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).

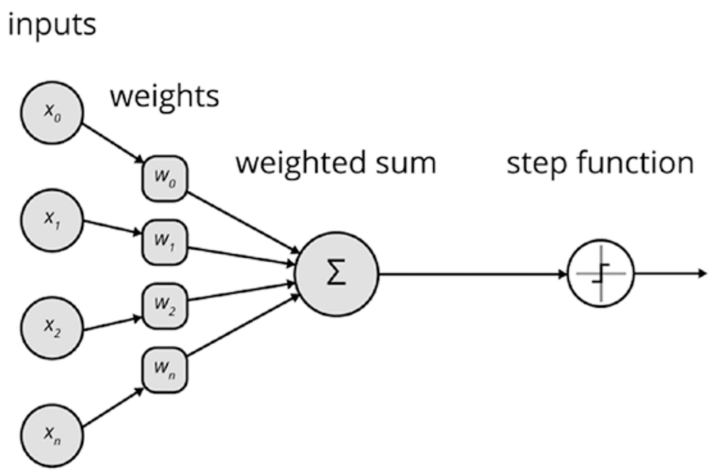

Figure 4-1. *Visualization of single perceptron model*

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

$$y = f\left(x, w^T\right)$$

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

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

where

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

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

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_2y
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.

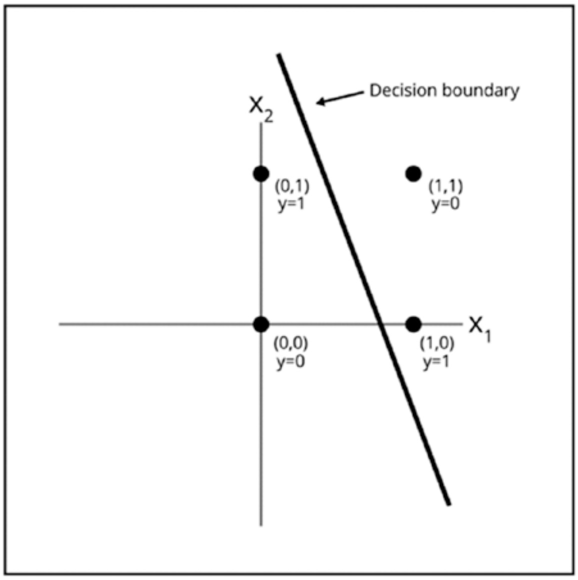

Figure 4-2. *XOR problem*

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")
}
```

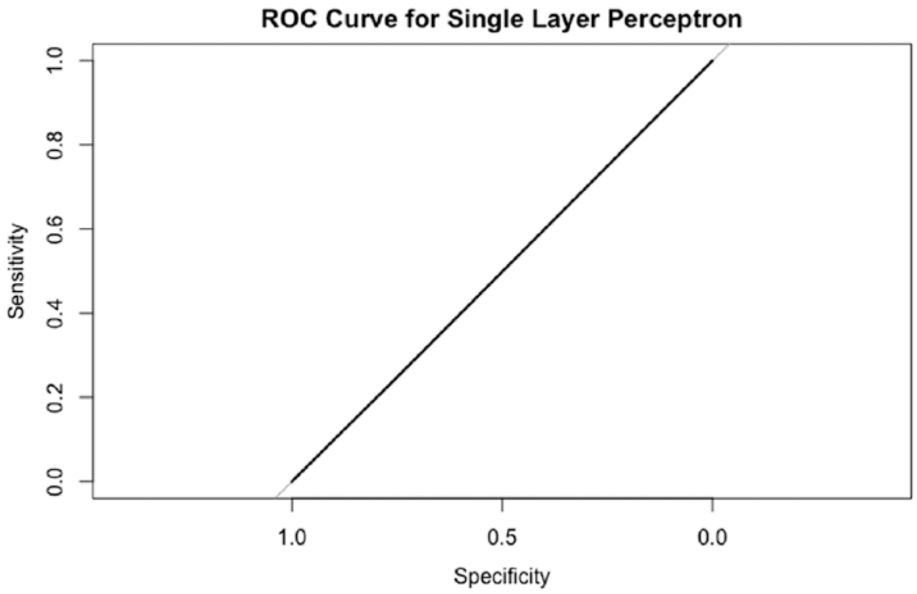

Figure 4-3. *ROC curve*

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")
```

