CHAPTER 6

Recurrent Neural Networks (RNNs)

Recurrent neural networks (RNNs) are models that were created to tackle problems within the scope of pattern recognition and are fundamentally built on the same concepts with respect to feed-forward MLPs. The difference is that although MLPs by definition have multiple layers, RNNs do not and instead have a directed cycle through which the inputs are transformed into outputs. I'll begin the chapter by covering several RNN models and end it with a practical application of RNNs.

Fully Recurrent Networks

Imagine that we have an input, x, that we're inputting into an RNN model, where we define the state as h, with the inputs being multiplied by a weight matrix, W. So far, everything is the same as it would be in previously described neural network models—but as stated before, RNNs perform the same task on the inputs over time. Because of this, to calculate the current state of a neural network, we derive the following equation:

$$h_t = f\left(W_t x + W_R h_{t-1} + b\right),\ where\ f = tanh, ReLU$$

$$y_t = f\left(Wh_t + b\right),\ where\ W_{t,r} = weights,\ h_t = hidden\ layer,\ b = bias$$

The key characteristic here is that when the neural network performs these operations, it "unfolds" into multiple new states, each of which is dependent on the prior states. Because these networks perform the same task for every input that's put in, in addition to the functional dependency of the model, RNNs are often referred to as *having memory*. Figure 6-1 illustrates an RNN.

T. Beysolow II, *Introduction to Deep Learning Using R*, DOI 10.1007/978-1-4842-2734-3_6

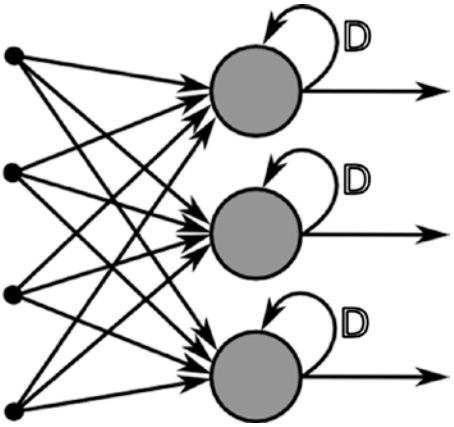

Figure 6-1. *Architecture of recurrent neural network*

This form of the RNN was developed in the 1980s. Similar to other neural networks, multiple layers of neurons are connected by weights, with each weight being altered via back-propagation methods. We alter our weights based on an evaluation statistic, which in this case is the weighted sum of the activation units at a given time step. The total error is the sum of all these individual weighted sums across all time steps. There may be teacher-driven target activations for some of the output units at certain time steps. For example, if the input sequence is a speech signal corresponding to a spoken digit, the final target output at the end of the sequence may be a label classifying the digit. For each sequence, its error is the sum of the deviations of all target signals from the corresponding activations computed by the network. For a training set of numerous sequences, the total error is the sum of the error of all individual sequences.

Training RNNs with Back-Propagation Through Time (BPPT)

Sepp Hochreiter and Jurgen Schmidhuber, among others, are considered among the greatest pioneers for development of training methods for deep learning. The standard method is called *back-propagation through time* (BPTT). BPTT is roughly the same as regular back-propagation, except it was created to deal with a specific problem that RNNs have, which is the fact that we are evolving a model through various time steps. For each training epoch, we begin by first training on reasonably small sequences and gradually increasing the length of the aforementioned training sequence. Intuitively, this is typically envisioned as training on a sequence of length 1,2, through *N*, where N is the maximum possible length of the sequence. Here is an equation describing this phenomenon more succinctly

$$\delta_{p,j}(t-1)=\sum_{h}^{m}\delta_{ph}(t)u_{hj}f'\left(s_{pj}(t-1)\right)$$

where t = time step, h = index for hidden node at t, j = hidden node at step $t = 1$, and δ = errors.

