R Deep Learning Essentials

Deep learning is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using model architectures.

This book will introduce you to the deep learning package H2O with R and help you understand the concepts of deep learning. We will start by setting up important deep learning packages available in R and then move toward building models related to neural networks, prediction, and deep prediction, all of this with the help of real-life examples. After installing the H2O package, you will learn about prediction algorithms. Moving ahead, concepts such as overfitting data, anomalous data, and deep prediction models are explained. Finally, the book will cover concepts relating to tuning and optimizing models.

Who this book is written for
This book caters to aspiring data scientists who are well versed in machine learning concepts with R and are looking to explore the deep learning paradigm using the packages available in R. You should have a fundamental understanding of the R language and be comfortable with statistical algorithms and machine learning techniques.

What you will learn from this book

- Set up the R package H2O to train deep learning models
- Understand the core concepts behind deep learning models
- Use auto-encoders to identify anomalous data or outliers
- Predict or classify data automatically using deep neural networks
- Build generalizable models using regularization to avoid overfitting the training data

Who this book is written for
This book caters to aspiring data scientists who are well versed in machine learning concepts with R and are looking to explore the deep learning paradigm using the packages available in R. You should have a fundamental understanding of the R language and be comfortable with statistical algorithms and machine learning techniques.

What you will learn from this book

- Set up the R package H2O to train deep learning models
- Understand the core concepts behind deep learning models
- Use auto-encoders to identify anomalous data or outliers
- Predict or classify data automatically using deep neural networks
- Build generalizable models using regularization to avoid overfitting the training data
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with Deep Learning'
- A synopsis of the book’s content
- More information on R Deep Learning Essentials
About the Author

Dr. Joshua F. Wiley is a lecturer at Monash University and a senior partner at Elkhart Group Limited, a statistical consultancy. He earned his PhD from the University of California, Los Angeles. His research focuses on using advanced quantitative methods to understand the complex interplays of psychological, social, and physiological processes in relation to psychological and physical health. In statistics and data science, Joshua focuses on biostatistics and is interested in reproducible research and graphical displays of data and statistical models. Through consulting at Elkhart Group Limited and his former work at the UCLA Statistical Consulting Group, Joshua has helped a wide array of clients, ranging from experienced researchers to biotechnology companies. He develops or codevelops a number of R packages including varian, a package to conduct Bayesian scale-location structural equation models, and MplusAutomation, a popular package that links R to the commercial Mplus software.
Preface

This book is about how to train and use deep learning models or deep neural networks in the R programming language and environment. This book is not intended to provide an in-depth theoretical coverage of deep neural networks, but it will give you enough background to help you understand their basics and use and interpret the results. This book will also show you some of the packages and functions available to train deep neural networks, optimize their hyperparameters to improve the accuracy of your model, and generate predictions or otherwise use the model you built. The book is intended to provide an easy-to-read coverage of the essentials in order to get going with real-life examples and applications.

What this book covers

Chapter 1, Getting Started with Deep Learning, shows how to get the R and H2O packages set up and installed on a computer or server along with covering all the basic concepts related to deep learning.

Chapter 2, Training a Prediction Model, covers how to build a shallow unsupervised neural network prediction model.

Chapter 3, Preventing Overfitting, explains different approaches that can be used to prevent models from overfitting the data in order to improve generalizability called regularization on unsupervised data.

Chapter 4, Identifying Anomalous Data, covers how to perform unsupervised deep learning in order to identify anomalous data, such as fraudulent activity or outliers.

Chapter 5, Training Deep Prediction Models, shows how to train deep neural networks to solve prediction and classification problems, such as image recognition.
Chapter 6, Tuning and Optimizing Models, explains how to adjust model tuning parameters to improve and optimize the accuracy and performance of deep learning models.