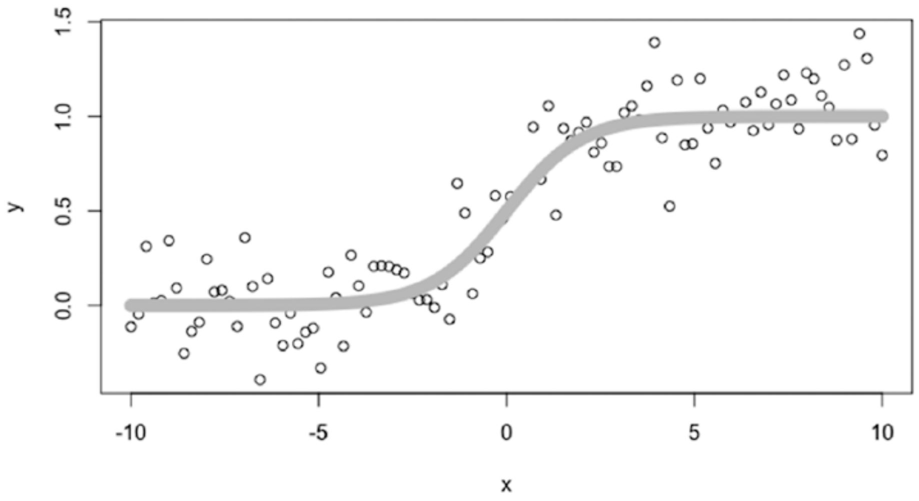

Figure 4-4. *Plotting generated data sequence*

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.

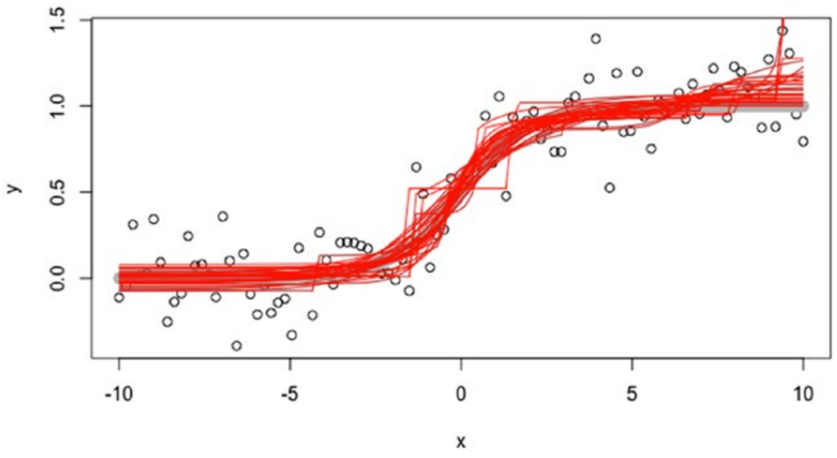

Figure 4-5. *Predicted lines laying over function representing data*

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).

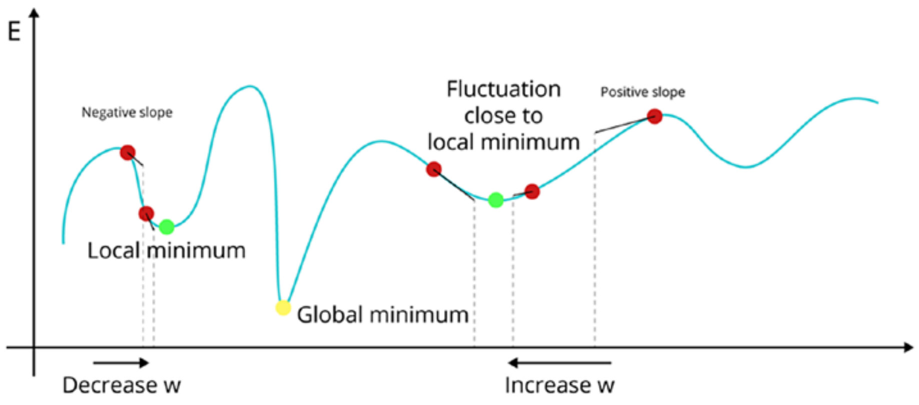

Figure 4-6. *Error over weight plot*

To alleviate this, the conjugate gradient algorithm is applied. *Conjugate gradient algorithms* differ from the traditional gradient descent method in that the learning rate is adjusted upon each iteration. Many types of conjugate gradient methods have been developed, but all of them have the same motivation underlying them. In the context of the MLP network, we're trying to find the weights that minimize the error function. To do this, we move in the direction of steepest descent, but we change the step size in such a way that it minimizes any possible "missteps" in searching for the global optimum. Let's take a simple example, where we're trying to solve

$$Ax = b$$

where x is an unknown vector, or weights vector in the context of the MLP network, A is the matrix of explanatory variables, and b is the response variable. Now look at the quadratic function