In detail, we can view the phenomena as the following: we define W as the matrix of weights for the output layer with the equation

$$W(t+1)=W(t)+\eta \boldsymbol{s}(t)\boldsymbol{e}_o(t)^T$$

where $\mathbf{e}_o$ = errors from the output layer:

$$\boldsymbol{e}_o(t)=\boldsymbol{d}(t)-\boldsymbol{y}(t)$$

We now have k sequences, through which we unfold the network into a regular feed-forward network that we've been observing up until this point. However, the recurrent layer in RNN model simultaneously takes the input from the preceding layer as well as the successive layer. To offset the change in weights that occurs from simultaneous inputting when back-propagating the errors, we average the updates that each layer receives.

Elman Neural Networks

RNN architectures received additional contributions from Jeffrey Elman, who is credited with creating the Elman network model named after him. Primarily, the architectures Elman constructed were for language-processing algorithms, but they can also be useful for any problem in which the input data is sequential or time-series based. Figure 6-2 shows the basic structure of an Elman network.

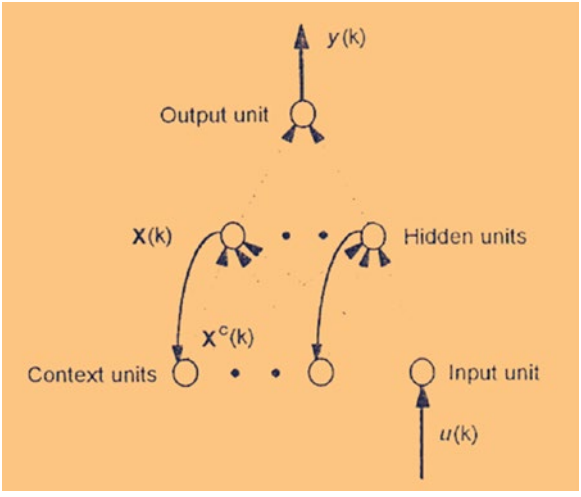

Figure 6-2. *Illustration of Elman network*

Elman included a layer of context units in this architecture that are distinguished by the fact that their functionality is highly concerned regarding prior internal states. One of the key distinctions of an Elman network is for the output of the hidden layer to feed into the context units in the preceding layer as well, but the weights that connect the context units and hidden layer have a constant value of 1, making the relationship linear. After this, the input layer and context layer simultaneously activate the hidden layer, whereupon the hidden layer also outputs a value while performing the update step. During the next epoch, the training sequence previously described occurs the same way, except here we observe that the layer with the context units now adopt the values of the hidden layer from the prior epoch. This feature of the context units colloquially has been described as the network *having memory*. Training this neural net requires a multitude of steps, the number of which ultimately depends on the length of the string being chosen.

Neural History Compressor

The *vanishing gradient* specifically refers to the gradients in earlier layers of a network becoming infinitesimally small. This occurs due to whatever activation function we use, usually a tanh or sigmoid. Because these activation functions "squash" the inputs into relatively small ranges to make interpolating the results easier, it makes deriving the gradients significantly more difficult. Repeat this process of squashing inputs after multiple stacked layers, and by the time we back-propagate to the first layer, our gradient has "vanished." The problem of vanishing gradients was partially dealt with via the creation of *neural history compressors*—an early generative model implemented as an unsupervised "stack" of recurrent neural networks. The input level learns to predict its next input from the previous input history. In the next higher-level RNN, the inputs are comprised of only the unpredictable inputs of a subset of the RNNs in the stack, which ensures that the internal state is recomputed rarely. Each high-level RNN thus learns a compressed representation of the information in the RNN below. By design, we can precisely reconstruct the input sequence from the sequence representation at the highest level. When we're using sequential data with considerable predictability, supervised learning can be utilized to classify substantially deeper sequences via the highest-level RNN.

Long Short-Term Memory (LSTM)

LSTM is an increasingly popular model whose strength is handling gaps of unknown size between signals in the noise of the data. Developed in the late 1990s by Sepp Hochreiter and Jurgen Schmidhuber, LSTMs are universal such that when enough network units are present, anything a computer can compute can be replicated with LSTMs, assuming we have a properly calibrated weight matrix. Figure 6-3 illustrates.