Appendix, Bibliography, contains the references for all the citations throughout the book.
Getting Started with Deep Learning

This chapter discusses deep learning, a powerful multi-layered architecture for pattern recognition, signal detection, and classification or prediction. Although deep learning is not new, it is only in the past decade that it has gained great popularity, due in part to advances in computational capacity and new ways of more efficiently training models, as well as the availability of ever-increasing amounts of data. In this chapter, you will learn what deep learning is, the R packages available for training such models, how to get your system set up for analysis, and how to connect R with H2O, which we will use for many of the examples and work in later chapters on how to actually train and use a deep learning model.

In this chapter, we will explore the following topics:

- What is deep learning?
- R packages that train deep learning models such as deep belief networks or deep neural networks
- Connecting R and H2O, the main package we will be using for deep learning
What is deep learning?

To understand what deep learning is, perhaps it is easiest to start with what is meant by regular machine learning. In general terms, machine learning is devoted to developing and using algorithms that learn from raw data in order to make predictions. Prediction is a very general term. For example, predictions from machine learning may include predicting how much money a customer will spend at a given company, or whether a particular credit card purchase is fraudulent. Predictions also encompass more general pattern recognition, such as what letters are present in a given image, or whether a picture is of a horse, dog, person, face, building, and so on. Deep learning is a branch of machine learning where a multi-layered (deep) architecture is used to map the relations between inputs or observed features and the outcome. This deep architecture makes deep learning particularly suitable for handling a large number of variables and allows deep learning to generate features as part of the overall learning algorithm, rather than feature creation being a separate step. Deep learning has proven particularly effective in the fields of image recognition (including handwriting as well as photo or object classification) and natural language processing, such as recognizing speech.

There are many types of machine learning algorithms. In this book, we are primarily going to focus on neural networks as these have been particularly popular in deep learning. However, this focus does not mean that it is the only technique available in machine learning or even deep learning, nor that other techniques are not valuable or even better suited, depending on the specific task. The next sections will discuss what neural networks and deep neural networks are conceptually in more depth.

Conceptual overview of neural networks

As their name suggests, neural networks draw their inspiration from neural processes and neurons in the body. Neural networks contain a series of neurons, or nodes, which are interconnected and process input. The connections between neurons are weighted, with these weights based on the function being used and learned from the data. Activation in one set of neurons and the weights (adaptively learned from the data) may then feed into other neurons, and the activation of some final neuron(s) is the prediction.

To make this process more concrete, an example from human visual perception may be helpful. The term grandmother cell is used to refer to the concept that somewhere in the brain there is a cell or neuron that responds specifically to a complex and specific object, such as your grandmother. Such specificity would require thousands of cells to represent every unique entity or object we encounter. Instead, it is thought that visual perception occurs by building up more basic pieces into complex representations. For example, the following is a picture of a square:
Rather than our visual system having cells, neurons that are activated only upon seeing the gestalt, or entirety, of a square, we can have cells that recognize horizontal and vertical lines, as shown in the following:

Figure 1.1

Figure 1.2
In this hypothetical case, there may be two neurons, one which is activated when it senses horizontal lines and another that is activated when it senses vertical lines. Finally, a higher-order process recognizes that it is seeing a square when both the lower order neurons are activated simultaneously.

Neural networks share some of these same concepts, with inputs being processed by a first layer of neurons that may go on to trigger another layer. Neural networks are sometimes shown as graphical models. In Figure 1.3, Inputs are data represented as squares. These may be pixels in an image, or different aspects of sounds, or something else. The next layer of Hidden neurons consists of neurons that recognize basic features, such as horizontal lines, vertical lines, or curved lines. Finally, the output may be a neuron that is activated by the simultaneous activation of two of the hidden neurons. In this book, observed data or features are depicted as squares, and unobserved or hidden layers as circles:

![Figure 1.3](image-url)
Neural networks are used to refer to a broad class of models and algorithms. Hidden neurons are generated based on some combination of the observed data, similar to a basis expansion in other statistical techniques; however, rather than choosing the form of the expansion, the weights used to create the hidden neurons are learned from the data. Neural networks can involve a variety of activation function(s), which are transformations of the weighted raw data inputs to create the hidden neurons. A common choice for activation functions is the sigmoid function: 
\[ \sigma(x) = \frac{1}{1 + e^{-x}} \] and the hyperbolic tangent function \( f(x) = \tanh(x) \). Finally, radial basis functions are sometimes used as they are efficient function approximators. Although there are a variety of these, the Gaussian form is common: 
\[ f(x) = \exp \left( -\frac{\|x-c\|^2}{2\sigma^2} \right) \].

