We define *history* as the sequences of observations, rewards, and actions that occur with respect to an agent and a given environment, denoted as $H_t = O_1, R_1, A_1, \ldots, O_{t-1}, R_{t-1}, A_{t-1}$. All subsequent observations, rewards, and actions are influenced by the history as it exists in a given experiment. This is what we define as the *state*, denoted as $S_t = f(H_t)$. The state of the environment is not visible to the agent, so it doesn't allow a bias for the actions an agent may pick. In contrast, the state of the agent is internal. The information also has a state, which is described as a *Markov process*. It should be noted that because this is an introductory book to deep learning, the application of reinforcement learning won't be as in depth as it would be in more advanced books. That said, it is my hope that from reading this book, those who currently find reinforcement learning problems inaccessible will be able to tackle these problems upon attainment of a solid understanding of the concepts addressed during the course of this text.

Summary

We now have reached the end of our review of the necessary components of optimization and machine learning. This chapter, as well as the prior chapter, should be used as a reference point for understanding some of the more complex algorithms we shall discuss in the chapters moving forward. Now, we'll progress into discussing the simplest model within the paradigm of deep learning: single layer perceptrons.

CHAPTER 4

Single and Multilayer Perceptron Models

With enough background now under our belt, it's time to begin our discussion of neural networks. We'll begin by looking at two of the commonest and simplest neural networks, whose use cases revolve around classification and regression.

Single Layer Perceptron (SLP) Model

The simplest of the neural network models, SLP, was designed by researchers McCulloch and Pitts. In the eyes of many machine learning scientists, SLP is viewed as the beginning of artificial intelligence and provided inspiration in developing other neural network models and machine learning models. The SLP architecture is such that a single neuron is connected by many synapses, each of which contains a weight (illustrated in Figure 4-1).

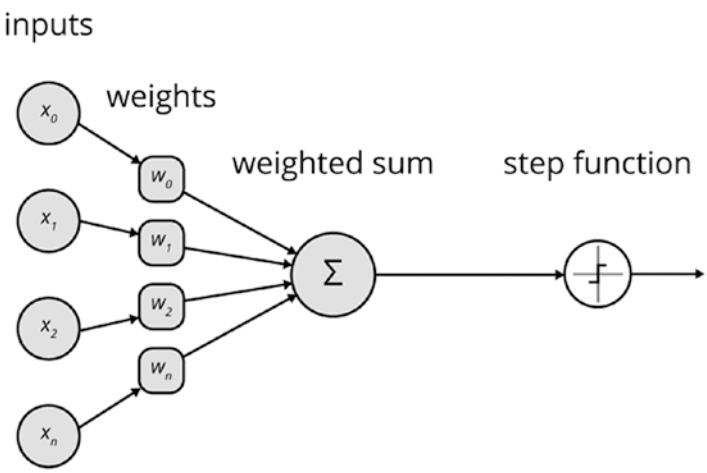

Figure 4-1. *Visualization of single perceptron model*

T. Beysolow II, *Introduction to Deep Learning Using R*, DOI 10.1007/978-1-4842-2734-3_4

The weights affect the output of the neuron, which in the example model will be a classification problem. The aggregate values of the weights multiplied by the input are then summed within the neuron and then fed into an activation function, the standard function being the logistic function:

Let the vector of inputs $x = [x_1, x_2, \ldots, x_n]^T$ and the vector of weights $w = [w_1, w_2, \ldots, w_n]$.

The output of the function is given by

$$y = f(x, w^T)$$

where the activation function, when using a logistic function, is the following:

$$f(x) = \frac{1}{1+e^{-x}}$$

Training the Perceptron Model

We begin the training process by initializing all the weights with values sampled randomly from a normal distribution. We can use a gradient descent method to train the model, with the objective being to minimize the error function. We describe the perceptron model as

$$\hat{y} = f(x, w^T) = \sigma\left(\sum_i^n x_i w_i\right)$$

where

$$\sigma = \frac{1}{1+e^{-x}},$$

$$\hat{y} = \begin{cases} 1 & if\ y \geq \pi^*, \\ 0 & elsewhere \end{cases}$$

where π^* = the threshold for log odds as described for logistic regression in Chapter 3.