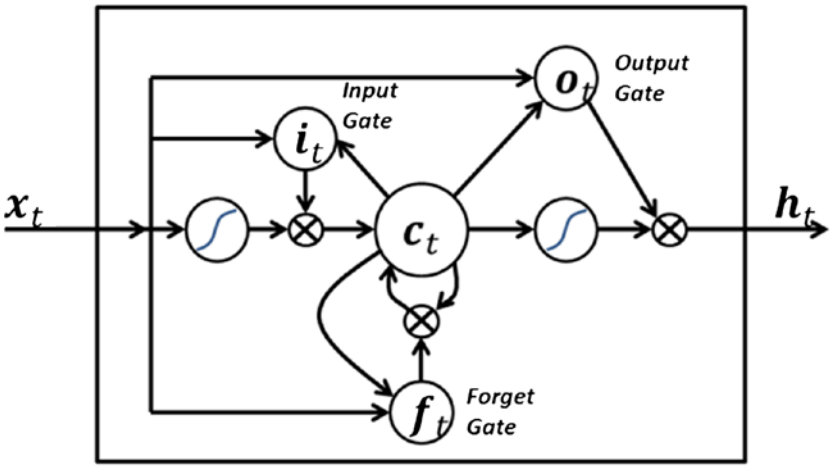

Figure 6-3. *Visualization of long short-term memory network*

The range of applications of LSTMs explains their popularity in part, as they are often used in the fields of robot control, time series prediction, speech recognition, and other tasks. In contrast to the units that we often see in other RNN architectures, LSTM networks contain blocks. Other key distinguishing factor of LSTMs is being able to "remember" a given value for extended periods of time and the gates within the model determining several attributes of the input sequence. Among the considerations of the gates are input significance, when should memory be kept or "garbage collection" occur and data be removed, and output value time. A typical implementation of an LSTM block is shown in Figure 6-3. The sigmoid units in a standard LSTM contain the equation

$$y = s\left(\sum_{i=1}^{N} w_i x_i\right),$$

where *s* is a squashing function (in many cases, often a logistic function or any activation function, as described in prior models). Looking at Figure 6-3, the sigmoid unit furthest to the left feeds the input to the LSTM block's "memory." From this point forward, the other units in the figure serve as the gates, which either permit or deny access into the LSTM memory. The unit entitled *i*, which we denote as the input gate of the diagram, will block all values from entering the memory that are very small (close to zero). The *forget gate*, the unit at the bottom of the figure, "forgets" whatever value it was remembering and discards this from the memory. The unit in the top righthand corner of the figure is the "output gate," which determines whether the value stored in the memory of the LSTM should be outputted. Occasionally, we observe units that are denoted by the following symbols: Π or Σ. Units that have the summation symbol are fed back into the LSTM

block as to facilitate remembrance of the same value over many time steps sans value decay. Typically this value is also inputted into the three gate units improve their respect decision making processes. The Haramard product, or entrywise product of matrices used in LSTMs, is given by the following in index notation:

$$\left(A \circ B\right)_{i,j} = A_{i,j}\ x\ B_{i,j}$$

Traditional LSTM

Above, we have the layers of an LSTM through which our data passes

$$f_t = \sigma_g\left(W_f x_t + U_f h_{t-1} + b_f\right),$$

$$i_t = \sigma_g\left(W_i x_t + U_i h_{t-1} + b_i\right),$$

$$o_t = \sigma_g\left(W_o x_t + U_o h_{t-1} + b_o\right),$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c\left(W_c x_t + U_c h_{t-1} + b_c\right),$$

$$h_t = o_t \circ \sigma_h\left(c_t\right),$$

where x = input vector, h_t = output vector, c_t = cell state, (W, U, b) = paramter matrices and vector, (f_t, i_t, and o_t) = remembered information, acquired information, and output, respectively, σ_g = sigmoid function, σ_c = original hyperbolic tangent, σ_h = original hyperbolic tangent.