In a shallow neural network such as is shown in Figure 1.3, with only a single hidden layer, from the hidden units to the outputs is essentially a standard regression or classification problem. The hidden units can be denoted by \( h \), the outputs by \( Y \). Different outputs can be denoted by subscripts \( i = 1, \ldots, k \) and may represent different possible classifications, such as (in our case) a circle or square. The paths from each hidden unit to each output are the weights and for the \( i \)th output are denoted by \( w_i \). These weights are also learned from the data, just like the weights used to create the hidden layer. For classification, it is common to use a final

\[ Y_i = \frac{e^{w_i^T h}}{\sum_{j} e^{w_j^T h}} \]

transformation, the softmax function, which is as this ensures that the estimates are positive (using the exponential function) and that the probability of being in any given class sums to one. For linear regression, the identity function, which returns its input, is commonly used. Confusion may arise as to why there are paths between every hidden unit and output as well as every input and hidden unit. These are commonly drawn to represent that a priori any of these relations are allowed to exist. The weights must then be learned from the data, with zero or near zero weights essentially equating to dropping unnecessary relations.

This only scratches the surface of the conceptual and practical aspects of neural networks. For a slightly more in-depth introduction to neural networks, see Chapter 11 of Hastie, T., Tibshirani, R., and Friedman, J. (2009), which is freely available at http://statweb.stanford.edu/~tibs/ElemStatLearn/, Chapter 16 of Murphy, K. P. (2012), and Chapter 5 of Bishop, C. M. (2006). Next, we will turn to a brief introduction to deep neural networks.
Deep neural networks

Perhaps the simplest, if not the most informative, definition of a **deep neural network** (DNN) is that it is a neural network with multiple hidden layers. Although a relatively simple conceptual extension of neural networks, such deep architecture provides valuable advances in terms of the capability of the models and new challenges in training them.

Using multiple hidden layers allows a more sophisticated build-up from simple elements to more complex ones. When discussing neural networks, we considered the outputs to be whether the object was a circle or a square. In a deep neural network, many circles and squares could be combined to form other more advanced shapes. One can consider two complexity aspects of a model’s architecture. One is how wide or narrow it is—that is, how many neurons there are in a given layer. The second is how deep it is, or how many layers of neurons there are. For data that truly has such deep architectures, a deep neural network can fit it more accurately with fewer parameters than a **neural network** (NN), because more layers (each with fewer neurons) can be a more efficient and accurate representation; for example, because the shallow NN cannot build more advanced shapes from basic pieces, in order to provide equal accuracy to the deep neural network it must represent each unique object. Again considering pattern recognition in images, if we are trying to train a model for text recognition the raw data may be pixels from an image. The first layer of neurons could be trained to capture different letters of the alphabet, and then another layer could recognize sets of these letters as words. The advantage is that the second layer does not have to directly learn from the pixels, which are noisy and complex. In contrast, a shallow architecture may require far more parameters, as each hidden neuron would have to be capable of going directly from pixels in an image to a complete word, and many words may overlap, creating redundancy in the model.

One of the challenges in training deep neural networks is how to efficiently learn the weights. The models are often complex and local minima abound, making the optimization problem a challenging one. One of the major advancements came in 2006, when it was shown that **Deep Belief Networks** (DBNs) could be trained one layer at a time (See Hinton, G. E., Osindero, S., and Teh, Y. W. (2006)). A DBN is a type of deep neural network where multiple hidden layers and connections between (but not within) layers (that is, a neuron in layer 1 may be connected to a neuron in layer 2, but may not be connected to another neuron in layer 1). This is the essentially the same definition of a **Restricted Boltzmann Machine** (RBM)—an example is shown in Figure 1.4—except that a RBM typically has one input layer and one hidden layer:
The restriction of no connections within a layer is valuable as it allows for much faster training algorithms to be used, such as the contrastive divergence algorithm. If several RBMs are stacked together, they can form a DBN. Essentially, the DBN can then be trained as a series of RBMs. The first RBM layer is trained and used to transform raw data into hidden neurons, which are then treated as a new set of inputs in a second RBM, and the process is repeated until all layers have been trained.