Widrow-Hoff (WH) Algorithm

Developed by Bernard Widrow and Macron Hoff in the late 1950s, this algorithm is used to train SLP models. Though similar to the gradient method used to train neural networks (mentioned earlier), the WH algorithm uses what is called an *instantaneous algorithm*, given by

$$w_i(k+1) = w_i(k) - \eta\left(\frac{\partial E}{\partial w_i}\right)1$$

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\sum_{m=1}^{M} 1\left(h_{\theta_x} - y(k)\right)\left(-\frac{\partial y(k)}{\partial w_i}\right)$$

$$= \sum_{m=1}^{M}\left(h_{\theta_x} - y(k)\right)\left(-x_i(k)\right)$$

$$= \sum_{m=1}^{M}\delta(k)x_i(k)$$

where

$$\delta(k) = \left(h_{\theta_x} - y(k)\right)$$

We can therefore summarize the preceding equations into the following:

$$w_i(k+1) = w_i(k) + \eta\delta(x)x_i(k)$$

In this manner, we have the same optimization problem that we would in any traditional gradient method. Our goal is to minimize the error of the model by adjusting the weights applied to the inputs of data via gradient descent. With a classification problem in mind, let's use logistic regression as our baseline indicator while also comparing it to a fixed rate perceptron indicator and the bold driver adaptive gradient using the WH algorithm.

Limitations of Single Perceptron Models

The main limitation of the SLP models that led to the development of subsequent neural network models is that perceptron models are only accurate when working with data that is clearly linearly separable. This obviously becomes difficult in situations with much more dense and complex data, and effectively eliminates this technique's usefulness from classification problems that we would encounter in a practical context. An example of this is the XOR problem. Imagine that we have two inputs, x_1 and x_2 for which a response, y, is given, such that the following is true:

	x_1	$x_2 y$
0	0	0
1	0	1
0	1	1
1	1	0

From the following example, we can see that the response variable is equal to 1 when either of the explanatory variables is equal to 1, but is equal to 0 when both explanatory variables are equal to each other. This situation is illustrated by Figure 4-2.

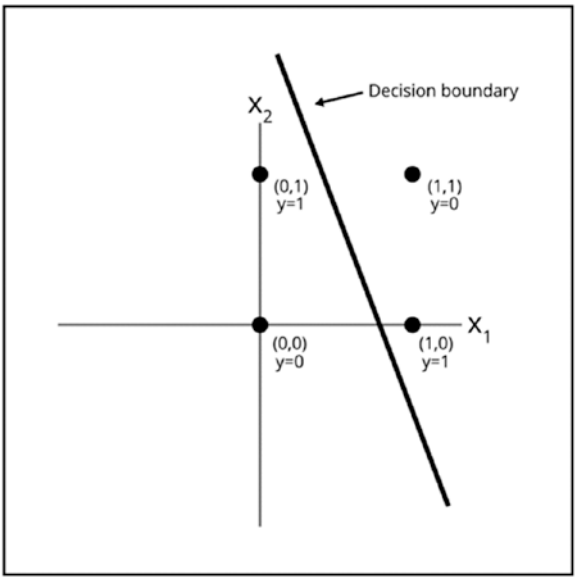

Figure 4-2. *XOR problem*

Let's now take a look at an example using SLP with data that is not rigidly linearly separable to get an understanding of how this model performs. For this example, I've created a simple example function of a single layer perceptron model. For the error function, I used 1 minus the AUC score, as this would give us a numerical quantity such that we could train the weight matrix via back-propagation using gradient descent. Readers may feel free to use the next function as well as change the parameters.