Training LSTMs

BPPT is used for LSTMs, but due to special features of the LSTM, we can also use gradient descent via BP as we would traditionally. Vanishing gradients in LSTMs are handled specifically by the *error carousel*. LSTMs "remember" their back-propagated errors, which are then fed back to each of the weight. Thus, regular back-propagation is effective at training an LSTM block to remember values for very long durations of time.

Structural Damping Within RNNs

If we're using a conjugate gradient method and it strays too far from the original x, the curvature estimate becomes inaccurate and we may observe an inability to converge upon the global optimum. Suggested by Martens and Sutskever, *structural dampening* is recommended when using conjugate gradient methods. With this method, we penalize large deviations from x, where the formula is given by

$$\tilde{f}_d(x+\Delta x) = \tilde{f}(x+\Delta x) + \lambda \left\|\Delta x\right\|^2,$$

where $\|\Delta x\|^2$ is the magnitude of the deviation. λ, similar to ridge regression, serves as a tuning parameter.

The tuning parameter is adaptive and is chosen via a process similar to that of the Levenburg-Marq algorithm described in Chapter 3. It is suggested that we find a reduction ratio, given by the following equation:

$$\rho \equiv \frac{f(x+\Delta x) - f(x)}{\tilde{f}(x+\Delta x) - \tilde{f}(x)}$$

Tuning Parameter Update Algorithm

Weights are updated at each time step and as such augmenting the value in this matrix can cause drastic changes in the output:

$$If\left(\rho > \frac{3}{4}\right)\{$$

$$\lambda \to \frac{2}{3}\lambda.$$

$$\}\ Else\ If\left(\rho < \frac{1}{4}\right)\{$$

$$\lambda \to \frac{3}{2}\lambda$$

$$\}\ Else\ If\left(\frac{1}{4} < \rho < \frac{3}{4}\right)\{$$

$$\lambda \to \lambda, (no\ update)$$

Practical Example of RNN: Pattern Detection

Let's take the example of trying to predict time series based sequential data. In this instance, we're going to try and predict the production of milk at different times of the year (Figures 6-4 and 6-5). Let's begin by examining our data to get an understanding of it:

```
#Clear the workspace
rm(list = ls())
#Load the necessary packages
require(rnn)

#Function to be used later
#Creating Training and Test Data Set
dataset <- function(data){
  x <- y <- c()
  for (i in 1:(nrow(data)-2)){
    x <- append(x, data[i, 2])
    y <- append(y, data[i+1, 2])
  }
  #Creating New DataFrame
  output <- cbind(x,y)
  return(output[1:nrow(output)-1,])
}
```

When working with time series data, we will have to perform a significant amount of data transformation. Particularly, we must create X and Y variables that are slightly different from the given data. From the `dataset()` function, we create a new X variable, which is time step t, from the original Y variable. We make a new Y variable that is t + 1 from the original Y variable. We then truncate the data by one row such that we remove the missing observation. Moving forward, let us load and visualize the data (shown in Figures 6-4 and 6-5):

```
#Monthly Milk Production: Pounds Per Cow
data <- read.table("/Users/tawehbeysolow/Downloads/monthly-milk-production-
pounds-p.csv", header = TRUE, sep = ",")
#Plotting Sequence
plot(data[,2], main = "Monthly Milk Production in Pounds", xlab = "Month",
ylab = "Pounds",
     lwd = 1.5, col = "cadetblue", type = "l")
#Ploting Histogram
hist(data[,2], main = "Histogram of Monthly Milk Production in Pounds", xlab
= "Pounds", col = "red")
```