$$f(x) = \frac{1}{2}x^T Ax - b^T + c$$

where c is a constant scalar. When considering an example where A is positive-definite, the optimal solution for minimizing $f(x)$ is the solution to $Ax = b$. When calculating the gradient, we find that $f'(x) = Ax - b$, meaning that the direction of steepest descent would be equal to $b - Ax$. Therefore, we want to adjust the weight vector, x, with the following equation:

$$x_k = x_{k-1} - \eta(b - Ax)$$

The operative part of this method is the transformation of the learning rate, η. By definition, η minimizes the function when the directional derivative of the function with respect to the learning rate is equal to zero. According to the chain rule:

$$\frac{df(x)}{d\eta} = f'(x)^T(-AE), E = y - \hat{y}$$

Finally, we determine the learning rate to therefore be the following:

$$\text{Let } b - Ax = r$$

$$r_k^{\,T} r_{k\text{-}1} = 0,$$

$$(b - Ax_k)^{\mathrm{T}} r_{k\text{-}1} = 0,$$

$$(b - A(x_{k\text{-}1} + \eta\, r_{k\text{-}1}))^{\mathrm{T}} r_{k\text{-}1} = 0,$$

$$(b - Ax_{k\text{-}1})^T r_{k\text{-}1} - \eta\, (Ar_{k\text{-}1}))^{\mathrm{T}} r_{k\text{-}1} = 0,$$

$$(b - Ax_{k\text{-}1})^T r_{k\text{-}1} = \eta\, r_{k-1}^{\,T} (Ar_{k\text{-}1})$$

$$r_k^{\,T} r_{k\text{-}1} = \eta\, r_{k-1}^{\,T} (Ar_{k\text{-}1}),$$

$$\eta\ = \frac{r_k^T r_{k-1}}{(Ar_{k-1})}$$

How Many Hidden Layers to Use and How Many Neurons Are in It

We typically choose to use hidden layers only in the event that data is not linearly separable. Whenever step, heaveside, or threshold activation functions are utilized, it is generally advisable to use two hidden layers. With respect to using more than one hidden layer, it's largely unnecessary because the increase in performance from using two or more layers is negligible in most situations. In situations where this may not be the case, experimentation by observing the RMSE, or another statistical indicator, over the number of hidden layers should be used as a method of deciding. Often, when adding a layer to a neural network model, this will be simple as editing an argument in a function or, in the case of some deep learning frameworks such as mxnet (featured in later chapters), passing values from a previous layer through an entirely new function. With respect to how many neurons should be within a given hidden layer, this must be tested for with the objective of minimizing the training error. Some suggest that it has to be between the input and output layer size, never more than twice the number of inputs, capturing .70-.90 variance of the initial data set—or to use the following formula: $\#\, Hidden\, Units = (\#\, inputs + \#\, outputs) * \frac{2}{3}$.

Briefly, let's look at the difference between the conjugate gradient training method and traditional gradient descent using the RNSS package in R with the following code:

```
#Conjugate Gradient Trained NN
conGradMLP <- mlp(x = x, y = y,
size = (2/3)*nrow(x)*2,
                  maxit = 200,
                  learnFunc = "SCG")
#Predicted Values
y_h <- predict(conGradMLP, x)
```

We begin by defining the neural network using the `mlp()` function, in which we specifically denote the `learnFunc` argument as SCG (scaled conjugate gradient). We also choose the `size` parameter (the number of neurons in a neural network) using the 2/3 rule mentioned earlier.

Now let's compare the MSE of both the MLP model shown prior and this one we've just constructed:

> MSE for Conjugate Gradient Descent Trained Model: 0.03533956
>
> MSE for Gradient Descent Trained Model: 0.03356279

Although only a slight difference in this instance, we can see that the conjugate gradient method yields a slightly inferior MSE value than the traditional gradient descent method in this instance. As such, it would be wise, given this trend of staying consistent, to pick the gradient descent trained method.

Summary

This chapter serves as an introduction into the world of neural networks. Moving forward, we will discuss models that have been developed for tasks that are generally beyond what SLP and MLP models are made for. Specifically, in Chapter 5, we will look at convolutional neural networks for image recognition as well as recurrent neural networks for time series prediction. Readers who don't feel comfortable yet with the concepts discussed in this chapter are advised to review Chapters 2 though 4 again before advancing to Chapter 5, because many of the concepts referred to in Chapter 5 are addressed at length in those chapters.

