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

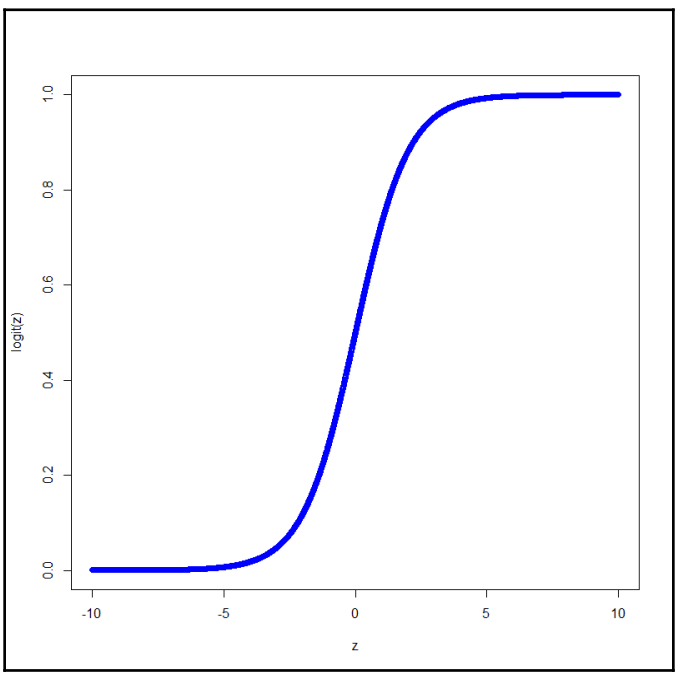
Logistic regression is a supervised machine learning approach for the classification of dichotomous/ordinal (order discrete) categories.

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:

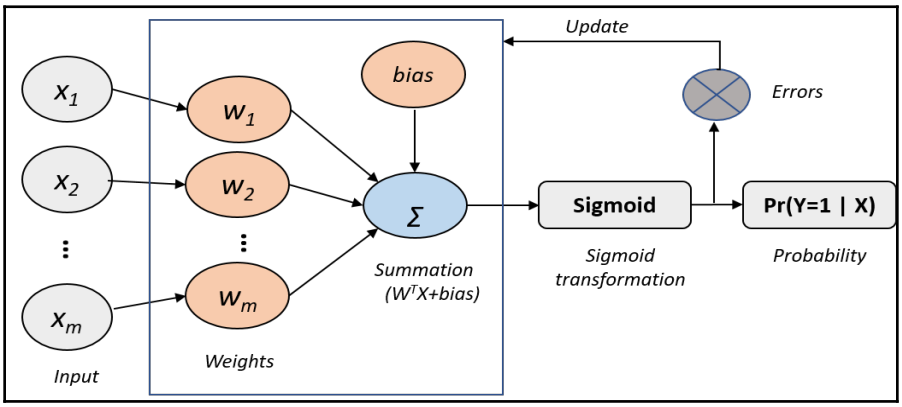


Sigmoid functional form

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 `h20` 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:

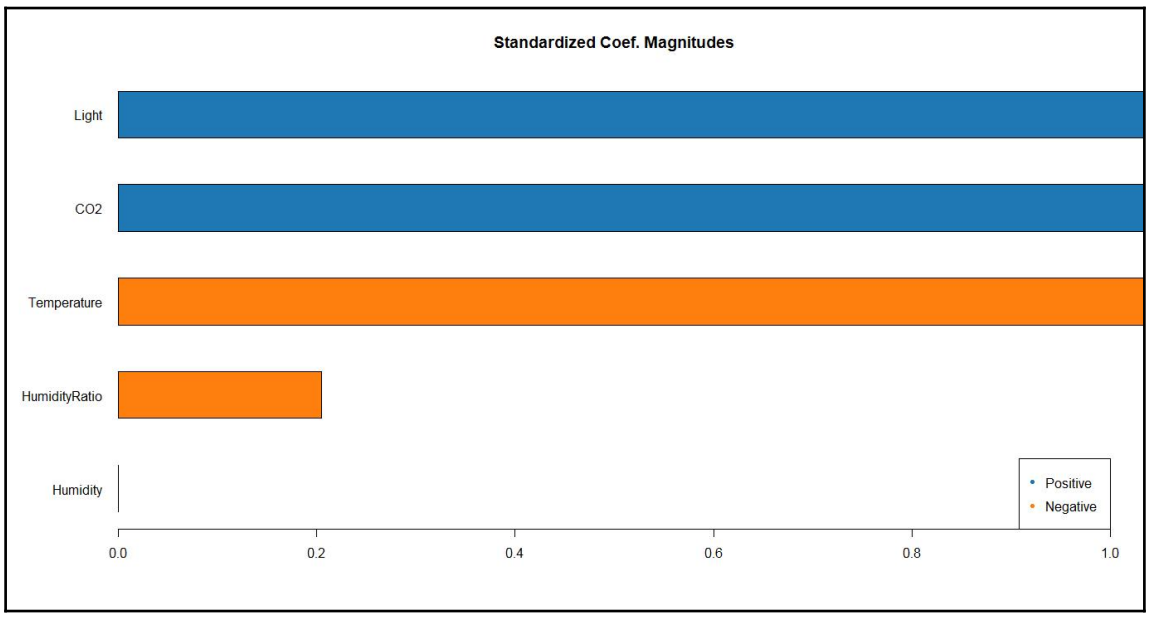
```
# Predict on test data
yhat <- h2o.predict(occupancy_train.glm, occupancy_test.hex)
```

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...