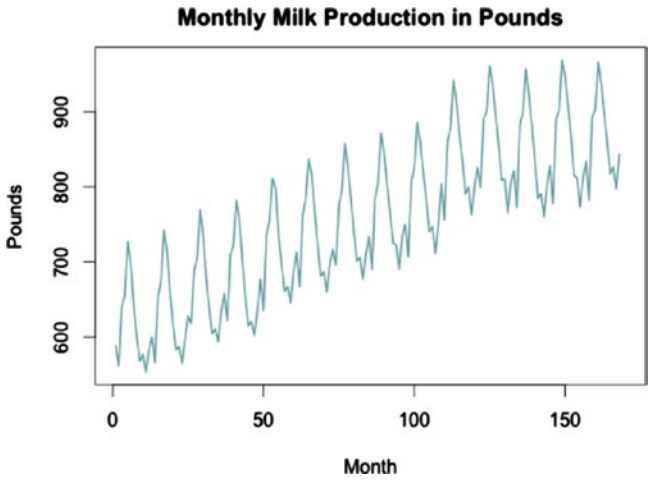

Figure 6-4. *Visualization of sequence*

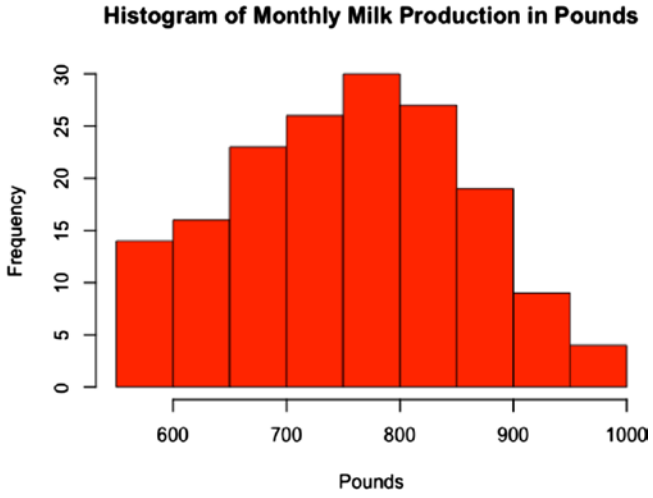

Figure 6-5. *Visualization of milk data via histogram*

As you can see, our data has a heavy right skew with respect to the frequency of values, despite the seemingly wide range of values.

Now that you've visually understood our data, let's move on and prepare our data to be inputted into the RNN:

```
#Creating Test and Training Sets
newData <- dataset(data = data)

#Creating Test and Train Data
rows <- sample(1:120, 120)
trainingData <- scale(newData[rows, ])
testData <- scale(newData[-rows, ])
```

I recommend that all users use max-min scaling prior to inputting their data into an RNN, because it significantly helps with reducing the errors from a given neural network. Similar to standard normalization, max-min scaling significantly reduces the range of your input data set, but it does so by classifying observations between 0 through 1 rather than by returning how many standard deviations away from the mean the data is. After we have performed this step, we can input our data. Users may feel free to experiment with the parameters, but I have trained the network for good performance.

Now let's evaluate our training and test results (shown in Figures 6-6 and 6-7):

```
#Max-Min Scaling
x <- trainingData[,1]
y <- trainingData[,2]

train_x <- (x - min(x))/(max(x)-min(x))
train_y <- (y - min(y))/(max(y)-min(y))

#RNN Model
RNN <- trainr(Y = as.matrix(train_x),X = as.matrix(train_y),
learningrate = 0.04, momentum = 0.1,
network_type = "rnn", numepochs = 700, hidden_dim = 3)

y_h <- predictr(RNN, as.matrix(train_x))
#Comparing Plots of Predicted Curve vs Actual Curve: Training Data
plot(train_y, col = "blue", type = "l", main = "Actual vs Predicted Curve",
lwd = 2)
lines(y_h, type = "l", col = "red", lwd = 1)
cat("Train MSE: ", mse(y_h, train_y))


#Test Data
testData <- scale(newData[-rows, ])
x <- testData[,1]
y <- testData[,2]
test_x <- (x - min(x))/(max(x)-min(x))
test_y <- (y - min(y))/(max(y)-min(y))
y_h2 <- predictr(RNN, as.matrix(x))
```

```
#Comparing Plots of Predicted Curve vs Actual Curve: Test Data
plot(test_y, col = "blue", type = "l", main = "Actual vs Predicted Curve",
lwd = 2)
lines(y_h3, type = "l", col = "red", lwd = 1)
cat("Test MSE: ", mse(y_h2, test_y))
```