We begin by setting some of the same parameters that we did with respect to our linear regression algorithm performed via gradient descent. (Review Chapter 3 if you need to review the specifics of gradient descent and how it's applied for parameter updating.) The only difference here is that we're using a different error function than the mean squared error used in regression:

```
singleLayerPerceptron <- function(x = x_train, y = y_train, max_iter = 1000, tol = .001){
#Initializing weights and other parameters
  weights <- matrix(rnorm(ncol(x_train)))
  x <- as.matrix(x_train)
  cost <- 0
  iter <- 1
  converged <- FALSE
```

Here, we define a function for a single layer perceptron, setting parameters similar to that of the linear regression via the gradient decent algorithm defined in Chapter 3. As always, we cross-validate (this section of code redacted, please check GitHub) our data upon each iteration to prevent the weights from overfitting. In the following code, we define the algorithm for the SLP described in the preceding section:

```
  while(converged == FALSE){
        #Our Log Odds Threshold here is the Average Log Odds
      weighted_sum <- 1/(1 + exp(-(x%*%weights)))
      y_h <- ifelse(weighted_sum <= mean(weighted_sum), 1, 0)
      error <- 1 - roc(as.factor(y_h), y_train)$auc
}
```

Finally, we train our algorithm using gradient descent with the error defined as 1 – AUC. In the following code, we define the processes that we repeat until we converge upon an optimal solution or the maximum number of iterations allowed:

```
#Weight Updates using Gradient Descent
#Error Statistic := 1 - AUC
if (abs(cost - error) > tol | iter < max_iter){
        cost <- error
        iter <-  iter + 1
        gradient <- matrix(ncol = ncol(weights), nrow = nrow(weights))
        for(i in 1:nrow(gradient)){
          gradient[i,1] <- (1/length(y_h))*(0.01*error)*(weights[i,1])
        }
(Next section redacted, please check github!)
```

As always, it's useful for readers to evaluate the results of their experiment. Figure 4-3 shows the AUC score summary statistics in addition to the last AUC score with its respective ROC curve plotted:

```
  #Performance Statistics
  cat("The AUC of the Trained Model is ", roc(as.factor(y_h), y_train)$auc)
  cat("\nTotal number of iterations: ", iter)
  curve <- roc(as.factor(y_h), y_train)
  plot(curve, main = "ROC Curve for Single Layer Perceptron")
}
```

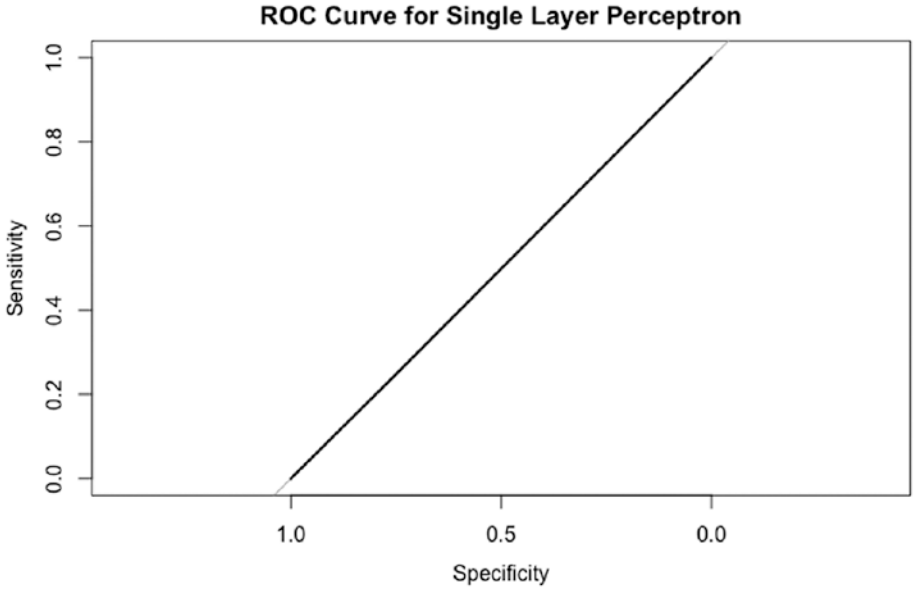

Figure 4-3. *ROC curve*

Summary Statistics

```
       Mean     Std.Dev       Min       Max     Range
1 0.4994949 0.03061466 0.3973214 0.6205357 0.2232143
```

Note that the AUC scores are considerably poor, with the average rating being no better than guessing. Sometimes the algorithm here reaches slightly better results, but this would still likely be insufficient for purposes of deployment. This is likely due to the fact that the classes aren't so clearly linearly separable, leading to misclassification with updates to the weight matrix upon each iteration.