CHAPTER 5

Convolutional Neural Networks (CNNs)

Similar to the concepts covered in Chapter 4 with respect to the multi-layer perceptron problem, convolutional neural networks (CNNs) also feature multiple layers used to calculate the output given a data set. This model's development can be traced back to the 1950s, where researchers Hubel and Wiesel modeled the animal visual cortex. At length in a 1968 paper, they discussed their findings, which identified both simple cells and complex cells within the brains of the monkeys and cats they studied. The simple cells, they observed, had a maximized output with regard to straight edges that were observed. In contrast, the receptive field in complex cells was observed to be considerably larger, and their outputs were relatively unaffected by the positions of edges within the aforementioned receptive field. Beyond image recognition, for which CNNs originally gained and still retain their notoriety, CNNs have considerable other applications, such as within the fields of natural language processing and reinforcement learning.

Structure and Properties of CNNs

CNNs are, broadly speaking, multi-layer neural network models. In keeping with the structure of the animal visual cortex as described by Hubel and Weisel, the model can be visually interpreted as shown in Figure 5-1.

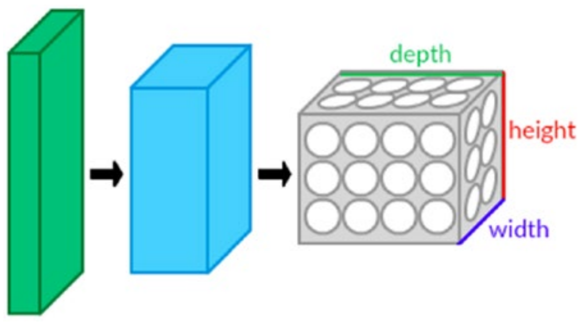

Figure 5-1. *Broad visual display of a CNN*

T. Beysolow II, *Introduction to Deep Learning Using R*, DOI 10.1007/978-1-4842-2734-3_5

Each block represents a different layer of the CNN, which I explain in greater detail later in this chapter. From left to right are the input, hidden (convolutional, pooling, and dropout layers), and fully connected layers. After the final layer, the model outputs a classification. Now consider Figure 5-2.

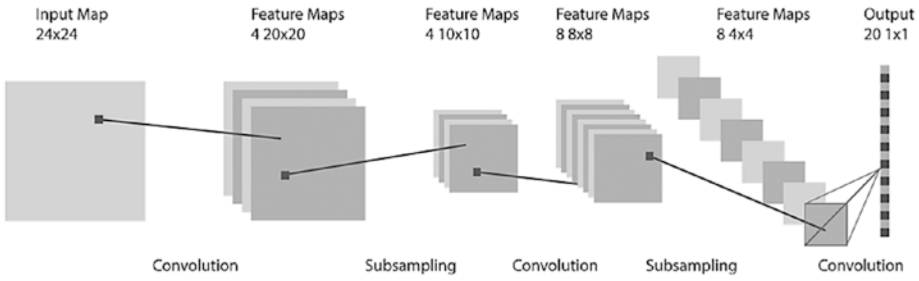

Figure 5-2. *CNN architecture diagram*

Fully connected layers enforce local connectivity between neurons and adjacent layers, as show in Figure 5-2. As such, the inputs of hidden layers are a subset of neurons from the layer preceding that hidden layer. This ensures that the learned subset neurons produce the best possible response. Also, the units share the same weight and bias in the feature/activation map, so that all the neurons in a given layer are analyzing/detecting the exact same feature.

As for *features* in the context of CNNs, I mean portions of an image that are distinct. This is what our filter compares the section of the image it is analyzing to, such that it can determine the degree to which the section of the image being scanned over is similar to the feature being analyzed. Assuming that we have enough training data and enough classes of images, these features are distinct enough that they help to distinguish one class from another.

Imagine we're looking at two images, specifically an X and O, such as in Figure 5-3.

Figure 5-3. *O and X example photo*

If we image both the X and the O as distinct images, to the human eye we can determine them as distinctly different letters. Among their distinguishing factors are that the center of the O is empty, whereas the center of the X features two intersecting lines. Examples of the feature maps of these values when visualized are shown in Figures 5-4 and 5-5.

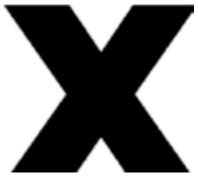

Figure 5-4. *Feature map of "X"*

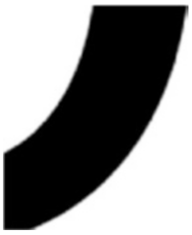

Figure 5-5. *Feature map of "O"*

These values are often represented as an entry within a matrix with –1 and 1 for black and white respectively. When dealing with color images, each pixel is typically represented as an entry in a matrix with a value of 1 or 256 for black and white respectively. Depending on the language being used, though, zero indexing may affect the representation of RGB values such that the bounds shift backwards by 1.