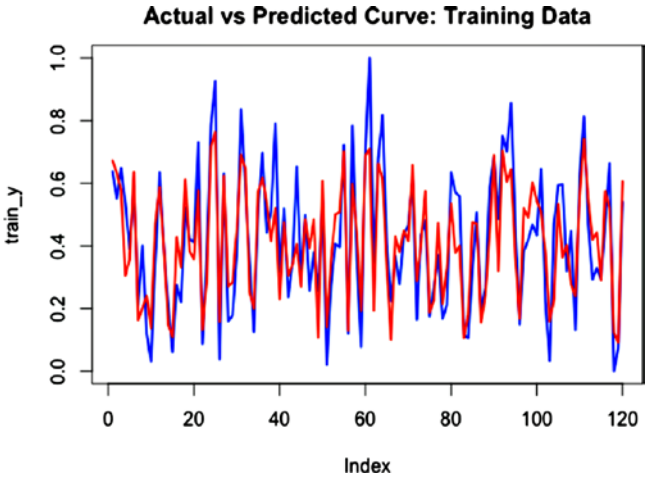

Figure 6-6. *Training data performance*

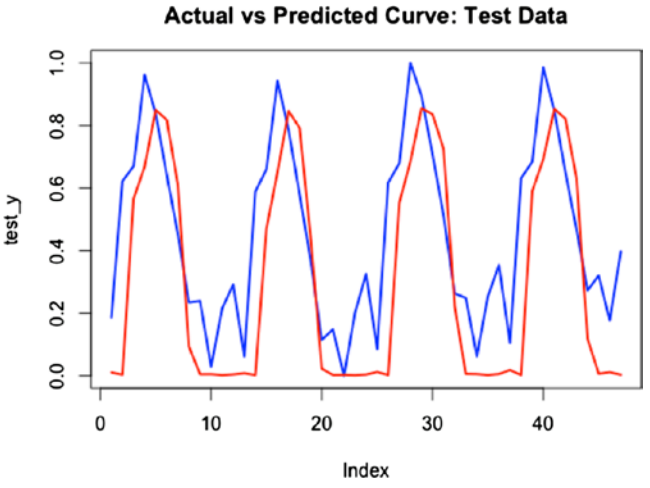

Figure 6-7. *Test set performance*

Respectively, the training and test set have MSEs of 0.01268307 and 0.06666131. Although the MSE for the training set is lower, this is likely just because the training set is significantly larger than the test set. We can see how the test performance is less accurate than the training set by visually comparing the curves in the respective plots. As you can see, the actual curve in both the training and test sets exhibits higher variance than the RNN can completely capture. If you're reading the e-book, the actual curve is in blue and the predicted curve is in red.

Summary

This chapter has effectively covered the most frequently mentioned RNN examples. It also walked the reader through tackling time series data problems. Chapter 7 addresses some of the most recent developments in deep learning and also explores how we can use these insights to tackle even more difficult problems.

CHAPTER 7

Autoencoders, Restricted Boltzmann Machines, and Deep Belief Networks

This chapter covers some of the newer and more advanced deep learning models that have been gaining popularity in the field. It is intended to help you understand some of the recent developments in the field of data science. To see how these models are applied in a practical context, see Chapters 10 and 11, where we will be utilizing these in practical examples.

Autoencoders

Prior to discussing restricted Boltzmann machines (RBMs), I want to address a set of related algorithms. *Autoencoders* are known as feature extractors, in that they are able to learn the encoding/representation of data. The data inputted to an RBM would be the same data that we would input to any machine learning algorithm, but for the sake of simplicity we can imagine it as an M x N matrix where each column is a unique feature and each row a unique observation of N features. It is an unsupervised learning method that uses back-propagation to find a way to reconstruct its own inputs. Developed by Geoffrey Hinton, along with other researchers, autoencoders address the problem of how to perform back-propagation without explicitly telling the autoencoder what to learn from.