The benefits of the realization that DBNs could be trained one layer at a time extend beyond just DBNs, however. DBNs are sometimes used as a pre-training stage for a deep neural network. This allows the comparatively fast, greedy layer-by-layer training to be used to provide good initial estimates, which are then refined in the deep neural network using other, slower, training algorithms such as back propagation.
So far we have been primarily focused on feed-forward neural networks, where the results from one layer and neuron feed forward to the next. Before closing this section, two specific kinds of deep neural networks that have grown in popularity are worth mentioning. The first is a Recurrent Neural Network (RNN) where neurons send feedback signals to each other. These feedback loops allow RNNs to work well with sequences. A recent example of an application of RNNs was to automatically generate click-bait such as One trick great hair salons don’t want you to know or Top 10 reasons to visit Los Angeles: #6 will shock you!. RNNs work well for such jobs as they can be seeded from a large initial pool of a few words (even just trending search terms or names) and then predict/generate what the next word should be. This process can be repeated a few times until a short phrase is generated, the click-bait. This example is drawn from a blog post by Lars Eidnes, available at http://larseidnes.com/2015/10/13/auto-generating-clickbait-with-recurrent-neural-networks/. The second type is a Convolutional Neural Network (CNN). CNNs are most commonly used in image recognition. CNNs work by having each neuron respond to overlapping subregions of an image. The benefits of CNNs are that they require comparatively minimal pre-processing yet still do not require too many parameters through weight sharing (for example, across subregions of an image). This is particularly valuable for images as they are often not consistent. For example, imagine ten different people taking a picture of the same desk. Some may be closer or farther away or at positions resulting in essentially the same image having different heights, widths, and the amount of image captured around the focal object.

As for neural networks, this description only provides the briefest of overviews as to what deep neural networks are and some of the use cases to which they can be applied. For an overview, see Schmidhuber, J. (2015) as well as Chapter 28 of Murphy, K. P. (2012).

R packages for deep learning
Although there are a number of R packages for machine learning, there are comparatively few available for neural networks and deep learning. In this section, we will see how to install all the necessary R packages and set them up to use neural networks and deep learning.

It is helpful to have a good integrated development environment (IDE) for working with R and doing data analysis. I use Emacs, a powerful text editor, along with Emacs Speaks Statistics (ESS), which helps Emacs work nicely with R. An easy way to get up-and-running is to use a modified distribution of Emacs designed to work nicely with R and for statistics. It is created and maintained by Vincent Goulet and is freely available at http://vgoulet.act.ulaval.ca/en/emacs/. Another popular R IDE is Rstudio (https://www.rstudio.com/). One advantage of both Emacs and Rstudio is that they are available on all major platforms (Windows, Mac, and Linux), so even if you switch computers you can have a consistent IDE experience.
Setting up reproducible results

Software for data science is advancing and changing rapidly. Although this is wonderful for progress, it can make reproducing someone else's results a challenge. Even your own code may not work when you go back to it a few months later. One way to address this issue is to make a record of what versions of software were used and ensure there is a snapshot of them available. For this book, we will use the R package `checkpoint` provided by Revolution Analytics; this works in connection with their server, which provides daily snapshots (checkpoints) of the Comprehensive R Archive Network (CRAN). To learn more about this process, you can read the online vignette for the package available at https://cran.r-project.org/web/packages/checkpoint/vignettes/checkpoint.html.