Components of CNN Architectures

This section covers the components of CNN architectures.

Convolutional Layer

This layer is where the majority of the computation in any given CNN occurs and as such is the first layer after input that an image passes through. Within a convolutional layer, we have filters that scan over a portion of the image. Every filter is not particularly large with respect to height and width, but all of them extend through the entirety of the length of this layer.

For example, imagine we're trying to classify an image as either a 1 or a 0, and the image in actuality is a 1. And imagine that the image has a black background, but the digit is outlined in white pixels. Figure 5-6 shows an example of what this image can be said to look like.

Figure 5-6. *Example image of "1"*

The computer will distinguish the white pixels as having a value of 1 and the black pixels as having a value of –1. When we input this image through the convolutional layer, the model extracts the unique features of an image, which usually are the colors, shapes, and edges that ultimately define a specific image. Once we have the features of a given training image, we perform what is known as filtering over this inputted image. *Filtering* is the process of taking an image feature, which in this case we can imagine as a 3 x 3 pixel square, and matching it with a patch of that inputted image, which is also a 3 x 3 pixel square. In Figure 5-7, we can see what the process of filtering looks like.

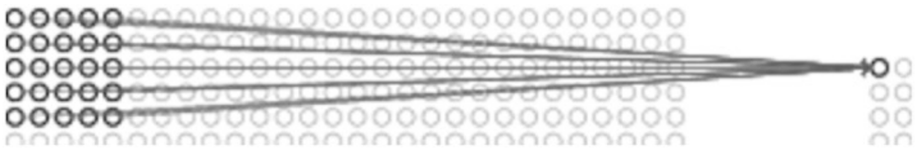

Figure 5-7. *Example of a filter*

We then multiply the number of the pixel of the feature by the corresponding number of the pixel of the image patch. In the example, we should gain an output of 1 or –1 for each operation. Intuitively, when the pixels match exactly, they should output to 1, and when they don't, they should output to –1. At the end, we take the average of the pixel products. If an image matches exactly, the average should be 1. If it doesn't, it will be considerable degrees lower than one. In this instance, imagine that the image patch and feature selected don't match at all. As such, when we take the average, it should output to –1. We place this product in the center of the position of the image patch we are analyzing with a given feature on what is called a *feature map*. This ultimately will be the output of the convolution layer and will be used in the following layer. The convolutional layer will, over different iterations, produce multiple feature maps. The process of matching a feature with a given image patch over every possible position is known as *convolving* an image. We denote the feature/activation map for a given CNN as

$$h_{i,j}^{k} = \tanh\left(\left(w^{k}x\right)_{i,j} + b_{k}\right)$$

where w^k is the weight, b_k is the bias, x is the value of the specific pixel, and tanh is for non-linearities in data. The subscripts *i,j* refer to the entry of the matrix that represents the feature/activation map. The weight, w^k, is ultimately what connects the pixels in the feature maps to the preceding layer. The convolution layer, ultimately, is a stack of the feature maps that were yielded from the operation described earlier. We then put the feature maps into the pooling layer. We calculate the spatial size of the output volume as

$$Spatial\ Size_{Output} = \frac{W - F + 2P}{S + 1}$$

where W = input volume size, F = size of receptive field of in convolution layer, P = amount of zero padding, and S = stride.

Pooling Layer

Between successive convolutional layers, it's common to place what is called a *pooling layer* in between. Simply stated, the pooling layer takes the feature maps produced in the convolution layer and "pools" them into an image. The pooling layer effectively performs dimensionality reduction, hence the prior emphasis on spatial representation, thereby reducing the complexity of the model. This can be compared to the process of pruning in decision trees and similarly helps to prevent overfitting of a given model. In the prototypical CNN model, the pooling layer has a 2 x 2 filter, a stride of 2, and every depth slice in the input is downsampled such that we move by 2 pixels with respect to height and width. These operations in the pooling layer help to discard 75% of the feature/activation map. This layer uses a max operation, which in the aforementioned example would be taking a max over 4 numbers, or the 4 pixels in any given feature/activation map.