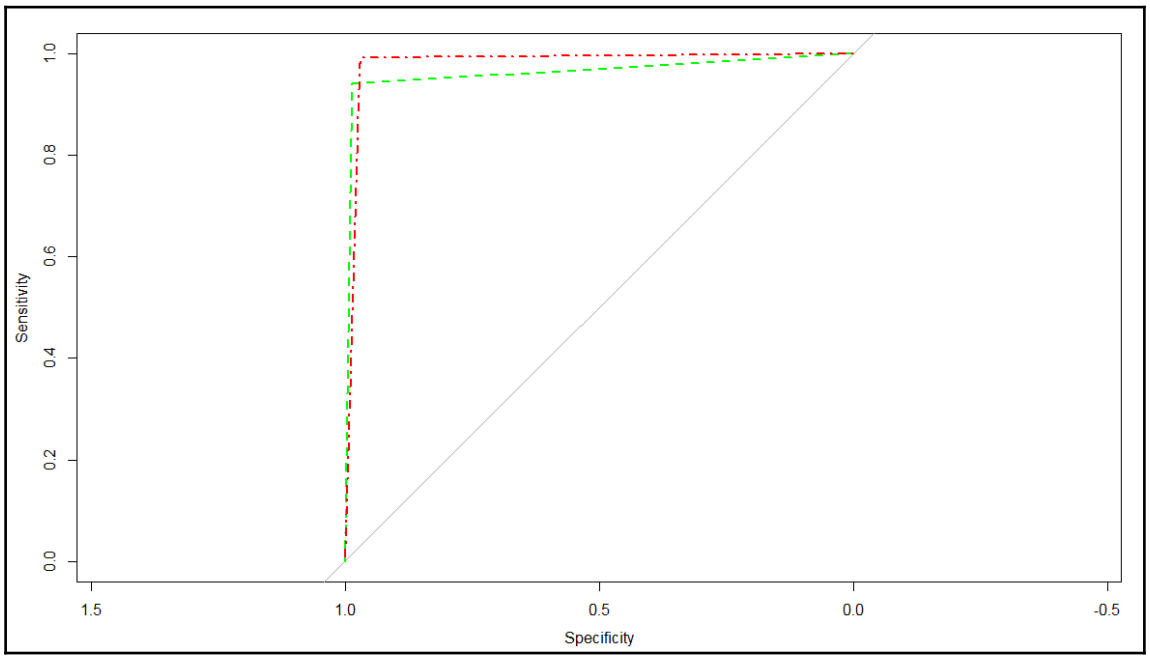
The performance of the model can be evaluated using AUC:

```
# 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:

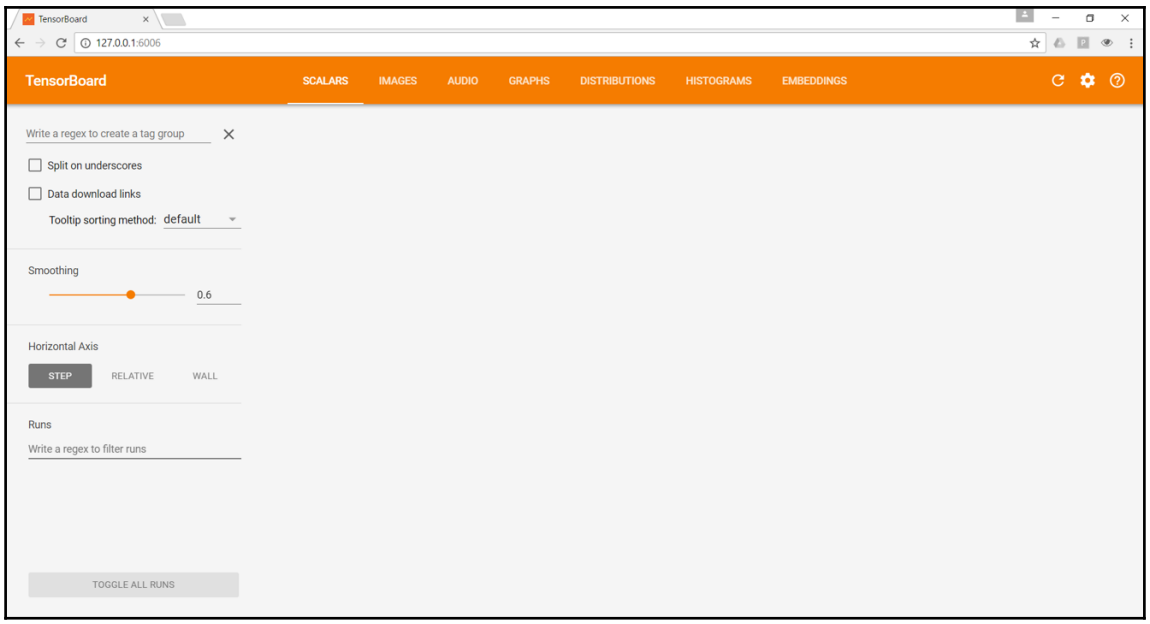
```
$ tensorboard --logdir home/log --port 6006
```

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.

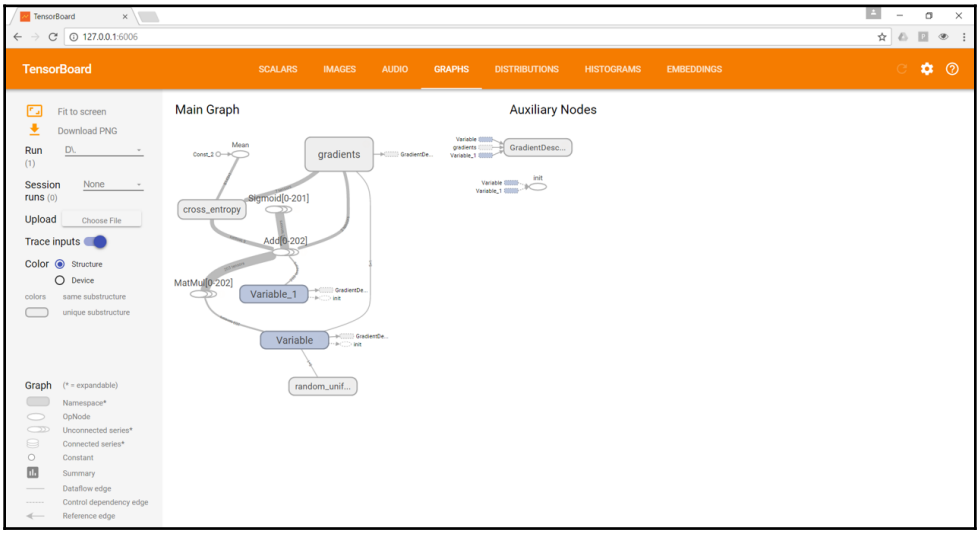
How to do it...

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:

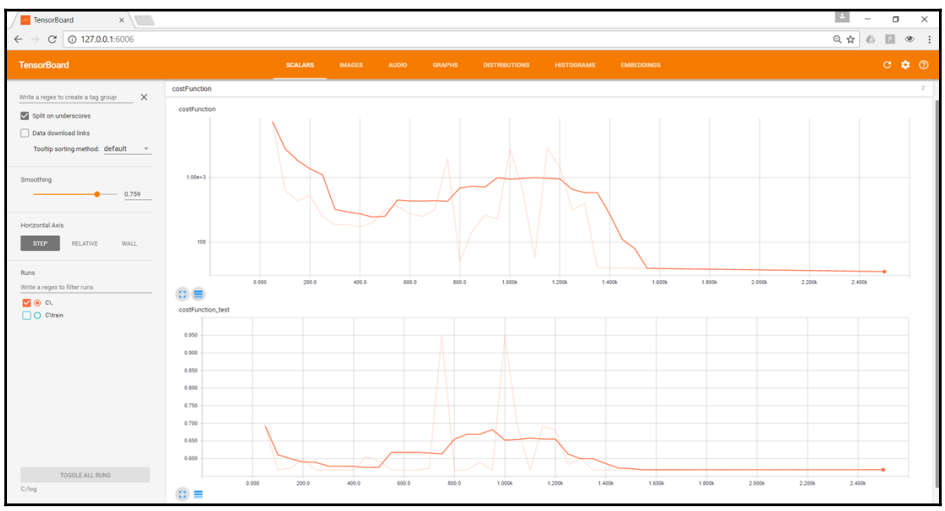
```
summary = tf$summary$merge_all()
```

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.