This book was written using R version 3.2.3, nicknamed Wooden Christmas-Tree, on Windows 10 Professional x64. Although this is the latest version of R at the time of writing, as new versions are released CRAN keeps copies of older R versions both as binaries (in the future at https://cran.r-project.org/bin/windows/base/old/) and as source tar balls (https://cran.r-project.org/src/base/R-3/), which can be used to compile the source to any operating system.

For H2O, one of the main R packages will be used for deep learning, we will also need Java installed. This book was written using the Java SE Development Kit 8 update 66 for 64 bit. You can download Java for your operating system at http://www.oracle.com/technetwork/java/javase/.

With those steps done, we are ready to get started. To use the checkpoint package, put all your R scripts for one project together in a single folder. Installing R packages using the checkpoint package is a somewhat circular process. The checkpoint package works by scanning R scripts in the project directory to see what packages are loaded (and therefore that it needs to install), by checking for calls to the `library()` or `require()` functions. Of course, we cannot actually use the `library()` function until we have installed the packages.

To begin with, create an R script in your project directory called `checkpoint.R` with the following code:

```r
## uncomment to install the checkpoint package
## install.packages("checkpoint")
library(checkpoint)

checkpoint("2016-02-20", R.version = "3.2.3")
```
Getting Started with Deep Learning

Once you have created the R script, you can uncomment and run the code to install the checkpoint package. You only need to do this once, so when you are done it’s best to comment the code out again so it is not re-installed each time you run the file. This is the file we will run each time we want to set up our R environment for this deep learning project. The checkpoint for this book is 20th February 2016 and we are using R version 3.2.3. Next, we can add `library()` calls for some packages we will need to be available by adding the following code to our `checkpoint.R` script (but note that these are not run yet!):

```r
## Chapter 1 ##
## Tools
library(RCurl)
library(jsonlite)
library(caret)
library(e1071)
## basic stats packages
library(statmod)
library(MASS)
```

**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:

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

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
- Zippeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux
Once we have added that code, save the file so that any changes are written to the disk, and then run the first couple of lines to load the checkpoint package and the call to `checkpoint()`. The results should look something like Figure 1.5:

```
> options(check=FALSE, help.type="text")
> options(STEM='ESS', str.dendrogram.last='', editor='emacsclient.exe', show.error.locations=TRUE)
> library(checkpoint)

checkpoint: Part of the Reproducible R Toolkit from Revolution Analytics
http://projects.revolutionanalytics.com/rrt/
> checkpoint("2015-01-21", R.version = "3.2.2")
Can I create directory ~/.checkpoint for internal checkpoint use?(y/n) y
Scanning for packages used in this project
[==========================================================================] 100%
- Discovered 7 packages
- Installing packages used in this project
  - Installing 'caret'
    also installing the dependencies 'colorspace', 'grid', 'nloptr', 'RcppEigen', 'RCOLOR Brewer', 'dichromat', 'munsell', 'labeling', 'Matrix', 'lme4', 'SparseM', 'MatrixModels', 'stringr', 'magrittr', 'digest', 'gttable', 'MASS', 'scales', 'mgcv', 'knitr', 'pilot', 'quantreg', 'codetools', 'iterators', 'Rcpp', 'stringr'
  - 'lattice', 'ggplot2', 'car', 'foreach', 'plyr', 'nlme', 'reshape2'.

- package 'colorspace' successfully unpacked and MDS sums checked
- package 'caret' successfully unpacked and MDS sums checked
  - Installing 'e1071'
  also installing the dependency 'class'

- package 'class' successfully unpacked and MDS sums checked
- package 'e1071' successfully unpacked and MDS sums checked
  - Installing 'jsonlite'
  package 'jsonlite' successfully unpacked and MDS sums checked
  - Previously installed 'MASS'
    - Installing 'Rcpp'
    also installing the dependency 'bitops'

- package 'bitops' successfully unpacked and MDS sums checked
- package 'Rcpp' successfully unpacked and MDS sums checked
  - Installing 'statmod'
    package 'statmod' successfully unpacked and MDS sums checked
    - checkpoint process complete
```

---

Figure 1.5
Getting Started with Deep Learning

The checkpoint package asks to create a directory to store specific versions of the packages used, and then finds all packages and installs them. The next sections show how to set up some specific R packages for deep learning.

