21	25	80	368	88	211	83	66	21	37
5	29	17	106	80	439	74	119	103	19	14
6	17	17	80	211	70	416	65	89	12	23
7	5	23	66	87	89	64	594	41	6	25
8	23	32	49	89	106	82	41	520	11	47
9	108	67	14	29	37	17	11	19	632	66
10	41	151	12	48	16	27	24	56	63	562

Flattening the second convolution layer

In this recipe, let's flatten the second convolution layer that we created.

Getting ready

The following is the input to the function defined in the recipe Creating the second convolution layer, `flatten_conv_layer`:

- `Layer`: This is the output of the second convolution layer, `layer_conv2`

How to do it...

1. Run the `flatten_conv_layer` function with the preceding input parameter:

```
flatten_lay <- flatten_conv_layer(layer_conv2)
```

2. Extract the flattened layer:

```
layer_flat <- flatten_lay$layer_flat
```

3. Extract the number of (input) features generated for each image:

```
num_features <- flatten_lay$num_features
```

How it works...

Prior to connecting the output of the (second) convolution layer with a fully connected network, in step 1, we reshape the four-dimensional convolution layer into a two-dimensional tensor. The first dimension (?) represents any number of input images (as rows) and the second dimension represents the flattened vector of features generated for each image of length 4,096; that is, 8 x 8 x 64 (as columns). Steps 2 and 3 validate the dimensions of the reshaped layers and input features.

Creating the first fully connected layer

In this recipe, let's create the first fully connected layer.

Getting ready

The following are the inputs to the function defined in the recipe *Using functions to flatten the densely connected layer*, `create_fc_layer`:

- `Input`: This is the flattened convolution layer; that is, `layer_flat`
- `Num_inputs`: This is the number of features created post flattening, `num_features`
- `Num_outputs`: This is the number of fully connected neurons output, `fc_size`
- `Use_relu`: This is the binary flag set to `TRUE` to incorporate non-linearity in the tensor

How to do it...

1. Run the `create_fc_layer` function with the preceding input parameters:

```
layer_fc1 = create_fc_layer(input=layer_flat,
num_inputs=num_features,
num_outputs=fc_size,
use_relu=TRUE)
```

How it works...

Here, we create a fully connected layer that returns a two-dimensional tensor. The first dimension (?) represents any number of (input) images and the second dimension represents the number of output neurons (here, 1,024).

Applying dropout to the first fully connected layer

In this recipe, let's apply dropout to the output of the fully connected layer to reduce the chance of overfitting. The dropout step involves removing some neurons randomly during the learning process.

Getting ready

The dropout is connected to the output of the layer. Thus, model initial structure is set up and loaded. For example, in dropout current layer `layer_fc1` is defined, on which dropout is applied.

How to do it...

1. Create a placeholder for dropout that can take probability as an input:

```
keep_prob <- tf$placeholder(tf$float32)
```

2. Use TensorFlow's dropout function to handle the scaling and masking of neuron outputs:

```
layer_fc1_drop <- tf$nn$dropout(layer_fc1, keep_prob)
```

How it works...

In steps 1 and 2, we can drop (or mask) out the output neurons based on the input probability (or percentage). The dropout is generally allowed during training and can be turned off (by assigning probability as `1` or `NULL`) during testing.

Creating the second fully connected layer with dropout

In this recipe, let's create the second fully connected layer along with dropout.

Getting ready

The following are the inputs to the function defined in the recipe *Using functions to flatten the densely connected layer*, `create_fc_layer`:

- `Input`: This is the output of the first fully connected layer; that is, `layer_fc1`
- `Num_inputs`: This is the number of features in the output of the first fully connected layer, `fc_size`
- `Num_outputs`: This is the number of the fully connected neurons output (equal to the number of labels, `num_classes`)
- `Use_relu`: This is the binary flag set to `FALSE`

How to do it...

1. Run the `create_fc_layer` function with the preceding input parameters:

```
layer_fc2 = create_fc_layer(input=layer_fc1_drop,
num_inputs=fc_size,
num_outputs=num_classes,
use_relu=FALSE)
```

2. Use TensorFlow's dropout function to handle the scaling and masking of neuron outputs:

```
layer_fc2_drop <- tf$nn$dropout(layer_fc2, keep_prob)
```

How it works...

In step 1, we create a fully connected layer that returns a two-dimensional tensor. The first dimension (?) represents any number of (input) images and the second dimension represents the number of output neurons (here, 10 class labels). In step 2, we provide the option for dropout primarily used during the training of the network.

Applying softmax activation to obtain a predicted class

In this recipe, we will normalize the outputs of the second fully connected layer using softmax activation such that each class has a (probability) value restricted between 0 and 1, and all the values across 10 classes add up to 1.

Getting ready

The activation function is applied at the end of the pipeline on predictions generated by the deep learning model. Before executing this step, all steps in the pipeline need to be executed. The recipe requires the TensorFlow library.

How to do it...

1. Run the `softmax` activation function on the output of the second fully connected layer:

```
y_pred = tf$nn$softmax(layer_fc2_drop)
```

2. Use the `argmax` function to determine the class number of the label. It is the index of the class with the largest (probability) value:

```
y_pred_cls = tf$argmax(y_pred, dimension=1L)
```

Defining the cost function used for optimization

The cost function is primarily used to evaluate the current performance of the model by comparing the true class labels (`y_true_cls`) with the predicted class labels (`y_pred_cls`). Based on the current performance, the optimizer then fine-tunes the network parameters, such as weights and biases, to further improve its performance.

Getting ready

The cost function definition is critical as it will decide optimization criteria. The cost function definition will require true classes and predicted classes to do comparison. The objective function used in this recipe is cross entropy, used in multi-classification problems.

How to do it...

1. Evaluate the current performance of each image using the cross entropy function in TensorFlow. As the cross entropy function in TensorFlow internally applies softmax normalization, we provide the output of the fully connected layer post dropout (`layer_fc2_drop`) as an input along with true labels (`y_true`):

```
cross_entropy =
tf$nn$softmax_cross_entropy_with_logits(logits=layer_fc2_drop,
labels=y_true)
```

In the current cost function, softmax activation function is embedded thus the activation function is not required to be defined separately.

2. Calculate the average of the cross entropy, which needs to be minimized using an optimizer:

```
cost = tf$reduce_mean(cross_entropy)
```

How it works...

In step 1, we define a cross entropy to evaluate the performance of classification. Based on the exact match between the true and predicted labels, the cross entropy function returns a value that is positive and follows a continuous distribution. As zero cross entropy ensures a full match, optimizers tend to minimize the cross entropy toward the value zero by updating the network parameters such as weights and biases. The cross entropy function returns a value for each individual image that needs to be further compressed into a single scalar value, which can be used in an optimizer. Hence, in step 2, we calculate a simple average of the cross entropy output and store it as *cost*.

Performing gradient descent cost optimization

In this recipe, let's define an optimizer that can minimize the cost. Post optimization, check for CNN performance.

Getting ready

The optimizer definition will require the `cost` recipe to be defined as it goes as input to the optimizer.

How to do it...

1. Run an Adam optimizer with the objective of minimizing the cost for a given `learning_rate`:

```
optimizer =
tf$train$AdamOptimizer(learning_rate=1e-4)$minimize(cost)
```

2. Extract the number of `correct_predictions` and calculate the mean percentage accuracy:

```
correct_prediction = tf$equal(y_pred_cls, y_true_cls)
accuracy = tf$reduce_mean(tf$cast(correct_prediction, tf$float32))
```

Executing the graph in a TensorFlow session

Until now, we have only created tensor objects and added them to a TensorFlow graph for later execution. In this recipe, we will learn how to create a TensorFlow session that can be used to execute (or run) the TensorFlow graph.

Getting ready

Before we run the graph, we should have TensorFlow installed and loaded in R. The installation details can be found in `Chapter 1`, *Getting Started*.

How to do it...

1. Load the `tensorflow` library and import the `numpy` package:

```
library(tensorflow)
np <- import("numpy")
```

2. Reset or remove any existing `default_graph`:

```
tf$reset_default_graph()
```

3. Start an `InteractiveSession`:

```
sess <- tf$InteractiveSession()
```

4. Initialize the `global_variables`:

```
sess$run(tf$global_variables_initializer())
```

5. Run iterations to perform optimization (`training`):

```
# Train the model
train_batch_size = 128L
for (i in 1:100) {
spls <- sample(1:dim(train_data$images)[1],train_batch_size)
if (i %% 10 == 0) {
train_accuracy <- accuracy$eval(feed_dict = dict(
x = train_data$images[spls,], y_true = train_data$labels[spls,],
keep_prob = 1.0))
cat(sprintf("step %d, training accuracy %g\n", i, train_accuracy))
}
optimizer$run(feed_dict = dict(
x = train_data$images[spls,], y_true = train_data$labels[spls,],
keep_prob = 0.5))
}
```

6. Evaluate the performance of the trained model on test data:

```
# Test the model
test_accuracy <- accuracy$eval(feed_dict = dict(
x = test_data$images, y_true = test_data$labels, keep_prob = 1.0))
cat(sprintf("test accuracy %g", test_accuracy))
```

How it works...

Steps 1 through 4 are, in a way, the default way to launch a new TensorFlow session. In step 4, the variables of weights and biases are initialized, which is mandatory before their optimization. Step 5 is primarily to execute the TensorFlow session for optimization. As we have a large number of training images, it becomes highly difficult (computationally) to calculate the optimum gradient taking all the images at once into the optimizer.

Hence, a small random sample of 128 images is selected to train the activation layer (weights and biases) in each iteration. In the current setup, we run 100 iterations and report training accuracy for every tenth iteration.

However, these can be increased based on the cluster configuration or computational power (CPU or GPU) to obtain higher model accuracy. In addition, a 50% dropout rate is used to train the CNN in each iteration. In step 6, we can evaluate the performance of the trained model on a test data of 10,000 images.

Evaluating the performance on test data

In this recipe, we will look into the performance of the trained CNN on test images using a confusion matrix and plots.

Getting ready

The prerequisite packages for plots are `imager` and `ggplot2`.

How to do it...

1. Get the `actual` or `true` class labels of test images:

```
test_true_class <- c(unlist(images.lab.test))
```

2. Get the predicted class labels of test images. Remember to add `1` to each class label, as the starting index of TensorFlow (the same as Python) is 0 and that of R is `1`:

```
test_pred_class <- y_pred_cls$eval(feed_dict = dict(
x = test_data$images, y_true = test_data$labels, keep_prob = 1.0))
test_pred_class <- test_pred_class + 1
```

3. Generate the confusion matrix with rows as true labels and columns as predicted labels:

```
table(actual = test_true_class, predicted = test_pred_class)
```

4. Generate a plot of the `confusion` matrix:

```
confusion <- as.data.frame(table(actual = test_true_class,
predicted = test_pred_class))
plot <- ggplot(confusion)
plot + geom_tile(aes(x=actual, y=predicted, fill=Freq)) +
scale_x_discrete(name="Actual Class") +
scale_y_discrete(name="Predicted Class") +
scale_fill_gradient(breaks=seq(from=-.5, to=4, by=.2)) +
labs(fill="Normalized\nFrequency")
```

5. Run a helper function to plot images:

```
check.image <- function(images.rgb,index,true_lab, pred_lab) {
require(imager)
# Testing the parsing: Convert each color layer into a matrix,
# combine into an rgb object, and display as a plot
img <- images.rgb[[index]]
img.r.mat <- as.cimg(matrix(img$r, ncol=32, byrow = FALSE))
img.g.mat <- as.cimg(matrix(img$g, ncol=32, byrow = FALSE))
img.b.mat <- as.cimg(matrix(img$b, ncol=32, byrow = FALSE))
img.col.mat <- imappend(list(img.r.mat,img.g.mat,img.b.mat),"c")
# Plot with actual and predicted label
plot(img.col.mat,main=paste0("True: ", true_lab,":: Pred: ",
pred_lab),xaxt="n")
axis(side=1, xaxp=c(10, 50, 4), las=1)
}
```

6. Plot random misclassified test images:

```
labels <-
c("airplane","automobile","bird","cat","deer","dog","frog","horse",
"ship","truck")
# Plot misclassified test images
plot.misclass.images <- function(images.rgb, y_actual,
y_predicted,labels){
# Get indices of misclassified
indices <- which(!(y_actual == y_predicted))
id <- sample(indices,1)
# plot the image with true and predicted class
true_lab <- labels[y_actual[id]]
pred_lab <- labels[y_predicted[id]]
check.image(images.rgb,index=id,
true_lab=true_lab,pred_lab=pred_lab)
}
plot.misclass.images(images.rgb=images.rgb.test,y_actual=test_true_
class,y_predicted=test_pred_class,labels=labels)
```

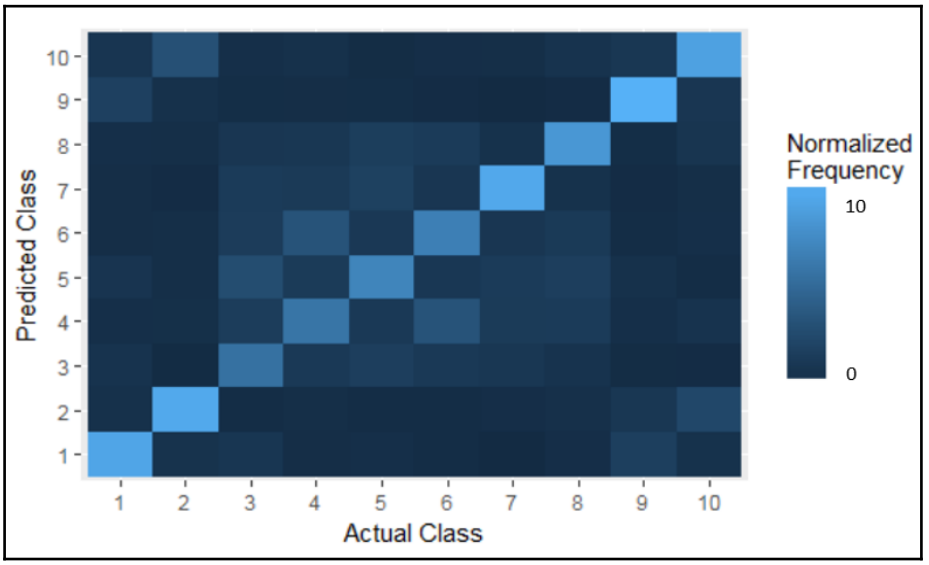
How it works...

In steps 1 through 3, we extract the true and predicted test class labels and create a confusion matrix. The following image shows the confusion matrix of the current test predictions:

	predicted									
actual	1	2	3	4	5	6	7	8	9	10
1	582	36	50	28	55	22	22	29	117	59
2	46	598	9	30	24	24	13	24	38	194
3	62	18	348	100	177	96	89	63	20	27
4	21	25	80	368	88	211	83	66	21	37
5	29	17	106	80	439	74	119	103	19	14
6	17	17	80	211	70	416	65	89	12	23
7	5	23	66	87	89	64	594	41	6	25
8	23	32	49	89	106	82	41	520	11	47
9	108	67	14	29	37	17	11	19	632	66
10	41	151	12	48	16	27	24	56	63	562

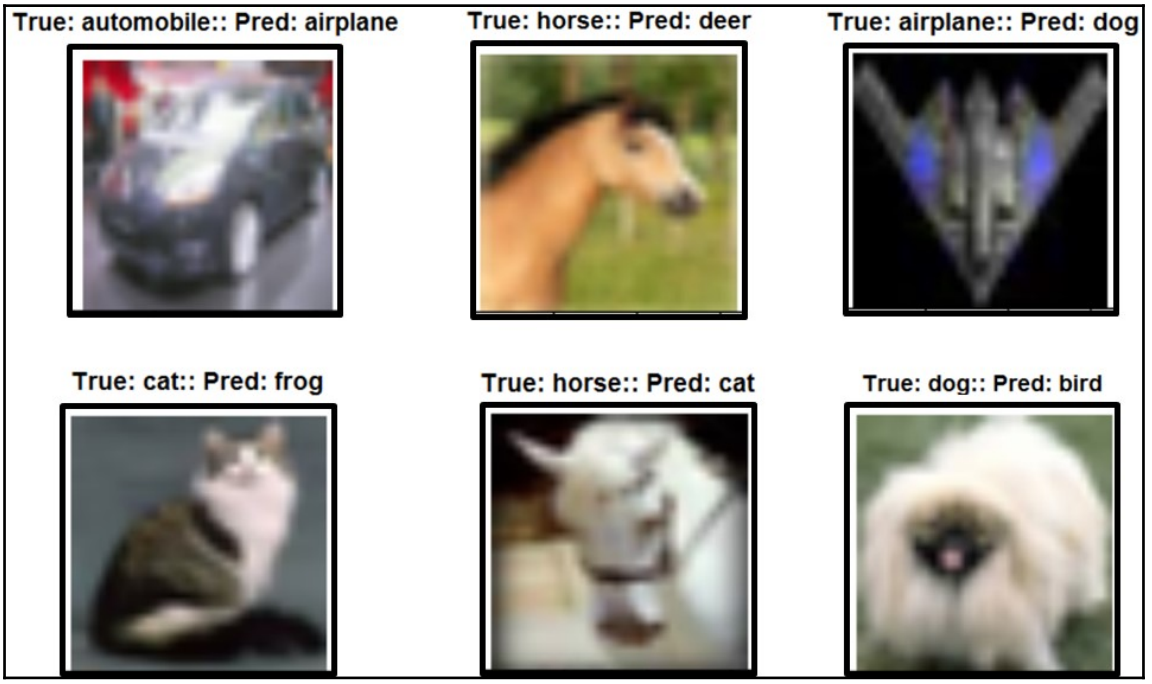
The test accuracy post 700 training iterations is only ~51% and can be further improved by increasing the number of iterations, increasing the batch size, configuring layer parameters such as the number of convolution layers (used 2), types of activation functions (used ReLU), number of fully connected layers (used two), optimization objective function (used accuracy), pooling (used max 2 x 2), dropout probability, and many others.

Step 4 is used to build a facet plot of the test confusion matrix, as shown in the following screenshot:



In step 5, we define a helper function to plot the image along with a header containing both true and predicted classes. The input parameters of the `check.image` function are (`test`) flattened input dataset (`images.rgb`), image number (`index`), true label (`true_lab`), and predicted label (`pred_lab`). Here, the red, green, and blue pixels are initially parsed out, converted into a matrix, appended as a list, and displayed as an image using the *plot* function.

In step 6, we plot misclassified test images using the helper function of step 5. The input parameters of the `plot.misclass.images` function are (`test`) flattened input dataset (`images.rgb`), a vector of true labels (`y_actual`), a vector of predicted labels (`y_predicted`), and a vector of unique ordered character labels (`labels`). Here, the indices of the misclassified images are obtained and an index is randomly selected to generate the plot. The following screenshot shows a set of six misclassified images with true and predicted labels:



4

Data Representation Using Autoencoders

This chapter will introduce unsupervised applications of deep learning using autoencoders. In this chapter, we will cover the following topics:

- Setting up autoencoders
- Data normalization
- Setting up a regularized autoencoder
- Fine-tuning the parameters of the autoencoder
- Setting up stacked autoencoders
- Setting up denoising autoencoders
- Building and comparing stochastic encoders and decoders
- Learning manifolds from autoencoders
- Evaluating the sparse decomposition

Introduction

Neural networks aim to find a non-linear relationship between input X with output y, as $y=f(x)$**.** An autoencoder is a form of unsupervised neural network which tries to find a relationship between features in space such that $h=f(x)$, which helps us learn the relationship between input space and can be used for data compression, dimensionality reduction, and feature learning.

An autoencoder consists of an encoder and decoder. The encoder helps encode the input x in a latent representation y, whereas a decoder converts back the y to x. Both the encoder and decoder possess a similar representation of form.

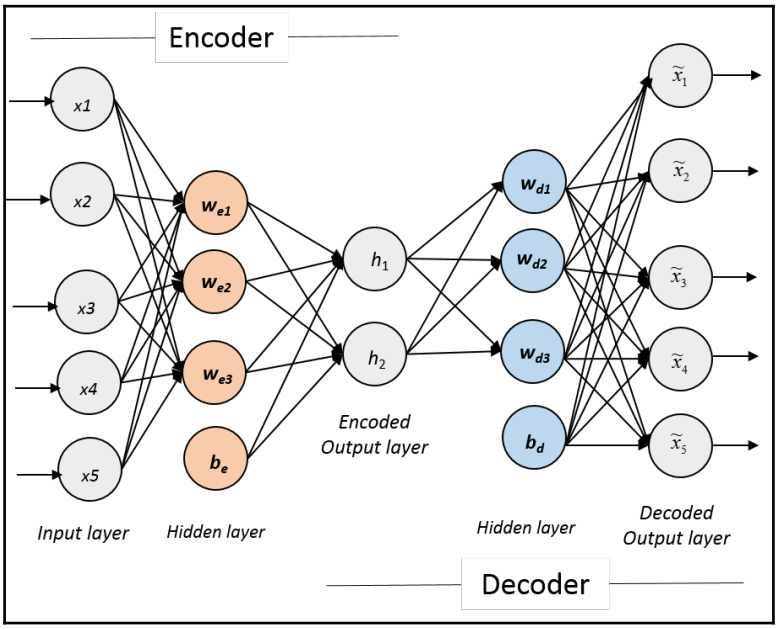
Here is a representation of a one layer autoencoder:

$$\boldsymbol{h = f(x) = \sigma(W_e^T X + b_e)}$$

$$\boldsymbol{\tilde{X} = f(h) = \sigma(W_d^T h + b_d)}$$

The coder encodes input X to h under a hidden layer contain, whereas the decoder helps to attain the original data from encoded output h. The matrices W_e and W_d represent the weights of the encoder and decoder layers, respectively. The function f is the activation function.

An illustration of an autoencoder is shown in the following diagram:



The constraints in the form of nodes allows the autoencoder to discover interesting structures within the data. For example, in the encoder in the preceding diagram, the five-input dataset must pass through three-node compression to get an encoded value *h.* The **Encoded Output layer** of an encoder can have a dimensionality that is the same, lower, or higher than the **input/output Decoded Output layer**. The **Encoded Output layer** with a fewer number of nodes than the input layer is referred to as an under-complete representation, and can be thought of as data compression transforming data into low-dimensional representation.

An **Encoded Output layer** with a larger number of input layers is referred to as an over-complete representation and is used in a **sparse autoencoder** as a regularization strategy. The objective of an autoencoder is to find *y*, capturing the main factors along the variation of data, which is similar to **Principal Component Analysis (PCA)**, and thus can be used for compression as well.

Setting up autoencoders

There exist a lot of different architectures of autoencoders distinguished by cost functions used to capture data representation. The most basic autoencoder is known as a vanilla autoencoder. It's a two-layer neural network with one hidden layer the same number of nodes at the input and output layers, with an objective to minimize the cost function. The typical choices, but not limited to, for a loss function are **mean square error** (**MSE**) for regression and cross entropy for classification. The current approach can be easily extended to multiple layers, also known as multilayer autoencoder.

The number of nodes plays a very critical role in autoencoders. If the number of nodes in the hidden layer is less than the input layer then an autoencoder is known as an **under-complete** autoencoder. A higher number of nodes in the hidden layer represents an **over-complete** autoencoder or sparse autoencoder.

The sparse autoencoder aims to impose sparsity in the hidden layer. This sparsity can be achieved by introducing a higher number of nodes than the input in the hidden layer or by introducing a penalty in the loss function that will move the weights for the hidden layer toward zero. Some autoencoders attain the sparsity by manually zeroing out the weight for nodes; these are referred to as **K-sparse autoencoders**. We will set up an autoencoder on the occupancy dataset discussed in `Chapter 1`, *Getting Started*. The hidden layer for the current example can be tweaked around.

Getting ready

Let's use the `Occupancy` dataset to set up an autoencoder:

- Download the `Occupancy` dataset as described in `Chapter 1`, *Getting Started*
- TensorFlow installation in R and Python

How to do it...

The current occupancy dataset as described in `Chapter 1`, *Getting Started,* is used to demonstrate the autoencoder setup in R using TensorFlow:

1. Set up the R TensorFlow environment.
2. The `load_occupancy_data` function can be used to load the data by setting the correct working directory path using `setwd`:

```
# Function to load Occupancy data
load_occupancy_data<-function(train){
xFeatures = c("Temperature", "Humidity", "Light", "CO2",
  "HumidityRatio")
yFeatures = "Occupancy"
  if(train){
    occupancy_ds <-
as.matrix(read.csv("datatraining.txt",stringsAsFactors = T))
  } else
  {
    occupancy_ds <-
as.matrix(read.csv("datatest.txt",stringsAsFactors = T))
  }
  occupancy_ds<-apply(occupancy_ds[, c(xFeatures, yFeatures)], 2,
FUN=as.numeric)
  return(occupancy_ds)
}
```

3. The train and test occupancy dataset can be loaded to the R environment with the following script:

```
occupancy_train <-load_occupancy_data(train=T)
occupancy_test <- load_occupancy_data(train = F)
```

Data normalization

Data normalization is a critical step in machine learning to bring data to a similar scale. It is also known as feature scaling and is performed as data preprocessing.

The correct normalization is very critical in neural networks, else it will lead to saturation within the hidden layers, which in turn leads to zero gradient and no learning will be possible.

Getting ready

There are multiple ways to perform normalization:

- **Min-max standardization**: The min-max retains the original distribution and scales the feature values between *[0, 1]*, with *0* as the minimum value of the feature and *1* as the maximum value. The standardization is performed as follows:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

  Here, x' is the normalized value of the feature. The method is sensitive to outliers in the dataset.

- **Decimal scaling**: This form of scaling is used where values of different decimal ranges are present. For example, two features with different bounds can be brought to a similar scale using decimal scaling as follows:

$$x'=x/10^n$$

- **Z-score:** This transformation scales the value toward a normal distribution with a zero mean and unit variance. The Z-score is computed as:

$$Z=(x-\mu)/\sigma$$

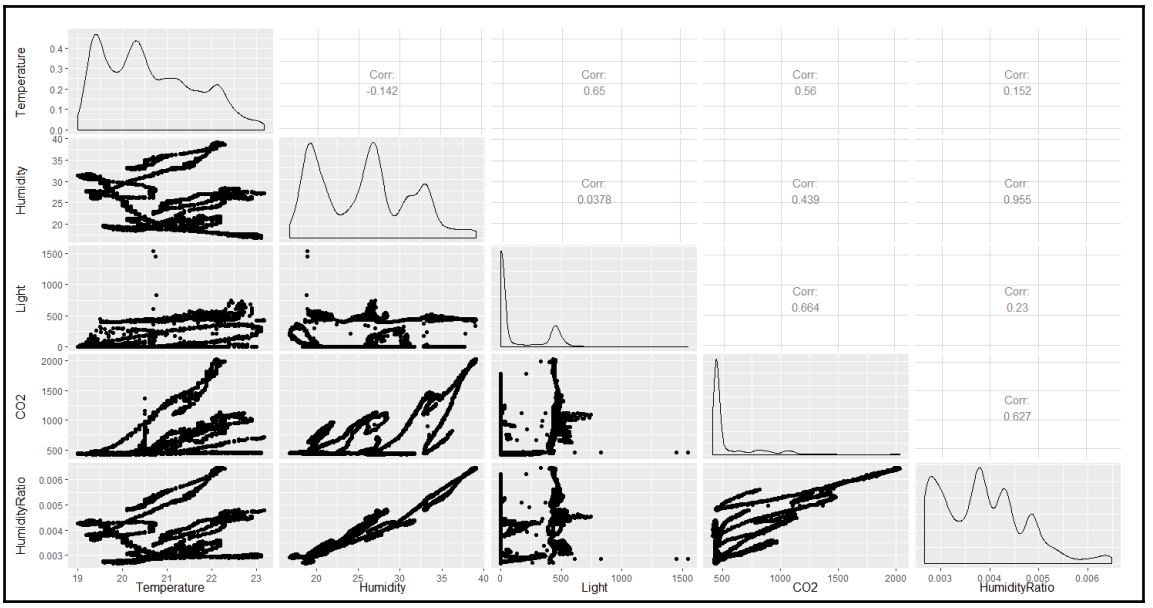
Here, μ is the mean and σ is the standard deviation of the feature. These distributions are very efficient for a dataset with a Gaussian distribution.

All the preceding methods are sensitive to outliers; there are other more robust approaches for normalization that you can explore, such as **Median Absolute Deviation** (**MAD**), tanh-estimator, and double sigmoid.

Visualizing dataset distribution

Let's look at the distribution of features for the occupation data:

```
> ggpairs(occupancy_train$data[, occupancy_train$xFeatures])
```



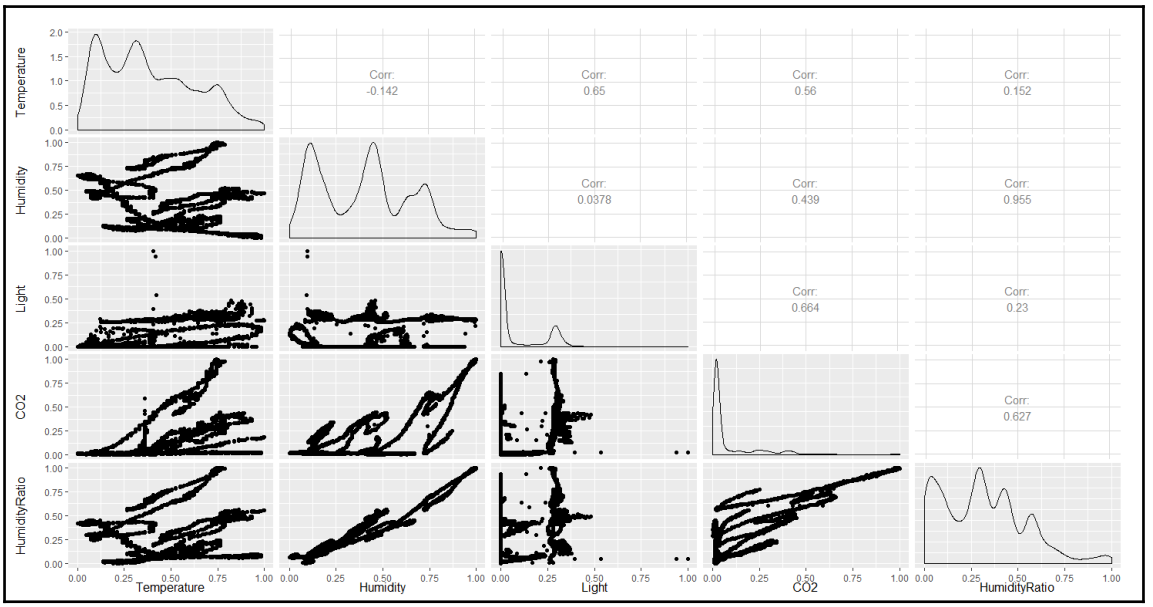
The figure shows that the features have linear correlations and the distributions are non-normal. The non-normality can be further validated using the Shapiro-Wilk test, using the `shapiro.test` function from R. Let's use min-max standardization for the occupation data.

How to do it...

1. Perform the following operation for data normalization:

```
minmax.normalize<-function(ds, scaler=NULL){
  if(is.null(scaler)){
    for(f in ds$xFeatures){
      scaler[[f]]$minval<-min(ds$data[,f])
      scaler[[f]]$maxval<-max(ds$data[,f])
      ds$data[,f]<-(ds$data[,f]-
scaler[[f]]$minval)/(scaler[[f]]$maxval-scaler[[f]]$minval)
    }
    ds$scaler<-scaler
  } else
  {
    for(f in ds$xFeatures){
      ds$data[,f]<-(ds$data[,f]-
scaler[[f]]$minval)/(scaler[[f]]$maxval-scaler[[f]]$minval)
    }
  }
  return(ds)
}
```

2. The `minmax.normalize` function normalizes the data using min-max normalization. When the `scaler` variable is `NULL`, it performs normalization using the dataset provided, or normalizes using `scaler` values. The normalized data pair plot is shown in the following figure:



This figure shows min-max normalization bringing the values within bounds *[0, 1]* and it does not change the distribution and correlations between features.

How to set up an autoencoder model

The next step is to set up the autoencoder model. Let's set up a vanilla autoencoder using TensorFlow:

1. Reset the `graph` and start `InteractiveSession`:

```
# Reset the graph and set-up a interactive session
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

2. Define the input parameter where `n` and `m` are the number of samples and features, respectively. To build, network `m` is used to set up the input parameter:

```
# Network Parameters
n_hidden_1 = 5 # 1st layer num features
n_input = length(xFeatures) # Number of input features
nRow<-nrow(occupancy_train)
```

   When `n_hidden_1` is low, the autoencoder is compressing the data and is referred to as an under-complete autoencoder; whereas, when `n_hidden_1` is large, then the autoencoder is sparse and is referred to as an over-complete autoencoder.

3. Define graph input parameters that include the input tensor and layer definitions for the encoder and decoder:

```
# Define input feature
x <- tf$constant(unlist(occupancy_train[, xFeatures]),
shape=c(nRow, n_input), dtype=np$float32)

# Define hidden and bias layer for encoder and decoders
hiddenLayerEncoder<-tf$Variable(tf$random_normal(shape(n_input,
n_hidden_1)), dtype=np$float32)
biasEncoder <- tf$Variable(tf$zeros(shape(n_hidden_1)),
dtype=np$float32)
hiddenLayerDecoder<-tf$Variable(tf$random_normal(shape(n_hidden_1,
n_input)))
biasDecoder <- tf$Variable(tf$zeros(shape(n_input)))
```

   The preceding script designs a single-layer encoder and decoder.

4. Define a function to evaluate the response:

```
auto_encoder<-function(x, hiddenLayerEncoder, biasEncoder){
  x_transform <- tf$nn$sigmoid(tf$add(tf$matmul(x,
hiddenLayerEncoder), biasEncoder))
  x_transform
}
```

   The `auto_encoder` function takes the node bias weights and computes the output. The same function can be used for `encoder` and `decoder` by passing respective weights.

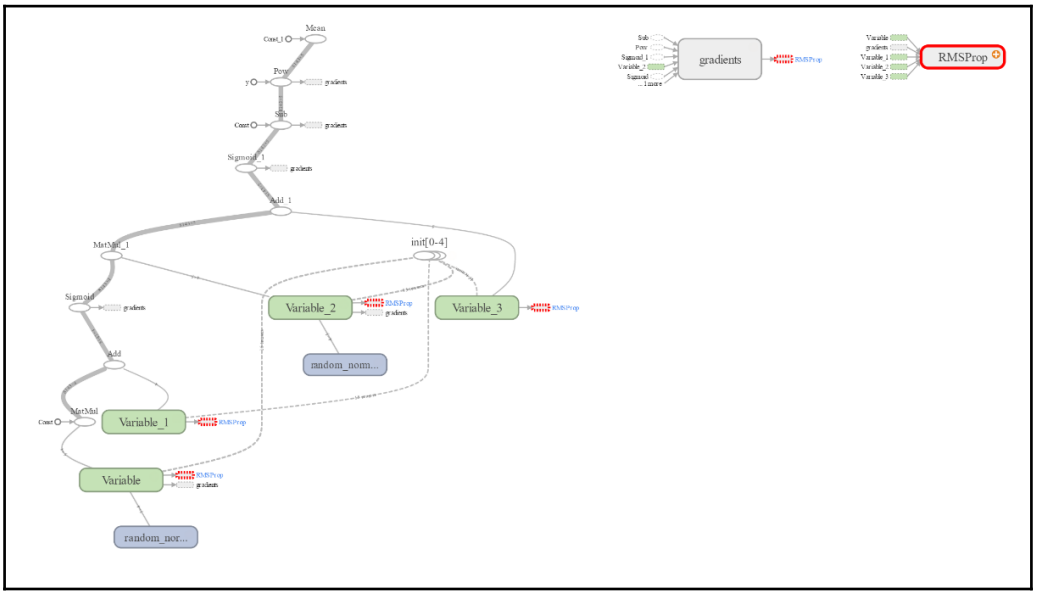
5. Create `encoder` and `decoder` objects by passing symbolic TensorFlow variables:

```
encoder_obj = auto_encoder(x,hiddenLayerEncoder, biasEncoder)
y_pred = auto_encoder(encoder_obj, hiddenLayerDecoder, biasDecoder)
```

6. The `y_pred` is the outcome from `decoder`, which takes the `encoder` object as input with nodes and bias weights:

```
Define loss function and optimizer module.
learning_rate = 0.01
cost = tf$reduce_mean(tf$pow(x - y_pred, 2))
optimizer = tf$train$RMSPropOptimizer(learning_rate)$minimize(cost)
```

The preceding script defines mean square error as the cost function, and uses `RMSPropOptimizer` from TensorFlow with 0.1 learning rate for the optimization of weights. The TensorFlow graph for the preceding model is shown in the following diagram:



Running optimization

The next step is to run optimizer optimization. Executing this process in TensorFlow consists of two steps:

1. The first step is parameter initialization of the variables defined in the graph. The initialization is performed by calling the `global_variables_initializer` function from TensorFlow:

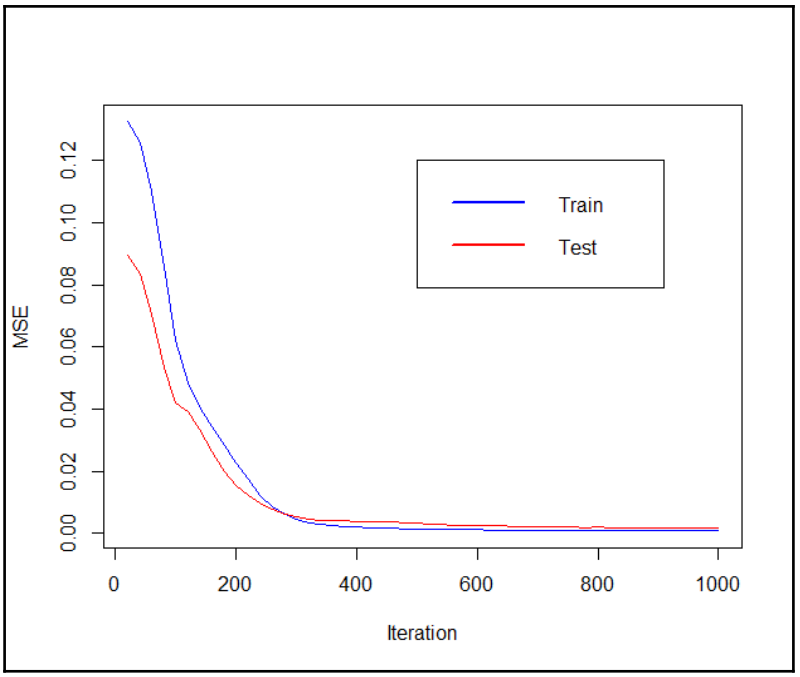
```
# Initializing the variables
init = tf$global_variables_initializer()
sess$run(init)
```

   Optimization is performed based on optimizing and monitoring the train and test performance:

```
costconvergence<-NULL
for (step in 1:1000) {
  sess$run(optimizer)
  if (step %% 20==0){
    costconvergence<-rbind(costconvergence, c(step, sess$run(cost),
sess$run(costt)))
    cat(step, "-", "Traing Cost ==>", sess$run(cost), "\n")
  }
}
```

2. The cost function from train and test can be observed to understand convergence of the model, as shown in the following figure:

```
costconvergence<-data.frame(costconvergence)
colnames(costconvergence)<-c("iter", "train", "test")
plot(costconvergence[, "iter"], costconvergence[, "train"], type =
"l", col="blue", xlab = "Iteration", ylab = "MSE")
lines(costconvergence[, "iter"], costconvergence[, "test"],
col="red")
legend(500,0.25, c("Train","Test"), lty=c(1,1),
lwd=c(2.5,2.5),col=c("blue","red"))
```



This graph shows that the model major `convergence` is at around **400** iterations; however, it is still converging at a very slow rate even after **1,000** iterations. The model is stable in both the train and holdout test datasets.

Setting up a regularized autoencoder

A regularized autoencoder extends the standard autoencoder by adding a regularization parameter to the `cost` function.

Getting ready

The regularized autoencoder is an extension of the standard autoencoder. The set-up will require:

1. TensorFlow installation in R and Python.
2. Implementation of a standard autoencoder.

How to do it...

The code setup for the autoencoder can directly be converted to a regularized autoencoder by replacing the cost definition with the following lines:

```
Lambda=0.01
cost = tf$reduce_mean(tf$pow(x - y_pred, 2))
Regularize_weights = tf$nn$l2_loss(weights)
cost = tf$reduce_mean(cost + lambda * Regularize_weights)
```

How it works...

As mentioned earlier, a regularized autoencoder extends the standard autoencoder by adding a regularization parameter to the cost function, shown as follows:

$$\sum L(x,\tilde{x}) + \lambda \left\| W_{ij}^2 \right\|$$

Here, λ is the regularization parameter and i and j are the node indexes with W representing the hidden layer weights for the autoencoder. The regularization autoencoder aims to ensure more robust encoding and prefers a low weight h function. The concept is further utilized to develop a contractive autoencoder, which utilizes the Frobenius norm of the Jacobian matrix on input, represented as follows:

$$L(\mathbf{X},\tilde{\mathbf{X}}) + \lambda \left\| J(x) \right\|^2$$

where **J(x)** is the Jacobian matrix and is evaluated as follows:

$$\left\| J(x) \right\|_F^2 = \sum_{ij} \frac{\partial h_j(x)}{\partial x_i}$$

For a linear encoder, a contractive encoder and regularized encoder converge to L2 weight decay. The regularization helps in making the autoencoder less sensitive to the input; however, the minimization of the cost function helps the model to capture the variation and remain sensitive to manifolds of high density. These autoencoders are also referred to as **contractive autoencoders**.

Fine-tuning the parameters of the autoencoder

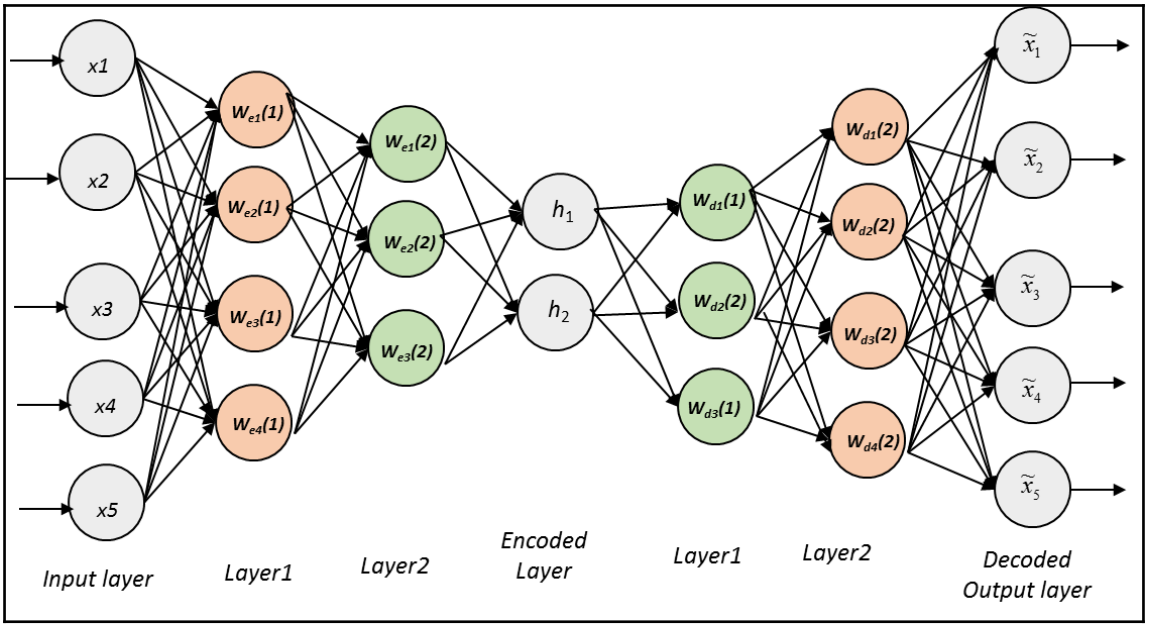
The autoencoder involves a couple of parameters to tune, depending on the type of autoencoder we are working on. The major parameters in an autoencoder include the following:

- Number of nodes in any hidden layer
- Number of hidden layers applicable for deep autoencoders
- Activation unit such as sigmoid, tanh, softmax, and ReLU activation functions
- Regularization parameters or weight decay terms on hidden unit weights
- Fraction of the signal to be corrupted in a denoising autoencoder
- Sparsity parameters in sparse autoencoders that control the expected activation of neurons in hidden layers
- Batch size, if using batch gradient descent learning; learning rate and momentum parameter for stochastic gradient descent
- Maximum iterations to be used for the training
- Weight initialization
- Dropout regularization if dropout is used

These hyperparameters can be trained by setting the problem as a grid search problem. However, each hyperparameter combination requires training the neuron weights for the hidden layer(s), which results in increasing computational complexity with an increase in the number of layers and number of nodes within each layer. To deal with these critical parameters and training issues, stacked autoencoder concepts have been proposed that train each layer separately to get pretrained weights, and then the model is fine-tuned using the obtained weights. This approach tremendously improves the training performance over the conventional mode of training.

Setting up stacked autoencoders

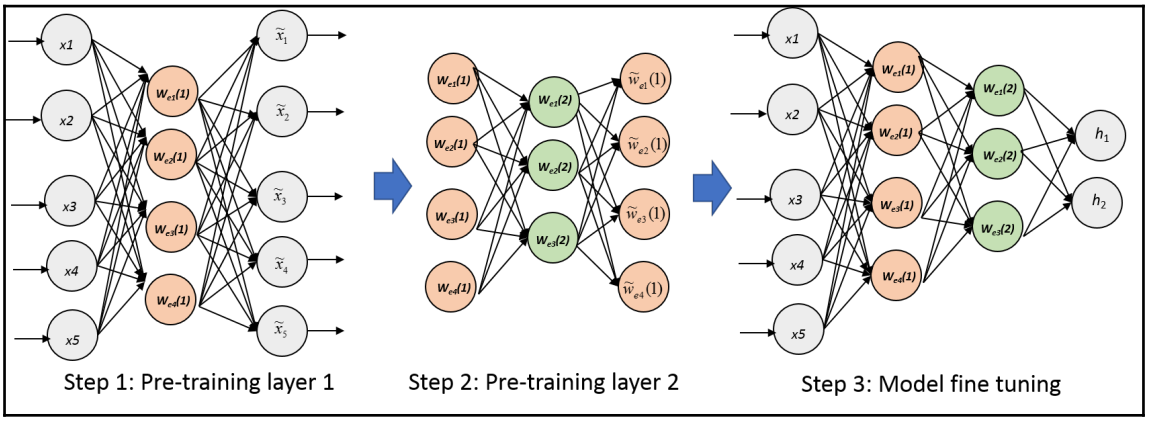
The stacked autoencoder is an approach to train deep networks consisting of multiple layers trained using the greedy approach. An example of a stacked autoencoder is shown in the following diagram:



An example of a stacked autoencoder

Getting ready

The preceding diagram demonstrates a stacked autoencoder with two layers. A stacked autoencoder can have *n* layers, where each layer is trained using one layer at a time. For example, the previous layer will be trained as follows:



Training of a stacked autoencoder

The initial pre-training of layer 1 is obtained by training it over the actual input x_i . The first step is to optimize the *We(1)* layer of the encoder with respect to output X. The second step in the preceding example is to optimize the weights *We(2)* in the second layer, using *We(1)* as input and output. Once all the layers of *We(i)* where *i*=1, 2, ...,*n* is number of layers are pretrained, model fine-tuning is performed by connecting all the layers together, as shown in step 3 of the preceding diagram. The concept can also be applied to denoising to train multilayer networks, which is known as a stacked denoising autoencoder. The code developed in a denoising autoencoder can be easily tweaked to develop a **stacked denoising autoencoder**, which is an extension of the stacked autoencoder.

The requirements for this recipe are:

1. R should be installed.
2. `SAENET` package, The package can be download from Cran using the command `install.packages("SAENET")`.

How to do it...

There are other popular libraries in R to develop stacked autoencoders. Let's utilize the `SAENET` package from R to set up a stacked autoencoder. The `SAENET` is a stacked autoencoder implementation, using a feedforward neural network using the `neuralnet` package from CRAN:

1. Get the `SAENET` package from the CRAN repository, if not installed already:

```
install.packages("SAENET")
```

2. Load all library dependencies:

```
require(SAENET)
```

3. Load the train and test occupancy dataset using `load_occupancy_data`:

```
occupancy_train <-load_occupancy_data(train=T)
occupancy_test <- load_occupancy_data(train = F)
```

4. Normalize the dataset using the `minmax.normalize` function:

```
# Normalize dataset
occupancy_train<-minmax.normalize(occupancy_train, scaler = NULL)
occupancy_test<-minmax.normalize(occupancy_test, scaler =
occupancy_train$scaler)
```

5. The stacked autoencoder model can be built using the `SAENET.train` train function from the `SAENET` package:

```
# Building Stacked Autoencoder
SAE_obj<-SAENET.train(X.train= subset(occupancy_train$data,
select=-c(Occupancy)), n.nodes=c(4, 3, 2), unit.type ="tanh",
lambda = 1e-5, beta = 1e-5, rho = 0.01, epsilon = 0.01,
max.iterations=1000)
```

The output of the last node can be extracted using the `SAE_obj[[n]]$X.output` command.

Setting up denoising autoencoders

Denoising autoencoders are a special kind of autoencoder with a focus on extracting robust features from the input dataset. Denoising autoencoders are similar to the previous model except with a major difference that the data is corrupted before training the network. Different approaches for corruption can be used such as masking, which induces random error into the data.

Getting ready

Let's use the CIFAR-10 image data to set up a denoising dataset:

- Download the CIFAR-10 dataset using the `download_cifar_data` function (covered in `Chapter 3`, *Convolution Neural Network*)
- TensorFlow installation in R and Python

How to do it...

We first need to read the dataset.

Reading the dataset

1. Load the `CIFAR` dataset using the steps explained in `Chapter 3`, *Convolution Neural Network*. The data files `data_batch_1` and `data_batch_2` are used to train. The `data_batch_5` and `test_batch` files are used for validation and testing, respectively. The data can be flattened using the `flat_data` function:

```
train_data <- flat_data(x_listdata = images.rgb.train)
test_data <- flat_data(x_listdata = images.rgb.test)
valid_data <- flat_data(x_listdata = images.rgb.valid)
```

2. The `flat_data` function flattens the dataset as *NCOL = (Height * Width * number of channels)*, thus the dimension of the dataset is (# of images X NCOL). The images in `CIFAR` are 32 x 32 with three RGB channels; thus, we obtain 3,072 columns after data flattening:

```
> dim(train_data$images)
[1] 40000  3072
```

Corrupting data to train

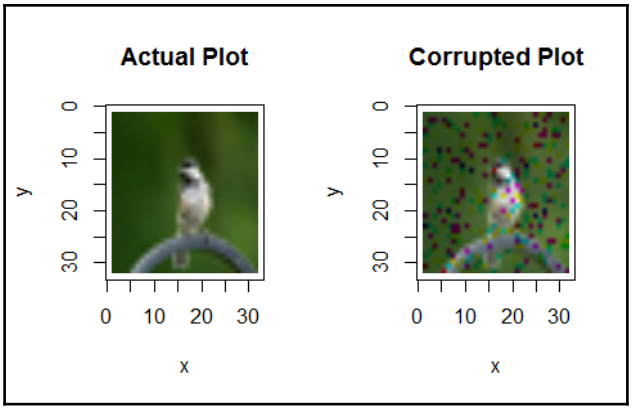
1. The next critical function needed to set up a denoising autoencoder is data corruption:

```
# Add noise using masking or salt & pepper noise method
add_noise<-function(data, frac=0.10, corr_type=c("masking",
"saltPepper", "none")){
  if(length(corr_type)>1) corr_type<-corr_type[1]
  # Assign a copy of data
  data_noise = data
  # Evaluate chaining parameters for autoencoder
  nROW<-nrow(data)
  nCOL<-ncol(data)
  nMask<-floor(frac*nCOL)
  if(corr_type=="masking"){
    for( i in 1:nROW){
      maskCol<-sample(nCOL, nMask)
      data_noise[i,maskCol,,]<-0
    }
  } else if(corr_type=="saltPepper"){
    minval<-min(data[,,1,])
    maxval<-max(data[,,1,])
    for( i in 1:nROW){
      maskCol<-sample(nCOL, nMask)
      randval<-runif(length(maskCol))
      ixmin<-randval<0.5
      ixmax<-randval>=0.5
      if(sum(ixmin)>0) data_noise[i,maskCol[ixmin],,]<-minval
      if(sum(ixmax)>0) data_noise[i,maskCol[ixmax],,]<-maxval
    }
  } else
  {
    data_noise<-data
  }
  return(data_noise)
}
```

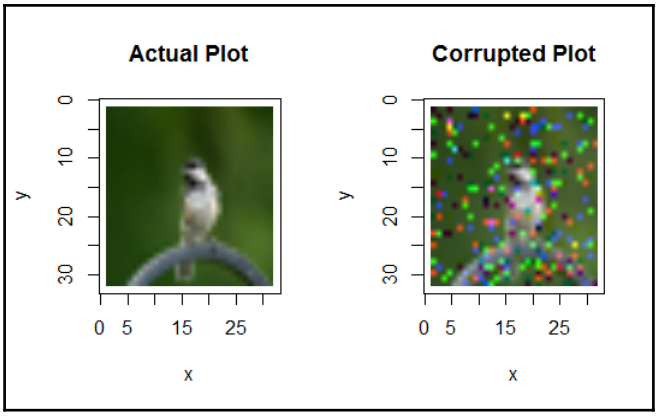
2. The CIFAR-10 data can be corrupted using the following script:

```
# Corrupting input signal
xcorr<-add_noise(train_data$images, frac=0.10, corr_type="masking")
```

3. An example image after corruption is as follows:



4. The preceding figure uses the masking approach to add noise. This method adds zero values at random image locations with a defined fraction. Another approach to add noise is by using salt & pepper noise. This method selects random locations in the image and replaces them, adding min or max values to the image using the flip of coin principle. An example of the salt and pepper approach for data corruption is shown in the following figure:



The data corruption helps the autoencoder to learn more robust representation.

Setting up a denoising autoencoder

The next step is to set up the autoencoder model:

1. First, reset the graph and start an interactive session as follows:

```
# Reset the graph and set-up an interactive session
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

2. The next step is to define two placeholders for the input signal and corrupt signal:

```
# Define Input as Placeholder variables
x = tf$placeholder(tf$float32, shape=shape(NULL, img_size_flat),
name='x')
x_corrput<-tf$placeholder(tf$float32, shape=shape(NULL,
img_size_flat), name='x_corrput')
```

   The `x_corrupt` will be used as input in the autoencoder, and *x* is the actual image that will be used as output.

3. Set up a denoising autoencoder function as shown in the following code:

```
# Setting-up denoising autoencoder
denoisingAutoencoder<-function(x, x_corrput, img_size_flat=3072,
hidden_layer=c(1024, 512), out_img_size=256){

  # Building Encoder
  encoder = NULL
  n_input<-img_size_flat
  curentInput<-x_corrput
  layer<-c(hidden_layer, out_img_size)
  for(i in 1:length(layer)){
    n_output<-layer[i]
    W = tf$Variable(tf$random_uniform(shape(n_input, n_output),
-1.0 / tf$sqrt(n_input), 1.0 / tf$sqrt(n_input)))
    b = tf$Variable(tf$zeros(shape(n_output)))
    encoder<-c(encoder, W)
    output = tf$nn$tanh(tf$matmul(curentInput, W) + b)
    curentInput = output
    n_input<-n_output
  }
  # latent representation
  z = curentInput
  encoder<-rev(encoder)
  layer_rev<-c(rev(hidden_layer), img_size_flat)
```

```
  # Build the decoder using the same weights
  decoder<-NULL
  for(i in 1:length(layer_rev)){
    n_output<-layer_rev[i]
    W = tf$transpose(encoder[[i]])
    b = tf$Variable(tf$zeros(shape(n_output)))
    output = tf$nn$tanh(tf$matmul(curentInput, W) + b)
    curentInput = output
  }
  # now have the reconstruction through the network
  y = curentInput
  # cost function measures pixel-wise difference
  cost = tf$sqrt(tf$reduce_mean(tf$square(y - x)))
  return(list("x"=x, "z"=z, "y"=y, "x_corrput"=x_corrput,
"cost"=cost))
}
```

4. Create the denoising object:

```
# Create denoising AE object
dae_obj<-denoisingAutoencoder(x, x_corrput, img_size_flat=3072,
hidden_layer=c(1024, 512), out_img_size=256)
```

5. Set the cost function:

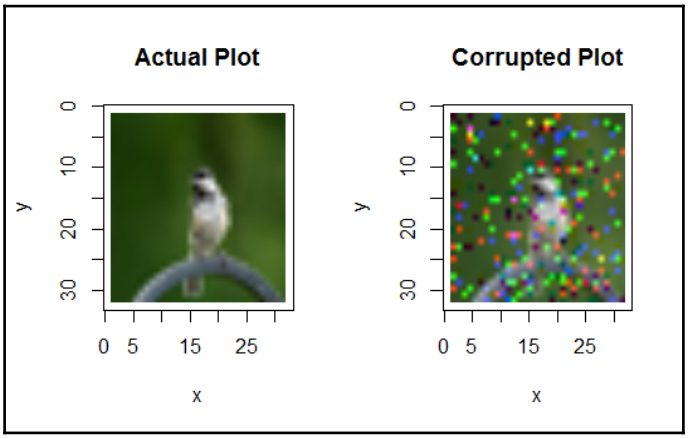
```
# Learning set-up
learning_rate = 0.001
optimizer =
tf$train$AdamOptimizer(learning_rate)$minimize(dae_obj$cost)
```

6. Run optimization:

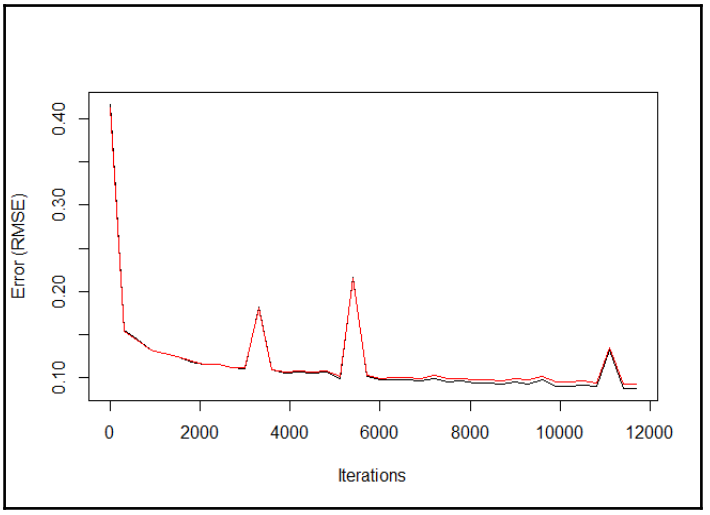
```
# We create a session to use the graph
sess$run(tf$global_variables_initializer())
for(i in 1:500){
  spls <- sample(1:dim(xcorr)[1],1000L)
  if (i %% 1 == 0) {
    x_corrput_ds<-add_noise(train_data$images[spls, ], frac = 0.3,
corr_type = "masking")
    optimizer$run(feed_dict = dict(x=train_data$images[spls, ],
x_corrput=x_corrput_ds))
    trainingCost<-dae_obj$cost$eval((feed_dict =
dict(x=train_data$images[spls, ], x_corrput=x_corrput_ds)))
    cat("Training Cost - ", trainingCost, "\n")
  }
}
```

How it works...

The autoencoder keeps learning about the function form for the feature to capture the relationship between input and output. An example of how the computer is visualizing the image after 1,000 iterations is shown in the following figure:



After 1,000 iterations, the computer can distinguish between a major part of the object and environment. As we run the algorithm further to fine-tune the weights, the computer keeps learning more features about the object itself, as shown in the following figure:



The preceding graph shows that the model is still learning, but the learning rate has become smaller over the iterations as it starts learning fine features about objects, as shown in the following image. There are instances when the model starts ascending instead of descending, due to batch gradient descent:

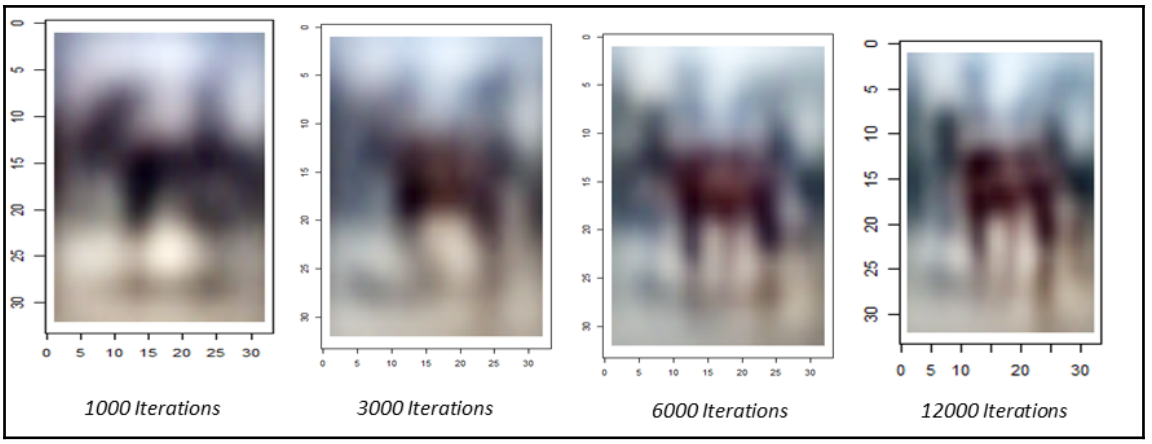
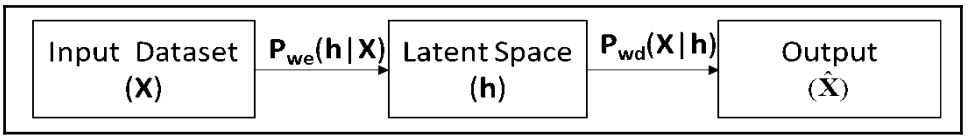


Illustration of learning using denoising autoencoder

Building and comparing stochastic encoders and decoders

Stochastic encoders fall into the domain of generative modeling, where the objective is to learn join probability *P(X)* over given data *X* transformed into another high-dimensional space. For example, we want to learn about images and produce similar, but not exactly the same, images by learning about pixel dependencies and distribution. One of the popular approaches in generative modeling is **Variational autoencoder** (**VAE**), which combines deep learning with statistical inference by making a strong distribution assumption on *h ~ P(h),* such as Gaussian or Bernoulli. For a given weight *W*, the *X* can be sampled from the distribution as *Pw(X|h)*. An example of VAE architecture is shown in the following diagram:



The cost function of VAE is based on log likelihood maximization. The cost function consists of reconstruction and regularization error terms:

Cost = Reconstruction Error + Regularization Error

The **reconstruction error** is how well we could map the outcome with the training data and the **regularization error** puts a penalty on the distribution formed at the encoder and decoder.

Getting ready

TensorFlow needs to be installed and loaded in the environment:

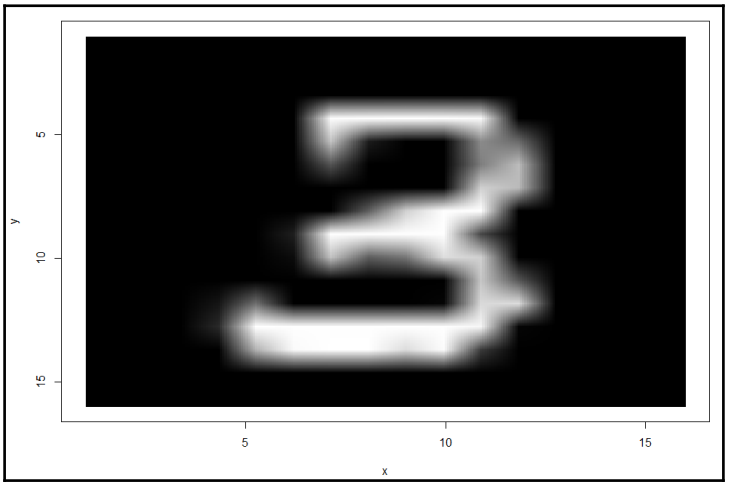
```
require(tensorflow)
```

Dependencies need to be loaded:

```
require(imager)
require(caret)
```

The `MNIST` dataset needs to be loaded. The dataset is normalized using the following script:

```
# Normalize Dataset
normalizeObj<-preProcess(trainData, method="range")
trainData<-predict(normalizeObj, trainData)
validData<-predict(normalizeObj, validData)
```



How to do it...

1. The `MNIST` dataset is used to demonstrate the concept of sparse decomposition. The `MNIST` dataset uses handwritten digits. It is downloaded from the `tensorflow` dataset library. The dataset consists of handwritten images of `28 x 28` pixels. It consists of 55,000 training examples, 10,000 test examples, and 5,000 test examples. The dataset can be downloaded from the `tensorflow` library using the following script:

```
library(tensorflow)
datasets <- tf$contrib$learn$datasets
mnist <- datasets$mnist$read_data_sets("MNIST-data", one_hot =
TRUE)
```

2. For computational simplicity, the `MNIST` image size is reduced from `28 x 28` pixels to `16 x 16` pixels using the following function:

```
# Function to reduce image size
reduceImage<-function(actds, n.pixel.x=16, n.pixel.y=16){
  actImage<-matrix(actds, ncol=28, byrow=FALSE)
  img.col.mat <- imappend(list(as.cimg(actImage)),"c")
  thmb <- resize(img.col.mat, n.pixel.x, n.pixel.y)
  outputImage<-matrix(thmb[,,1,1], nrow = 1, byrow = F)
  return(outputImage)
}
```

3. The following script can be used to prepare the `MNIST` training data with a 16 x 16 pixel image:

```
# Covert train data to 16 x 16  image
trainData<-t(apply(mnist$train$images, 1, FUN=reduceImage))
validData<-t(apply(mnist$test$images, 1, FUN=reduceImage))
```

4. The `plot_mnist` function can be used to visualize the selected `MNIST` image:

```
# Function to plot MNIST dataset
plot_mnist<-function(imageD, pixel.y=16){
  actImage<-matrix(imageD, ncol=pixel.y, byrow=FALSE)
  img.col.mat <- imappend(list(as.cimg(actImage)), "c")
  plot(img.col.mat)
}
```

Setting up a VAE model

1. Start a new TensorFlow environment:

```
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

2. Define network parameters:

```
n_input=256
n.hidden.enc.1<-64
```

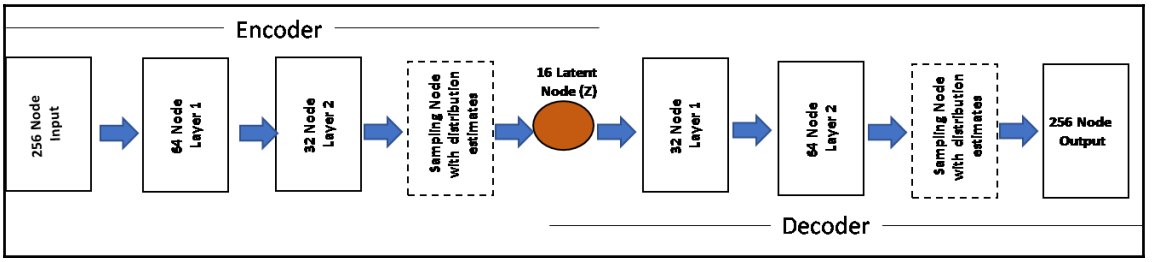
3. Start a new TensorFlow environment:

```
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

4. Define network parameters:

```
n_input=256
n.hidden.enc.1<-64
```

The preceding parameter will form a VAE network as follows:



5. Define the model initialization function, defining weights and biases at each layer of `encoder` and `decoder`:

```
model_init<-function(n.hidden.enc.1, n.hidden.enc.2,
                                n.hidden.dec.1,  n.hidden.dec.2,
                                n_input, n_h)
{ weights<-NULL
 ############################
 # Set-up Encoder
 ############################
 # Initialize Layer 1 of encoder
 weights[["encoder_w"]][["h1"]]=tf$Variable(xavier_init(n_input,
```

```
 n.hidden.enc.1))
 weights[["encoder_w"]]
[["h2"]]=tf$Variable(xavier_init(n.hidden.enc.1, n.hidden.enc.2))
weights[["encoder_w"]][["out_mean"]]=tf$Variable(xavier_init(n.hidd
en.enc.2, n_h))
weights[["encoder_w"]][["out_log_sigma"]]=tf$Variable(xavier_init(n
.hidden.enc.2, n_h))
weights[["encoder_b"]][["b1"]]=tf$Variable(tf$zeros(shape(n.hidden.
enc.1), dtype=tf$float32))
weights[["encoder_b"]][["b2"]]=tf$Variable(tf$zeros(shape(n.hidden.
enc.2), dtype=tf$float32))
weights[["encoder_b"]][["out_mean"]]=tf$Variable(tf$zeros(shape(n_h
), dtype=tf$float32))
weights[["encoder_b"]][["out_log_sigma"]]=tf$Variable(tf$zeros(shap
e(n_h), dtype=tf$float32))
 ############################
 # Set-up Decoder
 ############################
 weights[['decoder_w']][["h1"]]=tf$Variable(xavier_init(n_h,
n.hidden.dec.1))
weights[['decoder_w']][["h2"]]=tf$Variable(xavier_init(n.hidden.dec
.1, n.hidden.dec.2))
weights[['decoder_w']][["out_mean"]]=tf$Variable(xavier_init(n.hidd
en.dec.2, n_input))
weights[['decoder_w']][["out_log_sigma"]]=tf$Variable(xavier_init(n
.hidden.dec.2, n_input))
weights[['decoder_b']][["b1"]]=tf$Variable(tf$zeros(shape(n.hidden.
dec.1), dtype=tf$float32))
weights[['decoder_b']][["b2"]]=tf$Variable(tf$zeros(shape(n.hidden.
dec.2), dtype=tf$float32))
weights[['decoder_b']][["out_mean"]]=tf$Variable(tf$zeros(shape(n_i
nput), dtype=tf$float32))
weights[['decoder_b']][["out_log_sigma"]]=tf$Variable(tf$zeros(shap
e(n_input), dtype=tf$float32))
 return(weights)
}
```

The `model_init` function returns `weights`, which is a two-dimensional list. The first dimension captures the weight's association and type. For example, it describes if the `weights` variable is assigned to the encoder or decoder and if it stores the weight of the node or bias. The `xavier_init` function in `model_init` is used to assign initial weights for model training:

```
# Xavier Initialization using Uniform distribution
xavier_init<-function(n_inputs, n_outputs, constant=1){
  low = -constant*sqrt(6.0/(n_inputs + n_outputs))
  high = constant*sqrt(6.0/(n_inputs + n_outputs))
```

```
  return(tf$random_uniform(shape(n_inputs, n_outputs), minval=low,
maxval=high, dtype=tf$float32))
}
```

6. Set up the encoder evaluation function:

```
# Encoder update function
vae_encoder<-function(x, weights, biases){
  layer_1 = tf$nn$softplus(tf$add(tf$matmul(x, weights[['h1']]),
biases[['b1']]))
  layer_2 = tf$nn$softplus(tf$add(tf$matmul(layer_1,
weights[['h2']]), biases[['b2']]))
  z_mean = tf$add(tf$matmul(layer_2, weights[['out_mean']]),
biases[['out_mean']])
  z_log_sigma_sq = tf$add(tf$matmul(layer_2,
weights[['out_log_sigma']]), biases[['out_log_sigma']])
  return (list("z_mean"=z_mean, "z_log_sigma_sq"=z_log_sigma_sq))
}
```

The `vae_encoder` computes the mean and variance to sample the layer, using the weights and bias from the hidden layer.

7. Set up the decoder evaluation function:

```
# Decoder update function
vae_decoder<-function(z, weights, biases){
  layer1<-tf$nn$softplus(tf$add(tf$matmul(z, weights[["h1"]]),
biases[["b1"]]))
  layer2<-tf$nn$softplus(tf$add(tf$matmul(layer1, weights[["h2"]]),
biases[["b2"]]))
  x_reconstr_mean<-tf$nn$sigmoid(tf$add(tf$matmul(layer2,
weights[['out_mean']]), biases[['out_mean']]))
  return(x_reconstr_mean)
}
```

The `vae_decoder` function computes the mean and standard deviation associated with the sampling layer at output and average output.

8. Set up the function for reconstruction estimation:

```
# Parameter evaluation
network_ParEval<-function(x, network_weights, n_h){

  distParameter<-vae_encoder(x, network_weights[["encoder_w"]],
network_weights[["encoder_b"]])
  z_mean<-distParameter$z_mean
  z_log_sigma_sq <-distParameter$z_log_sigma_sq
```

```
  # Draw one sample z from Gaussian distribution
  eps = tf$random_normal(shape(BATCH, n_h), 0, 1, dtype=tf$float32)
  # z = mu + sigma*epsilon
  z = tf$add(z_mean, tf$multiply(tf$sqrt(tf$exp(z_log_sigma_sq)),
eps))
  # Use generator to determine mean of
  # Bernoulli distribution of reconstructed input
  x_reconstr_mean <- vae_decoder(z, network_weights[["decoder_w"]],
network_weights[["decoder_b"]])
  return(list("x_reconstr_mean"=x_reconstr_mean,
"z_log_sigma_sq"=z_log_sigma_sq, "z_mean"=z_mean))
}
```

9. Define the cost function for optimization:

```
# VAE cost function
vae_optimizer<-function(x, networkOutput){
  x_reconstr_mean<-networkOutput$x_reconstr_mean
  z_log_sigma_sq<-networkOutput$z_log_sigma_sq
  z_mean<-networkOutput$z_mean
  loss_reconstruction<--1*tf$reduce_sum(x*tf$log(1e-10 +
x_reconstr_mean)+
                                        (1-x)*tf$log(1e-10 + 1 -
x_reconstr_mean), reduction_indices=shape(1))
  loss_latent<--0.5*tf$reduce_sum(1+z_log_sigma_sq-
tf$square(z_mean)-
                                  tf$exp(z_log_sigma_sq),
reduction_indices=shape(1))
  cost = tf$reduce_mean(loss_reconstruction + loss_latent)
  return(cost)
}
```

10. Set up the model to train:

```
# VAE Initialization
x = tf$placeholder(tf$float32, shape=shape(NULL, img_size_flat),
name='x')
network_weights<-model_init(n.hidden.enc.1, n.hidden.enc.2,
                            n.hidden.dec.1,  n.hidden.dec.2,
                            n_input, n_h)
networkOutput<-network_ParEval(x, network_weights, n_h)
cost=vae_optimizer(x, networkOutput)
optimizer = tf$train$AdamOptimizer(lr)$minimize(cost)
```

11. Run optimization:

```
sess$run(tf$global_variables_initializer())
for(i in 1:ITERATION){
```

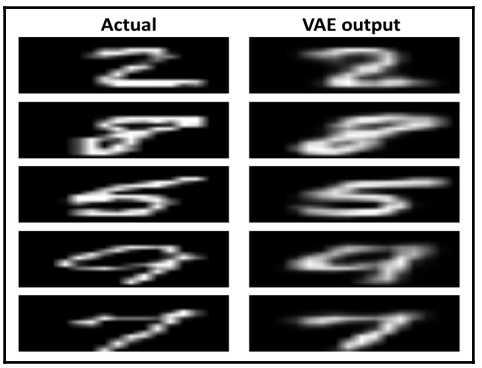
```
  spls <- sample(1:dim(trainData)[1],BATCH)
  out<-optimizer$run(feed_dict = dict(x=trainData[spls,]))
  if (i %% 100 == 0){
  cat("Iteration - ", i, "Training Loss - ",  cost$eval(feed_dict =
dict(x=trainData[spls,])), "\n")
  }
}
```

Output from the VAE autoencoder

1. The outcome can be generated using the following script:

```
spls <- sample(1:dim(trainData)[1],BATCH)
networkOutput_run<-sess$run(networkOutput, feed_dict =
dict(x=trainData[spls,]))

# Plot reconstructured Image
x_sample<-trainData[spls,]
NROW<-nrow(networkOutput_run$x_reconstr_mean)
n.plot<-5
par(mfrow = c(n.plot, 2), mar = c(0.2, 0.2, 0.2, 0.2), oma = c(3,
3, 3, 3))
pltImages<-sample(1:NROW,n.plot)
for(i in pltImages){
  plot_mnist(x_sample[i,])
  plot_mnist(networkOutput_run$x_reconstr_mean[i,])
}
```

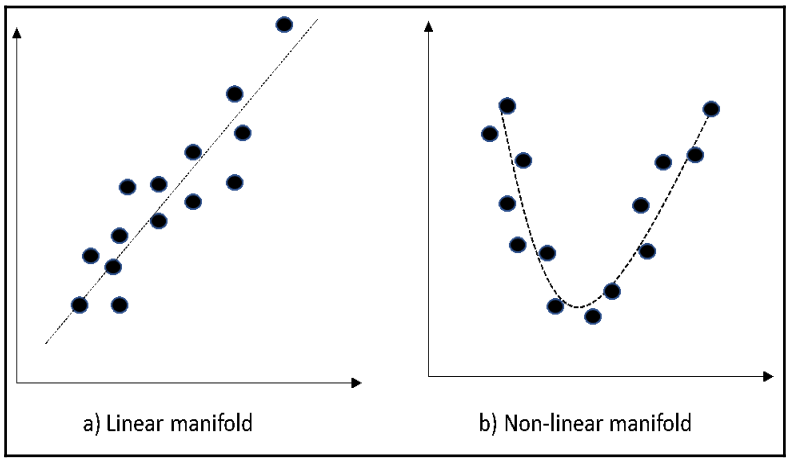
The outcome obtained after `20,000` iterations from the preceding VAE autoencoder is shown in the following figure:



Additionally, as VAE is a generative model, the outcome is not an exact replica of the input and would vary with runs, as a representative sample is extracted from the estimated distribution.