In keeping with the example described earlier, imagine that with a 2 x 2 filter, we're looking at 9 x 9 feature map, where we're analyzing the top lefthand corner with the following scores:

.88	0
0	.95

When using the max operation, we would choose .95, because it's the maximum value within the 2 x 2 window. Because we have a stride of 2, we move 2 pixels to the right, which should mean that we're looking at a 2 x 2 slice of the image where the top lefthand corner of the slice should be the third column of the feature map until we have a max pooled image, which is significantly reduced and therefore removes unnecessary complexities of the model. As a direct consequence of the max operation used in this layer, we needn't be as precise as the prior layer when analyzing the image, and therefore this helps to make a more robust model that can more easily classify inputs. What I mean by this specifically is that the values of the weights connecting each layer can be more generalized to all the training data that they have been exposed to, rather than overfitting in such a manner where the CNN wouldn't perform well out of sample.

The function that determines the spatial size of the output is given by

$$Spatial\ Size_{Output} = W_2\ x\ H_2\ x\ L_2,$$

where

$$W_2 = \frac{w_1 - F}{s+1},\ H_2 = \frac{H_1 - F}{S+1},\ L_2 = L_1$$

Rectified Linear Units (ReLU) Layer

Rectifiers are used as another term for an activation function. Typically, we apply the following function to the inputs to this layer

$$f(x) = \max(0,\ x)$$

where x = input to a neuron.

When applied to the feature map, we can imagine that any of the values of the feature map that would be negative now are zero. Specifically, this helps outline the feature map closer to the image it's most associated with. We do this to all of the feature maps to then get a "stack" of images.

Fully Connected (FC) Layer

Any neurons in this layer are connected to all the activation maps in the preceding layer. This layer is usually placed after a user-determined amount of convolutional, pooling, and ReLU layers. The images inputted to this layer will be significantly smaller than the original inputs due to the image reductions specified in the prior operations. In this layer, we scan the reduced images, which should correspond to each feature map, and turn each of the values given here into a list of values. This list then corresponds to one of the k images we put in. Following from the example used in the beginning of the chapter, we originally inputted a 1. After moving through all the layers, we take the average of the scores corresponding to this image, and then this is the probability of the image being a 1 or a 0. It should be noted that the only difference between this layer and the convolutional layer is the fact that the convolutional layers are only connected to a local region in the input and that many of the neurons in a convolutional layer volume share parameters. With this in mind, we can also convert between FC and convolutional layers when constructing a given architecture.

Loss Layer

This layer is where we compare the predicted labels from the actual labels of the images. When trying to classify and object from k possible feature levels, we would use a softmax loss classifier. Using a Euclidean function is also common for the purpose of regressing against the labels of the specific images. Their functions are given by the following:

I. Softmax loss function:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

II. Euclidean loss function:

$$E = \frac{1}{2N}\sum_{i=1}^{N}\left\|\hat{y}_i - y_i\right\|_2^2$$

III. Softmax normalization:

$$x_i' = \frac{1}{1+e^{-\left(\frac{x_i-\mu_i}{\sigma_I}\right)}}$$

When using the back-propagation algorithm, we make a confusion matrix comparing a 1 or a 0, where we subtract the label of the answer to the probability assigned. Following the example that we've been using, let's say 1 = 1, and 0 = 0, but when we input a 1, we only receive a probability of .85 that it is a 0, and a probability of .45 that it is a 1. Therefore, we would have a cumulative error of –.60. We then adjust each feature maps pixel, through the weights displayed in the fully connected layer, using a gradient method as described before, with a designated learning rate. We initialize the weights at 0 and stop the CNN at the point at which the loss tolerance has been reached or the maximum iterations threshold has been reached. The same considerations for convergence upon an optimal solution as described in prior chapters must be taken into consideration.

Tuning Parameters

Images sent to the input layer should be divisible by 2 more than once. Common image dimensions are 32 x 32, 64 x 62, and so on. Convolutional layers should have filters with dimensions of 3 x 3 or 5 x 5 at most, and zero-padding should be performed in such a way that it doesn't alter the spatial dimensions of the input. For the pooling layers, their dimensions should be 2 x 2 with a stride of 2 most often. With these parameters, 75% of the activations will be discarded. Pooling layers that are larger than 3 result in too much loss in the classification process. When describing neurons and their arrangements, hyper-parameters are most relevant to this conversation. Specifically, I will be referring to stride, depth, and zero-padding. Among the most important parameters in CNNs, *stride* is a fixed parameter that determines the number of pixels that slide through a filter. For example, if the stride is 2, then 2 pixels at a time slide through the filters. Typically, stride is no greater than 2, and no less than 1. *Zero-padding* is the size of the zeroes around the border of the input volume. Through controlling zero-padding, we can more carefully control size of the activation maps, and other outputs, from layer to layer. Finally, *depth* refers to the number of filters we choose for a given experiment, each of which is what ultimately searches over each image in the convolutional layer.