Now that we've seen the limitations of the SLP, let's move on to the successor to this model, the multi-layer perceptron, or MLP.

Multi-Layer Perceptron (MLP) Model

MLPs are distinguished from SLPs by the fact that there are hidden layers that affect the output of the model. This distinguishing factor also happens to be their strength, because it better allows them to handle XOR problems. Each neuron in this model receives an input from a neuron—or from the environment in the case of the input neuron. Each neuron is connected by a synapse, attached to which is a weight, similar to the SLP. Upon introducing one hidden layer, we can have the model represent a Boolean function, and introducing two layers allows the network to represent an arbitrary decision space.

Once we move past the SLP models, one of the more difficult and less obvious questions becomes what the actual architecture of the MLP should be and how this affects model performance. This section discusses some of the concerns the reader should keep in mind.

Converging upon a Global Optimum

By the design of the model, MLP models are not linear, and hence finding an optimal solution isn't nearly as simple as it would be in the case of an OLS regression. In MLP models, the standard algorithm used for training is the back-propagation algorithm, an extension of the earlier described Widrow-Hoff algorithm. It was first conceived in the 1980s by Rumelhart and McClelland and was seen as the first practical method for training MLP networks. It's one of the original methods by which MLP models were trained by using gradient descent. Let E be the error function for the multi-layer network, where

$$E(k)=\frac{1}{2}\sum_{i=1}^{M}\left(h(k)_{\theta_i}-y_i(k)\right)^2$$

We represent the weighted sum value of the individual neurons that is inputted into the hidden layer by the following:

$$s(k)_{h,j}=\sum_{i=1}^{M}w_{h,j,i}x_i(k)$$

Similarly, we represent the output from the hidden layer to the output layer as the following:

$$s(k)_{o,j}=\sum_{i=1}^{H}w_{o,j,i}o_{h,i}(k)$$

With the weights represented by the following:

$$w_{ij}(k+1)=w_{ij}(k)-\eta\frac{\partial E(k)}{\partial w_{ij}}$$

Back-propagation Algorithm for MLP Models:

1. Initialize all weights via sampling from normal distribution.
2. Input data and proceed to pass data through hidden layers to output layers.
3. Calculate the gradient and update weights accordingly.
4. Repeat steps 2 and 3 until algorithm converges upon tolerable loss threshold or maximum iterations have been reached.

After having reviewed this model conceptually, let's look at a toy example. Readers interested in applications of multi-layer perceptrons to practical example problems should pay particular attention to Chapter 10. In the following section of code, we generate new data and display it in the following plot (illustrated in Figure 4-4):

```
#Generating New Data
x <- as.matrix(seq(-10, 10, length = 100))
y <- logistic(x) + rnorm(100, sd = 0.2)

#Plotting Data
plot(x, y)
lines(x, logistic(x), lwd = 10, col = "gray")
```