Learning manifolds from autoencoders

Manifold learning is an approach in machine learning that assumes that data lies on a manifold of a much lower dimension. These manifolds can be linear or non-linear. Thus, the area tries to project the data from high-dimension space to a low dimension. For example, **principle component analysis** (**PCA**) is an example of linear manifold learning whereas an autoencoder is a **non-linear dimensionality reduction** (**NDR**) with the ability to learn non-linear manifolds in low dimensions. A comparison of linear and non-linear manifold learning is shown in the following figure:



As you can see from graph **a)**, the data is residing at a linear manifold, whereas in graph graph **b)**, the data is residing on a second-order non-linear manifold.

How to do it...

Let's take an output from the stacked autoencoder section and analyze how manifolds look when transferred into a different dimension.

Setting up principal component analysis

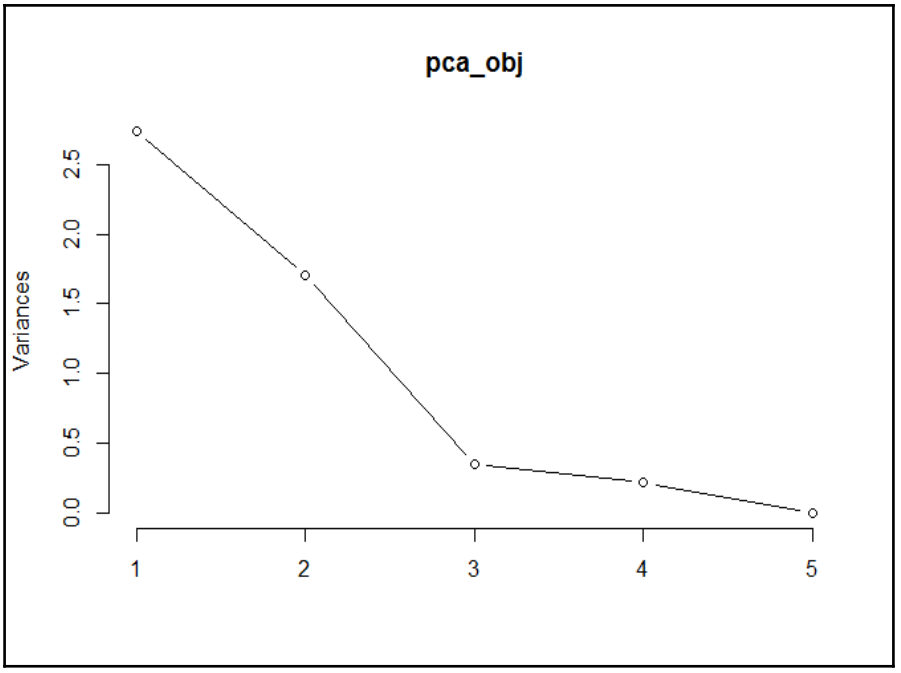
1. Before getting into non-linear manifolds, let's analyze principal component analysis on the occupancy data:

```
# Setting-up principal component analysis
pca_obj <- prcomp(occupancy_train$data,
                  center = TRUE,
                  scale. = TRUE)
                  scale. = TRUE)
```

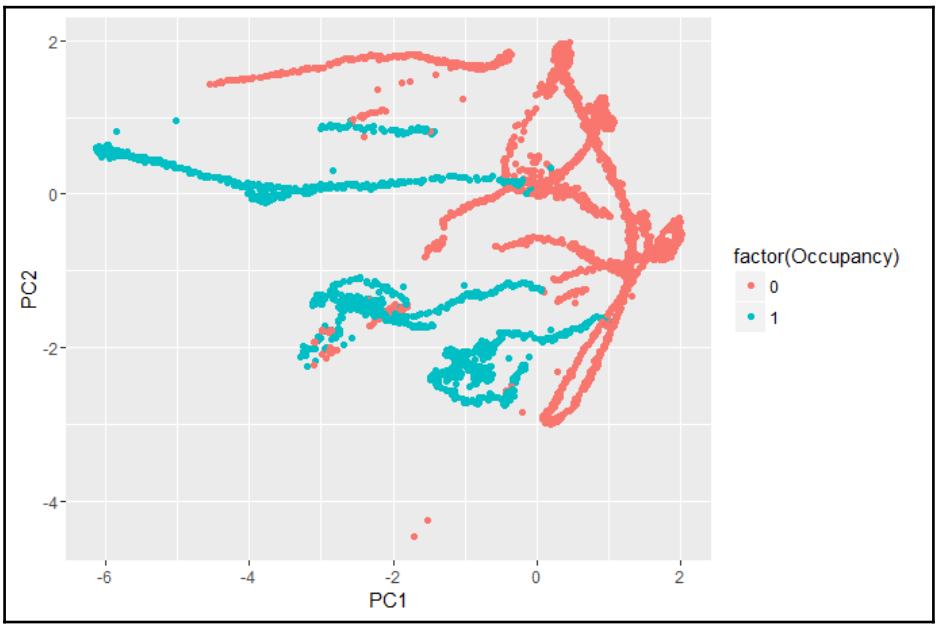
2. The preceding function will transform the data into six orthogonal directions specified as linear combinations of features. The variance explained by each dimension can be viewed using the following script:

```
plot(pca_obj, type = "l")
```

3. The preceding command will plot the variance across principal components, as shown in the following figure:



4. For the occupancy dataset, the first two principal components capture the majority of the variation, and when the principal component is plotted, it shows separation between the positive and negative classes for occupancy, as shown in the following figure:

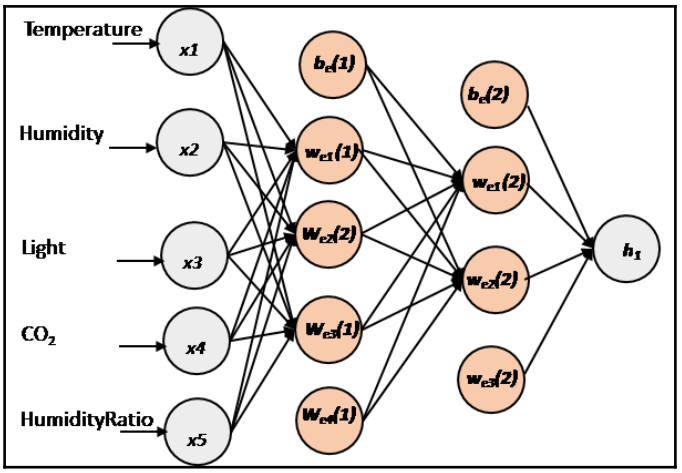


Output from the first two principal components

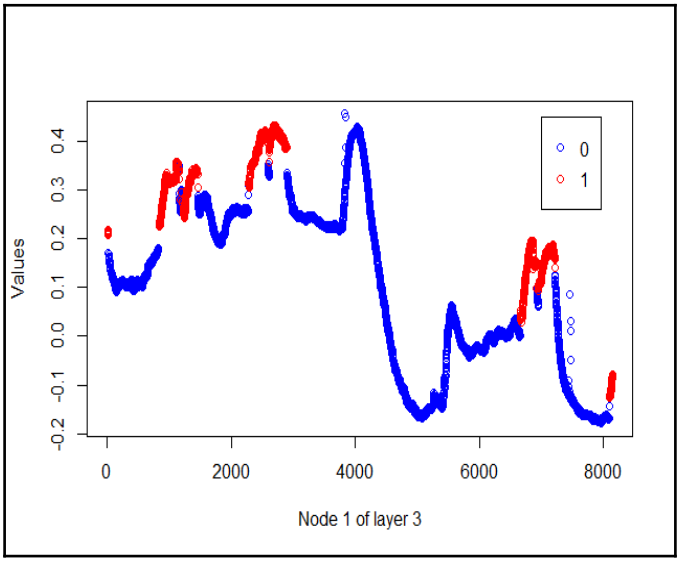
5. Let's visualize the manifold in a low dimension learned by the autoencoder. Let's use only one dimension to visualize the outcome, as follows:

```
SAE_obj<-SAENET.train(X.train= subset(occupancy_train$data,
select=-c(Occupancy)), n.nodes=c(4, 3, 1), unit.type ="tanh",
lambda = 1e-5, beta = 1e-5, rho = 0.01, epsilon = 0.01,
max.iterations=1000)
```

6. The encoder architecture for the preceding script is shown as follows:



The hidden layer outcome with one latent node from the stacked autoencoder is shown as follows:

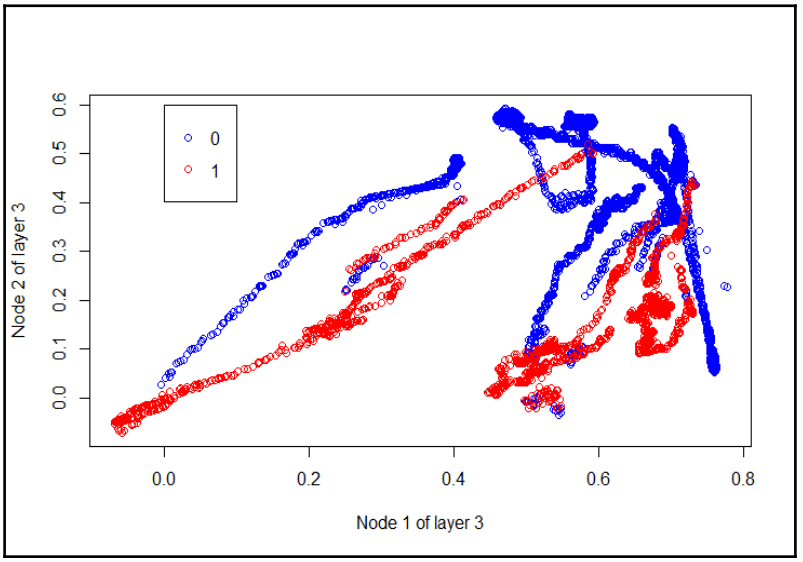


7. The preceding graph shows that occupancy is true at the peaks of the latent variables. However, the peaks are found at different values. Let's increase the latent variables 2, as captured by PCA. The model can be developed and data can be plotted using the following script:

```
SAE_obj<-SAENET.train(X.train= subset(occupancy_train$data,
```

```
select=-c(Occupancy)), n.nodes=c(4, 3, 2), unit.type ="tanh",
lambda = 1e-5, beta = 1e-5, rho = 0.01, epsilon = 0.01,
max.iterations=1000)

# plotting encoder values
plot(SAE_obj[[3]]$X.output[,1], SAE_obj[[3]]$X.output[,2],
col="blue", xlab = "Node 1 of layer 3", ylab = "Node 2 of layer 3")
ix<-occupancy_train$data[,6]==1
points(SAE_obj[[3]]$X.output[ix,1], SAE_obj[[3]]$X.output[ix,2],
col="red")
```

8. The encoded values with two layers are shown in the following diagram:



Evaluating the sparse decomposition

The sparse autoencoder is also known as over-complete representation and has a higher number of nodes in the hidden layer. The sparse autoencoders are usually executed with the sparsity parameter (regularization), which acts as a constraint and restricts the node to being active. The sparsity can also be assumed as nodes dropout caused due to sparsity constraints. The loss function for a sparse autoencoder consists of a reconstruction error, a regularization term to contain the weight decay, and KL divergence for sparsity constraint. The following representation gives a very good illustration of what we are talking about:

$$\|\mathbf{X} - f(\mathbf{h})\|^2 + \frac{\lambda}{2}\|\mathbf{W}\|^2 + \beta J_{KL}(\rho\|\hat{\rho})$$

where, $\mathbf{X}$ and $f(\mathbf{h})$ represent input matrix and output from decoder. The $\mathbf{W}$ represent weight matrix for nodes and λ and β are regularization parameter and penalty to sparsity term. The term $J_{KL}(\rho\|\hat{\rho})$ is KL divergence value captured for sparsity term ρ. KL divergence for given sparsity parameter can be computed as

$$J_{KL}(\rho\|\hat{\rho}) = \sum_{i=1}^{n'} \rho \log(\frac{\rho}{\hat{\rho}_i}) + (1-\rho)\log(\frac{1-\rho}{1-\hat{\rho}_i})$$

where, n' is number of nodes in layer $\mathbf{h}$ and $\hat{\rho}$ is vector of average activities in hidden layer on normalized neurons.

Getting ready

1. The dataset is loaded and set up.
2. Install and load the `autoencoder` package using the following script:

```
install.packages("autoencoder")
require(autoencoder)
```

How to do it...

1. The standard autoencoder code of TensorFlow can easily be extended to the sparse autoencoder module by updating the cost function. This section will introduce the autoencoder package of R, which comes with built-in functionality to run the sparse autoencoder:

```
### Setting-up parameter
nl<-3
N.hidden<-100
unit.type<-"logistic"
lambda<-0.001
rho<-0.01
beta<-6
```

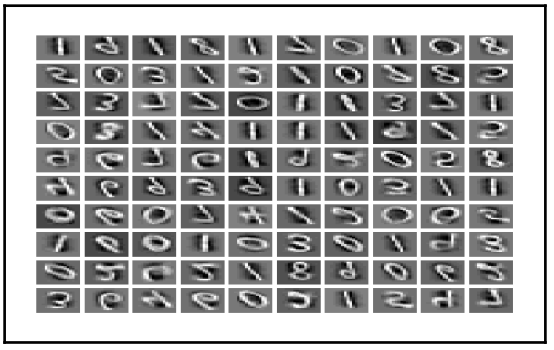
```
max.iterations<-2000
epsilon<-0.001

### Running sparse autoencoder
spe_ae_obj <- autoencode(X.train=trainData,  X.test = validData,
nl=nl, N.hidden=N.hidden,
unit.type=unit.type,lambda=lambda,beta=beta,
epsilon=epsilon,rho=rho,max.iterations=max.iterations, rescale.flag
= T)
```

The major parameters in the `autoencode` functions are as follows:

- `nl`: This is the number of layers including the input and output layer (the default is three).
- `N.hidden`: This is the vector with the number of neurons in each hidden layer.
- `unit.type`: This is the type of activation function to be used.
- `lambda`: This is the regularization parameter.
- `rho`: This is the sparsity parameter.
- `beta`: This is the penalty for the sparsity term.
- `max.iterations`: This is the maximum number of iterations.
- `epsilon`: This is the parameter for weight initialization. The weights are initialized using Gaussian distribution ~N(0, epsilon2).

How it works...

The following figure shows the shapes and orientation of digits from `MNIST` captured by the sparse autoencoder:



Filter generated by the sparse autoencoder to get the digit outcome

The filter learned by the sparse autoencoder can be visualized using the `visualize.hidden.units` function from the autoencoder package. The package plots the weight of the final layer with respect to the output. In the current scenario, 100 is the number of neurons in the hidden layer and 256 is the number of nodes in the output layer.

5
Generative Models in Deep Learning

In this chapter, we will cover the following topics:

- Comparing principal component analysis with the Restricted Boltzmann machine
- Setting up a Restricted Boltzmann machine for Bernoulli distribution input
- Training a Restricted Boltzmann machine
- Backward or reconstruction phase of RBM
- Understanding the contrastive divergence of the reconstruction
- Initializing and starting a new TensorFlow session
- Evaluating the output from an RBM
- Setting up a Restricted Boltzmann machine for Collaborative Filtering
- Performing a full run of training an RBM
- Setting up a Deep Belief Network
- Implementing a feed-forward backpropagation Neural Network
- Setting up a Deep Restricted Boltzmann Machine

Comparing principal component analysis with the Restricted Boltzmann machine

In this section, you will learn about two widely recommended dimensionality reduction techniques--**Principal component analysis** (**PCA**) and the **Restricted Boltzmann machine** (**RBM**). Consider a vector v in n-dimensional space. The dimensionality reduction technique essentially transforms the vector v into a relatively smaller (or sometimes equal) vector v' with m-dimensions ($m<n$). The transformation can be either linear or nonlinear.

PCA performs a linear transformation on features such that orthogonally adjusted components are generated that are later ordered based on their relative importance of variance capture. These m components can be considered as new input features, and can be defined as follows:

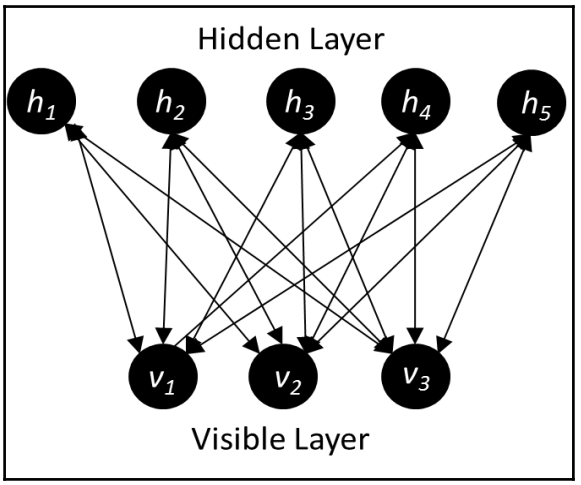
Vector $v' = \sum_{i=1}^{m} w_i c_i$

Here, w and c correspond to weights (loading) and transformed components, respectively.

Unlike PCA, RBMs (or DBNs/autoencoders) perform non-linear transformations using connections between visible and hidden units, as described in `Chapter 4`, *Data Representation Using Autoencoders*. The nonlinearity helps in better understanding the relationship with latent variables. Along with information capture, they also tend to remove noise. RBMs are generally based on stochastic distribution (either Bernoulli or Gaussian).

A large amount of Gibbs sampling is performed to learn and optimize the connection weights between visible and hidden layers. The optimization happens in two passes: a forward pass where hidden layers are sampled using given visible layers and a backward pass where visible layers are resampled using given hidden layers. The optimization is performed to minimize the reconstruction error.

The following image represents a restricted Boltzmann machine:



Getting ready

For this recipe, you will require R (the `rbm` and `ggplot2` packages) and the MNIST dataset. The MNIST dataset can be downloaded from the TensorFlow dataset library. The dataset consists of handwritten images of 28 x 28 pixels. It has 55,000 training examples and 10,000 test examples. It can be downloaded from the `tensorflow` library using the following script:

```
library(tensorflow)
datasets <- tf$contrib$learn$datasets
mnist <- datasets$mnist$read_data_sets("MNIST-data", one_hot = TRUE)
```

How to do it...

1. Extract the train dataset (`trainX` with all 784 independent variables and `trainY` with the respective 10 binary outputs):

```
trainX <- mnist$train$images
trainY <- mnist$train$labels
```

2. Run a PCA on the `trainX` data:

```
PCA_model <- prcomp(trainX, retx=TRUE)
```

3. Run an RBM on the `trainX` data:

```
RBM_model <- rbm(trainX, retx=TRUE, max_epoch=500,num_hidden =900)
```

4. Predict on the train data using the generated models. In the case of the RBM model, generate probabilities:

```
PCA_pred_train <- predict(PCA_model)
RBM_pred_train <- predict(RBM_model,type='probs')
```

5. Convert the outcomes into data frames:

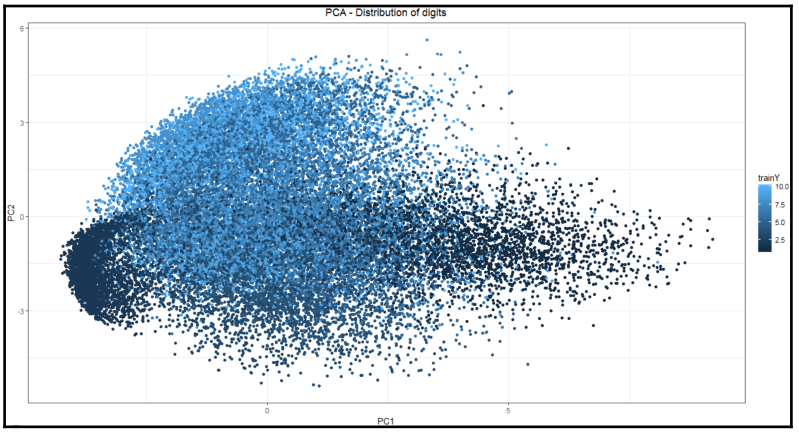
```
PCA_pred_train <- as.data.frame(PCA_pred_train)
 class="MsoSubtleEmphasis">RBM_pred_train <-
as.data.frame(as.matrix(RBM_pred_train))
```

6. Convert the 10-class binary `trainY` data frame into a numeric vector:

```
    trainY_num<-
as.numeric(stringi::stri_sub(colnames(as.data.frame(trainY))[max.co
l(as.data.frame(trainY),ties.method="first")],2))
```

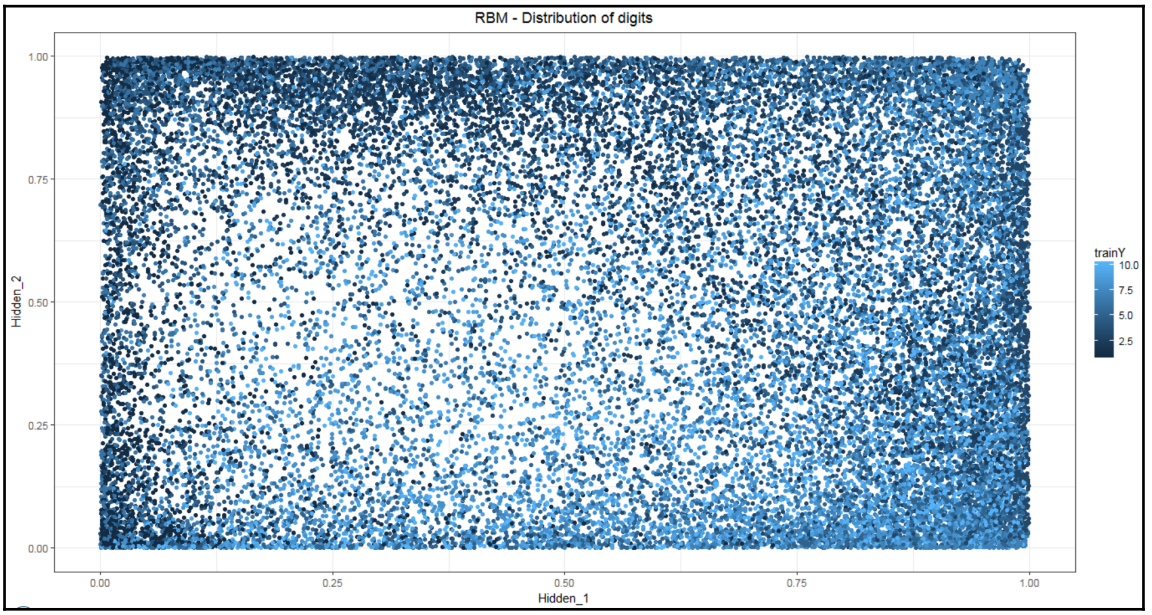
7. Plot the components generated using PCA. Here, the *x*-axis represents component 1 and the *y*-axis represents component 2. The following image shows the outcome of the PCA model:

```
ggplot(PCA_pred_train, aes(PC1, PC2))+
  geom_point(aes(colour = trainY))+
  theme_bw()+labs()+
  theme(plot.title = element_text(hjust = 0.5))
```



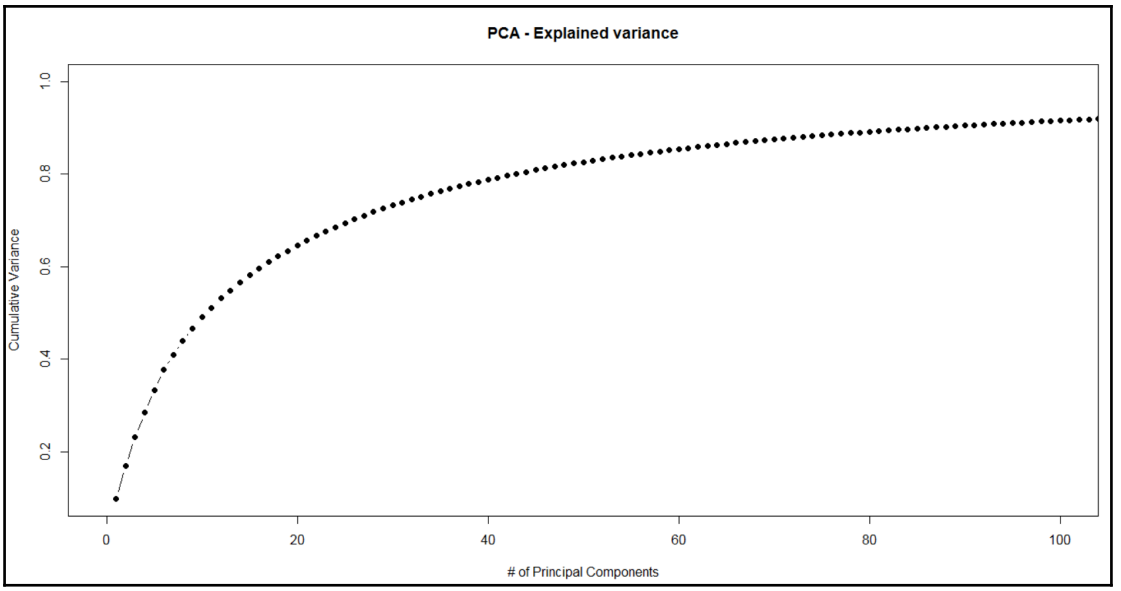
8. Plot the hidden layers generated using PCA. Here, the *x*-axis represents hidden 1 and *y*-axis represents hidden 2. The following image shows the outcome of the RBM model:

```
ggplot(RBM_pred_train, aes(Hidden_2, Hidden_3))+
  geom_point(aes(colour = trainY))+
  theme_bw()+labs()+
  theme(plot.title = element_text(hjust = 0.5))
```



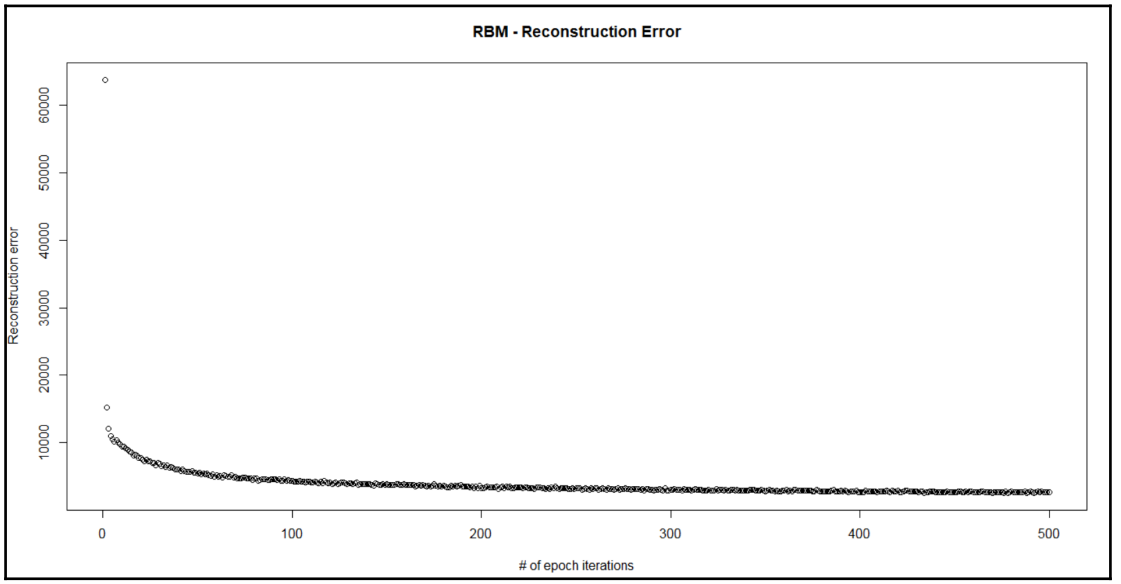
The following code and image shows the cumulative variance explained by the principal components:

```
var_explain <- as.data.frame(PCA_model$sdev^2/sum(PCA_model$sdev^2))
var_explain <- cbind(c(1:784),var_explain,cumsum(var_explain[,1]))
colnames(var_explain) <- c("PcompNo.","Ind_Variance","Cum_Variance")
plot(var_explain$PcompNo.,var_explain$Cum_Variance, xlim =
c(0,100),type='b',pch=16,xlab = "# of Principal Components",ylab =
"Cumulative Variance",main = 'PCA - Explained variance')
```



The following code and image shows the decrease in the reconstruction training error while generating an RBM using multiple epochs:

```
plot(RBM_model,xlab = "# of epoch iterations",ylab = "Reconstruction
error",main = 'RBM - Reconstruction Error')
```



Setting up a Restricted Boltzmann machine for Bernoulli distribution input

In this section, let's set up a restricted Boltzmann machine for Bernoulli distributed input data, where each attribute has values ranging from 0 to 1 (equivalent to a probability distribution). The dataset (MNIST) used in this recipe has input data satisfying a Bernoulli distribution.

An RBM comprises of two layers: a visible layer and a hidden layer. The visible layer is an input layer of nodes equal to the number of input attributes. In our case, each image in the MNIST dataset is defined using 784 pixels (28 x 28 size). Hence, our visible layer will have 784 nodes.

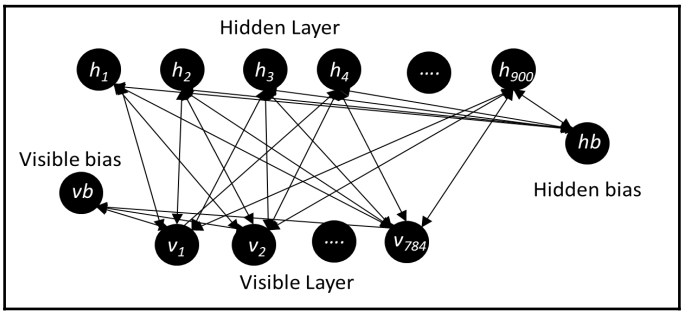
On the other hand, the hidden layer is generally user-defined. The hidden layer has a set of binary activated nodes, with each node having a probability of linkage with all other visible nodes. In our case, the hidden layer will have 900 nodes. As an initial step, all the nodes in the visible layer are connected with all the nodes in the hidden layer bidirectionally.

Each connection is defined using a weight, and hence a weight matrix is defined where the rows represent the number of input nodes and the columns represent the number of hidden nodes. In our case, the weight matrix (w) will be a tensor of dimensions 784 x 900.

In addition to weights, all the nodes in each layer are assisted by a bias node. The bias node of the visible layer will have connections with all the visible nodes (that is, the 784 nodes) and is represented with **vb**, whereas the bias node of the hidden layer will have connections with all the hidden nodes (that is, the 900 nodes) and is represented as **vh**.

A point to remember with RBMs is that there will be no connections among nodes within each layer. In other words, the connections will be interlayer, but not intralayer.

The following image represents an RBM with the visible layer, hidden layer, and interconnections:



Getting ready

This section provides the requirements for setting up an RBM.

- TensorFlow in R is installed and set up
- The `mnist` data is downloaded and loaded for setting up RBM

How to do it...

This section provides the steps to set up the visible and hidden layers of an RBM using TensorFlow:

1. Start a new interactive TensorFlow session:

```
# Reset the graph
tf$reset_default_graph()
# Starting session as interactive session
sess <- tf$InteractiveSession()
```

2. Define the model parameters. The `num_input` parameter defines the number of nodes in the visible layer and `num_hidden` defines the number of nodes in the hidden layer:

```
num_input<-784L
num_hidden<-900L
```

3. Create a placeholder variable for the weight matrix:

```
W <- tf$placeholder(tf$float32, shape = shape(num_input,
num_hidden))
```

4. Create placeholder variables of the visible and hidden biases:

```
vb <- tf$placeholder(tf$float32, shape = shape(num_input))
hb <- tf$placeholder(tf$float32, shape = shape(num_hidden))
```

Training a Restricted Boltzmann machine

Every training step of an RBM goes through two phases: the forward phase and the backward phase (or reconstruction phase). The reconstruction of visible units is fine tuned by making several iterations of the forward and backward phases.

Training a forward phase: In the forward phase, the input data is passed from the visible layer to the hidden layer and all the computation occurs within the nodes of the hidden layer. The computation is essentially to take a stochastic decision of each connection from the visible to the hidden layer. In the hidden layer, the input data (`X`) is multiplied by the weight matrix (`W`) and added to a hidden bias vector (`hb`).

The resultant vector of a size equal to the number of hidden nodes is then passed through a sigmoid function to determine each hidden node's output (or activation state). In our case, each input digit will produce a tensor vector of 900 probabilities, and as we have 55,000 input digits, we will have an activation matrix of the size 55,000 x 900. Using the hidden layer's probability distribution matrix, we can generate samples of activation vectors that can be used later to estimate negative phase gradients.

Getting ready

This section provides the requirements for setting up an RBM.

- TensorFlow in R is installed and set up
- The `mnist` data is downloaded and loaded for setting up the RBM
- The RBM model is set up as described in the recipe *Setting up a Restricted Boltzmann machine for Bernoulli distribution input*

Example of a sampling

Consider a constant vector `s1` equivalent to a tensor vector of probabilities. Then, create a new random uniformly distributed sample `s2` using the distribution of the constant vector `s1`. Then calculate the difference and apply a rectified linear activation function.

How to do it...

This section provides the steps to set up the script for running the RBM model using TensorFlow:

```
X = tf$placeholder(tf$float32, shape=shape(NULL, num_input))
prob_h0= tf$nn$sigmoid(tf$matmul(X, W) + hb)
h0 = tf$nn$relu(tf$sign(prob_h0 - tf$random_uniform(tf$shape(prob_h0))))
```

Use the following code to execute the graph created in TensorFlow:

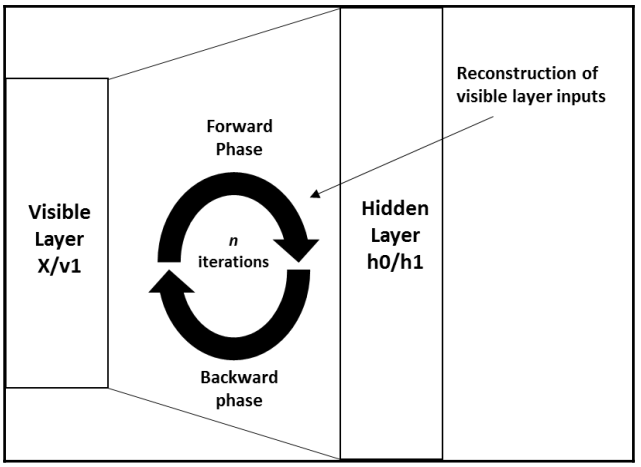
```
sess$run(tf$global_variables_initializer())
s1 <- tf$constant(value = c(0.1,0.4,0.7,0.9))
cat(sess$run(s1))
s2=sess$run(tf$random_uniform(tf$shape(s1)))
cat(s2)
cat(sess$run(s1-s2))
cat(sess$run(tf$sign(s1 - s2)))
cat(sess$run(tf$nn$relu(tf$sign(s1 - s2))))
```

Backward or reconstruction phase of RBM

In the reconstruction phase, the data from the hidden layer is passed back to the visible layer. The hidden layer vector of probabilities `h0` is multiplied by the transpose of the weight matrix `W` and added to a visible layer bias `vb`, which is then passed through a sigmoid function to generate a reconstructed input vector `prob_v1`.

A sample input vector is created using the reconstructed input vector, which is then multiplied by the weight matrix `W` and added to the hidden bias vector `hb` to generate an updated hidden vector of probabilities `h1`.

This is also called Gibbs sampling. In some scenarios, the sample input vector is not generated and the reconstructed input vector `prob_v1` is directly used to update the hi



Getting ready

This section provides the requirements for image reconstruction using the input probability vector.

- `mnist` data is loaded in the environment
- The RBM model is trained using the recipe *Training a Restricted Boltzmann machine*

How to do it...

This section covers the steps to perform backward reconstruction and evaluation:

1. The backward image reconstruction can be performed using the input probability vector with the following script:

```
prob_v1 = tf$nn$sigmoid(tf$matmul(h0, tf$transpose(W)) + vb)
v1 = tf$nn$relu(tf$sign(prob_v1 -
tf$random_uniform(tf$shape(prob_v1))))
h1 = tf$nn$sigmoid(tf$matmul(v1, W) + hb)
```

2. The evaluation can be performed using a defined metric, such as **mean squared error** (**MSE**), which is computed between the actual input data (`X`) and the reconstructed input data (`v1`). The MSE is computed after each epoch and the key objective is to minimize the MSE:

```
err = tf$reduce_mean(tf$square(X - v1))
```

Understanding the contrastive divergence of the reconstruction

As an initial start, the objective function can be defined as the minimization of the average negative log-likelihood of reconstructing the visible vector v where $P(v)$ denotes the vector of generated probabilities:

$$\arg\min(w) \; -E\left[\sum_{\vartheta \in V} \log P(\vartheta)\right]$$

Getting ready

This section provides the requirements for image reconstruction using the input probability vector.

- `mnist` data is loaded in the environment
- The images are reconstructed using the recipe *Backward or reconstruction phase*

How to do it...

This current recipe present the steps for, a **contrastive divergence** (**CD**) technique used to speed up the sampling process:

1. Compute a positive weight gradient by multiplying (outer product) the input vector `X` with a sample of the hidden vector `h0` from the given probability distribution `prob_h0`:

```
w_pos_grad = tf$matmul(tf$transpose(X), h0)
```

2. Compute a negative weight gradient by multiplying (outer product) the sample of the reconstructed input data `v1` with the updated hidden activation vector `h1`:

```
w_neg_grad = tf$matmul(tf$transpose(v1), h1)
```

3. Then, compute the `CD` matrix by subtracting the negative gradient from the positive gradient and dividing by the size of the input data:

```
CD = (w_pos_grad - w_neg_grad) / tf$to_float(tf$shape(X)[0])
```

4. Then, update the weight matrix `W` to `update_W` using a learning rate (*alpha*) and the CD matrix:

```
update_w = W + alpha * CD
```

5. Additionally, update the visible and hidden bias vectors:

```
update_vb = vb + alpha * tf$reduce_mean(X - v1)
update_hb = hb + alpha * tf$reduce_mean(h0 - h1)
```

How it works...

The objective function can be minimized using stochastic gradient descent by indirectly modifying (and optimizing) the weight matrix. The entire gradient can be further divided into two forms based on the probability density: positive gradient and negative gradient. The positive gradient primarily depends on the input data and the negative gradient depends only on the generated model.

In the positive gradient, the probability toward the reconstructing training data increases, and in the negative gradient, the probability of randomly generated uniform samples by the model decreases.

The CD technique is used to optimize the negative phase. In the CD technique, the weight matrix is adjusted in each iteration of reconstruction. The new weight matrix is generated using the following formula. The learning rate is defined as *alpha*, in our case:

$$W' = W + learning\ rate * CD$$

Initializing and starting a new TensorFlow session

A big part of calculating the error metric such as mean square error (MSE) is initialization and starting a new TensorFlow session. Here is how we proceed with it.

Getting ready

This section provides the requirements for starting a new TensorFlow session used to compute the error metric.

- `mnist` data is loaded in the environment
- The TensorFlow graph for the RBM is loaded

How to do it...