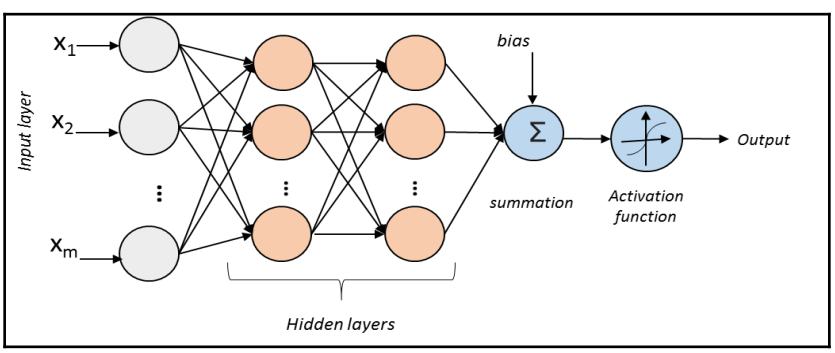
Starting with multilayer perceptrons

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.

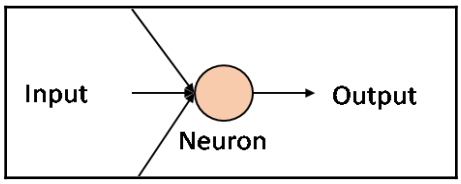
Getting ready

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...

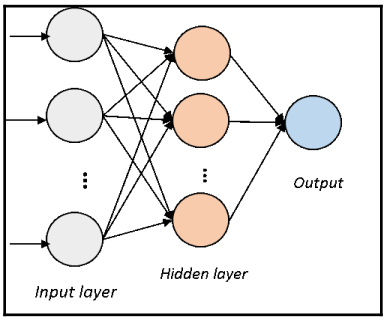
This section covers type activation functions in multilayer perceptrons. Activation is one of the critical component of ANN as it defines the output of that node based on the given input. There are many different activation functions used while building a neural network:

- **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...

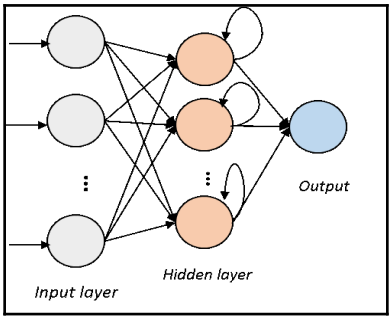
There are three main neural network architectures in neural networks:

- **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:



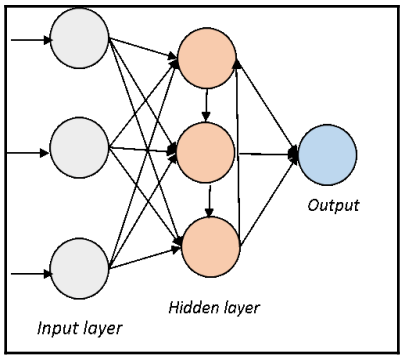
Feedforward architecture of neural networks

- **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...

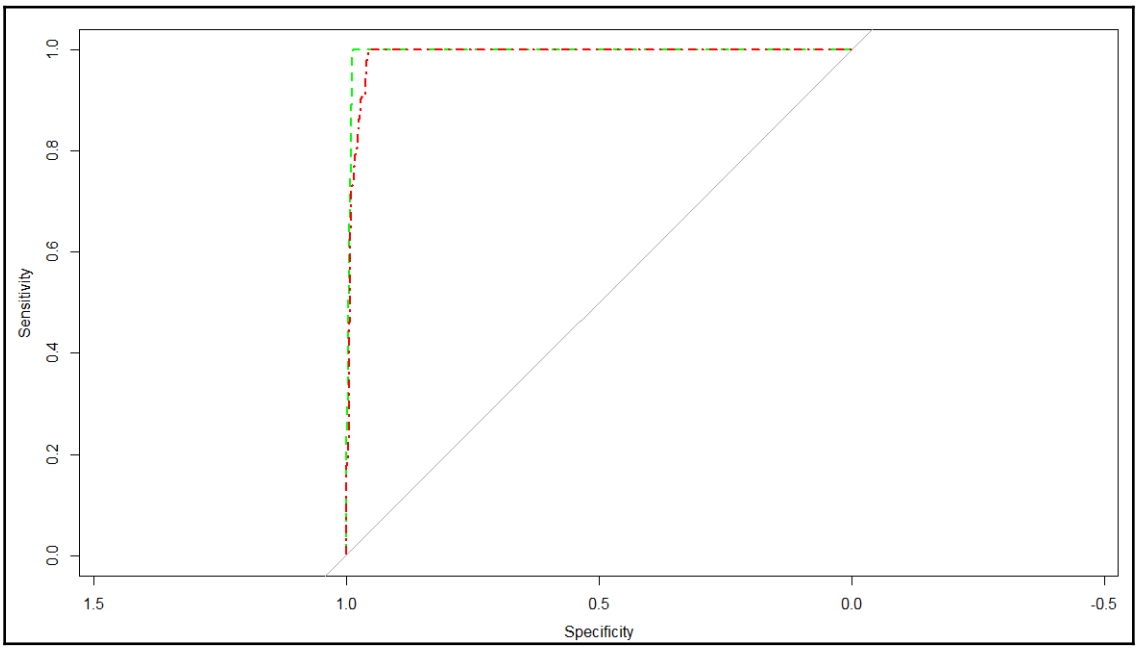
The performance of the model can be evaluated using AUC:

```
# 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