Neural networks

There are several packages in R that can fit basic neural networks. The nnet package is a recommended package and can fit feed-forward neural networks with one hidden layer, like the one shown in Figure 1.3. For more details on the nnet package, see Venables, W. N. and Ripley, B. D. (2002). The neuralnet package also fits shallow neural networks with one hidden layer, but can train them using back-propagation and allows custom error and neuron activation functions. Finally, we come to the RSNNS package, which is an R wrapper of the Stuttgart Neural Network Simulator (SNNS). The SNNS was originally written in C, but was ported to C++. RSNNS allows many types of models to fit in R. Common models are available using convenient wrappers, but the RSNNS package also makes many model components from SNNS available, making it possible to train a wide variety of models. For more details on the RSNNS package, see Bergmeir, C., and Benítez, J. M. (2012). We will see examples of how to use these models in Chapter 2, Training a Prediction Model. For now, we can install them by adding the following code to the checkpoint.R script and saving it. Saving is important because, if our changes to the R script are not written to the disk, the checkpoint() function will not see the changes and will not find and install the new packages:

```r
## neural networks
library(nnet)
library(neuralnet)
library(RSNNS)
```

Now, if we re-run the checkpoint() function and it is successful, R should tell us that it discovered eight packages and that it installed nnet, neuralnet, RSNNS, and Rcpp, a dependency for the RSNNS package.

The deepnet package

The deepnet package provides a number of tools for deep learning in R. Specifically, it can train RBMs and use these as part of DBNs to generate initial values to train deep neural networks. The deepnet package also allows for different activation functions, and the use of dropout for regularization. To install it, we follow the same process we used before adding the following code to the checkpoint.R script, saving it, and then re-running the checkpoint() function:

```r
## deep learning
library(deepnet)
```
The darch package

The darch package is based on Matlab code by George Hinton and stands for deep architectures. It can train RBMs and DBNs along with a variety of options related to each. A limitation of the darch package is that, because it is a pure R implementation, model training tends to be slow. To install it, we follow the same process we used before adding the following code to the checkpoint.R script, saving it, and then re-running the checkpoint() function:

```r
## deep learning
library(darch)
```

The H2O package

The H2O package provides an interface to the H2O software. H2O is written in Java and is fast and scalable. It provides not only deep learning functionality, but also a variety of other popular machine learning algorithms and models, and the model results can be stored as pure Java code to allow fast scoring, facilitating the deployment of models to solve real-world problems. To install it, we follow the same process we used before adding the following code to the checkpoint.R script, saving it, and then re-running the checkpoint() function:

```r
## deep learning
library(h2o)
```

Connecting R and H2O

Because H2O is Java-based software with an R wrapper, to connect R to it we must initialize an instance of H2O and also connect R with it, linking or passing data and model commands to it. In this section, we will show how to get everything set up to train a model using H2O.
Initializing H2O

To initialize an H2O cluster, we use the `h2o.init()` function. Initializing a cluster will also set up a lightweight web server that allows interaction with the software via a local webpage. Generally, the `h2o.init()` function has sensible default values, but we can customize many aspects of it, and it may be particularly good to customize the number of cores/threads to use as well as how much memory we are willing for it to use, which can be accomplished as in the following code using the `max_mem_size` and `nthreads` arguments. In the code that follows, we initialize an H2O cluster to use two threads and up to three gigabytes of memory. After the code, R will indicate the location of log files, the Java version, and details about the cluster:

```r
cl <- h2o.init(
    max_mem_size = "3G",
    nthreads = 2)
```

H2O is not running yet, starting it now...