This section provides the steps for optimizing the error using reconstruction from an RBM:

1. Initialize the current and previous vector of biases and matrices of weights:

```
cur_w = tf$Variable(tf$zeros(shape = shape(num_input, num_hidden),
dtype=tf$float32))
cur_vb = tf$Variable(tf$zeros(shape = shape(num_input),
dtype=tf$float32))
cur_hb = tf$Variable(tf$zeros(shape = shape(num_hidden),
dtype=tf$float32))
prv_w = tf$Variable(tf$random_normal(shape=shape(num_input,
num_hidden), stddev=0.01, dtype=tf$float32))
prv_vb = tf$Variable(tf$zeros(shape = shape(num_input),
dtype=tf$float32))
prv_hb = tf$Variable(tf$zeros(shape = shape(num_hidden),
dtype=tf$float32))
```

2. Start a new TensorFlow session:

```
sess$run(tf$global_variables_initializer())
```

3. Perform a first run with the full input data (**trainX**) and obtain the first set of weight matrix and bias vectors:

```
output <- sess$run(list(update_w, update_vb, update_hb), feed_dict
= dict(X=trainX,
W = prv_w$eval(),
vb = prv_vb$eval(),
hb = prv_hb$eval()))
prv_w <- output[[1]]
prv_vb <-output[[2]]
prv_hb <-output[[3]]
```

4. Let's look at the error of the first run:

```
sess$run(err, feed_dict=dict(X= trainX, W= prv_w, vb= prv_vb, hb=
prv_hb))
```

5. The full model for the RBM can be trained using the following script:

```
epochs=15
errors <- list()
weights <- list()
u=1
for(ep in 1:epochs){
  for(i in seq(0,(dim(trainX)[1]-100),100)){
    batchX <- trainX[(i+1):(i+100),]
    output <- sess$run(list(update_w, update_vb, update_hb),
feed_dict = dict(X=batchX,
W = prv_w,
vb = prv_vb,
hb = prv_hb))
    prv_w <- output[[1]]
    prv_vb <- output[[2]]
    prv_hb <-  output[[3]]
    if(i%%10000 == 0){
      errors[[u]] <- sess$run(err, feed_dict=dict(X= trainX, W=
prv_w, vb= prv_vb, hb= prv_hb))
      weights[[u]] <- output[[1]]
      u <- u+1
     cat(i , " : ")
    }
  }
  cat("epoch :", ep, " : reconstruction error : ",
errors[length(errors)][[1]],"\n")
}
```

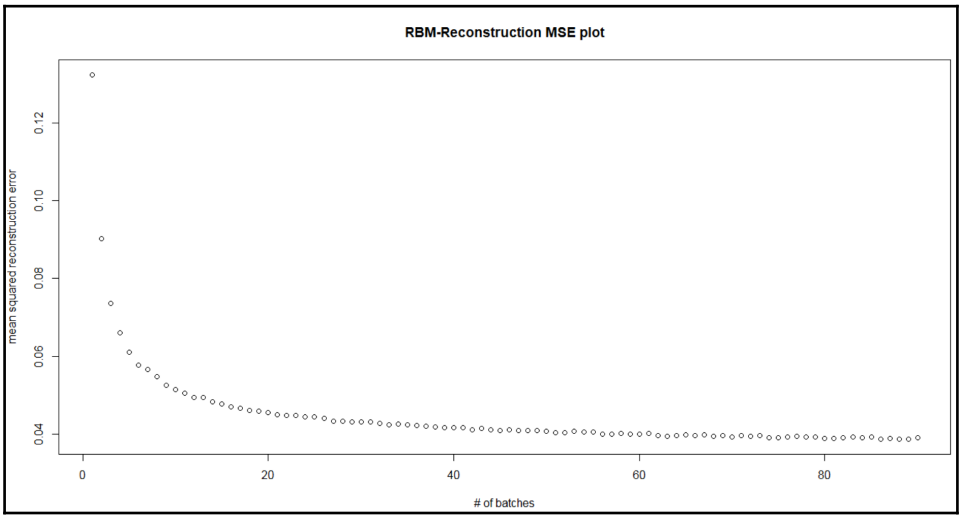
6. Plot reconstruction using mean squared errors:

```
error_vec <- unlist(errors)
plot(error_vec,xlab="# of batches",ylab="mean squared
reconstruction error",main="RBM-Reconstruction MSE plot")
```

How it works...

Here, we will run 15 epochs (or iterations) where, in each epoch, a batchwise (size = 100) optimization is performed. In each batch, the CD is computed and, accordingly, weights and biases are updated. To keep track of the optimization, MSE is calculated after every batch of 10,000 rows.

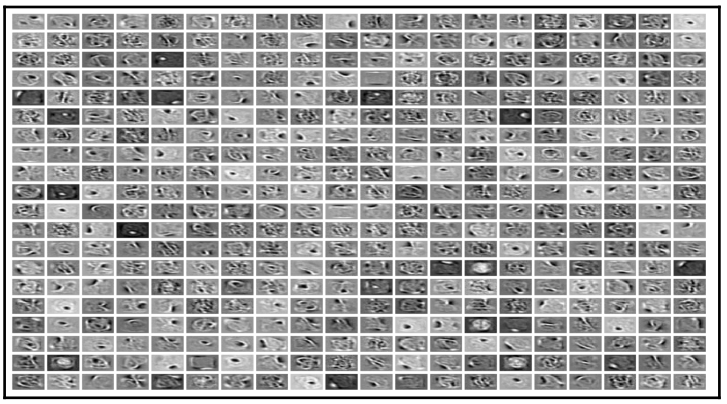
The following image shows the declining trend of mean squared reconstruction errors computed for 90 batches:



Evaluating the output from an RBM

Here, let's plot the weights of the final layer with respect to the output (reconstruction input data). In the current scenario, 900 is the number of nodes in the hidden layer and 784 is the number of nodes in the output (reconstructed) layer.

In the following image, the first 400 nodes in the hidden layer are seen:



Here, each tile represents a vector of connections formed between a hidden node and all the visible layer nodes. In each tile, the black region represents negative weights (weight < 0), the white region represents positive weights (weight > 1), and the grey region represents no connection (weight = 0). The higher the positive value, the greater the chance of activation in hidden nodes, and vice versa. These activations help determine which part of the input image is being determined by a given hidden node.

Getting ready

This section provides the requirements for running the evaluation recipe:

- `mnist` data is loaded in the environment
- The RBM model is executed using TensorFlow and the optimal weights are obtained

How to do it...

This recipe covers the steps for the evaluation of weights obtained from an RBM:

1. Run the following code to generate the image of 400 hidden nodes:

```
uw = t(weights[[length(weights)]])   # Extract the most recent
weight matrix
numXpatches = 20    # Number of images in X-axis (user input)
numYpatches=20      # Number of images in Y-axis (user input)
pixels <- list()
op <- par(no.readonly = TRUE)
par(mfrow = c(numXpatches,numYpatches), mar = c(0.2, 0.2, 0.2,
0.2), oma = c(3, 3, 3, 3))
for (i in 1:(numXpatches*numYpatches)) {
  denom <- sqrt(sum(uw[i, ]^2))
  pixels[[i]] <- matrix(uw[i, ]/denom, nrow = numYpatches, ncol =
numXpatches)
  image(pixels[[i]], axes = F, col = gray((0:32)/32))
}
par(op)
```

2. Select a sample of four actual input digits from the training data:

```
sample_image <- trainX[1:4,]
```

3. Then, visualize these sample digits using the following code:

```
mw=melt(sample_image)
mw$X3=floor((mw$X2-1)/28)+1
mw$X2=(mw$X2-1)%%28 + 1;
mw$X3=29-mw$X3
ggplot(data=mw)+geom_tile(aes(X2,X3,fill=value))+facet_wrap(~X1,nro
w=2)+
scale_fill_continuous(low='black',high='white')+coord_fixed(ratio=1
)+
  labs(x=NULL,y=NULL,)+
  theme(legend.position="none")+
  theme(plot.title = element_text(hjust = 0.5))
```

4. Now, reconstruct these four sample images using the final weights and biases obtained:

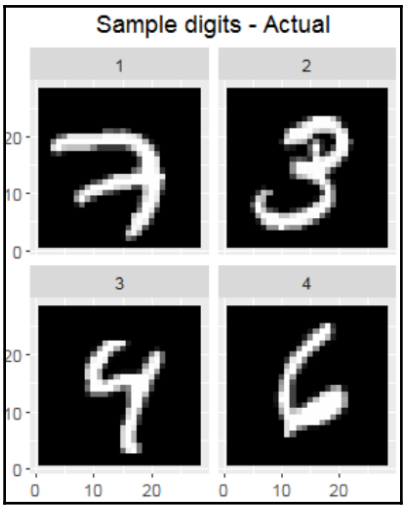
```
hh0 = tf$nn$sigmoid(tf$matmul(X, W) + hb)
vv1 = tf$nn$sigmoid(tf$matmul(hh0, tf$transpose(W)) + vb)
feed = sess$run(hh0, feed_dict=dict( X= sample_image, W= prv_w, hb=
prv_hb))
rec = sess$run(vv1, feed_dict=dict( hh0= feed, W= prv_w, vb=
prv_vb))
```

5. Then, visualize the reconstructed sample digits using the following code:

```
mw=melt(rec)
mw$X3=floor((mw$X2-1)/28)+1
mw$X2=(mw$X2-1)%%28 + 1
mw$X3=29-mw$X3
ggplot(data=mw)+geom_tile(aes(X2,X3,fill=value))+facet_wrap(~X1,nro
w=2)+
scale_fill_continuous(low='black',high='white')+coord_fixed(ratio=1
)+
  labs(x=NULL,y=NULL,)+
  theme(legend.position="none")+
  theme(plot.title = element_text(hjust = 0.5))
```

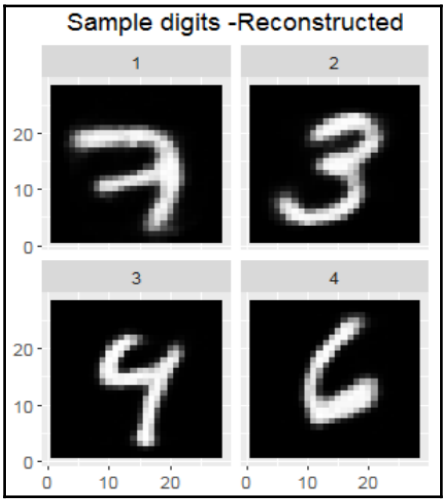
How it works...

The following image illustrates a raw image of the four sample digits:



The reconstructed images seemed to have had their noise removed, especially in the case of digits **3** and **6**.

The following image illustrates a reconstructed image of the same four digits:



Setting up a Restricted Boltzmann machine for Collaborative Filtering

In this recipe, you will learn how to build a collaborative-filtering-based recommendation system using an RBM. Here, for every user, the RBM tries to identify similar users based on their past behavior of rating various items, and then tries to recommend the next best item.

Getting ready

In this recipe, we will use the movielens dataset from the Grouplens research organization. The datasets (`movies.dat` and `ratings.dat`) can be downloaded from the following link. `Movies.dat` contains information of 3,883 movies and `Ratings.dat` contains information of 1,000,209 user ratings for these movies. The ratings range from 1 to 5, with 5 being the highest.

`http://files.grouplens.org/datasets/movielens/ml-1m.zip`

How to do it...

This recipe covers the steps for setting up collaborative filtering using an RBM.

1. Read the `movies.dat` datasets in R:

```
txt <- readLines("movies.dat", encoding = "latin1")
txt_split <- lapply(strsplit(txt, "::"), function(x)
as.data.frame(t(x), stringsAsFactors=FALSE))
movies_df <- do.call(rbind, txt_split)
names(movies_df) <- c("MovieID", "Title", "Genres")
movies_df$MovieID <- as.numeric(movies_df$MovieID)
```

2. Add a new column (`id_order`) to the movies dataset, as the current ID column (`UserID`) cannot be used to index movies because they range from 1 to 3,952:

```
movies_df$id_order <- 1:nrow(movies_df)
```

3. Read the `ratings.dat` dataset in R:

```
ratings_df <- read.table("ratings.dat",
sep=":",header=FALSE,stringsAsFactors = F)
ratings_df <- ratings_df[,c(1,3,5,7)]
colnames(ratings_df) <- c("UserID","MovieID","Rating","Timestamp")
```

4. Merge the movies and ratings datasets with `all=FALSE`:

```
merged_df <- merge(movies_df, ratings_df, by="MovieID",all=FALSE)
```

5. Remove the non-required columns:

```
merged_df[,c("Timestamp","Title","Genres")] <- NULL
```

6. Convert the ratings to percentages:

```
merged_df$rating_per <- merged_df$Rating/5
```

7. Generate a matrix of ratings across all the movies for a sample of 1,000 users:

```
num_of_users <- 1000
num_of_movies <- length(unique(movies_df$MovieID))
trX <- matrix(0,nrow=num_of_users,ncol=num_of_movies)
for(i in 1:num_of_users){
  merged_df_user <- merged_df[merged_df$UserID %in% i,]
  trX[i,merged_df_user$id_order] <- merged_df_user$rating_per
}
```

8. Look at the distribution of the `trX` training dataset. It seems to follow a Bernoulli distribution (values in the range of 0 to 1):

```
summary(trX[1,]); summary(trX[2,]); summary(trX[3,])
```

9. Define the input model parameters:

```
num_hidden = 20
num_input = nrow(movies_df)
```

10. Start a new TensorFlow session:

```
sess$run(tf$global_variables_initializer())
output <- sess$run(list(update_w, update_vb, update_hb), feed_dict
= dict(v0=trX,
W = prv_w$eval(),
vb = prv_vb$eval(),
hb = prv_hb$eval()))
prv_w <- output[[1]]
prv_vb <- output[[2]]
prv_hb <-  output[[3]]
sess$run(err_sum, feed_dict=dict(v0=trX, W= prv_w, vb= prv_vb, hb=
prv_hb))
```

11. Train the RBM using 500 epoch iterations and a batch size of 100:

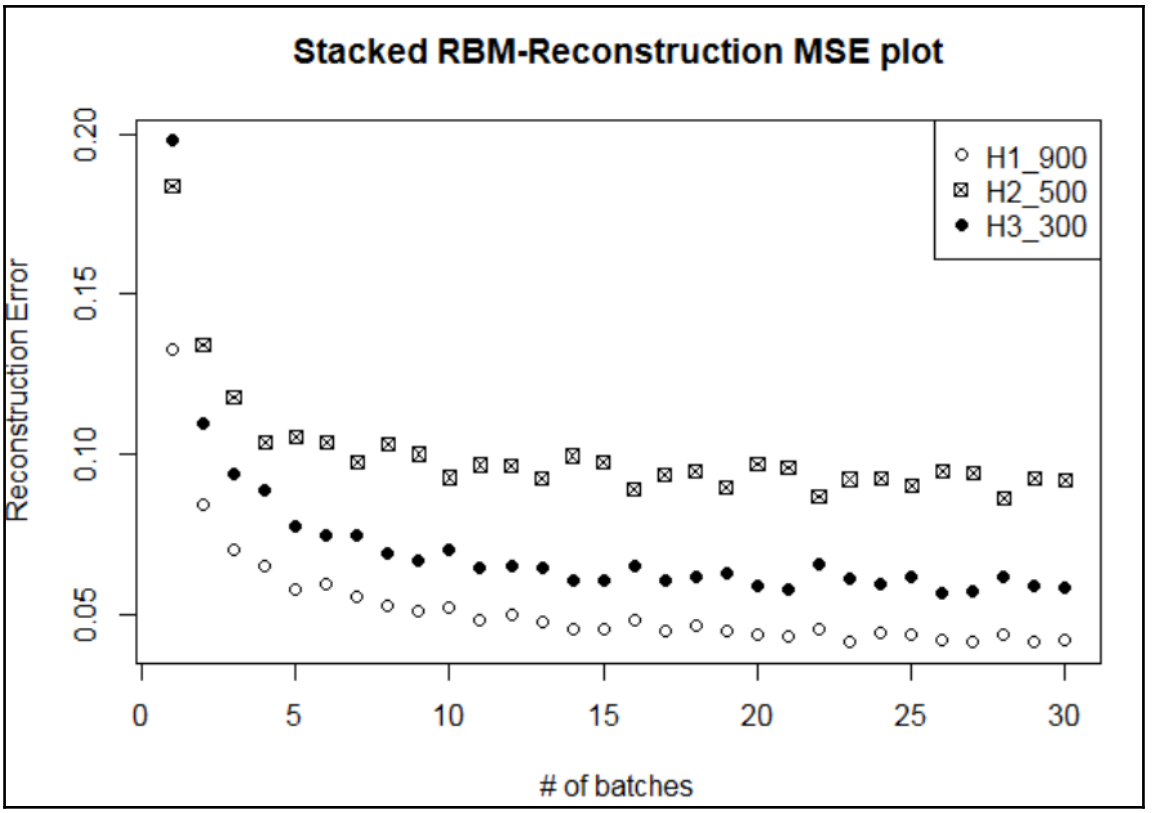
```
epochs= 500
errors <- list()
weights <- list()

for(ep in 1:epochs){
  for(i in seq(0,(dim(trX)[1]-100),100)){
    batchX <- trX[(i+1):(i+100),]
    output <- sess$run(list(update_w, update_vb, update_hb),
feed_dict = dict(v0=batchX,
W = prv_w,
vb = prv_vb,
hb = prv_hb))
    prv_w <- output[[1]]
    prv_vb <- output[[2]]
    prv_hb <-  output[[3]]
    if(i%%1000 == 0){
      errors <- c(errors,sess$run(err_sum,
feed_dict=dict(v0=batchX, W= prv_w, vb= prv_vb, hb= prv_hb)))
      weights <- c(weights,output[[1]])
      cat(i , " : ")
    }
  }
  cat("epoch :", ep, " : reconstruction error : ",
errors[length(errors)][[1]],"\n")
}
```

12. Plot reconstruction mean squared errors:

```
error_vec <- unlist(errors)
plot(error_vec,xlab="# of batches",ylab="mean squared
reconstruction error",main="RBM-Reconstruction MSE plot")
```

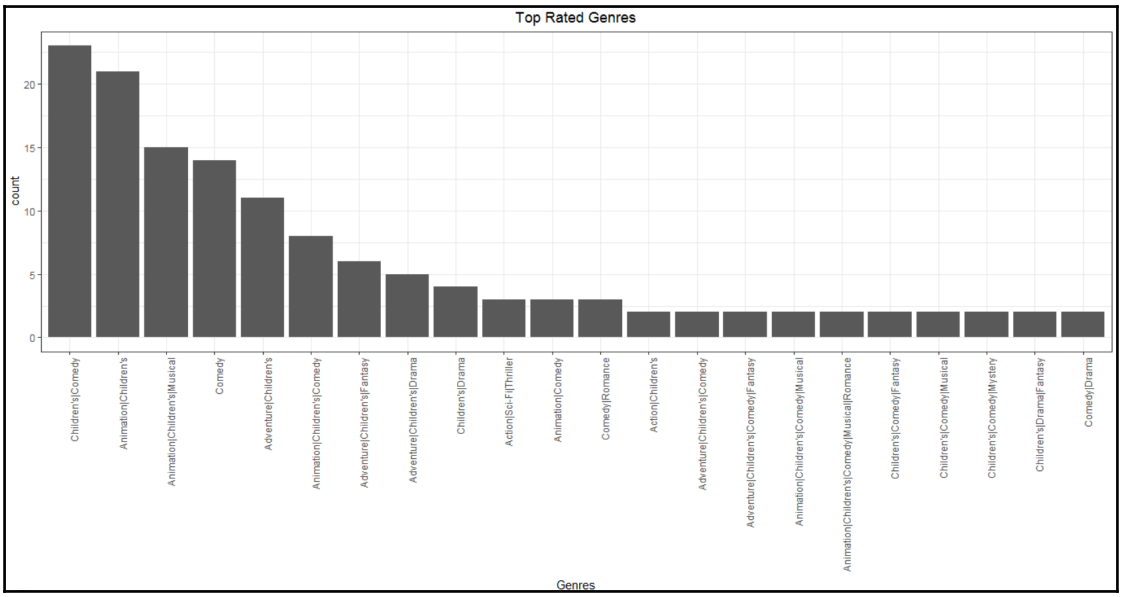
Performing a full run of training an RBM

Using the same RBM setup mentioned in the preceding recipe, train the RBM on the user ratings dataset (`trX`) using 20 hidden nodes. To keep a track of the optimization, the MSE is calculated after every batch of 1,000 rows. The following image shows the declining trend of mean squared reconstruction errors computed for 500 batches (equal to epochs):

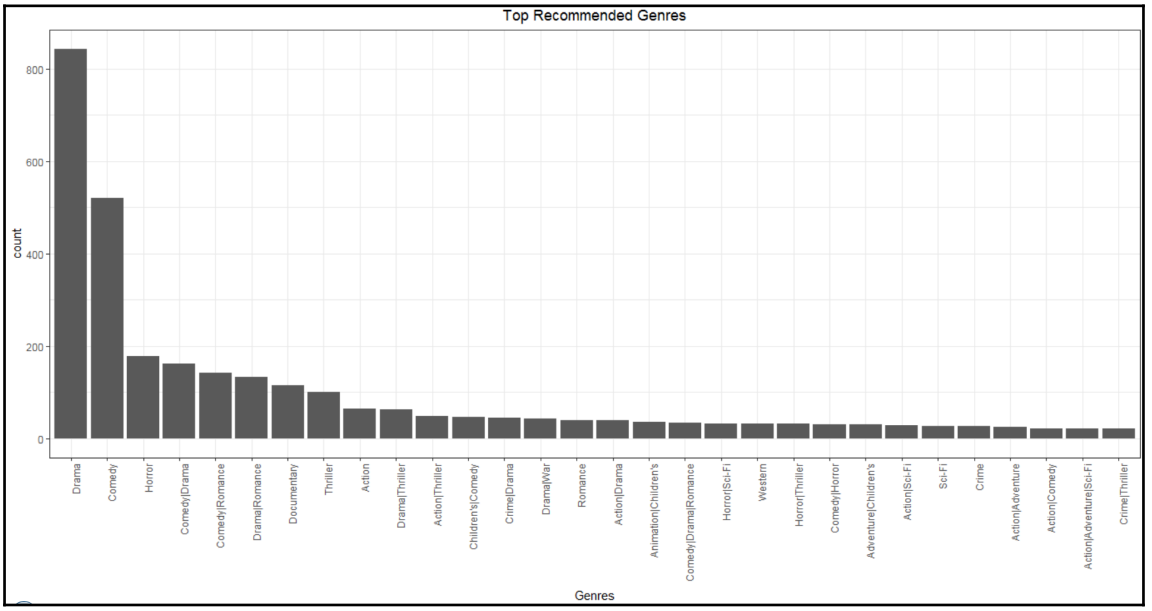


Looking into RBM recommendations: Let's now look into the recommendations generated by RBM-based collaborative filtering for a given user ID. Here, we will look into the top-rated genres and top-recommended genres of this user ID, along with the top 10 movie recommendations.

The following image illustrates a list of top-rated genres:



The following image illustrates a list of top-recommended genres:



Getting ready

This section provides the requirements for collaborative filtering the output evaluation:

- TensorFlow in R is installed and set up
- The `movies.dat` and `ratings.dat` datasets are loaded in environment
- The recipe *Setting up a Restricted Boltzmann machine for Collaborative Filtering* has been executed

How to do it...

This recipe covers the steps for evaluating the output from RBM-based collaborative filtering:

1. Select the ratings of a user:

```
inputUser = as.matrix(t(trX[75,]))
names(inputUser) <- movies_df$id_order
```

2. Remove the movies that were not rated by the user (assuming that they have yet to be seen):

```
inputUser <- inputUser[inputUser>0]
```

3. Plot the top genres seen by the user:

```
top_rated_movies <-
movies_df[as.numeric(names(inputUser)[order(inputUser,decreasing =
TRUE)]),]$Title
top_rated_genres <-
movies_df[as.numeric(names(inputUser)[order(inputUser,decreasing =
TRUE)]),]$Genres
top_rated_genres <-
as.data.frame(top_rated_genres,stringsAsFactors=F)
top_rated_genres$count <- 1
top_rated_genres <-
aggregate(count~top_rated_genres,FUN=sum,data=top_rated_genres)
top_rated_genres <- top_rated_genres[with(top_rated_genres, order(-
count)), ]
top_rated_genres$top_rated_genres <-
factor(top_rated_genres$top_rated_genres, levels =
top_rated_genres$top_rated_genres)
ggplot(top_rated_genres[top_rated_genres$count>1,],aes(x=top_rated_
genres,y=count))+
```

```
geom_bar(stat="identity")+
theme_bw()+
theme(axis.text.x = element_text(angle = 90, hjust = 1))+
labs(x="Genres",y="count",)+
theme(plot.title = element_text(hjust = 0.5))
```

4. Reconstruct the input vector to obtain the recommendation percentages for all the genres/movies:

```
hh0 = tf$nn$sigmoid(tf$matmul(v0, W) + hb)
vv1 = tf$nn$sigmoid(tf$matmul(hh0, tf$transpose(W)) + vb)
feed = sess$run(hh0, feed_dict=dict( v0= inputUser, W= prv_w, hb=
prv_hb))
rec = sess$run(vv1, feed_dict=dict( hh0= feed, W= prv_w, vb=
prv_vb))
names(rec) <- movies_df$id_order
```

5. Plot the top-recommended genres:

```
top_recom_genres <-
movies_df[as.numeric(names(rec)[order(rec,decreasing =
TRUE)]),]$Genres
top_recom_genres <-
as.data.frame(top_recom_genres,stringsAsFactors=F)
top_recom_genres$count <- 1
top_recom_genres <-
aggregate(count~top_recom_genres,FUN=sum,data=top_recom_genres)
top_recom_genres <- top_recom_genres[with(top_recom_genres, order(-
count)), ]
top_recom_genres$top_recom_genres <-
factor(top_recom_genres$top_recom_genres, levels =
top_recom_genres$top_recom_genres)
ggplot(top_recom_genres[top_recom_genres$count>20,],aes(x=top_recom
_genres,y=count))+
geom_bar(stat="identity")+
theme_bw()+
theme(axis.text.x = element_text(angle = 90, hjust = 1))+
labs(x="Genres",y="count",)+
theme(plot.title = element_text(hjust = 0.5))
```

6. Find the top 10 recommended movies:

```
top_recom_movies <-
movies_df[as.numeric(names(rec)[order(rec,decreasing =
TRUE)]),]$Title[1:10]
```

The following image shows the top 10 recommended movies:

```
> top_recom_movies
 [1] "Star Wars: Episode VI - Return of the Jedi (1983)"
 [2] "Matrix, The (1999)"
 [3] "Star Wars: Episode V - The Empire Strikes Back (1980)"
 [4] "Jurassic Park (1993)"
 [5] "Star Wars: Episode IV - A New Hope (1977)"
 [6] "Terminator 2: Judgment Day (1991)"
 [7] "Raiders of the Lost Ark (1981)"
 [8] "Star Wars: Episode I - The Phantom Menace (1999)"
 [9] "Men in Black (1997)"
[10] "Princess Bride, The (1987)"
```

Setting up a Deep Belief Network

Deep belief networks are a type of **Deep Neural Network** (**DNN**), and are composed of multiple hidden layers (or latent variables). Here, the connections are present only between the layers and not within the nodes of each layer. The DBN can be trained both as an unsupervised and supervised model.

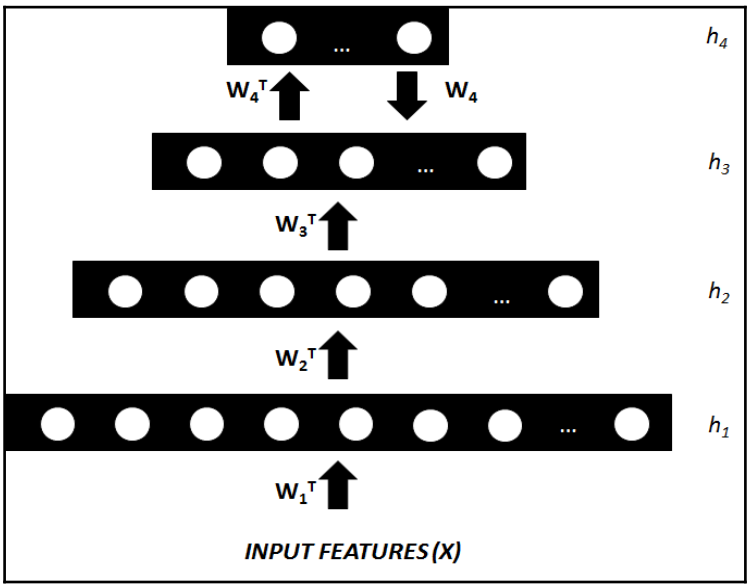
The unsupervised model is used to reconstruct the input with noise removal and the supervised model (after pretraining) is used to perform classification. As there are no connections within the nodes in each layer, the DBNs can be considered as a set of unsupervised RBMs or autoencoders, where each hidden layer serves as a visible layer to its subsequent connected hidden layer.

This kind of stacked RBM enhances the performance of input reconstruction where CD is applied across all layers, starting from the actual input training layer and finishing at the last hidden (or latent) layer.

DBNs are a type of graphical model that train the stacked RBMs in a greedy manner. Their networks tend to learn the deep hierarchical representation using joint distributions between the input feature vector i and hidden layers $h_{1,2....m}$:

$$P(i, h_1, h_2 \ldots h_m) = \left(\prod_{k=0}^{m-2} P(h_k|h_{k+1})\right) * P(h_{m-1}, h_m)$$

Here, $i = h_0$; $P(h_{k-1} | h_k)$ is a conditional distribution of reconstructed visible units on the hidden layers of the RBM at level k; $P(h_{m-1}, h_m)$ is the joint distribution of hidden and visible units (reconstructed) at the final RBM layer of the DBN. The following image illustrates a DBN of four hidden layers, where **W** represents the weight matrix:



DBNs can also be used to enhance the robustness of DNNs. DNNs face an issue of local optimization while implementing backpropagation. This is possible in scenarios where an error surface features numerous troughs, and the gradient descent, due to backpropagation occurs inside a local deep trough (not a global deep trough). DBNs, on the other hand, perform pretraining of the input features, which helps the optimization direct toward the global deepest trough, and then use backpropagation, to perform a gradient descent to gradually minimize the error rate.

Training a stack of three RBMs: In this recipe, we will train a DBN using three stacked RBMs, where the first hidden layer will have 900 nodes, the second hidden layer will have 500 nodes, and the third hidden layer will have 300 nodes.

Getting ready

This section provides the requirements for TensorFlow.

- The dataset is loaded and set up
- Load the `TensorFlow` package using the following script:

```
require(tensorflow)
```

How to do it...

This recipe covers the steps for setting up **Deep belief network** (**DBM**):

1. Define the number of nodes in each hidden layer as a vector:

```
RBM_hidden_sizes = c(900, 500 , 300 )
```

2. Generate an RBM function leveraging the codes illustrated in the *Setting up a Restricted Boltzmann Machine for Bernoulli distribution input* recipe with the following input and output parameters mentioned:

Type of Parameter	Parameter name	Parameter description
Input (Pre RBM)	*input_data*	Matrix of train MNIST data
Input (Pre RBM)	*num_input*	Number of independent variables
Input (Pre RBM)	*num_output*	Number of nodes in the corresponding hidden layer
Input (Pre RBM)	*epochs*	Number of iterations
Input (Pre RBM)	*alpha*	Learning rate for updating weight matrix
Input (Pre RBM)	*batchsize*	Number of observations per batch run
Output (Post RBM)	*output_data*	Matrix of reconstructed output
Output (Post RBM)	*error_list*	List of reconstruction errors for every run of 10 batch
Output (Post RBM)	*weight_list*	List of weight matrices for every run of 10 batch
Output (Post RBM)	*weight_final*	Matrix of final weights obtained after all epochs
Output (Post RBM)	*bias_final*	Vector of final biases obtained after all epochs

Here is the function for setting up up the RBM:

```
RBM <- function(input_data, num_input, num_output, epochs = 5,
alpha = 0.1, batchsize=100){
# Placeholder variables
```

```
vb <- tf$placeholder(tf$float32, shape = shape(num_input))
hb <- tf$placeholder(tf$float32, shape = shape(num_output))
W <- tf$placeholder(tf$float32, shape = shape(num_input,
num_output))
# Phase 1 : Forward Phase
X = tf$placeholder(tf$float32, shape=shape(NULL, num_input))
prob_h0= tf$nn$sigmoid(tf$matmul(X, W) + hb) #probabilities of the
hidden units
h0 = tf$nn$relu(tf$sign(prob_h0 -
tf$random_uniform(tf$shape(prob_h0)))) #sample_h_given_X
# Phase 2 : Backward Phase
prob_v1 = tf$nn$sigmoid(tf$matmul(h0, tf$transpose(W)) + vb)
v1 = tf$nn$relu(tf$sign(prob_v1 -
tf$random_uniform(tf$shape(prob_v1))))
h1 = tf$nn$sigmoid(tf$matmul(v1, W) + hb)
# calculate gradients
w_pos_grad = tf$matmul(tf$transpose(X), h0)
w_neg_grad = tf$matmul(tf$transpose(v1), h1)
CD = (w_pos_grad - w_neg_grad) / tf$to_float(tf$shape(X)[0])
update_w = W + alpha * CD
update_vb = vb + alpha * tf$reduce_mean(X - v1)
update_hb = hb + alpha * tf$reduce_mean(h0 - h1)
# Objective function
err = tf$reduce_mean(tf$square(X - v1))
# Initialize variables
cur_w = tf$Variable(tf$zeros(shape = shape(num_input, num_output),
dtype=tf$float32))
cur_vb = tf$Variable(tf$zeros(shape = shape(num_input),
dtype=tf$float32))
cur_hb = tf$Variable(tf$zeros(shape = shape(num_output),
dtype=tf$float32))
prv_w = tf$Variable(tf$random_normal(shape=shape(num_input,
num_output), stddev=0.01, dtype=tf$float32))
prv_vb = tf$Variable(tf$zeros(shape = shape(num_input),
dtype=tf$float32))
prv_hb = tf$Variable(tf$zeros(shape = shape(num_output),
dtype=tf$float32))
# Start tensorflow session
sess$run(tf$global_variables_initializer())
output <- sess$run(list(update_w, update_vb, update_hb), feed_dict
= dict(X=input_data,
W = prv_w$eval(),
vb = prv_vb$eval(),
hb = prv_hb$eval()))
prv_w <- output[[1]]
prv_vb <- output[[2]]
prv_hb <- output[[3]]
sess$run(err, feed_dict=dict(X= input_data, W= prv_w, vb= prv_vb,
```

```
hb= prv_hb))
errors <- weights <- list()
u=1
for(ep in 1:epochs){
for(i in seq(0,(dim(input_data)[1]-batchsize),batchsize)){
batchX <- input_data[(i+1):(i+batchsize),]
output <- sess$run(list(update_w, update_vb, update_hb), feed_dict
= dict(X=batchX,
W = prv_w,
vb = prv_vb,
hb = prv_hb))
prv_w <- output[[1]]
prv_vb <- output[[2]]
prv_hb <- output[[3]]
if(i%%10000 == 0){
errors[[u]] <- sess$run(err, feed_dict=dict(X= batchX, W= prv_w,
vb= prv_vb, hb= prv_hb))
weights[[u]] <- output[[1]]
u=u+1
cat(i , " : ")
}
}
cat("epoch :", ep, " : reconstruction error : ",
errors[length(errors)][[1]],"\n")
}
w <- prv_w
vb <- prv_vb
hb <- prv_hb
# Get the output
input_X = tf$constant(input_data)
ph_w = tf$constant(w)
ph_hb = tf$constant(hb)
out = tf$nn$sigmoid(tf$matmul(input_X, ph_w) + ph_hb)
sess$run(tf$global_variables_initializer())
return(list(output_data = sess$run(out),
error_list=errors,
weight_list=weights,
weight_final=w,
bias_final=hb))
}
```

3. Train the RBM for all three different types of hidden nodes in a sequence. In other words, first train RBM1 with 900 hidden nodes, then use the output of RBM1 as an input for RBM2 with 500 hidden nodes and train RBM2, then use the output of RBM2 as an input for RBM3 with 300 hidden nodes and train RBM3. Store the outputs of all three RBMs as a list, `RBM_output`:

```
inpX = trainX
RBM_output <- list()
for(i in 1:length(RBM_hidden_sizes)){
size <- RBM_hidden_sizes[i]
# Train the RBM
RBM_output[[i]] <- RBM(input_data= inpX,
num_input= ncol(trainX),
num_output=size,
epochs = 5,
alpha = 0.1,
batchsize=100)
# Update the input data
inpX <- RBM_output[[i]]$output_data
# Update the input_size
num_input = size
cat("completed size :", size,"\n")
}
```

4. Create a data frame of batch errors across three hidden layers:

```
error_df <-
data.frame("error"=c(unlist(RBM_output[[1]]$error_list),unlist(RBM_
output[[2]]$error_list),unlist(RBM_output[[3]]$error_list)),
"batches"=c(rep(seq(1:length(unlist(RBM_output[[1]]$error_list))),t
imes=3)),
"hidden_layer"=c(rep(c(1,2,3),each=length(unlist(RBM_output[[1]]$er
ror_list)))),
stringsAsFactors = FALSE)
```

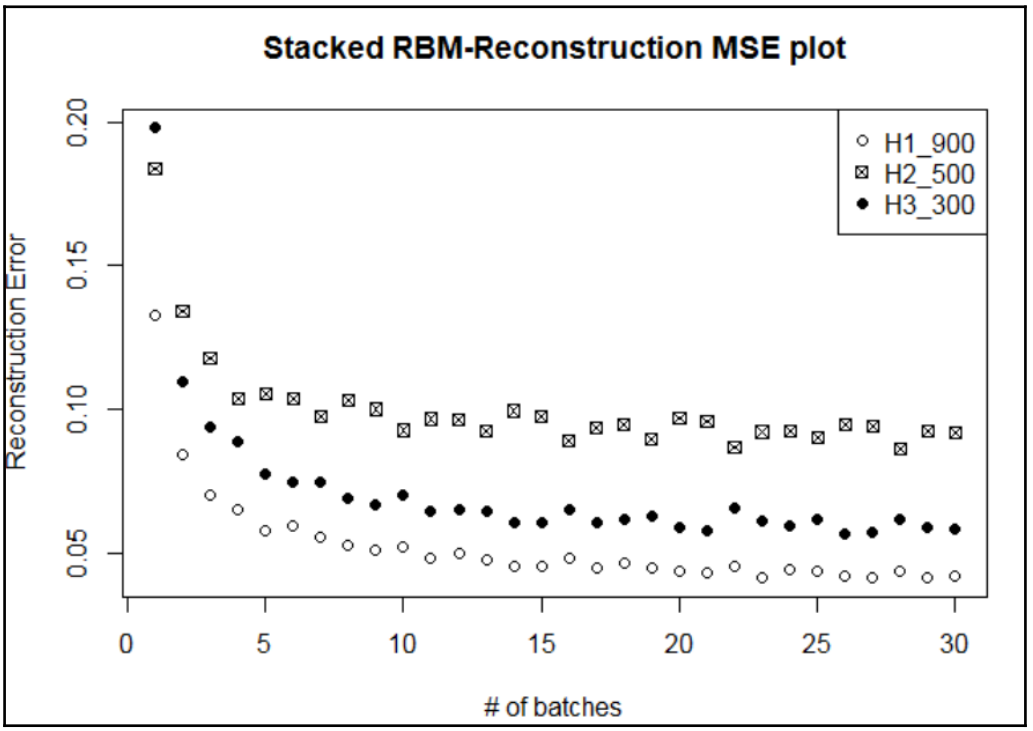
5. Plot reconstruction mean squared errors:

```
plot(error ~ batches,
xlab = "# of batches",
ylab = "Reconstruction Error",
pch = c(1, 7, 16)[hidden_layer],
main = "Stacked RBM-Reconstruction MSE plot",
data = error_df)
legend('topright',
c("H1_900","H2_500","H3_300"),
pch = c(1, 7, 16))
```

How it works...