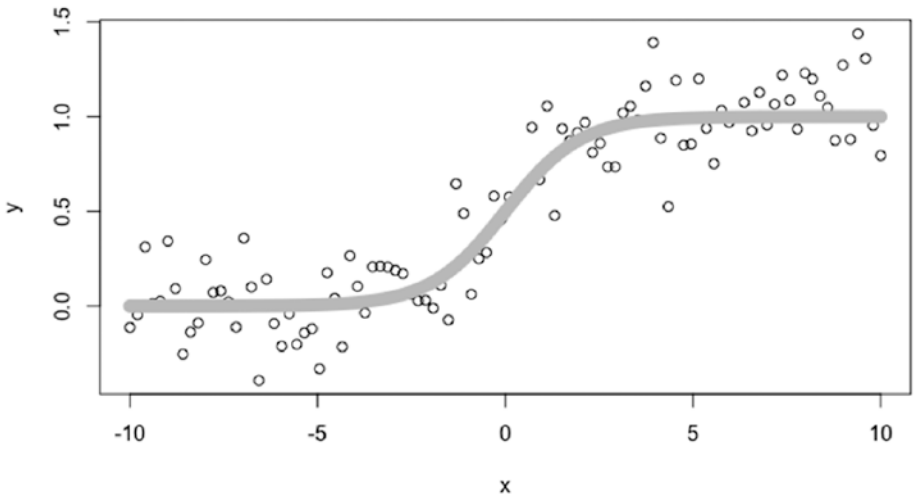

Figure 4-4. *Plotting generated data sequence*

Essentially, we have a logistic function around which the data is distributed such that there is variance around this logistic function. We then define the variable that holds the weights of the MLP model. I'm using the packaged monmlp, but users may also feel free to experiment with other implementations in packages such as RSNSS and h2o. Chapter 10 covers h2o briefly in the context of accessing deep learning models from the framework:

```
#Loading Required Packages
require(ggplot2)
require(lattice)
require(nnet)
require(pROC)
require(ROCR)
require(monmlp)
```

```
#Fitting Model
mlpModel <- monmlp.fit(x = x, y = y, hidden1 = 3, monotone = 1,
                       n.ensemble = 15, bag = TRUE)
mlpModel <- monmlp.predict(x = x, weights = mlpModel)

#Plotting predicted value over actual values
for(i in 1:15){
  lines(x, attr(mlpModel, "ensemble")[[i]], col = "red")
}
```

When plotting the predictions of the MLP model, we see the results shown in Figure 4-5.

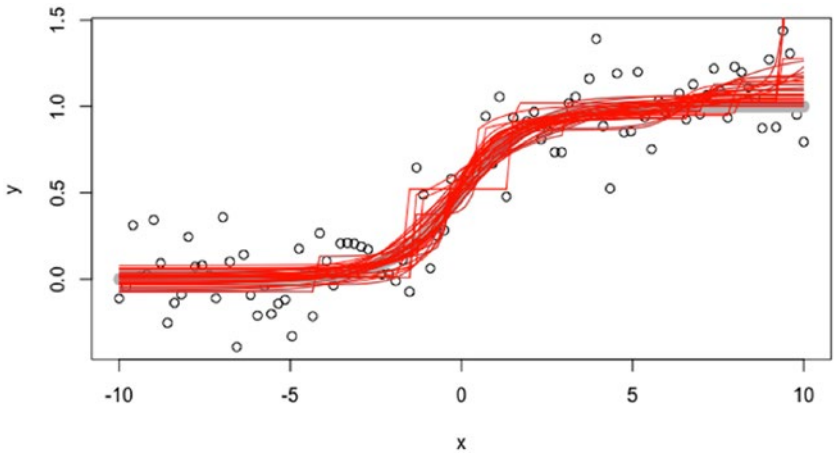

Figure 4-5. *Predicted lines laying over function representing data*

As you can see, there are instances in which the model captures some noise, evidenced by any deviations from the shape of the logistic function. But all the lines produced are overall a good generalization of the logistic function that underlies the pattern of the data. This is an easy display of the MLP model's ability to handle non-linear functions. Although a toy example, this concept holds true in practical examples.

Limitations and Considerations for MLP Models

It is often a problem when using a back-propagation algorithm, where the error is a function of the weights, that convergence upon a global optimum can be difficult to accomplish. As briefly alluded to before, when we are trying to optimize non-linear functions, many local minima obscure the global minimum. We can therefore be tricked into thinking we've found a model which can effectively solve the problem when in fact we've chosen a solution that doesn't effectively reach the global minimum (see Figure 4-6).