Autoencoders consist of two parts: the encoder and the decoder. Let's look at a simple example of what we will denote as an *n/p/n autoencoder architecture*. This architecture is denoted by $n, p, m, \mathbb{G}, \mathbb{F}, \mathcal{A}, \mathcal{B}, \mathcal{X}, \Delta,$ where the following are true:

1. $\mathbb{G}$ and $\mathbb{F}$ are sets.
2. n and p are positive integers where $0 < p < n$.
3. Let $\mathcal{A}$ be a function where $\mathcal{A} : \mathbb{G}^p \rightarrow \mathbb{F}^n$.
4. Let $\mathcal{B}$ be a function where $\mathcal{B} : \mathbb{F}^n \rightarrow \mathbb{G}^p$.

T. Beysolow II, *Introduction to Deep Learning Using R*, DOI 10.1007/978-1-4842-2734-3_7

5. $\mathcal{X} = \{x_1, \ldots, x_M\} \in \mathbb{F}^n$ and when targets are present, $\mathcal{Y} = \{y_1, \ldots, y_M\} \in \mathbb{F}^n$.
6. Δ is an L_p norm or some other loss/dissimilarity function.

For any $A \in \mathcal{A}$, and $B \in \mathcal{B}$, the autoencoder transforms the input x into an output vector:

$$\hat{x} = A \circ B(x) \in \mathbb{F}^n$$

Broadly, the problem we seek to solve by using an autoencoder is ultimately an optimization problem—in this case, it is to minimize the loss/dissimilarity function. We define this problem as the following:

$$min\ E(A,B) = \min_{A,B} \sum_{m=1}^{M} E(x_m) = \min_{A,B} \sum_{m=1}^{M} \Delta\left(A \circ B(x_m), x_m\right)$$

When targets are present:

$$min\ E(A,B) = \min_{A,B} \sum_{m=1}^{M} E(x_m, y_m) = \min_{A,B} \sum_{m=1}^{M} \Delta\left(A \circ B(x_m), y_m\right)$$

Linear Autoencoders vs. Principal Components Analysis (PCA)

For this example, let's look at the similarities between principal components analysis (PCA) and linear autoencoders. The primary focus of PCA is to find the linear transformations of the original data set that contain the most variability within them in. When translating this analysis to the original data set, we use this to achieve dimensionality reduction. Chapter 8 talks about PCA in greater detail, but I will explain the relation it has to linear autoencoders here. Plainly stated, *PCA* is an orthogonal linear transformation where we seek to maximize the variance within each principal component subject to the constraint that each principal component is uncorrelated with each other. Let us define y as the following:

$$y_i = Ax_i,$$

where $x \in \mathbb{R}^n$ and is the data set, and $A \in \mathbb{R}^{nxn}$ and is the orthogonal covariance matrix. As is the case with PCA, each principal component should be listed in order of decreasing variance. We define the direction of maximum variance as the following:

$$\hat{w} = arg\max_{w} \frac{w^T X^T X w}{w^T w}$$

This by definition is a constrained optimization problem, solvable by using Lagrangian multipliers. Therefore, we can remodel the problem as

$$\mathcal{L}(w,\lambda) = w^T C w - \lambda\left(w^t w - 1\right),$$

$$Cw - \lambda w = 0,$$

$$Cw = \lambda w$$

where $C = X^T X$.

Single layer autoencoders will yield almost the exact same eigenvectors as PCA. That said, PCA assumes a linear system in its derivation in contrast to autoencoders that don't. In the instance that we force linearity in an autoencoder, a similar answer will be reached.

To see applications of autoencoders, see Chapter 11, where we specifically use these models for anomaly detection and improving model performance for standard machine learning models.