Note: In case of errors look at the following log files:

```
C:\Users\jwile\AppData\Local\Temp\RtmpuelhZm/h2o_jwile_started_from_r.out
C:\Users\jwile\AppData\Local\Temp\RtmpuelhZm/h2o_jwile_started_from_r.err
```

java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b18, mixed mode)

Successfully connected to http://127.0.0.1:54321/

R is connected to the H2O cluster:

```r
H2O cluster uptime: 1 seconds 735 milliseconds
H2O cluster version: 3.6.0.8
H2O cluster name: H2O_started_from_R_jwile_ndx127
H2O cluster total nodes: 1
H2O cluster total memory: 2.67 GB
H2O cluster total cores: 4
H2O cluster allowed cores: 2
H2O cluster healthy: TRUE
```
Once the cluster is initialized, we can interface with it either using R or using the web interface available at the local host (127.0.0.1:54321); it is shown in Figure 1.6:

Figure 1.6

**Linking datasets to an H2O cluster**

There are a couple of ways to get data into an H2O cluster. If the dataset is already loaded into R, you can simply use the `as.h2o()` function as shown in the following code:

```r
h2oiris <- as.h2o(
  droplevels(iris[1:100, ])
)```
We can check the results by typing the R object, `h2oiris`, which is simply an object that holds a reference to the H2O data. The R API queries H2O when we try to print it:

```r
h2oiris
```

This returns the following output:

```
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
```

[100 rows x 5 columns]

We can also check the levels of factor variables, such as the `Species` variable, as shown in the following:

```r
h2o.levels(h2oiris, 5)
```

```
[1] setosa     versicolor
```

In real-world uses, it is more likely that the data already exists somewhere; rather than load the data into R only to export it into H2O (a costly operation as it creates an unnecessary copy of the data in R), we can just load data directly into H2O. First we will create a CSV file based on the built-in `mtcars` dataset, then we will tell the H2O instance to read the data using R. Printing again shows the data:

```r
write.csv(mtcars, file = "mtcars.csv")
```

```r
h2omtcars <- h2o.importFile(
  path = "mtcars.csv")
```

```r
c1  mpg cyl disp hp drat    wt  qsec vs am gear carb
1 Mazda RX4 21.0   6 160 110 3.90 2.620 16.46  0  1    4    4
2 Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02  0  1    4    4
3  Datsun 710 22.8   4 108  93 3.85 2.320 18.61  1  1    4    1
4  Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44  1  0    3    1
```

[16]
Finally, the data need not be located on the local disk. We can also ask H2O to read in data from a URL as shown in this last example, which uses a dataset made available from the UCLA Statistical Consulting Group:

```r
h2obin <- h2o.importFile(
  path = "http://www.ats.ucla.edu/stat/data/binary.csv"
)
```

```
<table>
<thead>
<tr>
<th>admit</th>
<th>gre</th>
<th>gpa</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>380</td>
<td>3.61</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>660</td>
<td>3.67</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>800</td>
<td>4.00</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>640</td>
<td>3.19</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>520</td>
<td>2.93</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>760</td>
<td>3.00</td>
</tr>
</tbody>
</table>
```

[400 rows x 4 columns]

**Summary**

This chapter presented a brief introduction to NNs and deep neural networks. Using multiple hidden layers, deep neural networks have been a revolution in machine learning by providing a powerful unsupervised learning and feature extraction component that can be standalone or integrated as part of a supervised model.

There are many applications of such models, and they are being increasingly used by large companies such as Google, Microsoft, and Facebook. Examples of tasks for deep learning are image recognition (for example, automatically tagging faces, or identifying keywords for an image), voice recognition, and text translation (for example, to go from English to Spanish, or vice versa). Work is even being done on text recognition such as sentiment analysis to try to identify whether a sentence or paragraph is generally positive or negative, particularly useful for evaluating perceptions about a product or service. Imagine being able to scrape reviews and social media for any mention of your product and being able to analyse whether it was being discussed more or less favourably than the month or year before!
This chapter also showed how to set up R and the necessary software and packages installed, in a reproducible way to match the versions used in this book.

In the next chapter, we will begin to train neural networks and generate our own predictions.
Where to buy this book

You can buy R Deep Learning Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.