Notable CNN Architectures

- *LeNet*: Developed in the 1990s by renowned deep learning researcher Yann LeCun, LeNet is a relatively simple architecture, all things considered. The purpose of this model was originally to classify digits, read zip codes, and perform general simple image classification. This is considered the analogue to a "Hello World" program that any developer first writes in a given language, because it's considered to be the first successful CNN application to a practical task. As Figure 5-8 illustrates, the layers involved are as follows:
    - input, conv layer, ReLU, pooling layer, conv layer, ReLU, pooling layer, fully connected, ReLu, fully connected, and softmax classifier.

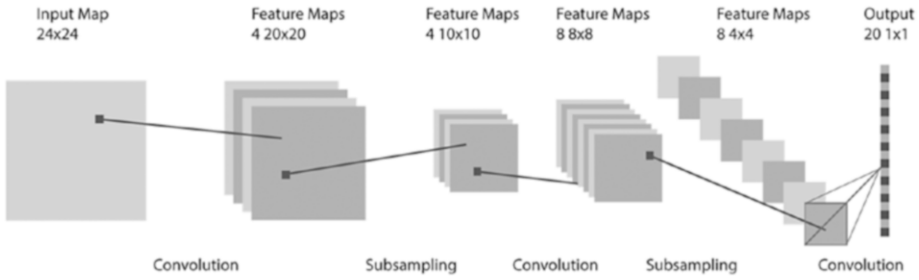

Figure 5-8. *Visualization of LeNet*

- *GoogLeNet (Inception)*: This architecture won the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) competition in 2014 in homage to Yann LeCun's LeNet. It was developed by Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Ehran, Cinven Vanhoucke, and Andrew Rabinovich. The name GoogLeNet is derived from the fact that a considerable number of the developers of the architecture work at Google Inc. In their paper "Going Deeper with Convolutions," they describe an architecture that allows for "increasing the depth and width of the network while keeping the computational budget constraint." As Figure 5-9 illustrates, the structure as proposed is as follows:
    - Input, conv. layer, max pool, conv layer, pooling layer (with max function), inception (2 layers), max pool, inception, inception (5 layers), max pool, inception (2 layers), average pooling layer, dropout, ReLU, softmax classifier.

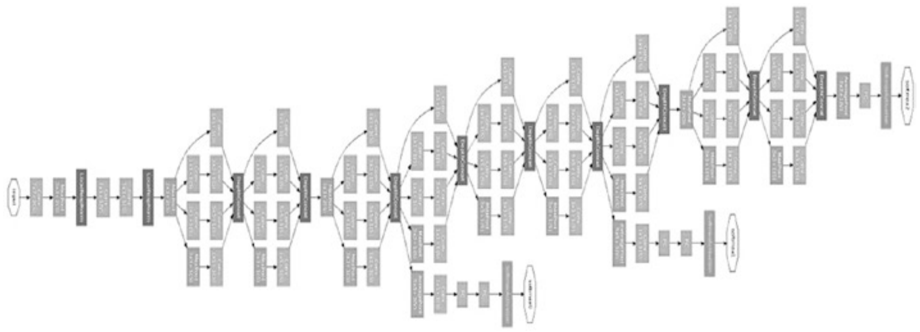

Figure 5-9. *GoogLeNet architecture*

The focus of the inception architecture is that through the orientation in the layers as described earlier, the CNN model allows for "increasing the number of units at each stage" without doing so to the point where the model becomes too complex. Overall, the model seeks to process visual information at various scales and then aggregate the calculations to the next stage so higher levels of abstraction are analyzed simultaneously.

- *AlexNet*: Developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, this won the ILSVRC in 2012. Similar in architecture to LeNet, AlexNet uses “non-saturating neurons” and efficiently implements the GPU for the convolution layers. The neurons in fully connected layers are connected to all neurons in the previous layer, response-normalization layers follow the first and second convolutional layers, and the kernel of layers two, four, and five are connected only to the kernel maps in the previous layer, which would be on the same GPU. The architecture is as follows (and shown in Figure 5-10):
    - Convolutional (5 layers), fully connected (3 layers), [output is 1000-way softmax classifier]