Assessing the performance of training three stacked RBMs: Here, we will run five epochs (or iterations) for each RBM. Each epoch will perform batchwise (size = 100) optimization. In each batch, CD is computed and, accordingly, weights and biases are updated.

To keep a track of optimization, the MSE is calculated after every batch of 10,000 rows. The following image shows the declining trend of mean squared reconstruction errors computed for 30 batches for three RBMs separately:



Implementing a feed-forward backpropagation Neural Network

In this recipe, we will implement a willow neural network with backpropagation. The input of the neural network is the outcome of the third (or last) RBM. In other words, the reconstructed raw data (`trainX`) is actually used to train the neural network as a supervised classifier of (10) digits. The backpropagation technique is used to further fine-tune the performance of classification.

Getting ready

This section provides the requirements for TensorFlow.

- The dataset is loaded and set up
- The `TensorFlow` package is set up and loaded

How to do it...

This section covers the steps for setting up a feed-forward backpropagation Neural Network:

1. Let's define the input parameters of the neural network as function parameters. The following table describes each parameter:

Parameter name	Parameter description
Xdata	Matrix of train input MNIST data
Ydata	Matrix of train output MNIST data
Xtestdata	Matrix of test input MNIST data
Ytestdata	Matrix of test output MNIST data
input_size	Number of attributes (or pixels) in train data
learning_rate	Learning rate for updating weight matrix
momentum	Increment in size of steps to jump out of local minima
epochs	Number of iterations
batchsize	Number of observations per batch run
rbm_list	List of outcomes of stacked RBM
dbn_sizes	Vector of sizes of hidden layers in stacked RBM

The neural network function will have a structure as shown in the following script:

```
NN_train <- function(Xdata,Ydata,Xtestdata,Ytestdata,input_size,
learning_rate=0.1,momentum = 0.1,epochs=10,
batchsize=100,rbm_list,dbn_sizes){
library(stringi)
```

```
## insert all the codes mentioned in next 11 points
}
```

2. Initialize a weight and bias list of length 4, with the first being a tensor of random normal distribution (with a standard deviation of 0.01) of dimensions 784 x 900, the second being 900 x 500, the third being 500 x 300, and the fourth being 300 x 10:

```
weight_list <- list()
bias_list <- list()
# Initialize variables
for(size in c(dbn_sizes,ncol(Ydata))){
#Initialize weights through a random uniform distribution
weight_list <-
c(weight_list,tf$random_normal(shape=shape(input_size, size),
stddev=0.01, dtype=tf$float32))
#Initialize bias as zeroes
bias_list <- c(bias_list, tf$zeros(shape = shape(size),
dtype=tf$float32))
input_size = size
}
```

3. Check whether the outcome of the stacked RBM conforms to the sizes of the hidden layers mentioned in the `dbn_sizes` parameter:

```
#Check if expected dbn_sizes are correct
if(length(dbn_sizes)!=length(rbm_list)){
stop("number of hidden dbn_sizes not equal to number of rbm outputs
generated")
# check if expected sized are correct
for(i in 1:length(dbn_sizes)){
if(dbn_sizes[i] != dbn_sizes[i])
stop("Number of hidden dbn_sizes do not match")
}
}
```

4. Now, place the weights and biases in suitable positions within `weight_list` and `bias_list`:

```
for(i in 1:length(dbn_sizes)){
weight_list[[i]] <- rbm_list[[i]]$weight_final
bias_list[[i]] <- rbm_list[[i]]$bias_final
}
```

5. Create placeholders for the input and output data:

```
input <- tf$placeholder(tf$float32, shape =
shape(NULL,ncol(Xdata)))
output <- tf$placeholder(tf$float32, shape =
shape(NULL,ncol(Ydata)))
```

6. Now, use the weights and biases obtained from the stacked RBM to reconstruct the input data and store each RBM's reconstructed data in the list `input_sub`:

```
input_sub <- list()
weight <- list()
bias <- list()
for(i in 1:(length(dbn_sizes)+1)){
weight[[i]] <- tf$cast(tf$Variable(weight_list[[i]]),tf$float32)
bias[[i]] <- tf$cast(tf$Variable(bias_list[[i]]),tf$float32)
}
input_sub[[1]] <- tf$nn$sigmoid(tf$matmul(input, weight[[1]]) +
bias[[1]])
for(i in 2:(length(dbn_sizes)+1)){
input_sub[[i]] <- tf$nn$sigmoid(tf$matmul(input_sub[[i-1]],
weight[[i]]) + bias[[i]])
}
```

7. Define the cost function--that is, the mean squared error of difference between prediction and actual digits:

```
cost = tf$reduce_mean(tf$square(input_sub[[length(input_sub)]] -
output))
```

8. Implement backpropagation for the purpose of minimizing the cost:

```
train_op <- tf$train$MomentumOptimizer(learning_rate,
momentum)$minimize(cost)
```

9. Generate the prediction results:

```
predict_op =
tf$argmax(input_sub[[length(input_sub)]],axis=tf$cast(1.0,tf$int32)
)
```

10. Perform iterations of training:

```
train_accuracy <- c()
test_accuracy <- c()
for(ep in 1:epochs){
for(i in seq(0,(dim(Xdata)[1]-batchsize),batchsize)){
```

```
batchX <- Xdata[(i+1):(i+batchsize),]
batchY <- Ydata[(i+1):(i+batchsize),]
#Run the training operation on the input data
sess$run(train_op,feed_dict=dict(input = batchX,
output = batchY))
}
for(j in 1:(length(dbn_sizes)+1)){
# Retrieve weights and biases
weight_list[[j]] <- sess$run(weight[[j]])
bias_list[[j]] <- sess$ run(bias[[j]])
}
train_result <- sess$run(predict_op, feed_dict = dict(input=Xdata,
output=Ydata))+1
train_actual <-
as.numeric(stringi::stri_sub(colnames(as.data.frame(Ydata))[max.col
(as.data.frame(Ydata),ties.method="first")],2))
test_result <- sess$run(predict_op, feed_dict =
dict(input=Xtestdata, output=Ytestdata))+1
test_actual <-
as.numeric(stringi::stri_sub(colnames(as.data.frame(Ytestdata))[max
.col(as.data.frame(Ytestdata),ties.method="first")],2))
train_accuracy <-
c(train_accuracy,mean(train_actual==train_result))
test_accuracy <- c(test_accuracy,mean(test_actual==test_result))
cat("epoch:", ep, " Train Accuracy: ",train_accuracy[ep]," Test
Accuracy : ",test_accuracy[ep],"\n")
}
```

11. Finally, return a list of four outcomes, which are train accuracy (`train_accuracy`), test accuracy (`test_accuracy`), a list of weight matrices generated in each iteration (`weight_list`), and a list of bias vectors generated in each iteration (`bias_list`):

```
return(list(train_accuracy=train_accuracy,
test_accuracy=test_accuracy,
weight_list=weight_list,
bias_list=bias_list))
```

12. Run the iterations for the defined neural network for training:

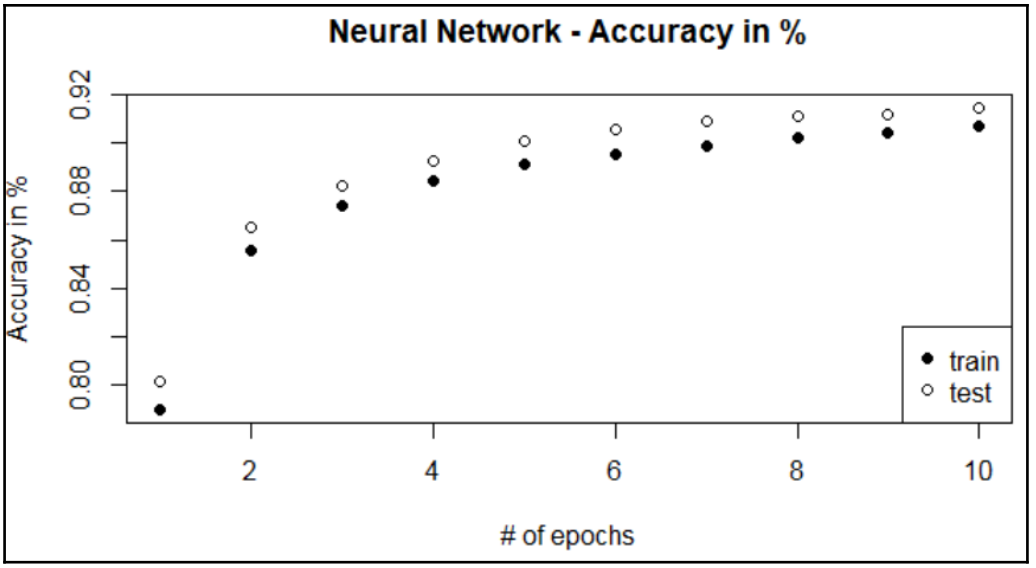
```
NN_results <- NN_train(Xdata=trainX,
Ydata=trainY,
Xtestdata=testX,
Ytestdata=testY,
input_size=ncol(trainX),
rbm_list=RBM_output,
dbn_sizes = RBM_hidden_sizes)
```

13. The following code is used to plot the train and test accuracy:

```
accuracy_df <-
data.frame("accuracy"=c(NN_results$train_accuracy,NN_results$test_a
ccuracy),
"epochs"=c(rep(1:10,times=2)),
"datatype"=c(rep(c(1,2),each=10)),
stringsAsFactors = FALSE)
plot(accuracy ~ epochs,
xlab = "# of epochs",
ylab = "Accuracy in %",
pch = c(16, 1)[datatype],
main = "Neural Network - Accuracy in %",
data = accuracy_df)
legend('bottomright',
c("train","test"),
pch = c( 16, 1))
```

How it works...

Assessing the train and test performance of the neural network: The following image shows the increasing trend of train and test accuracy observed while training the neural network:



Setting up a Deep Restricted Boltzmann Machine

Unlike DBNs, **Deep Restricted Boltzmann Machines** (**DRBM**) are undirected networks of interconnected hidden layers with the capability to learn joint probabilities over these connections. In the current setup, centering is performed where visible and hidden variables are subtracted from offset bias vectors after every iteration. Research has shown that centering optimizes the performance of DRBMs and can reach higher log-likelihood values in comparison with traditional RBMs.

Getting ready

This section provides the requirements for setting up a DRBM:

- The `MNIST` dataset is loaded and set up
- The `tensorflow` package is set up and loaded

How to do it...

This section covers detailed the steps for setting up the DRBM model using TensorFlow in R:

1. Define the parameters for the DRBM:

```
learning_rate = 0.005
momentum = 0.005
minbatch_size = 25
hidden_layers = c(400,100)
biases = list(-1,-1)
```

2. Define a sigmoid function using a hyperbolic arc tangent *[(log(1+x) -log(1-x))/2]*:

```
arcsigm <- function(x){
return(atanh((2*x)-1)*2)
}
```

3. Define a sigmoid function using only a hyperbolic tangent $[(e^{x}-e^{-x})/(e^{x}+e^{-x})]$:

```
sigm <- function(x){
return(tanh((x/2)+1)/2)
}
```

4. Define a `binarize` function to return a matrix of binary values (0,1):

```
binarize <- function(x){
# truncated rnorm
trnrom <- function(n, mean, sd, minval = -Inf, maxval = Inf){
qnorm(runif(n, pnorm(minval, mean, sd), pnorm(maxval, mean, sd)),
mean, sd)
}
return((x > matrix(
trnrom(n=nrow(x)*ncol(x),mean=0,sd=1,minval=0,maxval=1), nrow(x),
ncol(x)))*1)
}
```

5. Define a `re_construct` function to return a matrix of pixels:

```
re_construct <- function(x){
x = x - min(x) + 1e-9
x = x / (max(x) + 1e-9)
return(x*255)
}
```

6. Define a function to perform `gibbs` activation for a given layer:

```
gibbs <- function(X,l,initials){
if(l>1){
bu <- (X[l-1][[1]] -
matrix(rep(initials$param_O[[l-1]],minbatch_size),minbatch_size,byr
ow=TRUE)) %*%
initials$param_W[l-1][[1]]
} else {
bu <- 0
}
if((l+1) < length(X)){
td <- (X[l+1][[1]] -
matrix(rep(initials$param_O[[l+1]],minbatch_size),minbatch_size,byr
ow=TRUE))%*%
t(initials$param_W[l][[1]])
} else {
td <- 0
}
X[[l]] <-
binarize(sigm(bu+td+matrix(rep(initials$param_B[[l]],minbatch_size)
,minbatch_size,byrow=TRUE)))
return(X[[l]])
}
```

7. Define a function to perform the reparameterization of bias vectors:

```
reparamBias <- function(X,l,initials){
if(l>1){
bu <- colMeans((X[[l-1]] -
matrix(rep(initials$param_O[[l-1]],minbatch_size),minbatch_size,byr
ow=TRUE))%*%
initials$param_W[[l-1]])
} else {
bu <- 0
}
if((l+1) < length(X)){
td <- colMeans((X[[l+1]] -
matrix(rep(initials$param_O[[l+1]],minbatch_size),minbatch_size,byr
ow=TRUE))%*%
t(initials$param_W[[l]]))
} else {
td <- 0
}
initials$param_B[[l]] <- (1-momentum)*initials$param_B[[l]] +
momentum*(initials$param_B[[l]] + bu + td)
return(initials$param_B[[l]])
}
```

8. Define a function to perform the reparameterization of offset bias vectors:

```
reparamO <- function(X,l,initials){
initials$param_O[[l]] <- colMeans((1-
momentum)*matrix(rep(initials$param_O[[l]],minbatch_size),minbatch_
size,byrow=TRUE) + momentum*(X[[l]]))
return(initials$param_O[[l]])
}
```

9. Define a function to initialize weights, biases, offset biases, and input matrices:

```
DRBM_initialize <- function(layers,bias_list){
# Initialize model parameters and particles
param_W <- list()
for(i in 1:(length(layers)-1)){
param_W[[i]] <- matrix(0L, nrow=layers[i], ncol=layers[i+1])
}
param_B <- list()
for(i in 1:length(layers)){
param_B[[i]] <- matrix(0L, nrow=layers[i], ncol=1) + bias_list[[i]]
}
param_O <- list()
for(i in 1:length(param_B)){
param_O[[i]] <- sigm(param_B[[i]])
```

```
}
param_X <- list()
for(i in 1:length(layers)){
param_X[[i]] <- matrix(0L, nrow=minbatch_size, ncol=layers[i]) +
matrix(rep(param_O[[i]],minbatch_size),minbatch_size,byrow=TRUE)
}
return(list(param_W=param_W,param_B=param_B,param_O=param_O,param_X
=param_X))
}
```

10. Use the MNIST train data (`trainX`) introduced in the previous recipes. Standardize the `trainX` data by dividing it by 255:

```
X <- trainX/255
```

11. Generate the initial weight matrices, bias vectors, offset bias vectors, and input matrices:

```
layers <- c(784,hidden_layers)
bias_list <-
list(arcsigm(pmax(colMeans(X),0.001)),biases[[1]],biases[[2]])
initials <-DRBM_initialize(layers,bias_list)
```

12. Subset a sample (`minbatch_size`) of the input data `X`:

```
batchX <- X[sample(nrow(X))[1:minbatch_size],]
```

13. Perform a set of 1,000 iterations. Within each iteration, update the initial weights and biases 100 times and plot the images of the weight matrices:

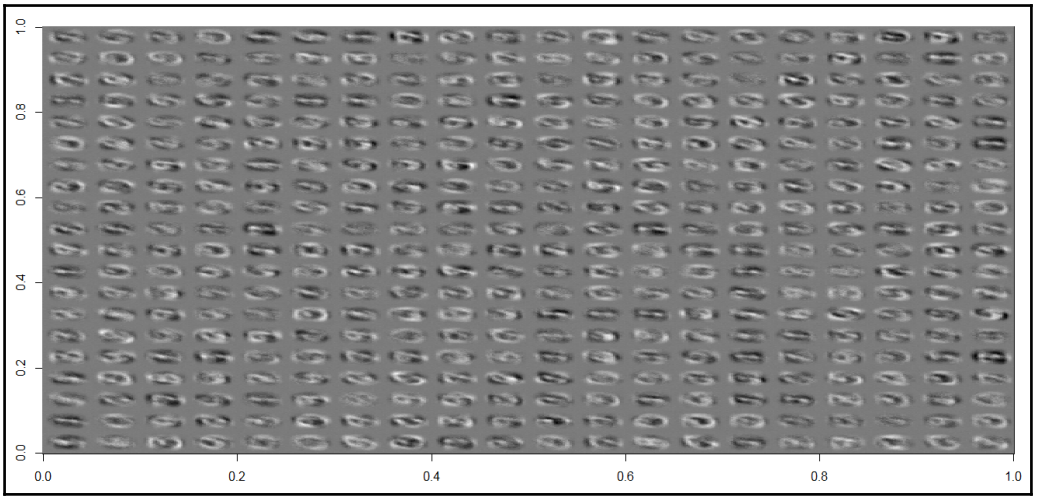
```
for(iter in 1:1000){

# Perform some learnings
for(j in 1:100){
# Initialize a data particle
dat <- list()
dat[[1]] <- binarize(batchX)
for(l in 2:length(initials$param_X)){
dat[[l]] <- initials$param_X[l][[1]]*0 +
matrix(rep(initials$param_O[l][[1]],minbatch_size),minbatch_size,by
row=TRUE)
}

# Alternate gibbs sampler on data and free particles
for(l in rep(c(seq(2,length(initials$param_X),2),
seq(3,length(initials$param_X),2)),5)){
dat[[l]] <- gibbs(dat,l,initials)
```

```
}
for(l in rep(c(seq(2,length(initials$param_X),2),
seq(1,length(initials$param_X),2)),1)){
initials$param_X[[l]] <- gibbs(initials$param_X,l,initials)
}

# Parameter update
for(i in 1:length(initials$param_W)){
initials$param_W[[i]] <- initials$param_W[[i]] +
(learning_rate*((t(dat[[i]] -
matrix(rep(initials$param_O[i][[1]],minbatch_size),minbatch_size,by
row=TRUE)) %*%
(dat[[i+1]] -
matrix(rep(initials$param_O[i+1][[1]],minbatch_size),minbatch_size,
byrow=TRUE))) -
(t(initials$param_X[[i]] -
matrix(rep(initials$param_O[i][[1]],minbatch_size),minbatch_size,by
row=TRUE)) %*%
(initials$param_X[[i+1]] -
matrix(rep(initials$param_O[i+1][[1]],minbatch_size),minbatch_size,
byrow=TRUE))))/nrow(batchX))
}

for(i in 1:length(initials$param_B)){
initials$param_B[[i]] <-
colMeans(matrix(rep(initials$param_B[[i]],minbatch_size),minbatch_s
ize,byrow=TRUE) + (learning_rate*(dat[[i]] -
initials$param_X[[i]])))
}

# Reparameterization
for(l in 1:length(initials$param_B)){
initials$param_B[[l]] <- reparamBias(dat,l,initials)
}
for(l in 1:length(initials$param_O)){
initials$param_O[[l]] <- reparamO(dat,l,initials)
}
}

# Generate necessary outputs
cat("Iteration:",iter," ","Mean of W of VL-
HL1:",mean(initials$param_W[[1]]),"  ","Mean of W of HL1-
HL2:",mean(initials$param_W[[2]]) ,"\n")
cat("Iteration:",iter," ","SDev of W of VL-
HL1:",sd(initials$param_W[[1]]),"  ","SDev of W of HL1-
HL2:",sd(initials$param_W[[2]]) ,"\n")

# Plot weight matrices
```

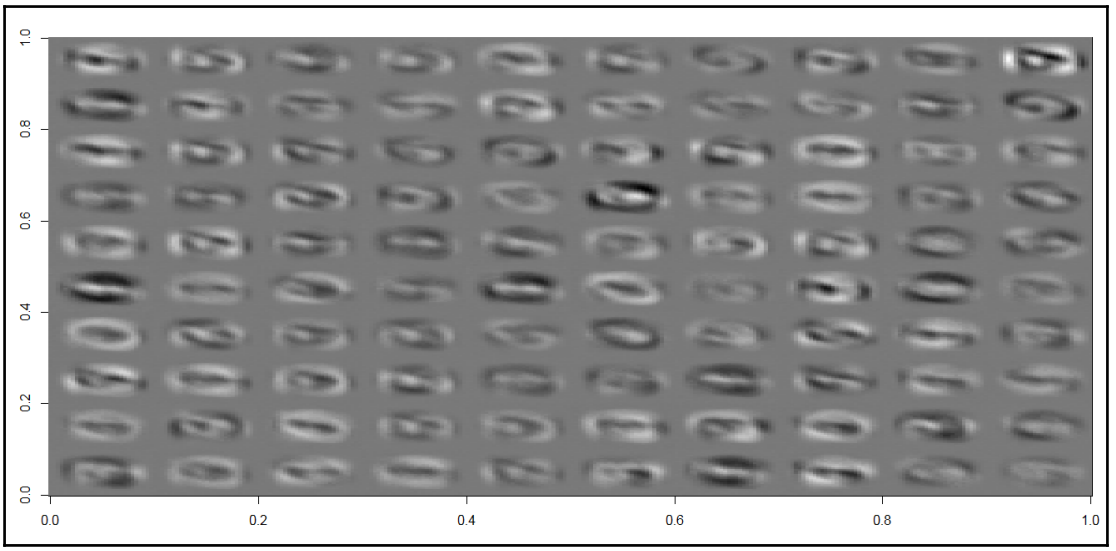
```
W=diag(nrow(initials$param_W[[1]]))
for(l in 1:length(initials$param_W)){
W = W %*% initials$param_W[[l]]
m = dim(W)[2] * 0.05
w1_arr <- matrix(0,28*m,28*m)
i=1
for(k in 1:m){
for(j in 1:28){
vec <- c(W[(28*j-28+1):(28*j),(k*m-m+1):(k*m)])
w1_arr[i,] <- vec
i=i+1
}
}
w1_arr = re_construct(w1_arr)
w1_arr <- floor(w1_arr)
image(w1_arr,axes = TRUE, col = grey(seq(0, 1, length = 256)))
}
}
```

How it works...

As the preceding DRBM is trained using two hidden layers, we generate two weight matrices. The first weight matrix defines the connection between the visible layer and the first hidden layer. The second weight matrix defines the connection between the first and second hidden layer. The following image shows pixel images of the first weight matrices:



The following image shows the second pixel images of the second weight matrix:



6
Recurrent Neural Networks

The current chapter will introduce Recurrent Neural Networks used for the modeling of sequential datasets. In this chapter, we will cover:

- Setting up a basic Recurrent Neural Network
- Setting up a bidirectional RNN model
- Setting up a deep RNN model
- Setting up a Long short-term memory based sequence model

Setting up a basic Recurrent Neural Network

Recurrent Neural Networks (**RNN**) are used for sequential modeling on datasets where high autocorrelation exists among observations. For example, predicting patient journeys using their historical dataset or predicting the next words in given sentences. The main commonality among these problem statements is that input length is not constant and there is a sequential dependence. Standard neural network and deep learning models are constrained by fixed size input and produce a fixed length output. For example, deep learning neural networks built on occupancy datasets have six input features and a binomial outcome.

Getting ready

Generative models in machine learning domains are referred to as models that have an ability to generate observable data values. For example, training a generative model on an images repository to generate new images like it. All generative models aim to compute the joint distribution over given datasets, either implicitly or explicitly:

1. Install and set up TensorFlow.
2. Load required packages:

```
library(tensorflow)
```

How to do it...

The section will provide steps to set-up an RNN model.

1. Load the `MNIST` dataset:

```
# Load mnist dataset from tensorflow library
datasets <- tf$contrib$learn$datasets
mnist <- datasets$mnist$read_data_sets("MNIST-data", one_hot =
TRUE)
```

2. Reset the graph and start an interactive session:

```
# Reset the graph and set-up a interactive session
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

3. Reduce image size to 16 x 16 pixels using the `reduceImage` function from `Chapter 4`, *Data Representation using Autoencoders*:

```
# Covert train data to 16 x 16  pixel image
trainData<-t(apply(mnist$train$images, 1, FUN=reduceImage))
validData<-t(apply(mnist$test$images, 1, FUN=reduceImage))
```

4. Extract labels for the defined `train` and `valid` datasets:

```
labels <- mnist$train$labels
labels_valid <- mnist$test$labels
```

5. Define model parameters such as size of input pixels (`n_input`), step size (`step_size`), number of hidden layers (`n.hidden`), and number of outcome classes (`n.classes`):

```
# Define Model parameter
n_input<-16
step_size<-16
n.hidden<-64
n.class<-10
```

6. Define training parameters such as learning rate (`lr`), number of inputs per batch run (`batch`), and number of iterations (`iteration`):

```
lr<-0.01
batch<-500
iteration = 100
```

7. Define a function `rnn` that takes in batch input dataset (`x`), weight matrix (`weight`), and bias vector (`bias`); and returns a final outcome predicted vector of a most basic RNN:

```
# Set up a most basic RNN
rnn<-function(x, weight, bias){
  # Unstack input into step_size
  x = tf$unstack(x, step_size, 1)
  # Define a most basic RNN
  rnn_cell = tf$contrib$rnn$BasicRNNCell(n.hidden)
  # create a Recurrent Neural Network
  cell_output = tf$contrib$rnn$static_rnn(rnn_cell, x,
dtype=tf$float32)
  # Linear activation, using rnn inner loop
  last_vec=tail(cell_output[[1]], n=1)[[1]]
  return(tf$matmul(last_vec, weights) + bias)
}
Define a function eval_func to evaluate mean accuracy using actual
(y) and predicted labels (yhat):
# Function to evaluate mean accuracy
eval_acc<-function(yhat, y){
  # Count correct solution
  correct_Count = tf$equal(tf$argmax(yhat,1L), tf$argmax(y,1L))
  # Mean accuracy
  mean_accuracy = tf$reduce_mean(tf$cast(correct_Count,
tf$float32))
  return(mean_accuracy)
}
```

8. Define `placeholder` variables (`x` and `y`) and initialize weight matrix and bias vector:

```
with(tf$name_scope('input'), {
# Define placeholder for input data
x = tf$placeholder(tf$float32, shape=shape(NULL, step_size,
n_input), name='x')
y <- tf$placeholder(tf$float32, shape(NULL, n.class), name='y')

# Define Weights and bias
weights <- tf$Variable(tf$random_normal(shape(n.hidden, n.class)))
bias <- tf$Variable(tf$random_normal(shape(n.class)))
})
```

9. Generate the predicted labels:

```
# Evaluate rnn cell output
yhat = rnn(x, weights, bias)
Define the loss function and optimizer
cost =
tf$reduce_mean(tf$nn$softmax_cross_entropy_with_logits(logits=yhat,
labels=y))
optimizer = tf$train$AdamOptimizer(learning_rate=lr)$minimize(cost)
```

10. Run the optimization post initializing a session using the global variables initializer:

```
sess$run(tf$global_variables_initializer())
for(i in 1:iteration){
  spls <- sample(1:dim(trainData)[1],batch)
  sample_data<-trainData[spls,]
  sample_y<-labels[spls,]
  # Reshape sample into 16 sequence with each of 16 element
  sample_data=tf$reshape(sample_data, shape(batch, step_size,
n_input))
  out<-optimizer$run(feed_dict = dict(x=sample_data$eval(),
y=sample_y))
  if (i %% 1 == 0){
    cat("iteration - ", i, "Training Loss - ",  cost$eval(feed_dict
= dict(x=sample_data$eval(), y=sample_y)), "\n")
  }
}
```

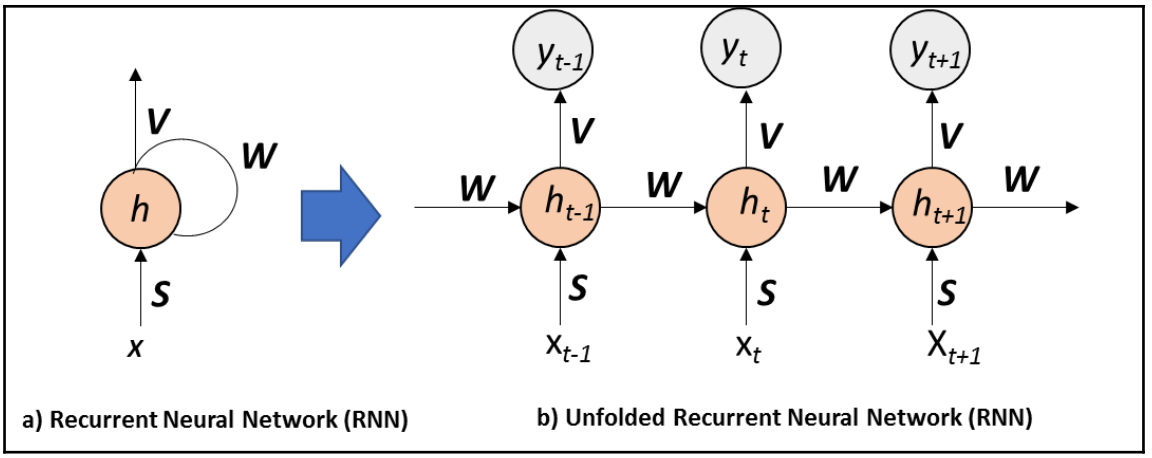
11. Get the mean accuracy on `valid_data`:

```
valid_data=tf$reshape(validData, shape(-1, step_size, n_input))
cost$eval(feed_dict=dict(x=valid_data$eval(), y=labels_valid))
```

How it works...

Any changes to the structure require model retraining. However, these assumptions may not be valid for a lot of sequential datasets, such as text-based classifications that may have varying input and output. RNN architecture helps to address the issue of variable input length.

The standard architecture for RNN with input and output is shown in the following figure:



Recurrent Neural Network architecture

The RNN architecture can be formulated as follows:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, x_t; \mathbf{S}, \mathbf{W})$$

Where h_t is state at time/index *t* and x_t is input at time/index *t*. The matrix *W* represents weights to connect hidden nodes and *S* connects input with the hidden layer. The output node at time/index *t* is related to state h_t as shown as follows:

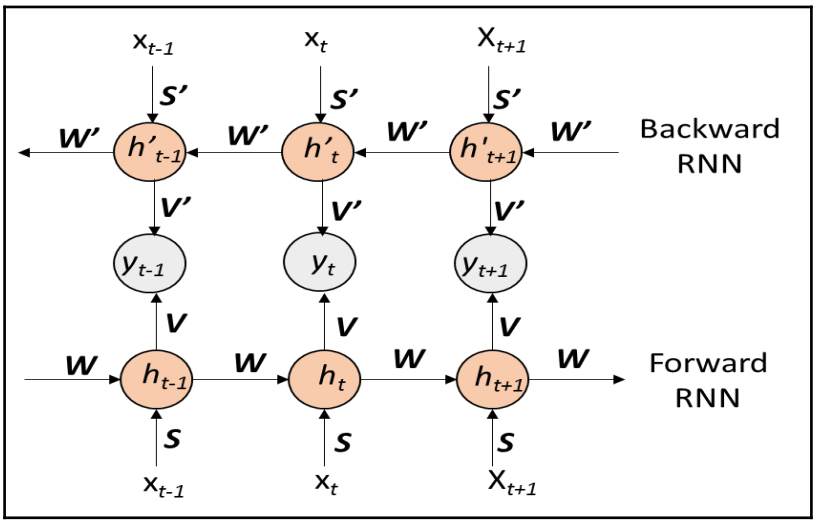
$$\mathbf{y}_t = f(\mathbf{h}_t; \mathbf{V})$$

In the previous Equations layer, weights remain constant across state and time.

Setting up a bidirectional RNN model

Recurrent Neural Networks focus on capturing the sequential information at time *t* by using historical states only. However, bidirectional RNN train the model from both directions using two RNN layers with one moving forwards from start to end and another RNN layer moving backwards from end to start of sequence.

Thus, the model is dependent on historical and future data. The bidirectional RNN models are useful where causal structure exists such as in text and speech. The unfolded structure of bidirectional RNN is shown in the following figure:



Unfolded bidirectional RNN architecture

Getting ready

Install and set up TensorFlow:

1. Load required packages:

```
library(tensorflow)
```

2. Load `MNIST` dataset.
3. The image from `MNIST` dataset is reduced to 16 x 16 pixels and normalized (Details are discussed in the *Setting-up RNN model* section).

How to do it...

This section covers the steps to set-up a bidirectional RNN model.

1. Reset the graph and start an interactive session:

```
# Reset the graph and set-up a interactive session
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

2. Reduce image size to 16 x 16 pixels using the `reduceImage` function from `Chapter 4`, *Data Representation using Autoencoders*:

```
# Covert train data to 16 x 16  pixel image
trainData<-t(apply(mnist$train$images, 1, FUN=reduceImage))
validData<-t(apply(mnist$test$images, 1, FUN=reduceImage))
```

3. Extract labels for the defined `train` and `valid` datasets:

```
labels <- mnist$train$labels
labels_valid <- mnist$test$labels
```

4. Define model parameters such as the size of input pixels (`n_input`), step size (`step_size`), number of hidden layers (`n.hidden`), and number of outcome classes (`n.classes`):

```
# Define Model parameter
n_input<-16
step_size<-16
n.hidden<-64
n.class<-10
```

5. Define training parameters such as learning rate (`lr`), number of inputs per batch run (`batch`), and number of iterations (`iteration`):

```
lr<-0.01
batch<-500
iteration = 100
```

6. Define a function to perform `bidirectional` Recurrent Neural Network:

```
bidirectionRNN<-function(x, weights, bias){
  # Unstack input into step_size
  x = tf$unstack(x, step_size, 1)
  # Forward lstm cell
  rnn_cell_forward = tf$contrib$rnn$BasicRNNCell(n.hidden)
```

```
  # Backward lstm cell
  rnn_cell_backward = tf$contrib$rnn$BasicRNNCell(n.hidden)
  # Get lstm cell output
  cell_output =
tf$contrib$rnn$static_bidirectional_rnn(rnn_cell_forward,
rnn_cell_backward, x, dtype=tf$float32)
  # Linear activation, using rnn inner loop last output
  last_vec=tail(cell_output[[1]], n=1)[[1]]
  return(tf$matmul(last_vec, weights) + bias)
}
```

7. Define an `eval_func` function to evaluate mean accuracy using actual (`y`) and predicted labels (`yhat`):

```
# Function to evaluate mean accuracy
eval_acc<-function(yhat, y){
  # Count correct solution
  correct_Count = tf$equal(tf$argmax(yhat,1L), tf$argmax(y,1L))
  # Mean accuracy
  mean_accuracy = tf$reduce_mean(tf$cast(correct_Count,
tf$float32))
  return(mean_accuracy)
}
```

8. Define `placeholder` variables (`x` and `y`) and initialize weight matrix and bias vector:

```
with(tf$name_scope('input'), {
# Define placeholder for input data
x = tf$placeholder(tf$float32, shape=shape(NULL, step_size,
n_input), name='x')
y <- tf$placeholder(tf$float32, shape(NULL, n.class), name='y')

# Define Weights and bias
weights <- tf$Variable(tf$random_normal(shape(n.hidden, n.class)))
bias <- tf$Variable(tf$random_normal(shape(n.class)))
})
```

9. Generate the predicted labels:

```
# Evaluate rnn cell output
yhat = bidirectionRNN(x, weights, bias)
```

10. Define the loss function and optimizer:

```
cost =
tf$reduce_mean(tf$nn$softmax_cross_entropy_with_logits(logits=yhat,
```

```
labels=y))
optimizer = tf$train$AdamOptimizer(learning_rate=lr)$minimize(cost)
```

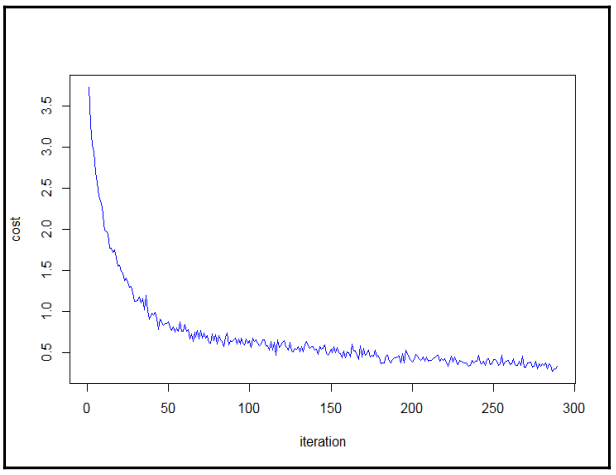
11. Run the optimization post initializing a session using global variables initializer:

```
sess$run(tf$global_variables_initializer())
# Running optimization
for(i in 1:iteration){
  spls <- sample(1:dim(trainData)[1],batch)
  sample_data<-trainData[spls,]
  sample_y<-labels[spls,]
  # Reshape sample into 16 sequence with each of 16 element
  sample_data=tf$reshape(sample_data, shape(batch, step_size,
n_input))
  out<-optimizer$run(feed_dict = dict(x=sample_data$eval(),
y=sample_y))
  if (i %% 1 == 0){
    cat("iteration - ", i, "Training Loss - ",  cost$eval(feed_dict
= dict(x=sample_data$eval(), y=sample_y)), "\n")
  }
}
```

12. Get the mean accuracy on valid data:

```
valid_data=tf$reshape(validData, shape(-1, step_size, n_input))
cost$eval(feed_dict=dict(x=valid_data$eval(), y=labels_valid))
```

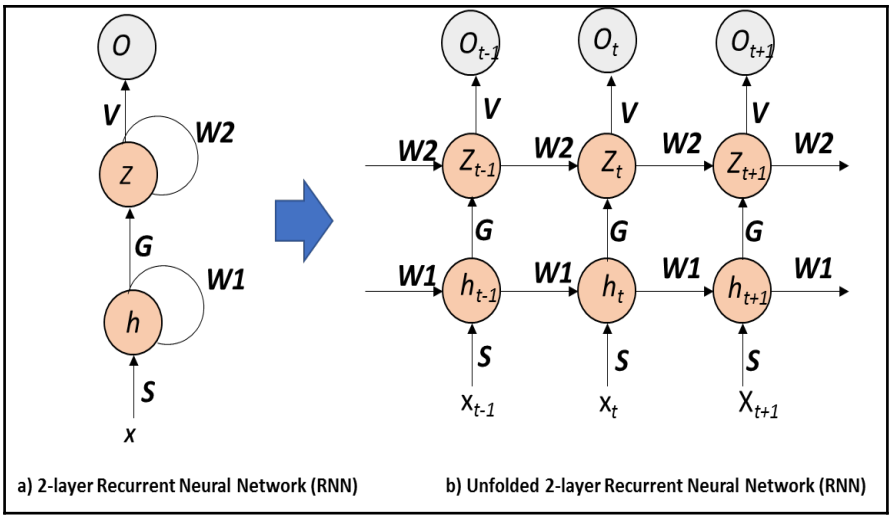
13. The convergence of cost function for RNN is shown in the following figure:



Bidirectional Recurrent Neural Network convergence plot on MNIST dataset

Setting up a deep RNN model

The RNN architecture is composed of input, hidden, and output layers. A RNN network can be made deep by decomposing the hidden layer into multiple groups or by adding computational nodes within RNN architecture such as including model computation such as multilayer perceptron for micro learning. The computational nodes can be added between input-hidden, hidden-hidden, and hidden-output connection. An example of a multilayer deep RNN model is shown in the following figure:



An example of two-layer Deep Recurrent Neural Network architecture

How to do it...

The RNN models in TensorFlow can easily be extended to Deep RNN models by using `MultiRNNCell`. The previous `rnn` function can be replaced with the `stacked_rnn`function to achieve a deep RNN architecture:

1. Define the number of layers in the deep RNN architecture:

```
num_layers <- 3
```

2. Define a `stacked_rnn` function to perform multi-hidden layers deep RNN:

```
stacked_rnn<-function(x, weight, bias){
  # Unstack input into step_size
  x = tf$unstack(x, step_size, 1)
```

```
  # Define a most basic RNN
  network = tf$contrib$rnn$GRUCell(n.hidden)
  # Then, assign stacked RNN cells
  network =
tf$contrib$rnn$MultiRNNCell(lapply(1:num_layers,function(k,network)
{network},network))
  # create a Recurrent Neural Network
  cell_output = tf$contrib$rnn$static_rnn(network, x,
dtype=tf$float32)
  # Linear activation, using rnn inner loop
  last_vec=tail(cell_output[[1]], n=1)[[1]]
  return(tf$matmul(last_vec, weights) + bias)
}
```

Setting up a Long short-term memory based sequence model

In sequence learning the objective is to capture short-term and long-term memory. The short-term memory is captured very well by standard RNN, however, they are not very effective in capturing long-term dependencies as the gradient vanishes (or explodes rarely) within an RNN chain over time.

The gradient vanishes when the weights have small values that on multiplication vanish over time, whereas in contrast, scenarios where weights have large values keep increasing over time and lead to divergence in the learning process. To deal with the issue **Long Short Term Memory** (**LSTM**) is proposed.

How to do it...

The RNN models in TensorFlow can easily be extended to LSTM models by using `BasicLSTMCell`. The previous `rnn` function can be replaced with the `lstm` function to achieve an LSTM architecture:

```
# LSTM implementation
lstm<-function(x, weight, bias){
  # Unstack input into step_size
  x = tf$unstack(x, step_size, 1)
  # Define a lstm cell
  lstm_cell = tf$contrib$rnn$BasicLSTMCell(n.hidden, forget_bias=1.0,
state_is_tuple=TRUE)
  # Get lstm cell output
```

```
    cell_output = tf$contrib$rnn$static_rnn(lstm_cell, x, dtype=tf$float32)
    # Linear activation, using rnn inner loop last output
    last_vec=tail(cell_output[[1]], n=1)[[1]]
    return(tf$matmul(last_vec, weights) + bias)
}
```

For brevity the other parts of the code are not replicated.

How it works...

The LSTM has a similar structure to RNN, however, the basic cell is very different as traditional RNN uses single **multi-layer perceptron** (**MLP**), whereas a single cell of LSTM includes four input layers interacting with each other. These three layers are:

- forget gate
- input gate
- output gate

The *forget gate* in LSTM decides which information to throw away and it depends on the last hidden state output h_{t-1}, X_t, which represents input at time *t*.

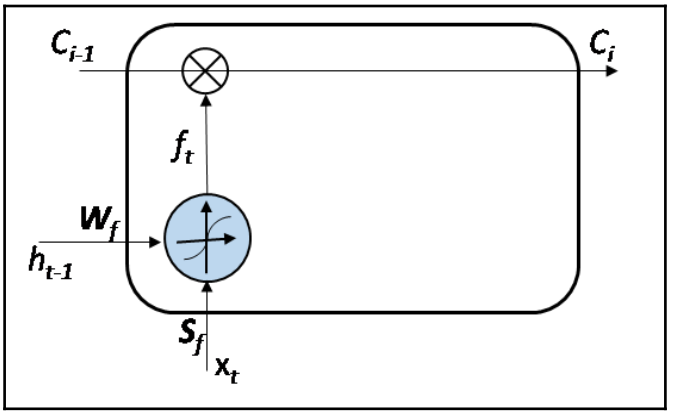


Illustration of forget gate

In the earlier figure, C_t represents cell state at time *t*. The input data is represented by X_t and the hidden state is represented as h_{t-1}. The earlier layer can be formulated as:

$$f_t = \sigma(\mathbf{S}_f \mathbf{x}_t + \mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{b}_f)$$

The *input gate* decides update values and decides the candidate values of the memory cell and updates the cell state, as shown in the following figure:

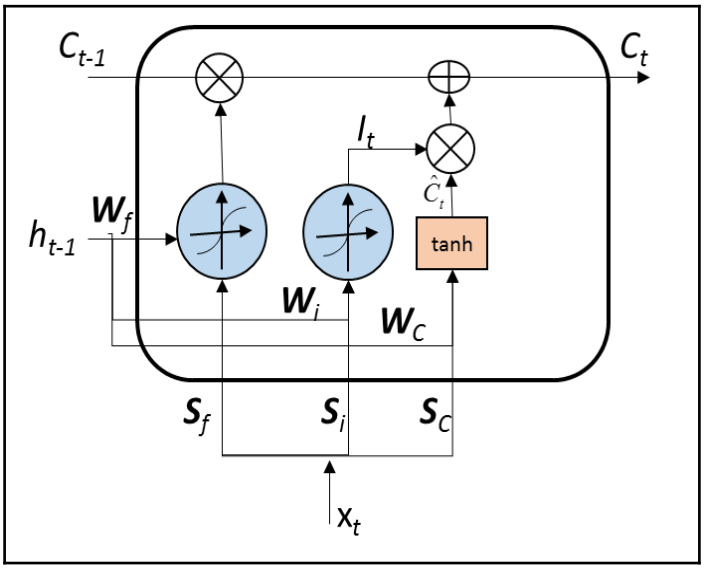


Illustration of input gate

- The input i_t at time t is updated as:

$$i_t = \sigma(\mathbf{S}_i \mathbf{x}_t + \mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{b}_i)$$

$$\hat{C}_t = \tanh(\mathbf{S}_C \mathbf{x}_t + \mathbf{W}_C \mathbf{h}_{t-1} + \mathbf{b}_C)$$

- The expected value of current state $\hat{C}_t$ and the output from input gate i_t is used to update the current state C_t at time t as:

$$C_t = I_t * \hat{C}_t + f_t * C_{t-1}$$

The output gates, as shown in the following figure, compute the output from the LSTM cell based on input X_t , previous layer output h_{t-1}, and current state C_t:

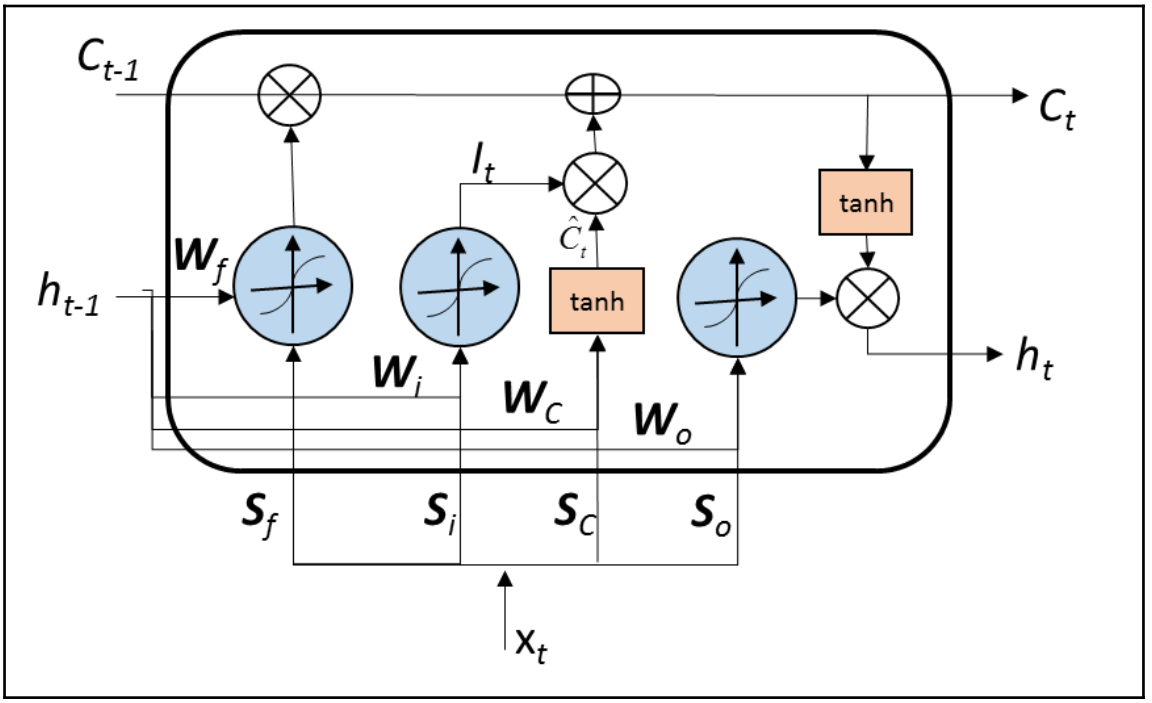


Illustration of output gate

The output based on *output gate* can be computed as follows:

$$O_t = \sigma(\mathbf{S}_o \mathbf{x}_t + \mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{b}_o)$$

$$h_t = O_t * \tanh(C_t)$$

7
Reinforcement Learning

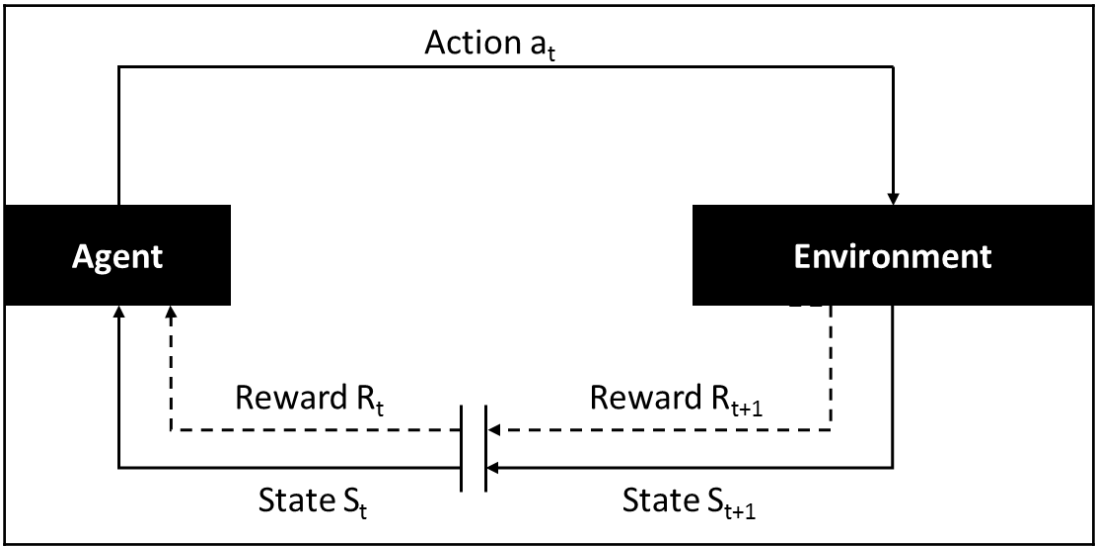
The current chapter will introduce Reinforcement Learning. We will cover the following topics:

- Setting up a Markov Decision Process
- Performing model-based learning
- Performing model-free learning

Introduction

Reinforcement Learning (**RL**) is an area in machine learning that is inspired by psychology, such as how agents (software programs) can take actions in order to maximize cumulative rewards.

The RL is reward-based learning where the reward comes at the end or is distributed during the learning. For example, in chess, the reward will be assigned to winning or losing the game whereas in games such as tennis, every point won is a reward. Some of the commercial examples of RL are DeepMind from Google uses RL to master parkour. Similarly, Tesla is developing AI-driven technology using RL. An example of reinforcement architecture is shown in the following figure:



Interaction of an agent with environment in Reinforcement Learning

The basic notations for RL are as follows:

- **T(s, a, s')**: Represents the transition model for reaching state s' when action a is taken at state s
- $\boldsymbol{P}$: Represents a policy which defines what action to take at every possible state $s \in S$
- **R(s)**: Denotes the reward received by agent at state s

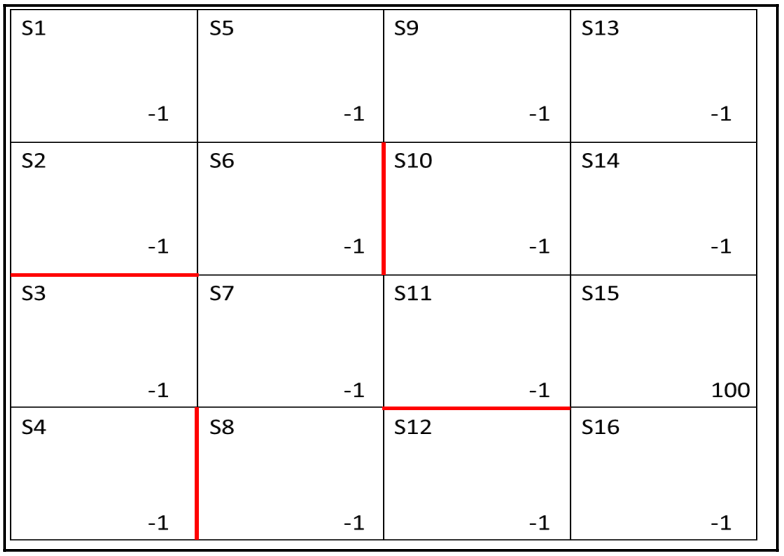
The current chapter will look at how to set up reinforcement models using R. The next sub-section will introduce `MDPtoolbox` from R.

Setting up a Markov Decision Process

The **Markov Decision Process** (**MDP**) forms the basis of setting up RL, where the outcome of a decision is semi-controlled; that is, it is partly random and partly controlled (by the decision-maker). An MDP is defined using a set of possible states (**S**), a set of possible actions (**A**), a real-values reward function (**R**), and a set of transition probabilities from one state to another state for a given action (**T**). In addition, the effects of an action performed on one state depends only on that state and not on its previous states.

Getting ready

In this section, let us define an agent travelling across a 4 x 4 grid, as shown in following figure:



A sample 4 x 4 grid of 16 states

This grid has 16 states (*S1*, *S2*....*S16*). In each state, the agent can perform four actions (*up*, *right*, *down*, *left*). However, the agent will be restricted to some actions based on the following constraints:

- The states across the edges shall be restricted to actions which point only toward states in the grid. For example, an agent in *S1* is restricted to the *right* or *down* action.

- Some state transitions have barriers, marked in red. For example, the agent cannot go *down* from *S2* to *S3*.

Each state is also assigned to a reward. The objective of the agent is to reach the destination with minimum moves, thereby achieving the maximum reward. Except state *S15* with a reward value of 100, all the remaining states have a reward value of *-1*.

Here, we will use the `MDPtoolbox` package in R.

How to do it...

This section will show you how to set up RL models using `MDPtoolbox` in R:

1. Install and load the required package:

```
Install.packages("MDPtoolbox")
library(MDPtoolbox)
```

2. Define the transition probabilities for action. Here, each row denotes `from state` and each column denotes `to state`. As we have 16 states, the transition probability matrix of each action shall be a 16 x 16 matrix, with each row adding upto 1:

```
up<- matrix(c(1      ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0.7   ,     0.2   ,     0     ,     0     ,     0
,     0.1   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0     ,     0     ,     0.8   ,     0.05  ,     0
,     0   ,     0.15  ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0     ,     0     ,     0.7   ,     0.3   ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0.1   ,     0     ,     0     ,     0     ,     0.7
,     0.1   ,     0     ,     0     ,     0.1   ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0     ,     0.05  ,     0     ,     0     ,     0.7
,     0.15    ,     0.1   ,     0     ,     0     ,     0     ,
0     ,     0     ,   0     ,     0     ,     0     ,     0     ,
                0     ,     0     ,     0.05  ,     0     ,     0
,     0.7   ,     0.15  ,     0.05  ,     0     ,     0     ,
0.05  ,     0     ,   0     ,     0     ,     0     ,     0     ,
                0     ,     0     ,     0     ,     0     ,     0
```

```
,     0   ,     0.7   ,     0.2   ,     0     ,     0     ,     0
,     0.1   ,   0     ,     0     ,     0     ,     0     ,
                0     ,     0     ,     0     ,     0     ,     0.05
,     0   ,     0     ,     0     ,     0.85  ,     0.05  ,     0
,     0     ,   0.05  ,     0     ,     0     ,     0     ,
                0     ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0.7   ,     0.2   ,
0.05  ,     0     ,   0     ,     0.05  ,     0     ,     0     ,
                0     ,     0     ,     0     ,     0     ,     0
,     0   ,     0.05  ,     0     ,     0     ,     0.7   ,     0.2
,     0     ,   0     ,     0     ,     0.05  ,     0     ,
                0     ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0.05  ,     0     ,     0     ,     0
,     0.9   ,   0     ,     0     ,     0     ,     0.05  ,
                0     ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0.1   ,     0     ,     0
,     0     ,   0.9   ,     0     ,     0     ,     0     ,
                0     ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0.1   ,     0
,     0     ,   0.7   ,     0.2   ,     0     ,     0     ,
                0     ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,
0.05  ,     0     ,   0     ,     0.8   ,     0.15  ,     0     ,
                0     ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0.8   ,     0.2   ),
nrow=16, ncol=16, byrow=TRUE)
left<- matrix(c(1     ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0.05  ,     0.9   ,     0     ,     0     ,
0     ,   0.05  ,     0     ,     0     ,     0     ,     0     ,
0     ,     0   ,     0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0.9   ,     0.05  ,     0
,     0   ,     0.05  ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0.05  ,     0.9   ,     0
,     0   ,     0     ,     0.05  ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0.8 ,     0     ,     0     ,     0     ,     0.1
,     0.05  ,     0     ,     0     ,     0.05  ,     0     ,
0     ,     0     ,   0     ,     0     ,     0     ,     0     ,
                0   ,     0.8   ,     0     ,     0     ,     0.05
,     0.1   ,     0.05  ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0.8   ,     0     ,     0
,     0.05  ,     0.1   ,     0.05  ,     0     ,     0     ,
0     ,     0     ,   0     ,     0     ,     0     ,     0     ,
```

```
                    0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0.1   ,     0.8   ,     0     ,     0     ,     0
,     0.1   ,   0     ,     0     ,     0     ,     0     ,
                    0   ,     0     ,     0     ,     0     ,     0.8
,     0   ,     0     ,     0     ,     0.1   ,     0.05  ,     0
,     0     ,   0.05  ,     0     ,     0     ,     0     ,
                    0   ,     0     ,     0     ,     0     ,     0
,     0.8   ,     0     ,     0     ,     0.05  ,     0.1   ,
0.05  ,     0     ,   0     ,     0     ,     0     ,     0     ,
                    0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0.8   ,     0     ,     0     ,     0.1   ,     0.1
,     0     ,   0     ,     0     ,     0     ,     0     ,
                    0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0.8   ,     0     ,     0     ,     0
,     0.2   ,   0     ,     0     ,     0     ,     0     ,
                    0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0.8   ,     0     ,     0
,     0     ,   0.2   ,     0     ,     0     ,     0     ,
                    0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0.8   ,     0
,     0     ,   0.05  ,     0.1   ,     0.05  ,     0     ,
                    0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0.8
,     0     ,   0     ,     0.05  ,     0.1   ,     0.05  ,
                    0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0.8   ,   0     ,     0     ,     0.05  ,     0.15),
nrow=16, ncol=16, byrow=TRUE)
down<- matrix(c(0.1   ,     0.8   ,     0     ,     0     ,     0.1
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                    0.05      ,     0.9   ,     0     ,     0     ,
0     ,   0.05  ,     0     ,     0     ,     0     ,     0     ,
0     ,     0   ,     0     ,     0     ,     0     ,     0     ,
                    0   ,     0     ,     0.1   ,     0.8   ,     0
,     0   ,     0.1   ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                    0   ,     0     ,     0.1   ,     0.9   ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                    0.05      ,     0     ,     0     ,     0     ,
0.15  ,   0.8   ,     0     ,     0     ,     0     ,     0     ,
0     ,     0   ,     0     ,     0     ,     0     ,     0     ,
                    0   ,     0     ,     0     ,     0     ,     0
,     0.2   ,     0.8   ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                    0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0.2   ,     0.8   ,     0     ,     0     ,     0
```

```
,     0     ,   0     ,     0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0.1   ,     0.9   ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0     ,     0     ,     0.05
,     0   ,     0     ,     0     ,     0.1   ,     0.8   ,     0
,     0     ,   0.05  ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0.2   ,     0.8
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0.05  ,     0.8
,     0     ,   0     ,     0     ,     0.05  ,     0     ,
                0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0.05  ,     0     ,     0     ,     0
,     0.9   ,   0     ,     0     ,     0     ,     0.05  ,
                0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0.2   ,     0.8   ,     0     ,     0     ,
                0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0.05  ,     0.15  ,     0.8   ,     0     ,
                0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0.2   ,     0.8   ,
                0   ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     1),
nrow=16, ncol=16, byrow=TRUE)
right<- matrix(c(0.2 ,     0.1   ,     0     ,     0     ,     0.7
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0.1   ,     0.1   ,     0     ,     0     ,
0     ,   0.8   ,     0     ,     0     ,     0     ,     0     ,
0     ,     0   ,     0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0.2   ,     0     ,     0
,     0   ,     0.8   ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0.1   ,     0.9   ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0     ,     0     ,     0.2
,     0.1   ,     0     ,     0     ,     0.7   ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0     ,     0     ,     0
,     0.9   ,     0.1   ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     0     ,
                0   ,     0     ,     0     ,     0     ,     0
```

```
,     0.05   ,     0.1   ,     0     ,     0     ,     0     ,
0.85  ,     0     ,   0     ,     0     ,     0     ,     0     ,
                    0  ,     0     ,     0     ,     0     ,     0
,     0   ,     0.1   ,     0.2   ,     0     ,     0     ,     0
,     0.7   ,   0     ,     0     ,     0     ,     0     ,
                    0  ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0.2   ,     0     ,     0
,     0     ,   0.8   ,     0     ,     0     ,     0     ,
                    0  ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0.1   ,     0
,     0     ,   0     ,     0.9   ,     0     ,     0     ,
                    0  ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0.1
,     0     ,   0     ,     0     ,     0.9   ,     0     ,
                    0  ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0.2   ,   0     ,     0     ,     0     ,     0.8   ,
                    0  ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   1     ,     0     ,     0     ,     0     ,
                    0  ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     1     ,     0     ,     0     ,
                    0  ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     1     ,     0     ,
                    0  ,     0     ,     0     ,     0     ,     0
,     0   ,     0     ,     0     ,     0     ,     0     ,     0
,     0     ,   0     ,     0     ,     0     ,     1),
nrow=16, ncol=16, byrow=TRUE)
```

3. Define a list of transition probability matrices:

```
TPMs <- list(up=up, left=left,
down=down, right=right)
```

4. Define a reward matrix of dimensions: 16 (number of states) x 4 (number of actions):

```
Rewards<- matrix(c(-1, -1, -1, -1,
                   -1, -1, -1, -1,
                   -1, -1, -1, -1,
                   -1, -1, -1, -1,
                   -1, -1, -1, -1,
                   -1, -1, -1, -1,
                   -1, -1, -1, -1,
                   -1, -1, -1, -1,
```

```
                  -1, -1, -1, -1,
                  -1, -1, -1, -1,
                  -1, -1, -1, -1,
                  -1, -1, -1, -1,
                  -1, -1, -1, -1,
                  100, 100, 100, 100,
                  -1, -1, -1, -1),
nrow=16, ncol=4, byrow=TRUE)
```

5. Test whether the defined `TPMs` and `Rewards` satisfy a well-defined MDP. If it returns an empty string, then the MDP is valid:

```
mdp_check(TPMs, Rewards)
```

Performing model-based learning

As the name suggests, the learning is augmented using a predefined model. Here, the model is represented in the form of transition probabilities and the key objective is to determine the optimal policy and value functions using these predefined model attributes (that is, `TPMs`). The policy is defined as a learning mechanism of an agent, traversing across multiple states. In other words, identifying the best action of an agent in a given state, to traverse to a next state, is termed a policy.

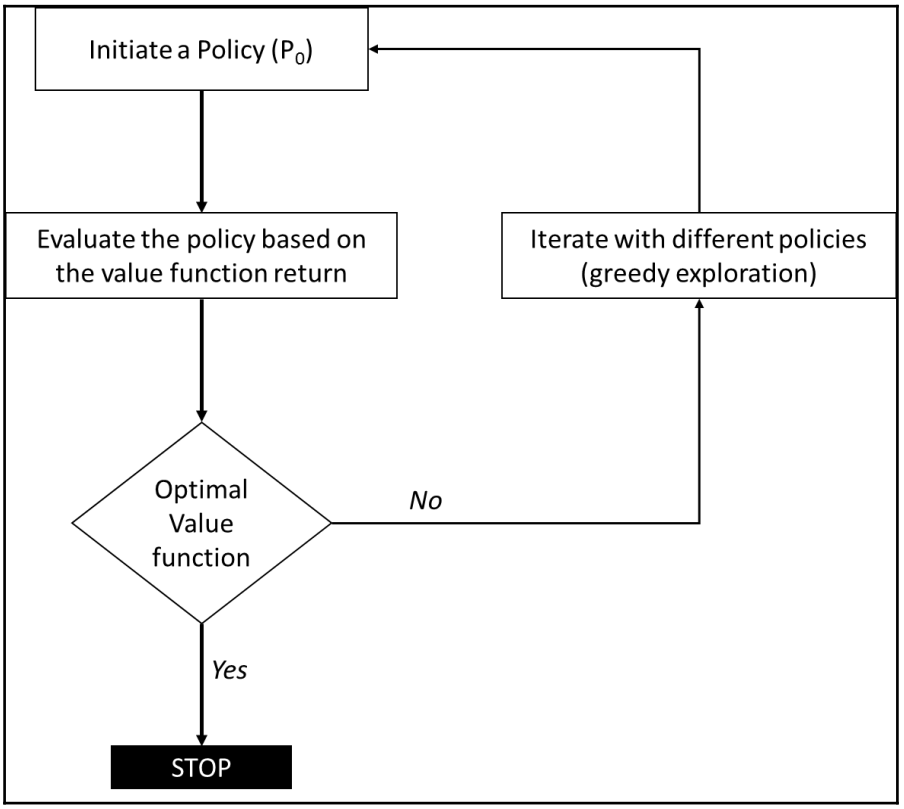
The objective of the policy is to maximize the cumulative reward of transitioning from the start state to the destination state, defined as follows, where *P(s)* is the cumulative policy *P* from a start state *s*, and *R* is the reward of transitioning from state *st* to state s_{t+1} by performing an action at.

$$P(s) = Max[\sum_{t} R^{a_t}_{s_t s_{t+1}}]$$

The value function is of two types: the state-value function and the state-action value function. In the state-value function, for a given policy, it is defined as an expected reward to be in a particular state (including start state), whereas in the state-action value function, for a given policy, it is defined as an expected reward to be in a particular state (including the start state) and undertake a particular action.

Now, a policy is said to be optimal provided it returns the maximum expected cumulative reward, and its corresponding states are termed optimal state-value functions or its corresponding states and actions are termed optimal state-action value functions.

In model-based learning, the following iterative steps are performed in order to obtain an optimum policy, as shown in the following figure:



Iterative steps to find an optimum policy

In this section, we shall evaluate the policy using the state-value function. In each iteration, the policies are dynamically evaluated using the Bellman equation, as follows, where V_i denotes the value at iteration i, P denotes an arbitrary policy of a given state s and action a, T denotes the transition probability from state s to state s' due to an action a, R denotes the reward at state s' while traversing from the state s post an action a, and Υ denotes a discount factor in the range of (0,1). The discount factor ensures higher importance to starting learning steps than later.

$$V_{i+1}(s) = \sum_{a} P(s,a) \sum_{s'} T_{ss'}^{a}[R_{ss'}^{a} + \Upsilon V_i(s')]$$

How to do it...

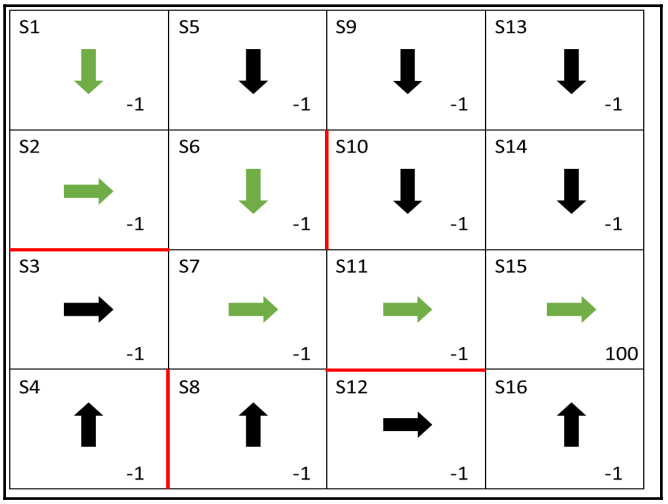
This section shows you how to set up model-based RL:

1. Run the policy iteration using the state-action value function with the discount factor $\Upsilon = 0.9$:

```
mdp_policy<- mdp_policy_iteration(P=TPMs, R=Rewards, discount=0.9)
```

2. Get the best (optimum) policy P* as shown in the following figure. The arrows marked in green show the direction of traversing *S1* to *S15*:

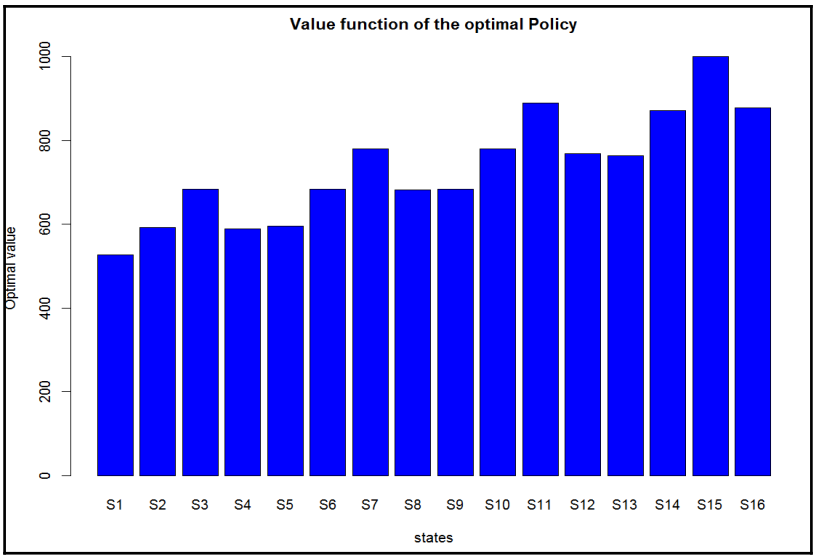
```
mdp_policy$policy
names(TPMs)[mdp_policy$policy]
```



Optimum policy using model-based iteration with an optimum path from *S1* to *S15*

3. Get the optimum value function V* for each state and plot them as shown in the following figure:

```
mdp_policy$V
names(mdp_policy$V) <- paste0("S",1:16)
barplot(mdp_policy$V,col="blue",xlab="states",ylab="Optimal
value",main="Value function of the optimal Policy",width=0.5)
```



Value functions of the optimal policy

Performing model-free learning

Unlike model-based learning, where dynamics of transitions are explicitly provided (as transition probabilities from one state to another state), in model-free learning, the transitions are supposed to be deduced and learned directly from the interaction between states (using actions) rather explicitly provided. Widely used frameworks of mode-free learning are **Monte Carlo** methods and the **Q-learning** technique. The former is simple to implement but convergence takes time, whereas the latter is complex to implement but is efficient in convergence due to off-policy learning.

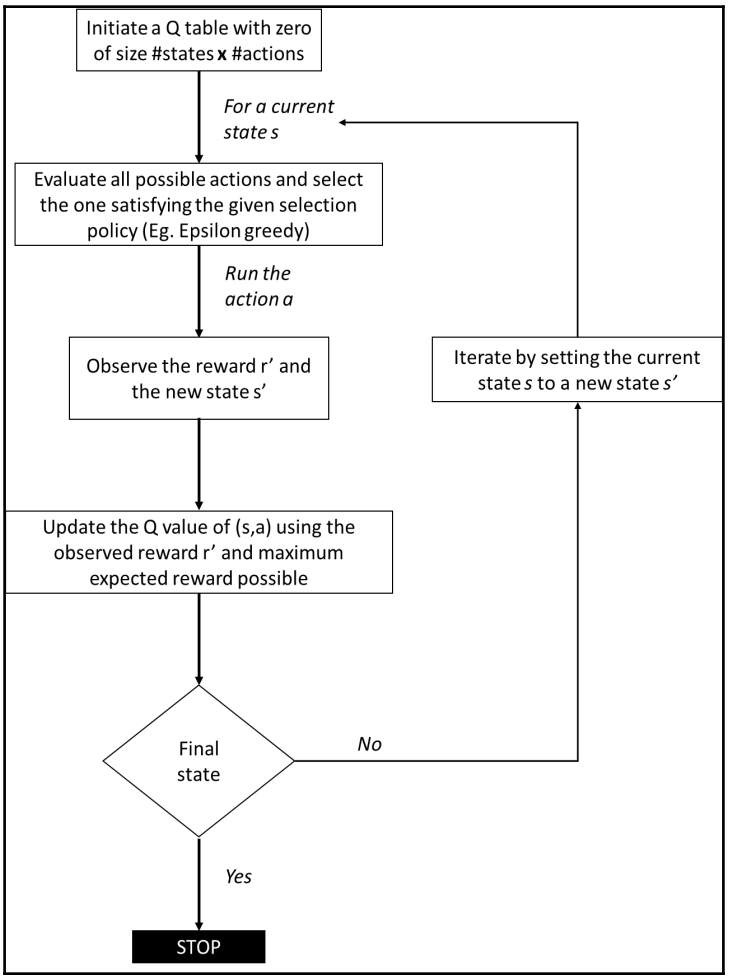
Getting ready

In this section, we will implement the Q-learning algorithm in R. The simultaneous exploration of the surrounding environment and exploitation of existing knowledge is termed off-policy convergence. For example, an agent in a particular state first explores all the possible actions of transitioning into next states and observes the corresponding rewards, and then exploits current knowledge to update the existing state-action value using the action generating the maximum possible reward.

The Q learning returns a 2D Q-table of the size of the number of states x the number of actions. The values in the Q-table are updated based on the following formula, where Q denotes the value of state s and action a, r' denotes the reward of the next state for a selected action a, Υ denotes the discount factor, and α denotes the learning rate:

$$Q_{new}(s,a) = Q_{old}(s,a) + \alpha\left[r' + \Upsilon * \max_{a'} Q_{expected\ optimal\ value}(s',a') - Q_{old}(s,a)\right]$$

The framework for Q-learning is shown in the following figure:



Framework of Q-learning

How to do it...

The section provide steps for how to set up Q-learning:

1. Define 16 states:

```
states <- c("s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9",
"s10", "s11", "s12", "s13", "s14", "s15", "s16")
```

2. Define four actions:

```
actions<- c("up", "left", "down", "right")
```

3. Define the `transitionStateAction` function, which can simulate the transitions from one state *s* to another state *s'* using an action *a*. The function takes in the current state *s* and selected action *a*, and it returns the next state *s'* and corresponding reward *r'*. In case of constrained action, the next state returned is the current state *s* and the existing reward *r*:

```
transitionStateAction<- function(state, action) {
  # The default state is the existing state in case of constrained
action
next_state<- state
if (state == "s1"&& action == "down") next_state<- "s2"
if (state == "s1"&& action == "right") next_state<- "s5"
if (state == "s2"&& action == "up") next_state<- "s1"
if (state == "s2"&& action == "right") next_state<- "s6"
if (state == "s3"&& action == "right") next_state<- "s7"
if (state == "s3"&& action == "down") next_state<- "s4"
if (state == "s4"&& action == "up") next_state<- "s3"
if (state == "s5"&& action == "right") next_state<- "s9"
if (state == "s5"&& action == "down") next_state<- "s6"
if (state == "s5"&& action == "left") next_state<- "s1"
if (state == "s6"&& action == "up") next_state<- "s5"
if (state == "s6"&& action == "down") next_state<- "s7"
if (state == "s6"&& action == "left") next_state<- "s2"
if (state == "s7"&& action == "up") next_state<- "s6"
if (state == "s7"&& action == "right") next_state<- "s11"
if (state == "s7"&& action == "down") next_state<- "s8"
if (state == "s7"&& action == "left") next_state<- "s3"
if (state == "s8"&& action == "up") next_state<- "s7"
if (state == "s8"&& action == "right") next_state<- "s12"
if (state == "s9"&& action == "right") next_state<- "s13"
if (state == "s9"&& action == "down") next_state<- "s10"
if (state == "s9"&& action == "left") next_state<- "s5"
if (state == "s10"&& action == "up") next_state<- "s9"
if (state == "s10"&& action == "right") next_state<- "s14"
```

```
if (state == "s10"&& action == "down") next_state<- "s11"
if (state == "s11"&& action == "up") next_state<- "s10"
if (state == "s11"&& action == "right") next_state<- "s15"
if (state == "s11"&& action == "left") next_state<- "s7"
if (state == "s12"&& action == "right") next_state<- "s16"
if (state == "s12"&& action == "left") next_state<- "s8"
if (state == "s13"&& action == "down") next_state<- "s14"
if (state == "s13"&& action == "left") next_state<- "s9"
if (state == "s14"&& action == "up") next_state<- "s13"
if (state == "s14"&& action == "down") next_state<- "s15"
if (state == "s14"&& action == "left") next_state<- "s10"
if (state == "s15"&& action == "up") next_state<- "s14"
if (state == "s15"&& action == "down") next_state<- "s16"
if (state == "s15"&& action == "left") next_state<- "s11"
if (state == "s16"&& action == "up") next_state<- "s15"
if (state == "s16"&& action == "left") next_state<- "s12"
  # Calculate reward
if (next_state == "s15") {
reward<- 100
  } else {
reward<- -1
  }

return(list(state=next_state, reward=reward))
}
```

4. Define a function to perform Q-learning using `n` iterations:

```
Qlearning<- function(n, initState, termState,
epsilon, learning_rate) {
  # Initialize a Q-matrix of size #states x #actions with zeroes
Q_mat<- matrix(0, nrow=length(states), ncol=length(actions),
dimnames=list(states, actions))
  # Run n iterations of Q-learning
for (i in 1:n) {
Q_mat<- updateIteration(initState, termState, epsilon,
learning_rate, Q_mat)
  }
return(Q_mat)
}
   updateIteration<- function(initState, termState, epsilon,
learning_rate, Q_mat) {
state<- initState # set cursor to initial state
while (state != termState) {
    # Select the next action greedily or randomnly
if (runif(1) >= epsilon) {
action<- sample(actions, 1) # Select randomnly
    } else {
```

```
action<- which.max(Q_mat[state, ]) # Select best action
    }
    # Extract the next state and its reward
response<- transitionStateAction(state, action)
    # Update the corresponding value in Q-matrix (learning)
Q_mat[state, action] <- Q_mat[state, action] + learning_rate *
      (response$reward + max(Q_mat[response$state, ]) -
Q_mat[state, action])
state<- response$state # update with next state
  }
return(Q_mat)
}
```

5. Set learning parameters such as `epsilon` and `learning_rate`:

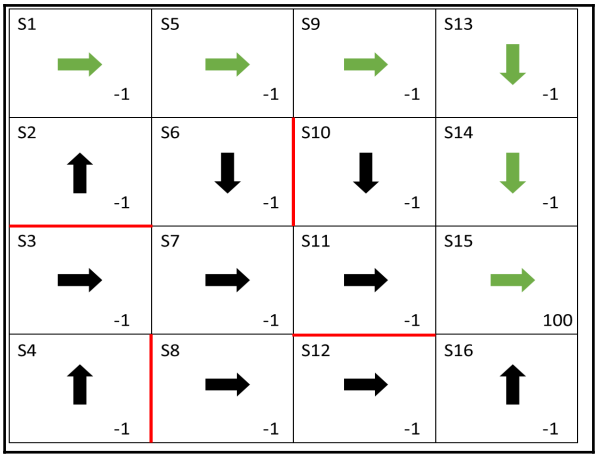
```
epsilon<- 0.1
learning_rate<- 0.9
```

6. Get the Q-table after 500k iterations:

```
Q_mat<- Qlearning(500, "s1", "s15", epsilon, learning_rate)
Q_mat
```

7. Get the best (optimum) policy P*, as shown in the following figure. The arrows marked in green shows the direction of traversing *S1* to *S15:*

```
actions[max.col(Q_mat)]
```



Optimum policy using model-free iteration with an optimum path from *S1* to *S15*

8
Application of Deep Learning in Text Mining

The current chapter we will cover the following topics:

- Performing preprocessing of textual data and extraction of sentiments
- Analyzing documents using tf-idf
- Performing sentiment prediction using LSTM network
- Application using text2vec examples

Performing preprocessing of textual data and extraction of sentiments

In this section, we will use Jane Austen's bestselling novel Pride and Prejudice, published in 1813, for our textual data preprocessing analysis. In R, we will use the `tidytext` package by Hadley Wickham to perform tokenization, stop word removal, sentiment extraction using predefined sentiment lexicons, **term frequency - inverse document frequency** (**tf-idf**) matrix creation, and to understand pairwise correlations among *n*-grams.