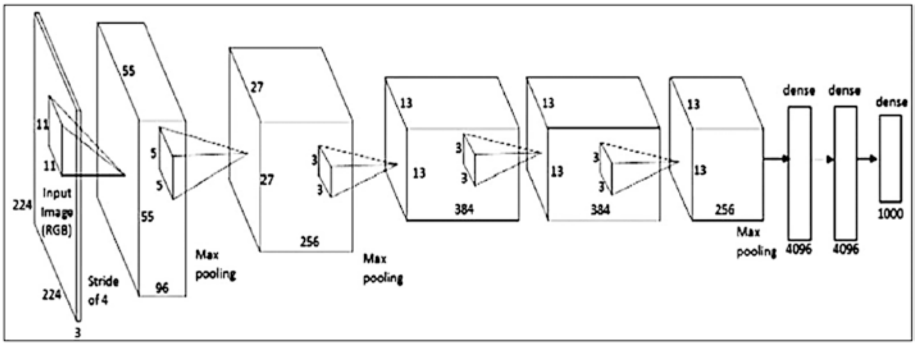

Figure 5-10. *AlexNet architecture*

- *VGGNet*: This took second place to AlexNet in the ILSVRC 2014 competition. VGGNet was developed by Karen Simonyan and Andrew Zisserman from the University of Oxford. The receptive field is 3 x 3, with 1 x 1 filters, stride is 2, and max pooling size is 2 x 2. The architecture is such that the input is fed through several convolutional layers, to three fully connected layers (the first and second layers have 4096 channels, and the final is a softmax layer that performs 1000-way classification).
- *ResNet*: The first-place winner of ILSVRC 2015, ResNet features 152 layers—far exceeding the amount of the previously mentioned networks. It was developed by Kaimin He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, all of whom are from Microsoft Research. The purpose of this architecture is to form a network that learns residual functions with references to the layer inputs, rather than a network learning unreferenced functions. The end result is a network that is considerably easier to learn, significantly easier to optimize, and that gains accuracy from increased depth, rather than one that loses accuracy from that depth.

Regularization

When multi-layer perceptrons have more than one layer, they are known to have the ability to approximate a given target, which then would lead to overfitting. To prevent overfitting, *regularizing* the input data is often recommended, however this is a slightly different process in the case of CNNs, we can use: 1) *DropOut*, which is taken from the inspiration of a phenomena observed within the human brain. This is where a given hidden layer has the probability of not being passed through with the probability we set as a hyperparameter. 2) Stochastic pooling, where the activation is picked randomly. *Stochastic Pooling* doesn't require hyper-parameters and can be used as a heuristic, so to speak, with other regularization techniques. 3) *DropConnect*, which is a generalization of dropout, where each connection can be dropped with the probability of 1 – p. Each unit in this layer inputs data from random units in the preceding layer, which change upon every iteration. This helps ensure that the weights don't overfit. 4) Weight Decay, which functions similarly to L1/L2 regularization, where we heavily penalize large weight vectors.

Of these methods, there has been a considerable amount of enthusiasm around using DropOut in CNNs, because it's been shown to be an effective and powerful technique. Beyond preventing overfitting, DropOut has been observed to improve the computational efficiency of networks with large amounts of parameters, as this form of regularization causes a network to in effect become smaller during a given iteration. After all these iterations, the smaller networks' performance can be averaged into a general prediction of what a complete network would have performed as. Secondly, it is observed that the DropOut layer introduces randomized performance in the network that allows noise within the data to be averaged over, such that its masking of signals within the data is diminished.

It's not uncommon to use L1 regularization either, but be aware of the fact that the weight vectors in this instance can often shrink to 0—sometimes enough so that we can be left with a sparsely populated weight matrix. The negative effect of this type of regularization is that the inputs to certain layers that contain important information may become entirely unnoticed due to a "dead" connection between layers. In contrast, though, when you feel specifically that you want very explicit feature selection, L1 regularization may yield significantly better performance.

L2 regularization is traditionally seen as the standard method by which regularization is performed in CNNs, because it tends to penalize abnormally large weights and favor those that are generally mild in their proportion relative to the entirety of the matrix. In contrast to L1 regularization, you get a considerably more populated weight matrix, which will cause the network to feed more data from a given layer to the next. As such, feature selection will be less stark than when using L1 regularization.

The final type of regularization you should know about would be an addendum to either L1 or L2 regularization via enforcing limits on a given weight's norm size. As such, this would allow the parameter updates to have a hard limit and therefore limit the number of possible solutions a given network can yield. This would help to train the network faster via the limitation of possible solutions, and in the optimal solution prevent the parameters from updating too far in the incorrect direction.

Summary

We have sufficiently covered the concepts of CNNs and walked through all the architectures that are most recent at this time. See Chapter 11 for an applied example of a CNN, specifically with respect to the preprocessing of image data—a highly important step in the constructing of image recognition software. Moving forward, we will discuss recurrent neural networks (RNs) and the intricacies of detecting patterns in time series–based data.