In this section, instead of storing text as a string or a corpus or a **document term matrix** (**DTM**), we process them into a tabular format of one token per row.

How to do it...

Here is how we go about preprocessing:

1. Load the required packages:

```
load_packages=c("janeaustenr","tidytext","dplyr","stringr","ggplot2
","wordcloud","reshape2","igraph","ggraph","widyr","tidyr")
lapply(load_packages, require, character.only = TRUE)
```

2. Load the `Pride and Prejudice` dataset. The `line_num` attribute is analogous to the line number printed in the book:

```
Pride_Prejudice <- data.frame("text" = prideprejudice,
                              "book" = "Pride and Prejudice",
                              "line_num" =
1:length(prideprejudice),
                              stringsAsFactors=F)
```

3. Now, perform tokenization to restructure the one-string-per-row format to a one-token-per-row format. Here, the token can refer to a single word, a group of characters, co-occurring words (*n*-grams), sentences, paragraphs, and so on. Currently, we will tokenize sentence into singular words:

```
Pride_Prejudice <- Pride_Prejudice %>% unnest_tokens(word,text)
```

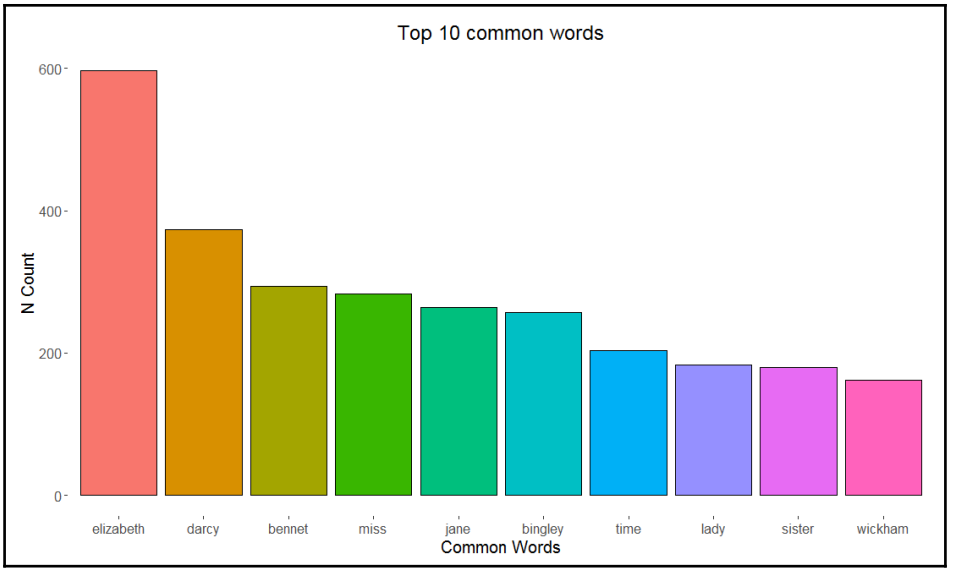
4. Then, remove the commonly occurring words such as *the*, *and*, *for*, and so on using the `stop words` removal corpus:

```
data(stop_words)
Pride_Prejudice <- Pride_Prejudice %>% anti_join(stop_words,
by="word")
```

5. Extract the most common textual words used:

```
most.common <- Pride_Prejudice %>% dplyr::count(word, sort = TRUE)
```

6. Visualize the top 10 common occurring words, as shown in the following figure:



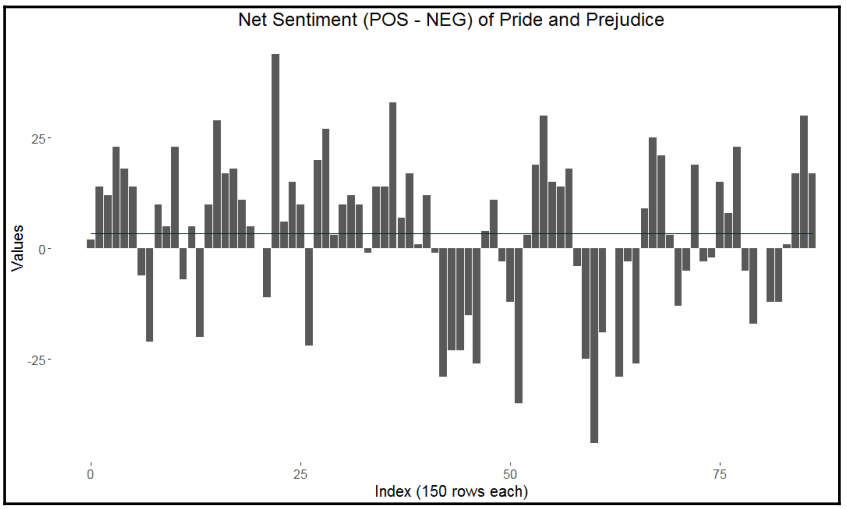
Top 10 common words

```
most.common$word  <- factor(most.common$word , levels =
most.common$word)
ggplot(data=most.common[1:10,], aes(x=word, y=n, fill=word)) +
  geom_bar(colour="black", stat="identity")+
  xlab("Common Words") + ylab("N Count")+
  ggtitle("Top 10 common words")+
  guides(fill=FALSE)+
  theme(plot.title = element_text(hjust = 0.5))+
  theme(text = element_text(size = 10))+
  theme(panel.background = element_blank(), panel.grid.major =
element_blank(),panel.grid.minor = element_blank())
```

7. Then, extract sentiments at a higher level (that is positive or negative) using the `bing` lexicon.

```
Pride_Prejudice_POS_NEG_sentiment <- Pride_Prejudice %>%
inner_join(get_sentiments("bing"), by="word") %>%
dplyr::count(book, index = line_num %/% 150, sentiment) %>%
spread(sentiment, n, fill = 0) %>% mutate(net_sentiment = positive
- negative)
```

8. Visualize the sentiments across small sections (150 words) of text, as shown in the following figure:



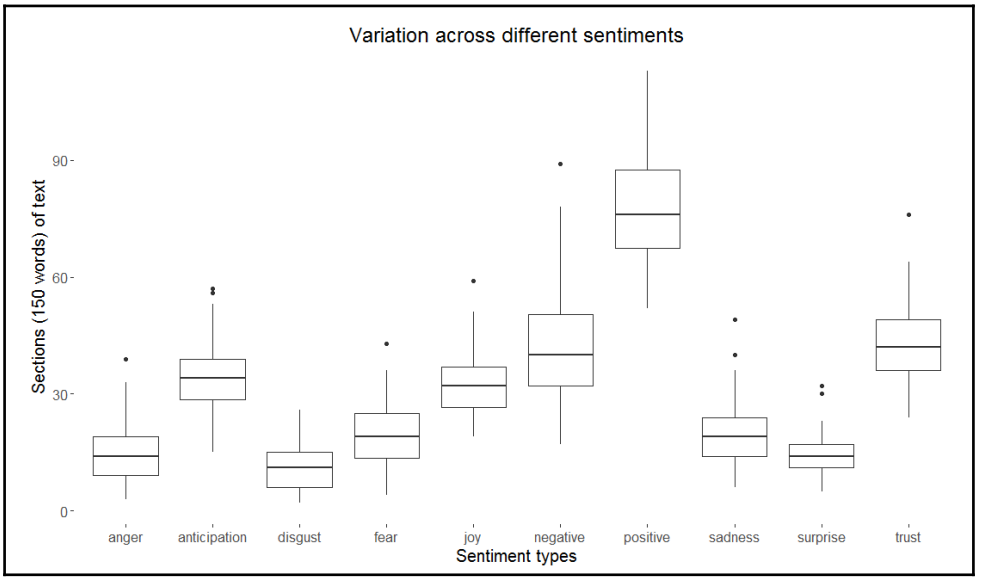
Distribution of the number of positive and negative words across sentences of 150 words each

```
ggplot(Pride_Prejudice_POS_NEG_sentiment, aes(index,
net_sentiment))+
  geom_col(show.legend = FALSE) +
  geom_line(aes(y=mean(net_sentiment)),color="blue")+
  xlab("Section (150 words each)") + ylab("Values")+
  ggtitle("Net Sentiment (POS - NEG) of Pride and Prejudice")+
  theme(plot.title = element_text(hjust = 0.5))+
  theme(text = element_text(size = 10))+
  theme(panel.background = element_blank(), panel.grid.major =
element_blank(),panel.grid.minor = element_blank())
```

9. Now extract sentiments at a granular level (namely positive, negative, anger, disgust, surprise, trust, and so on.) using the `nrc` lexicon:

```
Pride_Prejudice_GRAN_sentiment <- Pride_Prejudice %>%
inner_join(get_sentiments("nrc"), by="word") %>% dplyr::count(book,
index = line_num %/% 150, sentiment) %>% spread(sentiment, n, fill
= 0)
```

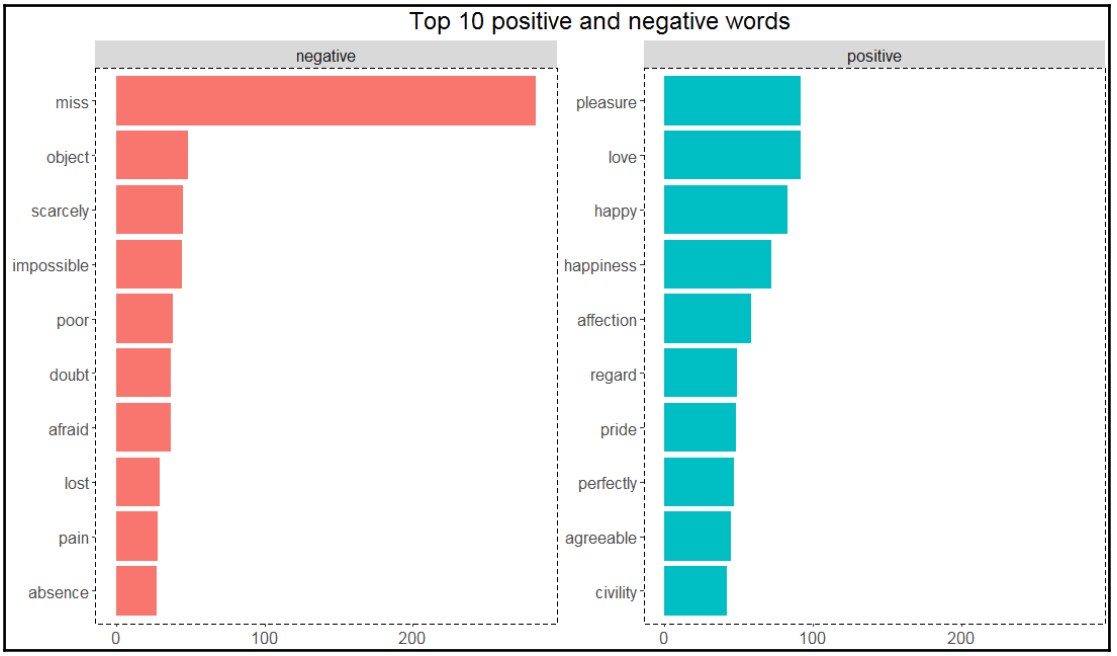
10. Visualize the variation across different sentiments defined, as shown in the following figure:



Variation across different types of sentiments

```
ggplot(stack(Pride_Prejudice_GRAN_sentiment[,3:12]), aes(x = ind, y
= values)) +
  geom_boxplot()+
  xlab("Sentiment types") + ylab("Sections (150 words) of text")+
  ggtitle("Variation across different sentiments")+
  theme(plot.title = element_text(hjust = 0.5))+
  theme(text = element_text(size = 15))+
  theme(panel.background = element_blank(), panel.grid.major =
element_blank(),panel.grid.minor = element_blank())
```

11. Extract the most occurring positive and negative words based on the `bing` lexicon, and visualize them as shown in the following figure:



Top 10 positive and negative words in the novel Pride and Prejudice

```
POS_NEG_word_counts <- Pride_Prejudice %>%
inner_join(get_sentiments("bing"), by="word") %>%
dplyr::count(word, sentiment, sort = TRUE) %>% ungroup()
POS_NEG_word_counts %>% group_by(sentiment) %>% top_n(10) %>%
ungroup() %>% mutate(word = reorder(word, n)) %>% ggplot(aes(word,
n, fill = sentiment)) + geom_col(show.legend = FALSE) +
facet_wrap(~sentiment, scales = "free_y") + ggtitle("Top 10
positive and negative words")+ coord_flip() + theme(plot.title =
element_text(hjust = 0.5))+ theme(text = element_text(size = 15))+
labs(y = NULL, x = NULL)+ theme(panel.background =
element_blank(),panel.border = element_rect(linetype = "dashed",
fill = NA))
```

12. Generate a sentiment word cloud as shown in the following figure:

Word cloud of positive and negative words

```
Prejudice %>%
inner_join(get_sentiments("bing"), by = "word") %>%
dplyr::count(word, sentiment, sort = TRUE) %>% acast(word ~
sentiment, value.var = "n", fill = 0) %>% comparison.cloud(colors =
c("red", "green"), max.words = 100,title.size=2, use.r.layout=TRUE,
random.order=TRUE, scale=c(6,0.5)
```

13. Now analyze sentiments across the chapters of the book:

    1. Extract the chapters, and perform tokenization:

```
austen_books_df <-
as.data.frame(austen_books(),stringsAsFactors=F)
austen_books_df$book <- as.character(austen_books_df$book)
Pride_Prejudice_chapters <- austen_books_df %>%
group_by(book) %>% filter(book == "Pride & Prejudice") %>%
mutate(chapter = cumsum(str_detect(text, regex("^chapter
[\\divxlc]", ignore_case = TRUE)))) %>% ungroup() %>%
unnest_tokens(word, text)
```

    2. Extract the set `positive` and `negative` words from the `bing` lexicon:

```
bingNEG <- get_sentiments("bing") %>%
  filter(sentiment == "negative")
bingPOS <- get_sentiments("bing") %>%
  filter(sentiment == "positive")
```

3. Get the count of words for each chapter:

```
wordcounts <- Pride_Prejudice_chapters %>%
group_by(book, chapter) %>%
dplyr::summarize(words = n())
```

4. Extract the ratio of positive and negative words:

```
POS_NEG_chapter_distribution <- merge (
Pride_Prejudice_chapters %>%
semi_join(bingNEG, by="word") %>%
group_by(book, chapter) %>%
dplyr::summarize(neg_words = n()) %>%
left_join(wordcounts, by = c("book", "chapter")) %>%
mutate(neg_ratio = round(neg_words*100/words,2)) %>%
filter(chapter != 0) %>%
ungroup(),
Pride_Prejudice_chapters %>%
semi_join(bingPOS, by="word") %>%
group_by(book, chapter) %>%
dplyr::summarize(pos_words = n()) %>%
left_join(wordcounts, by = c("book", "chapter")) %>%
mutate(pos_ratio = round(pos_words*100/words,2)) %>%
filter(chapter != 0) %>%
ungroup() )
```

5. Generate a sentiment flag for each chapter based on the proportion of positive and negative words:

```
POS_NEG_chapter_distribution$sentiment_flag <-
ifelse(POS_NEG_chapter_distribution$neg_ratio >
POS_NEG_chapter_distribution$pos_ratio,"NEG","POS")
table(POS_NEG_chapter_distribution$sentiment_flag)
```

How it works...

As mentioned earlier, we have used Jane Austen's famous novel *Pride and Prejudice* in this section, detailing the steps involved in tidying the data, and extracting sentiments using (publicly) available lexicons.

Steps 1 and 2 show the loading of the required `cran` packages and the required text. Steps 3 and 4 perform unigram tokenization and stop word removal. Steps 5 and 6 extract and visualize the top 10 most occurring words across all the 62 chapters. Steps 7 to 12 demonstrate high and granular-level sentiments using two widely used lexicons `bing` and `nrc`.

Both the lexicons contains a list of widely used English words that are tagged to sentiments. In `bing`, each word is tagged to one of the high level binary sentiments (positive or negative), and in `nrc`, each word is tagged to one of the granular-level multiple sentiments (positive, negative, anger, anticipation, joy, fear, disgust, trust, sadness, and surprise).

Each 150-word-long sentence is tagged to a sentiment, and the same has been shown in the figure showing the *Distribution of number of positive and negative words across sentences of 150 words each*. In step 13, chapter-wise sentiment tagging is performed using maximum occurrence of positive or negative words from the `bing` lexicon. Out of 62 chapters, 52 have more occurrences of positive lexicons, and 10 have more occurrences of negative lexicons.

Analyzing documents using tf-idf

In this section, we will learn how to analyze documents quantitatively. A simple way is to look at the distribution of unigram words across the document and their frequency of occurrence, also termed as **term frequency** (**tf**). The words with higher frequency of occurrence generally tend to dominate the document.

However, one would disagree in case of generally occurring words such as the, is, of, and so on. Hence, these are removed by stop word dictionaries. Apart from these stop words, there might be some specific words that are more frequent with less relevance. Such kinds of words are penalized using their **inverse document frequency** (**idf**) values. Here, the words with higher frequency of occurrence are penalized.

The statistic tf-idf combines these two quantities (by multiplication) and provides a measure of importance or relevance of each word for a given document across multiple documents (or a corpus).

In this section, we will generate a tf-idf matrix across chapters of the book *Pride and Prejudice*.

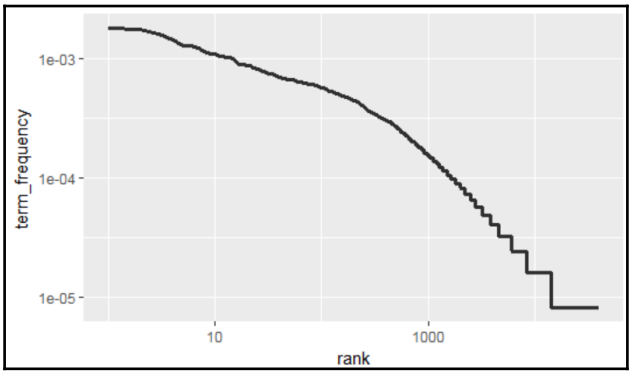
How to do it...

Here is how we go about analyzing documents using tf-idf:

1. Extract the text of all 62 chapters in the book `Pride and Prejudice`. Then, return chapter-wise occurrence of each word. The total words in the book are approx 1.22M.

```
Pride_Prejudice_chapters <- austen_books_df %>%
group_by(book) %>%
filter(book == "Pride & Prejudice") %>%
mutate(linenumber = row_number(),
chapter = cumsum(str_detect(text, regex("^chapter [\\divxlc]",
                                        ignore_case = TRUE)))) %>%
ungroup() %>%
unnest_tokens(word, text) %>%
count(book, chapter, word, sort = TRUE) %>%
ungroup()
```

2. Calculate the rank of words such that the most frequently occurring words have lower ranks. Also, visualize the term frequency by rank, as shown in the following figure:



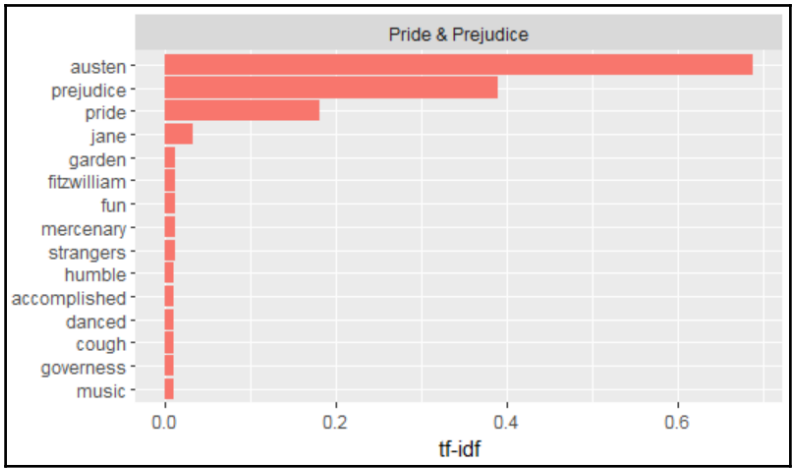
This figure shows lower ranks for words with higher term-frequency (ratio) value

```
freq_vs_rank <- Pride_Prejudice_chapters %>%
mutate(rank = row_number(),
       term_frequency = n/totalwords)
freq_vs_rank %>%
  ggplot(aes(rank, term_frequency)) +
  geom_line(size = 1.1, alpha = 0.8, show.legend = FALSE) +
  scale_x_log10() +
  scale_y_log10()
```

3. Calculate the `tf-idf` value for each word using the `bind_tf-idf` function:

```
Pride_Prejudice_chapters <- Pride_Prejudice_chapters %>%
bind_tf_idf(word, chapter, n)
```

4. Extract and visualize the top 15 words with higher values of tf-idf, as shown in the following figure:



tf-idf values of top 15 words

```
Pride_Prejudice_chapters %>%
  select(-totalwords) %>%
  arrange(desc(tf_idf))

Pride_Prejudice_chapters %>%
  arrange(desc(tf_idf)) %>%
  mutate(word = factor(word, levels = rev(unique(word)))) %>%
  group_by(book) %>%
  top_n(15) %>%
  ungroup %>%
  ggplot(aes(word, tf_idf, fill = book)) +
  geom_col(show.legend = FALSE) +
  labs(x = NULL, y = "tf-idf") +
  facet_wrap(~book, ncol = 2, scales = "free") +
  coord_flip()
```

How it works...

As mentioned earlier, one can observe that the tf-idf scores of very common words such as *the* are close to zero and those of fewer occurrence words such as the proper noun *Austen* is close to one.

Performing sentiment prediction using LSTM network

In this section, we will use LSTM networks to perform sentiment analysis. Along with the word itself, the LSTM network also accounts for the sequence using recurrent connections, which makes it more accurate than a traditional feed-forward neural network.

Here, we shall use the `movie reviews` dataset `text2vec` from the `cran` package. This dataset consists of 5,000 IMDb movie reviews, where each review is tagged with a binary sentiment flag (positive or negative).

How to do it...

Here is how you can proceed with sentiment prediction using LSTM:

1. Load the required packages and movie reviews dataset:

```
load_packages=c("text2vec","tidytext","tensorflow")
lapply(load_packages, require, character.only = TRUE)
data("movie_review")
```

2. Extract the movie reviews and labels as a dataframe and matrix respectively. In movie reviews, add an additional attribute `"Sno"` denoting the review number. In the labels matrix, add an additional attribute related to `negative flag`.

```
reviews <- data.frame("Sno" = 1:nrow(movie_review),
                          "text"=movie_review$review,
                          stringsAsFactors=F)

labels <- as.matrix(data.frame("Positive_flag" =
movie_review$sentiment,"negative_flag" = (1
                    movie_review$sentiment)))
```

3. Extract all the unique words across the reviews, and get their count of occurrences (*n*). Also, tag each word with a unique integer (`orderNo`). Thus, each word is encoded using a unique integer, which shall be later used in the LSTM network.

```
reviews_sortedWords <- reviews %>% unnest_tokens(word,text) %>%
dplyr::count(word, sort = TRUE)
reviews_sortedWords$orderNo <- 1:nrow(reviews_sortedWords)
reviews_sortedWords <- as.data.frame(reviews_sortedWords)
```

4. Now, assign the tagged words back to the reviews based on their occurrences:

```
reviews_words <- reviews %>% unnest_tokens(word,text)
reviews_words <-
plyr::join(reviews_words,reviews_sortedWords,by="word")
```

5. Using the outcome of step 4, create a list of reviews with each review transformed into a set of encoded numbers representing those words:

```
reviews_words_sno <- list()
for(i in 1:length(reviews$text))
{
  reviews_words_sno[[i]] <- c(subset(reviews_words,Sno==i,orderNo))
}
```

6. In order to facilitate equal-length sequences to the LSTM network, let's restrict the review length to 150 words. In other words, reviews longer than 150 words will be truncated to the first 150, whereas shorter reviews will be made 150 words long by prefixing with the required number of zeroes. Thus, we now add in a new word **0**.

```
reviews_words_sno <- lapply(reviews_words_sno,function(x)
{
  x <- x$orderNo
  if(length(x)>150)
  {
    return (x[1:150])
  }
  else
  {
  return(c(rep(0,150-length(x)),x))
  }
})
```

7. Now split the 5,000 reviews into training and testing reviews using a 70:30 split ratio. Also, bind the list of train and test reviews row wise into a matrix format, with rows representing reviews and columns representing the position of a word:

```
train_samples <-
caret::createDataPartition(c(1:length(labels[1,1])),p =
0.7)$Resample1

train_reviews <- reviews_words_sno[train_samples]
test_reviews <- reviews_words_sno[-train_samples]

train_reviews <- do.call(rbind,train_reviews)
test_reviews <- do.call(rbind,test_reviews)
```

8. Similarly, also split the labels into train and test accordingly:

```
train_labels <- as.matrix(labels[train_samples,])
test_labels <- as.matrix(labels[-train_samples,])
```

9. Reset the graph, and start an interactive TensorFlow session:

```
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

10. Define model parameters such as size of input pixels (`n_input`), step size (`step_size`), number of hidden layers (`n.hidden`), and number of outcome classes (`n.classes`):

```
n_input<-15
step_size<-10
n.hidden<-2
n.class<-2
```

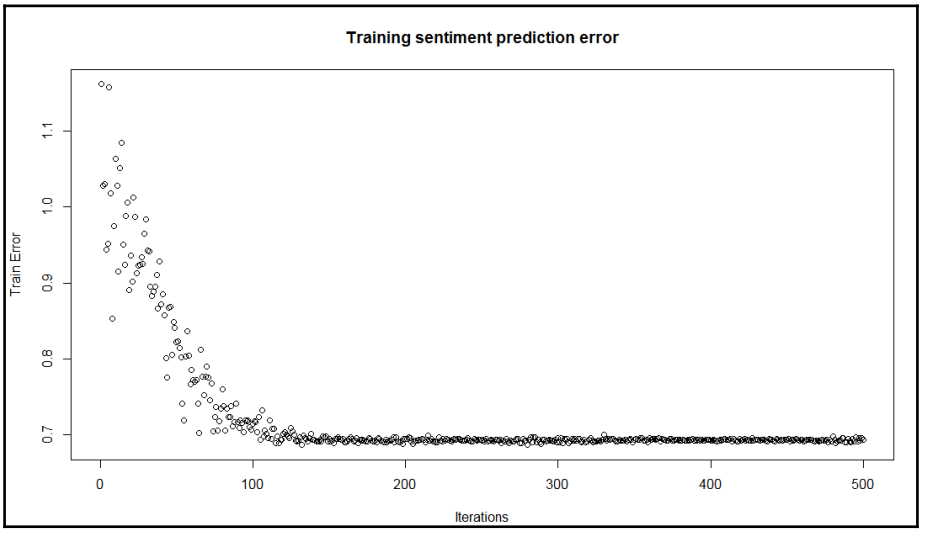
11. Define training parameters such as learning rate (`lr`), number of inputs per batch run (`batch`), and number of iterations (`iteration`):

```
lr<-0.01
batch<-200
iteration = 500
```

12. Based on the RNN and LSTM functions defined in `Chapter 6`, *Recurrent Neural Networks,* from the section *Run the optimization post initializing a session using global variables initializer*.

```
sess$run(tf$global_variables_initializer())
train_error <- c()
for(i in 1:iteration){
  spls <- sample(1:dim(train_reviews)[1],batch)
  sample_data<-train_reviews[spls,]
  sample_y<-train_labels[spls,]
  # Reshape sample into 15 sequence with each of 10 elements
  sample_data=tf$reshape(sample_data, shape(batch, step_size,
n_input))
  out<-optimizer$run(feed_dict = dict(x=sample_data$eval(session =
sess), y=sample_y))
  if (i %% 1 == 0){
    cat("iteration - ", i, "Training Loss - ",  cost$eval(feed_dict
= dict(x=sample_data$eval(), y=sample_y)), "\n")
  }
  train_error <-  c(train_error,cost$eval(feed_dict =
dict(x=sample_data$eval(), y=sample_y)))
}
```

13. Plot the reduction in training errors across iterations as shown in the following figure:



Distribution of sentiment prediction error of training dataset

```
plot(train_error, main="Training sentiment prediction error",
xlab="Iterations", ylab = "Train Error")
```

14. Get the error of test data:

```
test_data=tf$reshape(test_reviews, shape(-1, step_size, n_input))
cost$eval(feed_dict=dict(x= test_data$eval(), y=test_labels))
```

How it works...

In steps 1 to 8, the movie reviews dataset is loaded, processed, and transformed into a set of train and test matrices, which can be directly used to train an LSTM network. Steps 9 to 14 are used to run LSTM using TensorFlow, as described in `Chapter 6`, *Recurrent Neural Networks*. The figure *Distribution of sentiment prediction error of training dataset* shows the decline in training errors across 500 iterations.

Application using text2vec examples

In this section, we will analyze the performance of logistic regression on various examples of `text2vec`.

How to do it...

Here is how we apply `text2vec`:

1. Load the required packages and dataset:

```
library(text2vec)
library(glmnet)
data("movie_review")
```

2. Function to perform Lasso logistic regression, and return the train and test `AUC` values:

```
logistic_model <- function(Xtrain,Ytrain,Xtest,Ytest)
{
  classifier <- cv.glmnet(x=Xtrain, y=Ytrain,
  family="binomial", alpha=1, type.measure = "auc",
  nfolds = 5, maxit = 1000)
  plot(classifier)
  vocab_test_pred <- predict(classifier, Xtest, type = "response")
```

```
  return(cat("Train AUC : ", round(max(classifier$cvm), 4),
  "Test AUC : ",glmnet:::auc(Ytest, vocab_test_pred),"\n"))
}
```

3. Split the movies review data into train and test in an 80:20 ratio:

```
train_samples <-
caret::createDataPartition(c(1:length(labels[1,1])),p =
0.8)$Resample1
train_movie <- movie_review[train_samples,]
test_movie <- movie_review[-train_samples,]
```

4. Generate a DTM of all vocabulary words (without any stop word removal), and asses its performance using Lasso logistic regression:

```
train_tokens <- train_movie$review %>% tolower %>% word_tokenizer
test_tokens <- test_movie$review %>% tolower %>% word_tokenizer

vocab_train <-
create_vocabulary(itoken(train_tokens,ids=train$id,progressbar =
FALSE))

# Create train and test DTMs
vocab_train_dtm <- create_dtm(it =
itoken(train_tokens,ids=train$id,progressbar = FALSE),
                              vectorizer =
vocab_vectorizer(vocab_train))
vocab_test_dtm <- create_dtm(it =
itoken(test_tokens,ids=test$id,progressbar = FALSE),
                              vectorizer =
vocab_vectorizer(vocab_train))

dim(vocab_train_dtm)
dim(vocab_test_dtm)

# Run LASSO (L1 norm) Logistic Regression
logistic_model(Xtrain = vocab_train_dtm,
               Ytrain = train_movie$sentiment,
               Xtest = vocab_test_dtm,
               Ytest = test_movie$sentiment)
```

5. Perform pruning using a list of stop words, and then assess the performance using Lasso logistic regression:

```
data("stop_words")
vocab_train_prune <-
create_vocabulary(itoken(train_tokens,ids=train$id,progressbar =
FALSE),
```

```
                                              stopwords = stop_words$word)

vocab_train_prune <-
prune_vocabulary(vocab_train_prune,term_count_min = 15,
                                          doc_proportion_min = 0.0005,
                                          doc_proportion_max = 0.5)

vocab_train_prune_dtm <- create_dtm(it =
itoken(train_tokens,ids=train$id,progressbar = FALSE),
                                 vectorizer =
vocab_vectorizer(vocab_train_prune))
vocab_test_prune_dtm <- create_dtm(it =
itoken(test_tokens,ids=test$id,progressbar = FALSE),
                                vectorizer =
vocab_vectorizer(vocab_train_prune))

logistic_model(Xtrain = vocab_train_prune_dtm,
               Ytrain = train_movie$sentiment,
               Xtest = vocab_test_prune_dtm,
               Ytest = test_movie$sentiment)
```

6. Generate a DTM using *n*-grams (uni and bigram words), and then assess the performance using Lasso logistic regression:

```
vocab_train_ngrams <-
create_vocabulary(itoken(train_tokens,ids=train$id,progressbar =
FALSE),
                                           ngram = c(1L, 2L))

vocab_train_ngrams <-
prune_vocabulary(vocab_train_ngrams,term_count_min = 10,
                                           doc_proportion_min = 0.0005,
                                           doc_proportion_max = 0.5)

vocab_train_ngrams_dtm <- create_dtm(it =
itoken(train_tokens,ids=train$id,progressbar = FALSE),
                                      vectorizer =
vocab_vectorizer(vocab_train_ngrams))
vocab_test_ngrams_dtm <- create_dtm(it =
itoken(test_tokens,ids=test$id,progressbar = FALSE),
                                    vectorizer =
vocab_vectorizer(vocab_train_ngrams))


logistic_model(Xtrain = vocab_train_ngrams_dtm,
               Ytrain = train_movie$sentiment,
               Xtest = vocab_test_ngrams_dtm,
               Ytest = test_movie$sentiment)
```

7. Perform feature hashing, and then asses the performance using Lasso logistic regression:

```
vocab_train_hashing_dtm <- create_dtm(it =
itoken(train_tokens,ids=train$id,progressbar = FALSE),
                                      vectorizer =
hash_vectorizer(hash_size = 2^14, ngram = c(1L, 2L)))
vocab_test_hashing_dtm <- create_dtm(it =
itoken(test_tokens,ids=test$id,progressbar = FALSE),
                                     vectorizer =
hash_vectorizer(hash_size = 2^14, ngram = c(1L, 2L)))

logistic_model(Xtrain = vocab_train_hashing_dtm,
               Ytrain = train_movie$sentiment,
               Xtest = vocab_test_hashing_dtm,
               Ytest = test_movie$sentiment)
```

8. Using tf-idf transformation on full vocabulary DTM, assess the performance using Lasso logistic regression:

```
vocab_train_tfidf <- fit_transform(vocab_train_dtm, TfIdf$new())
vocab_test_tfidf <- fit_transform(vocab_test_dtm, TfIdf$new())

logistic_model(Xtrain = vocab_train_tfidf,
               Ytrain = train_movie$sentiment,
               Xtest = vocab_test_tfidf,
               Ytest = test_movie$sentiment)
```

How it works...

Steps 1 to 3 loads necessary packages, datasets, and functions required to assess different examples of `text2vec`. Logistic regression is implemented using the `glmnet` package with L1 penalty (Lasso regularization). In step 4, a DTM is created using all the vocabulary words present in the train movie reviews, and the test `auc` value is 0.918. In step 5, the train and test DTMs are pruned using stop words and frequency of occurrence.

The test `auc` value is observed as 0.916, not much decrease compared to using all the vocabulary words. In step 6, along with single words (or uni-grams), bi-grams are also added to the vocabulary. The test `auc` value increases to 0.928. Feature hashing is then performed in step 7, and the test `auc` value is 0.895. Though the `auc` value reduced, hashing is meant to improve run-time performance of larger datasets. Feature hashing is widely popularized by Yahoo. Finally, in step 8, we perform tf-idf transformation, which returns a test `auc` value of 0.907.

9
Application of Deep Learning to Signal processing

The current chapter will present a case study of creating new music notes using generative modeling techniques such as RBM. In this chapter, we will cover the following topics:

- Introducing and preprocessing music MIDI files
- Building an RBM model
- Generating new music notes

Introducing and preprocessing music MIDI files

In this recipe, we will read a repository of **Musical Instrument Digital Interface** (**MIDI**) files and preprocess them into a suitable format for an RBM. MIDI is one of the formats of storing musical notes, which can be converted to other formats such as `.wav`, `.mp3`, `.mp4`, and so on. MIDI file formats store various kinds of events such as Note-on, Note-off, Tempo, Time Signature, End of track, and so on. However, we will primarily be focusing on the type of note--when it was turned **on**, and when it was turned **off**.

Each song is encoded into a binary matrix, where rows represent time, and columns represent both turned on and turned off notes. At each time, a note is turned on and the same note is turned off. Suppose that, out of n notes, note i is turned on and turned off at time j, then positions $Mji = 1$ and $Mj(n+i) = 1$, and the rest $Mj = 0$.

All the rows together form a song. Currently, in this chapter, we will be leveraging Python codes to encode MIDI songs into binary matrices, which can later be used in a Restricted Boltzmann Machine (RBM).

Getting ready

Let's look at the prerequisites to preprocess MIDI files:

1. Download the MIDI song repository:

```
https://github.com/dshieble/Music_RBM/tree/master/Pop_Music_Midi
```

2. Download the Python codes to manipulate MIDI songs:

```
https://github.com/dshieble/Music_RBM/blob/master/midi_manipulation.py
```

3. Install the `"reticulate"` package, which provides the R interface to Python:

```
Install.packages("reticulate")
```

4. Import Python libraries:

```
use_condaenv("python27")
midi <-
import_from_path("midi",path="C:/ProgramData/Anaconda2/Lib/site-
packages")
np <- import("numpy")
msgpack <-
import_from_path("msgpack",path="C:/ProgramData/Anaconda2/Lib/site-
packages")
psys <- import("sys")
tqdm <-
import_from_path("tqdm",path="C:/ProgramData/Anaconda2/Lib/site-
packages")
midi_manipulation_updated <-
import_from_path("midi_manipulation_updated",path="C:/Music_RBM")
glob <- import("glob")
```

How to do it...

Now that we have set up all the essentials, let's look at the function to define MIDI files:

1. Define the function to read the MIDI files and encode them into a binary matrix:

```
get_input_songs <- function(path){
  files = glob$glob(paste0(path,"/*mid*"))
  songs <- list()
  count <- 1
  for(f in files){
    songs[[count]] <-
np$array(midi_manipulation_updated$midiToNoteStateMatrix(f))
    count <- count+1
  }
  return(songs)
}
path <- 'Pop_Music_Midi'
input_songs <- get_input_songs(path)
```

Building an RBM model

In this recipe, we will build an RBM model as discussed (in detail) in `Chapter 5`, *Generative Models in Deep Learning*.

Getting ready

Let's set up our system for the model:

1. In Piano, the lowest note is 24 and the highest is 102; hence, the range of notes is 78. Thus, the number of columns in the encoded matrix is 156 (that is, 78 for note-on and 78 for note-off):

```
lowest_note = 24L
highest_note = 102L
note_range = highest_note-lowest_note
```

2. We will create notes for 15 number of steps at a time with 2,340 nodes in the input layer and 50 nodes in the hidden layer:

```
num_timesteps  = 15L
num_input      = 2L*note_range*num_timesteps
num_hidden       = 50L
```

3. The learning rate (alpha) is 0.1:

```
alpha<-0.1
```

How to do it...

Looking into the steps of building an RBM model:

1. Define the `placeholder` variables:

```
vb <- tf$placeholder(tf$float32, shape = shape(num_input))
hb <- tf$placeholder(tf$float32, shape = shape(num_hidden))
W <- tf$placeholder(tf$float32, shape = shape(num_input,
num_hidden))
```

2. Define a forward pass:

```
X = tf$placeholder(tf$float32, shape=shape(NULL, num_input))
prob_h0= tf$nn$sigmoid(tf$matmul(X, W) + hb)
h0 = tf$nn$relu(tf$sign(prob_h0 -
tf$random_uniform(tf$shape(prob_h0))))
```

3. Then, define a backward pass:

```
prob_v1 = tf$matmul(h0, tf$transpose(W)) + vb
v1 = prob_v1 + tf$random_normal(tf$shape(prob_v1), mean=0.0,
stddev=1.0, dtype=tf$float32)
h1 = tf$nn$sigmoid(tf$matmul(v1, W) + hb)
```

4. Calculate positive and negative gradients accordingly:

```
w_pos_grad = tf$matmul(tf$transpose(X), h0)
w_neg_grad = tf$matmul(tf$transpose(v1), h1)
CD = (w_pos_grad - w_neg_grad) / tf$to_float(tf$shape(X)[0])
update_w = W + alpha * CD
update_vb = vb + alpha * tf$reduce_mean(X - v1)
update_hb = hb + alpha * tf$reduce_mean(h0 - h1)
```

5. Define the objective function:

```
err = tf$reduce_mean(tf$square(X - v1))
```

6. Initialize the current and previous variables:

```
cur_w = tf$Variable(tf$zeros(shape = shape(num_input, num_hidden),
dtype=tf$float32))
```

```
cur_vb = tf$Variable(tf$zeros(shape = shape(num_input),
dtype=tf$float32))
cur_hb = tf$Variable(tf$zeros(shape = shape(num_hidden),
dtype=tf$float32))
prv_w = tf$Variable(tf$random_normal(shape=shape(num_input,
num_hidden), stddev=0.01, dtype=tf$float32))
prv_vb = tf$Variable(tf$zeros(shape = shape(num_input),
dtype=tf$float32))
prv_hb = tf$Variable(tf$zeros(shape = shape(num_hidden),
dtype=tf$float32))
```

7. Start a TensorFlow session:

```
sess$run(tf$global_variables_initializer())
song = np$array(trainX)
song =
song[1:(np$floor(dim(song)[1]/num_timesteps)*num_timesteps),]
song = np$reshape(song, newshape=shape(dim(song)[1]/num_timesteps,
dim(song)[2]*num_timesteps))
output <- sess$run(list(update_w, update_vb, update_hb), feed_dict
= dict(X=song,
W = prv_w$eval(),
vb = prv_vb$eval(),
hb = prv_hb$eval()))
prv_w <- output[[1]]
prv_vb <- output[[2]]
prv_hb <-  output[[3]]
sess$run(err, feed_dict=dict(X= song, W= prv_w, vb= prv_vb, hb=
prv_hb))
```

8. Run `200` training epochs:

```
epochs=200
errors <- list()
weights <- list()
u=1
for(ep in 1:epochs){
  for(i in seq(0,(dim(song)[1]-100),100)){
    batchX <- song[(i+1):(i+100),]
    output <- sess$run(list(update_w, update_vb, update_hb),
feed_dict = dict(X=batchX,
W = prv_w,
vb = prv_vb,
hb = prv_hb))
    prv_w <- output[[1]]
    prv_vb <- output[[2]]
    prv_hb <-  output[[3]]
    if(i%%500 == 0){
```

```
      errors[[u]] <- sess$run(err, feed_dict=dict(X= song, W=
prv_w, vb= prv_vb, hb= prv_hb))
      weights[[u]] <- output[[1]]
      u <- u+1
      cat(i , " : ")
    }
  }
  cat("epoch :", ep, " : reconstruction error : ",
errors[length(errors)][[1]],"\n")
}
```

Generating new music notes

In this recipe, we will generate new sample music notes. New musical notes can be generated by altering parameter `num_timesteps`. However, one should keep in mind to increase the timesteps, as it can become computationally inefficient to handle increased dimensionality of vectors in the current setup of RBM. These RBMs can be made efficient in learning by creating their stacks (namely **Deep Belief Networks**). Readers can leverage the DBN codes of `Chapter 5`, *Generative Models in Deep Learning,* to generate new musical notes.

How to do it...

1. Create new sample music:

```
hh0 = tf$nn$sigmoid(tf$matmul(X, W) + hb)
vv1 = tf$nn$sigmoid(tf$matmul(hh0, tf$transpose(W)) + vb)
feed = sess$run(hh0, feed_dict=dict( X= sample_image, W= prv_w, hb=
prv_hb))
rec = sess$run(vv1, feed_dict=dict( hh0= feed, W= prv_w, vb=
prv_vb))
S = np$reshape(rec[1,],newshape=shape(num_timesteps,2*note_range))
```

2. Regenerate the MIDI file:

```
midi_manipulation$noteStateMatrixToMidi(S,
name=paste0("generated_chord_1"))
generated_chord_1
```

10
Transfer Learning

In this chapter, we will discuss the concept of Transfer Learning. The following are the topics that will be covered:

- Illustrating the use of a pretrained model
- Setting up the Transfer Learning model
- Building an image classification model
- Training a deep learning model on a GPU
- Comparing performance using CPU and GPU

Introduction

A lot of development has happened within the deep learning domain in recent years, to enhance algorithmic efficacy and computational efficiency across different domains such as text, images, audio, and video. However, when it comes to training on new datasets, machine learning usually rebuilds the model from scratch, as is done in traditional data science problem solving. This becomes challenging when a new big dataset need to be trained as it will require very high computation power a lot of and time to reach the desired model efficacy.

Transfer Learning is a mechanism to learn new scenarios from existing models. This approach is very useful to train on big datasets, not necessarily from a similar domain or problem statement. For example, researchers have shown examples of Transfer Learning where they have trained Transfer Learning for completely different problem scenarios, such as when a model built using classifications of cat and dog is used for classifying objects such as aeroplane vs automobile.

In terms of analogy, it's more about passing the learned relationship to new architecture in order to fine-tune weights. An example of how Transfer Learning is used is shown in the following figure:

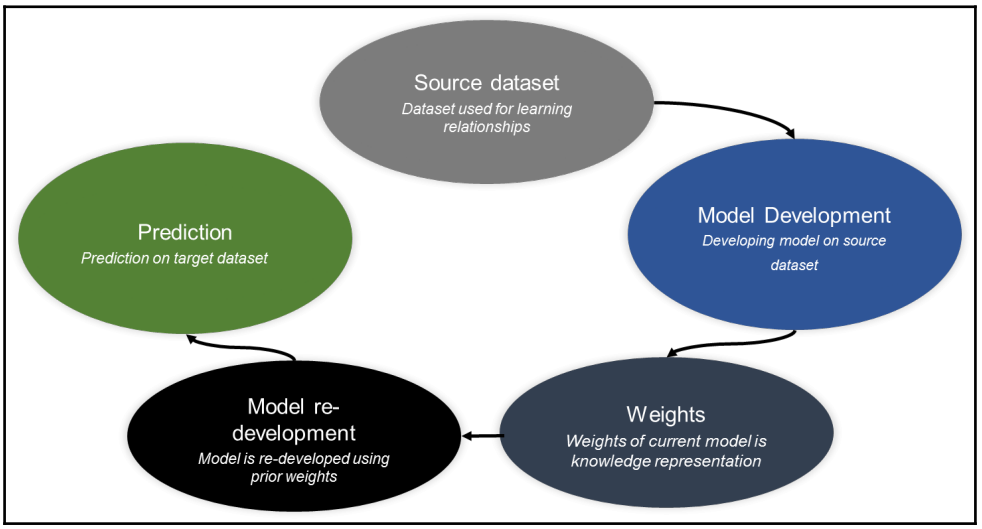


Illustration of Transfer Learning flow

The figure shows the steps of Transfer Learning, where the weights/architectures from a predeveloped deep learning model are reused to predict a new problem statement. Transfer Learning helps provide a good starting point for deep learning architectures. There are different open source projects going on in different domains, which facilitate Transfer Learning, for example, ImageNet (`http://image-net.org/index`) is an open source project for image classification where a lot of different architectures such as Alexnet, VGG16, and VGG19 have been developed. Similarly, in text mining, there is a Word2Vec representation of Google News trained using three billion running words.

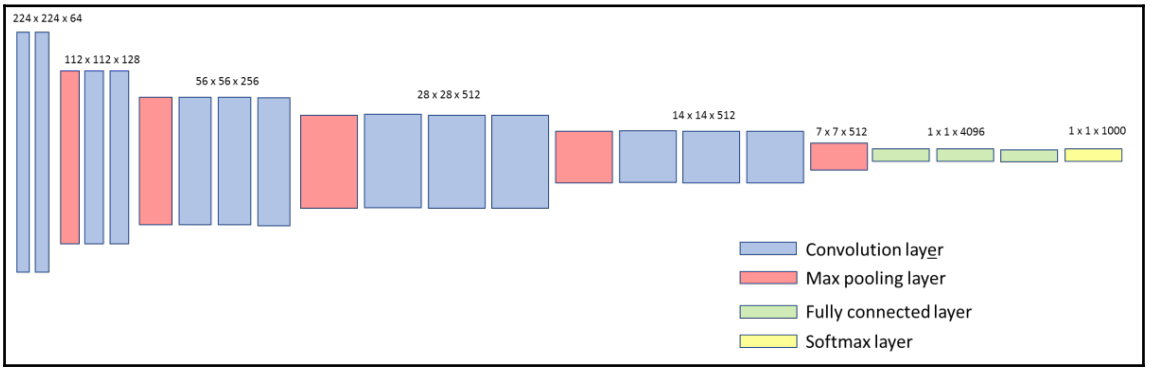
Details on word2vec can be found at `https://code.google.com/archive/p/word2vec/`.

Illustrating the use of a pretrained model

The current recipe will cover the set-up for using a pretrained model. We will use TensorFlow to demonstrate the recipe. The current recipe will use VGG16 architecture built using the ImageNet as dataset. The ImageNet is an open source image repository of images used for building image recognition algorithms. The database has more than 10 millions tagged images and more than 1 million images have bounding box to capture objects.

Lot of different deep learning architectures are developed using ImageNet dataset. Once of the popular one is VGG networks are convolution neural networks proposed by Zisserman and Simonyan (2014) and trained over ImageNet data with 1,000 classes. The current recipe will consider VGG16 variant of VGG architecture which is known for it's simplicity. The network uses input of 224 x 224 RGB image. The network utilizes 13 convolution layers with different width x height x depth. The maximum pooling layer is used to reduce volume size. The network uses 5 maxpooling layer. The output from convolution layer is passed through 3 fully connected layer. Outcome from fully connected layer go through softmax function to evaluate probability of 1000 classes.

The detailed architecture of VGG16 is shown in the following figure:



VGG16 architecture

Getting ready

The section covers required to use VGG16 pretrained model for classification.

1. Download VGG16 weights from `http://download.tensorflow.org/models/vgg_16_2016_08_28.tar.gz`. The file can be downloaded using the following script:

```
require(RCurl)
URL <-
'http://download.tensorflow.org/models/vgg_16_2016_08_28.tar.gz'
download.file(URL,destfile="vgg_16_2016_08_28.tar.gz",method="libcurl")
```

2. Install TensorFlow in Python.
3. Install R and the `tensorflow` package in R.
4. Download a sample image from `http://image-net.org/download-imageurls`.

How to do it...

The current section provides steps to use pretrained models:

1. Load `tensorflow` in R:

```
require(tensorflow)
```

2. Assign the `slim` library from TensorFlow:

```
slimobj = tf$contrib$slim
```

   The `slim` library in TensorFlow is used to maintain complex neural network models in terms of definition, training, and evaluation.

3. Reset graph in TensorFlow:

```
tf$reset_default_graph()
```

4. Define input images:

```
# Resizing the images
input.img= tf$placeholder(tf$float32, shape(NULL, NULL, NULL, 3))
scaled.img = tf$image$resize_images(input.img, shape(224,224))
```

5. Redefine the VGG16 network:

```
# Define VGG16 network
library(magrittr)
VGG16.model<-function(slim, input.image){
  vgg16.network = slim$conv2d(input.image, 64, shape(3,3),
scope='vgg_16/conv1/conv1_1') %>%
    slim$conv2d(64, shape(3,3), scope='vgg_16/conv1/conv1_2')  %>%
    slim$max_pool2d( shape(2, 2), scope='vgg_16/pool1')  %>%
    slim$conv2d(128, shape(3,3), scope='vgg_16/conv2/conv2_1')  %>%
    slim$conv2d(128, shape(3,3), scope='vgg_16/conv2/conv2_2')  %>%
    slim$max_pool2d( shape(2, 2), scope='vgg_16/pool2')  %>%
    slim$conv2d(256, shape(3,3), scope='vgg_16/conv3/conv3_1')  %>%
    slim$conv2d(256, shape(3,3), scope='vgg_16/conv3/conv3_2')  %>%
    slim$conv2d(256, shape(3,3), scope='vgg_16/conv3/conv3_3')  %>%
    slim$max_pool2d(shape(2, 2), scope='vgg_16/pool3')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv4/conv4_1')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv4/conv4_2')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv4/conv4_3')  %>%
    slim$max_pool2d(shape(2, 2), scope='vgg_16/pool4')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv5/conv5_1')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv5/conv5_2')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv5/conv5_3')  %>%
    slim$max_pool2d(shape(2, 2), scope='vgg_16/pool5')  %>%
    slim$conv2d(4096, shape(7, 7), padding='VALID',
scope='vgg_16/fc6')  %>%
    slim$conv2d(4096, shape(1, 1), scope='vgg_16/fc7') %>%
    slim$conv2d(1000, shape(1, 1), scope='vgg_16/fc8')  %>%
    tf$squeeze(shape(1, 2), name='vgg_16/fc8/squeezed')
  return(vgg16.network)
}
```

6. The preceding function defines the network architecture used for the VGG16 network. The network can be assigned using the following script:

```
vgg16.network<-VGG16.model(slim, input.image = scaled.img)
```

7. Load the VGG16 weights `vgg_16_2016_08_28.tar.gz` downloaded in the *Getting started* section:

```
# Restore the weights
restorer = tf$train$Saver()
sess = tf$Session()
restorer$restore(sess, 'vgg_16.ckpt')
```

8. Download a sample test image. Let's download an example image from a `testImgURL` location as shown in following script:

```
# Evaluating using VGG16 network
testImgURL<-
"http://farm4.static.flickr.com/3155/2591264041_273abea408.jpg"
img.test<-tempfile()
download.file(testImgURL,img.test, mode="wb")
read.image <- readJPEG(img.test)
# Clean-up the temp file
file.remove(img.test)
```

The preceding script downloads the following image from URL mention in variable `testImgURL`. The following is the downloaded image:

Sample image used to evaluate imagenet

9. Determine the class using the VGG16 pretrained model:

```
## Evaluate
size = dim(read.image)
imgs = array(255*read.image, dim = c(1, size[1], size[2], size[3]))
VGG16_eval = sess$run(vgg16.network, dict(images = imgs))
probs = exp(VGG16_eval)/sum(exp(VGG16_eval))
```

The maximum probability achieved is 0.62 for class 672, which refers to the category--mountain bike, all-terrain bike, off-roader--in the VGG16 trained dataset.

Setting up the Transfer Learning model

The current recipe will cover Transfer Learning using the CIFAR-10 dataset. The previous recipe presented how to use a pretrained model. The current recipe will demonstrate how to use a pretrained model for different problem statements.

We will use another very good deep learning package, MXNET, to demonstrate the concept with another architecture, Inception. To simplify the computation, we will reduce the problem complexity from 10 classes to two classes: aeroplane and automobile. The recipe focuses on data preparation for Transfer Learning using Inception-BN.

Getting ready

The section prepares for the upcoming section for setting-up Transfer Learning model.

1. Download the CIFAR-10 dataset from `http://www.cs.toronto.edu/~kriz/cifar.html`. The `download.cifar.data` function from `Chapter 3`, *Convolution Neural Networks,* can be used to download the dataset.
2. Install the `imager` package:

```
install.packages("imager")
```

How to do it...

The current part of the recipe will provide a step-by-step guide to prepare the dataset for the Inception-BN pretrained model.

1. Load the dependent packages:

```
# Load packages
require(imager)
source("download_cifar_data.R")
The download_cifar_data consists of function to download and read
CIFAR10 dataset.
```

2. Read the downloaded CIFAR-10 dataset:

```
# Read Dataset and labels
DATA_PATH<-paste(SOURCE_PATH, "/Chapter 4/data/cifar-10-batches-
bin/", sep="")
labels <- read.table(paste(DATA_PATH, "batches.meta.txt", sep=""))
cifar_train <- read.cifar.data(filenames =
```

```
c("data_batch_1.bin","data_batch_2.bin","data_batch_3.bin","data_ba
tch_4.bin"))
```

3. Filter the dataset for aeroplane and automobile. This is an optional step and is done to reduce complexity later:

```
# Filter data for Aeroplane and Automobile with label 1 and 2,
respectively
Classes = c(1, 2)
images.rgb.train <- cifar_train$images.rgb
images.lab.train <- cifar_train$images.lab
ix<-images.lab.train%in%Classes
images.rgb.train<-images.rgb.train[ix]
images.lab.train<-images.lab.train[ix]
rm(cifar_train)
```

4. Transform to image. This step is required as the CIFAR-10 dataset is a 32 x 32 x 3 image, which is flattened to a 1024 x 3 format:

```
# Function to transform to image
transform.Image <- function(index, images.rgb) {
  # Convert each color layer into a matrix,
  # combine into an rgb object, and display as a plot
  img <- images.rgb[[index]]
  img.r.mat <- as.cimg(matrix(img$r, ncol=32, byrow = FALSE))
  img.g.mat <- as.cimg(matrix(img$g, ncol=32, byrow = FALSE))
  img.b.mat <- as.cimg(matrix(img$b, ncol=32, byrow = FALSE))

  # Bind the three channels into one image
  img.col.mat <- imappend(list(img.r.mat,img.g.mat,img.b.mat),"c")
  return(img.col.mat)
}
```

5. The next step involve padding images with zeros:

```
  # Function to pad image
  image.padding <- function(x) {
  img_width <- max(dim(x)[1:2])
  img_height <- min(dim(x)[1:2])
  pad.img <- pad(x, nPix = img_width - img_height,
                 axes = ifelse(dim(x)[1] < dim(x)[2], "x", "y"))
  return(pad.img)
}
```

6. Save the image to a specified folder:

```
# Save train images
MAX_IMAGE<-length(images.rgb.train)

# Write Aeroplane images to aero folder
sapply(1:MAX_IMAGE, FUN=function(x, images.rgb.train,
images.lab.train){
  if(images.lab.train[[x]]==1){
    img<-transform.Image(x, images.rgb.train)
    pad_img <- image.padding(img)
    res_img <- resize(pad_img, size_x = 224, size_y = 224)
    imager::save.image(res_img, paste("train/aero/aero", x,
".jpeg", sep=""))
  }
}, images.rgb.train=images.rgb.train,
images.lab.train=images.lab.train)


# Write Automobile images to auto folder
sapply(1:MAX_IMAGE, FUN=function(x, images.rgb.train,
images.lab.train){
  if(images.lab.train[[x]]==2){
    img<-transform.Image(x, images.rgb.train)
    pad_img <- image.padding(img)
    res_img <- resize(pad_img, size_x = 224, size_y = 224)
    imager::save.image(res_img, paste("train/auto/auto", x,
".jpeg", sep=""))
  }
}, images.rgb.train=images.rgb.train,
images.lab.train=images.lab.train)
```

The preceding script saves the aeroplane images into the `aero` folder and the automobile images in the `auto` folder.

7. Convert to the recording format `.rec` supported by MXNet. This conversion requires `im2rec.py` MXnet module from Python as conversion is not supported in R. However, it can be called from R once MXNet is installed in Python using the system command. The splitting of the dataset into train and test can be obtained using the following file:

```
System("python ~/mxnet/tools/im2rec.py --list True --recursive True
--train-ratio 0.90 cifar_224/pks.lst cifar_224/trainf/")
```

The preceding script will generate two list files: `pks.lst_train.lst` and `pks.lst_train.lst`. The splitting of train and validation is controlled by the `–train-ratio` parameter in the preceding script. The number of classes is based on the number of folders in the `trainf` directory. In this scenario, two classes are picked: automotive and aeroplane.

8. Convert the `*.rec` file for training and validation dataset:

```
# Creating .rec file from training sample list
System("python ~/mxnet/tools/im2rec.py --num-thread=4 --pass-
through=1 /home/prakash/deep\ learning/cifar_224/pks.lst_train.lst
/home/prakash/deep\ learning/cifar_224/trainf/")

# Creating .rec file from validation sample list
System("python ~/mxnet/tools/im2rec.py --num-thread=4 --pass-
through=1 /home/prakash/deep\ learning/cifar_224/pks.lst_val.lst
/home/prakash/deep\ learning/cifar_224/trainf/")
```

The preceding script will create the `pks.lst_train.rec` and `pks.lst_val.rec` files to be used in the next recipe to train the model using a pretrained model.

Building an image classification model

The recipe focuses on building an image classification model using Transfer Learning. It will utilize the dataset prepared in the previous recipes and use the Inception-BN architecture. The BN in Inception-BN stands for **batch normalization**. Details of the Inception model in computer vision can be found in Szegedy et al. (2015).

Getting ready

The section cover's the prerequisite to set-up a classification model using INCEPTION-BN pretrained model.

1. Convert images into `.rec` file for train and validation.
2. Download the Inception-BN architecture from `http://data.dmlc.ml/models/imagenet/inception-bn/`.
3. Install R and the `mxnet` package in R.

How to do it...

1. Load the `.rec` file as iterators. The following is the function to load the `.rec` data as iterators:

```
# Function to load data as iterators
data.iterator <- function(data.shape, train.data, val.data,
BATCHSIZE = 128) {
  # Load training data as iterator
  train <- mx.io.ImageRecordIter(
    path.imgrec = train.data,
    batch.size  = BATCHSIZE,
    data.shape  = data.shape,
    rand.crop   = TRUE,
    rand.mirror = TRUE)
  # Load validation data as iterator
  val <- mx.io.ImageRecordIter(
    path.imgrec = val.data,
    batch.size  = BATCHSIZE,
    data.shape  = data.shape,
    rand.crop   = FALSE,
    rand.mirror = FALSE
  )
  return(list(train = train, val = val))
}
```

   In the preceding function, `mx.io.ImageRecordIter` reads batches of images from the `RecordIO` (`.rec`) files.

2. Load data using the `data.iterator` function:

```
# Load dataset
data  <- data.iterator(data.shape = c(224, 224, 3),
                       train.data = "pks.lst_train.rec",
                       val.data  = "pks.lst_val.rec",
                       BATCHSIZE = 8)
train <- data$train
val   <- data$val
```

3. Load the Inception-BN pretrained model from the `Inception-BN` folder:

```
# Load Inception-BN model
inception_bn <- mx.model.load("Inception-BN", iteration = 126)
symbol <- inception_bn$symbol
The different layers of the model can be viewed using function
symbol$arguments
```

4. Get the layers of the Inception-BN model:

```
# Load model information
internals <- symbol$get.internals()
outputs <- internals$outputs
flatten <- internals$get.output(which(outputs == "flatten_output"))
```

5. Define a new layer to replace the `flatten_output` layer:

```
# Define new layer
new_fc <- mx.symbol.FullyConnected(data = flatten,
                                   num_hidden = 2,
                                   name = "fc1")
new_soft <- mx.symbol.SoftmaxOutput(data = new_fc,
                                    name = "softmax")
```

6. Initialize weights for the newly defined layer. To retrain the last layer, weight initialization is performed using the following script:

```
# Re-initialize the weights for new layer
arg_params_new <- mxnet:::mx.model.init.params(
  symbol = new_soft,
  input.shape = c(224, 224, 3, 8),
  output.shape = NULL,
  initializer = mxnet:::mx.init.uniform(0.2),
  ctx = mx.cpu(0)
)$arg.params
fc1_weights_new <- arg_params_new[["fc1_weight"]]
fc1_bias_new <- arg_params_new[["fc1_bias"]]
```

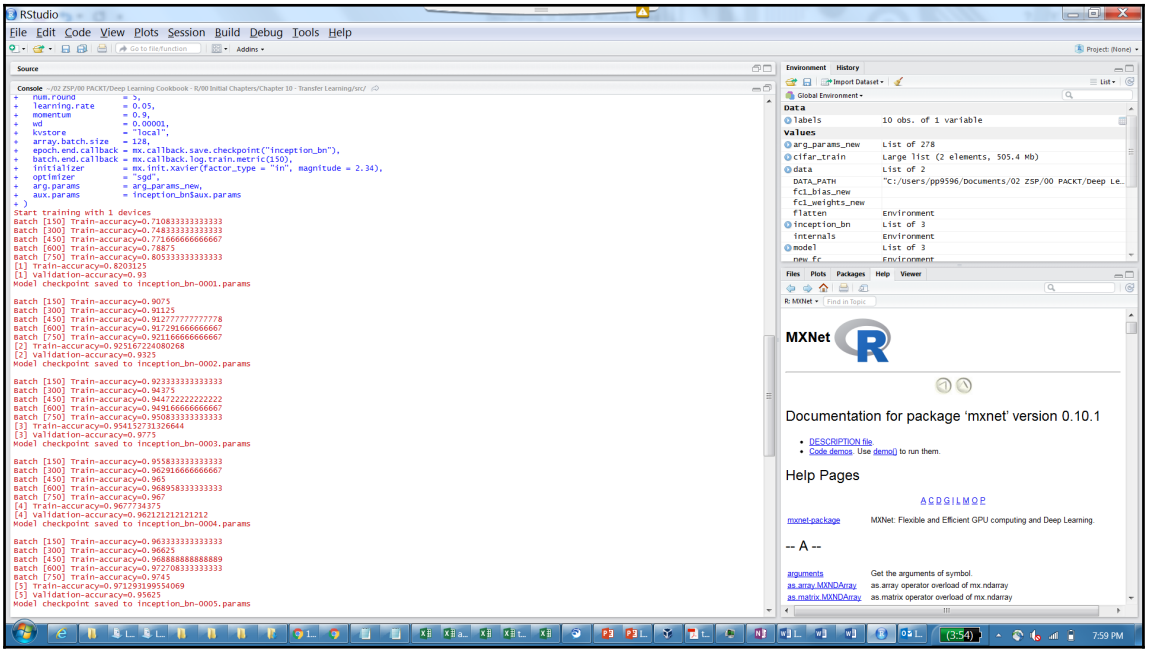
In the aforementioned layer, weights are assigned using uniform distribution between [*-0.2, 0.2*]. The `ctx` define the device on which the execution is to be performed.

7. Retrain the model:

```
# Mode re-train
model <- mx.model.FeedForward.create(
  symbol             = new_soft,
  X                  = train,
  eval.data          = val,
  ctx                = mx.cpu(0),
  eval.metric        = mx.metric.accuracy,
  num.round          = 5,
  learning.rate      = 0.05,
  momentum           = 0.85,
  wd                 = 0.00001,
```

```
    kvstore            = "local",
    array.batch.size   = 128,
    epoch.end.callback = mx.callback.save.checkpoint("inception_bn"),
    batch.end.callback = mx.callback.log.train.metric(150),
    initializer        = mx.init.Xavier(factor_type = "in", magnitude
= 2.34),
    optimizer          = "sgd",
    arg.params         = arg_params_new,
    aux.params         = inception_bn$aux.params
  )
```

The preceding model is set to run on CPU with five rounds, using accuracy as the evaluation metric. The following screenshot shows the execution of the described model:



Output from Inception-BN model, trained using CIFAR-10 dataset

The trained model has produced a training accuracy of 0.97 and a validation accuracy of 0.95.

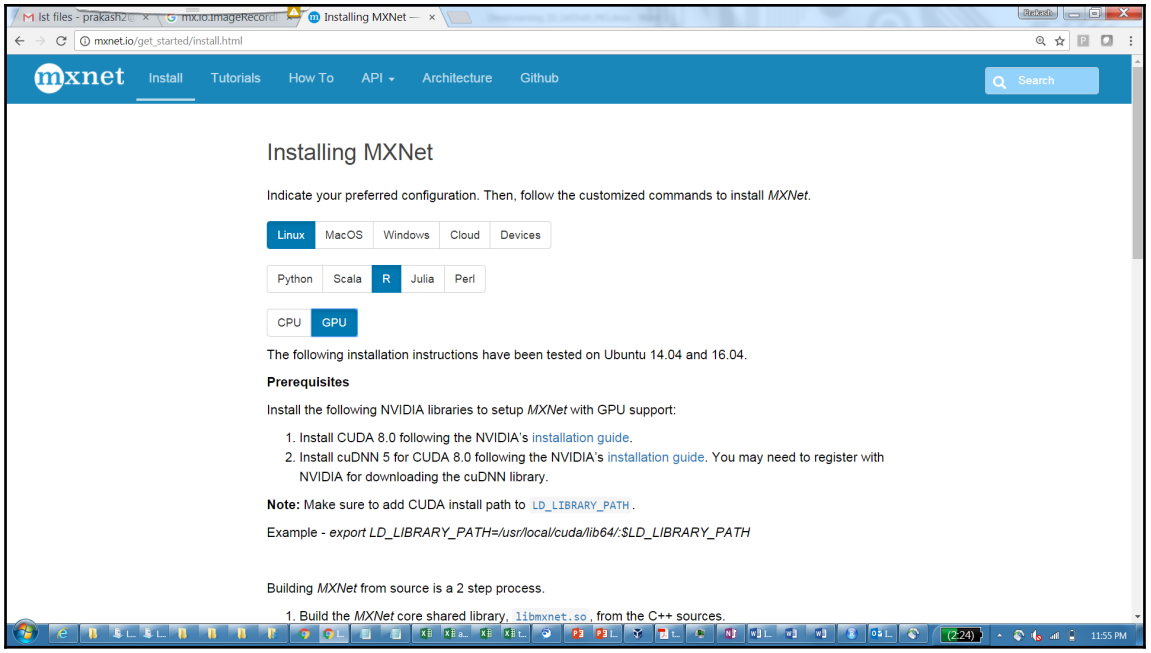
Training a deep learning model on a GPU

The **Graphical processing unit** (**GPU**) is hardware used for rendering images using a lot of cores. Pascal is the latest GPU micro architecture released by NVIDIA. The presence of hundreds of cores in GPU helps enhance the computation. This section provides the recipe for running a deep learning model using GPU.

Getting ready

This section provides dependencies required to run GPU and CPU:

1. The experiment performed in this recipe uses GPU hardware such as GTX 1070.
2. Install `mxnet` for GPU. To install `mxnet` for GPU for a specified machine, follow the installation instruction from `mxnet.io`. Select the requirement as shown in the screenshot, and follow the instructions:



Steps to get installation instruction for MXNet

How to do it...

Here is how you train a deep learning model on a GPU:

1. The Inception-BN-transferred learning recipe discussed in the previous section can be made to run on the GPU installed and the configured machine by changing the device settings as shown in the following script:

```
# Mode re-train
model <- mx.model.FeedForward.create(
  symbol             = new_soft,
  X                  = train,
  eval.data          = val,
  ctx                = mx.gpu(0),
  eval.metric        = mx.metric.accuracy,
  num.round          = 5,
  learning.rate      = 0.05,
  momentum           = 0.85,
  wd                 = 0.00001,
  kvstore            = "local",
  array.batch.size   = 128,
  epoch.end.callback = mx.callback.save.checkpoint("inception_bn"),
  batch.end.callback = mx.callback.log.train.metric(150),
  initializer        = mx.init.Xavier(factor_type = "in", magnitude
= 2.34),
  optimizer          = "sgd",
  arg.params         = arg_params_new,
  aux.params         = inception_bn$aux.params
)
```

In the aforementioned model, the device setting is changed from `mx.cpu` to `mx.gpu`. The CPU for five iteration tools takes ~2 hours of computational effort, whereas the same iteration is completed in ~15 min with GPU.

Comparing performance using CPU and GPU

One of the questions with device change is why so much improvement is observed when the device is switched from CPU to GPU. As the deep learning architecture involves a lot of matrix computations, GPUs help expedite these computations using a lot of parallel cores, which are usually used for image rendering.

The power of GPU has been utilized by a lot of algorithms to accelerate the execution. The following recipe provides some benchmarks of matrix computation using the `gpuR` package. The `gpuR` package is a general-purpose package for GPU computing in R.

Getting ready

The section covers requirement to set-up a comparison between GPU Vs CPU.

1. Use GPU hardware installed such as GTX 1070.
2. CUDA toolkit installation using URL `https://developer.nvidia.com/cuda-downloads`.
3. Install the `gpuR` package:

```
install.packages("gpuR")
```

4. Test `gpuR`:

```
library(gpuR)
# verify you have valid GPUs
detectGPUs()
```

How to do it...

Let's get started by loading the packages:

1. Load the package, and set the precision to `float` (by default, the GPU precision is set to a single digit):

```
library("gpuR")
options(gpuR.default.type = "float")
```

2. Matrix assignment to GPU:

```
# Assigning a matrix to GPU
A<-matrix(rnorm(1000), nrow=10)

vcl1 = vclMatrix(A)
```

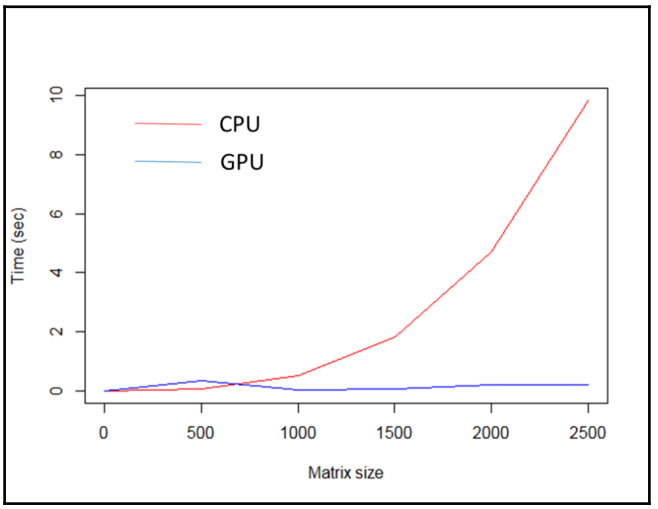
The output of the preceding command will contain details of the object. An illustration is shown in the following script:

```
> vcl1
An object of class "fvclMatrix"
Slot "address":
<pointer: 0x000000001822e180>

Slot ".context_index":
[1] 1

Slot ".platform_index":
[1] 1

Slot ".platform":
[1] "Intel(R) OpenCL"

Slot ".device_index":
[1] 1

Slot ".device":
[1] "Intel(R) HD Graphics 530"
```

3. Let's consider the evaluation of CPU vs GPU. As most of the deep learning will be using GPUs for matrix computation, the performance is evaluated by matrix multiplication using the following script:

```
# CPU vs GPU performance
DF <- data.frame()
evalSeq<-seq(1,2501,500)
for (dimpower in evalSeq){
  print(dimpower)
  Mat1 = matrix(rnorm(dimpower^2), nrow=dimpower)
  Mat2 = matrix(rnorm(dimpower^2), nrow=dimpower)
  now <- Sys.time()
  Matfin = Mat1%*%Mat2
  cpu <- Sys.time()-now
  now <- Sys.time()
  vcl1 = vclMatrix(Mat1)
  vcl2 = vclMatrix(Mat2)
  vclC = vcl1 %*% vcl2
  gpu <- Sys.time()-now
  DF <- rbind(DF,c(nrow(Mat1), cpu, gpu))
}
DF<-data.frame(DF)
colnames(DF) <- c("nrow", "CPU_time", "gpu_time")
```

The preceding script computes the matrix multiplication using CPU and GPU; the time is stored for different dimensions of the matrix. The output from the preceding script is shown in the following diagram:



Comparison between CPU and GPU

The graph shows that the computation effort required by CPUs increases exponentially with the CPU. Thus, GPUs help expedite it drastically.

There's more...

The GPUs are new arena in machine learning computing, and a lot of packages have been developed in R to access GPUs while keeping you in a familiar R environment such as `gputools`, `gmatrix`, and `gpuR`. The other algorithms are also developed and implemented while accessing GPUs to enhance their computational power such as `RPUSVM`, which implements SVM using GPUs. Thus, the topic requires a lot of creativity with some exploration to deploy algorithms while utilizes full capability of hardware.

See also

To learn more on parallel computing using R, go through *Mastering Parallel Programming with R* by Simon R. Chapple et al. (2016).

Index

E

F

G

H

I

J

Q

R

S

T

U

V

W

Z