Restricted Boltzmann Machines

In the 1980s, Geoffrey Hinton, David Ackley, and Terrence Sejnowski developed this algorithm, which can be described as a type of stochastic neural network. At the time, it represented a breakthrough in the science of deep learning because it was among the first models to be able to learn the internal representations of data and have an ability to solve difficult combinatorics problems. The standard restricted Boltzmann machine has a binary-valued hidden and visible unit, consisting of a matrix of weights, *W,* associated with the connection between a given set of hidden units and visible units, and a bias weight. The hidden, visible, and bias units can be thought as analogous to those same units that appear in a multilayer perceptron model. Given these, the energy of a configuration is stated as the following:

$$E(v,h) = -\sum_{i=1}^{N} a_i v_i - \sum_{j=1}^{N} b_j h_j - \sum_{i=1}^{N}\sum_{j=1}^{N} v_i w_{i,j} h_j$$

This energy function is similar to the output neurons of a Hopfield network (see Figure 7-1), which is a particular type of RNN. Created in the 1980s by John Hopfield, the inputs, as with other RNN models, typically would be data that we suspect to have some underlying pattern (a time series for example). The weighted sum of all inputs is calculated, whereupon it is inputted into a linear classifier such as a logistic function. We define the output as the following:

$$\hat{y} = \begin{cases} 1, & \sum w_i x_i \geq 0 \\ -1, & \sum w_i x_i < 0 \end{cases}$$

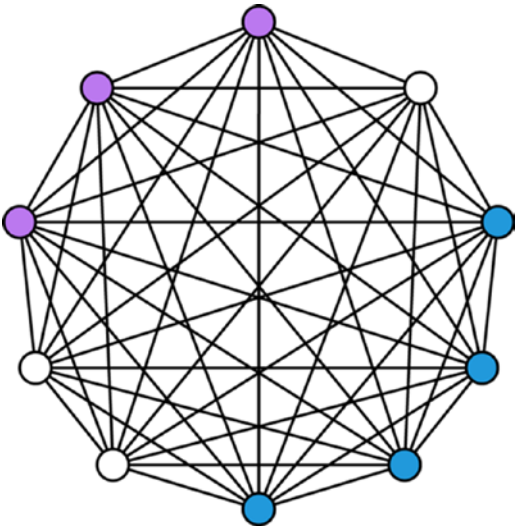

Figure 7-1. *Visualization of a Hopfield network*

After data is inputted to the model, all the nodes in the network receive specific values. The network is then subjected to a number of iterations using asynchronous or synchronous updating. After a stopping criterion is reached, the values within the neurons are displayed. The primary motivation for Hopfield networks is to discover the patterns stored in the weight matrix.

When referring back to the RBM model, the probability distributions that underlie the data are defined as

$$P(v,h)=\frac{1}{Z}e^{-E(v,h)},\ Z=\sum e^{-E(v,h)},$$

$$P(v)=\frac{1}{Z}\sum e^{-E(v,h)}$$

where $e^{-E(v,h)}$ = the exponential function, and the superscript is the negative value of the energy function previously described.

RBMs and bipartite graphs share similar properties. As such, the activations from the hidden units are mutually independent given the activations from the visible units such that

$$P(v|h)=\prod_{i=1}^{N}P(h_j|v),\ P(h|v)=\prod_{j=1}^{N}P(h_j|v),$$

and the individual activation probabilities are

$$P\left(h_j = 1 \mid v\right) = \sigma\left(b_i + \sum_{j=1}^{N} w_{i,j} v_i\right),\ P\left(v_i = 1 \mid h\right) = \sigma\left(a_j + \sum_{i=1}^{M} w_{i,j} h_i\right),$$

$$\sigma = \frac{1}{1 + e^{-k(x - x_0)}}$$

where a = activation unit.

The values of the visible units of an RBM can be derived from a multinomial distribution, whereas the values of the hidden units are derived from a Bernoulli distribution. In the instance that we use a softmax function for the visible units, we have the following function:

$$P\left(v_i^k = 1 \mid h\right) = \frac{exp\left(a_i^k + \sum W_{i,j}^k h_j\right)}{\sum_{k=1}^{K} exp\left(a_i^k + \sum_{j=1} W_{i,j}^k \boldsymbol{h}_j\right)}$$

The optimization of the weights inside an RBM is performed traditionally by using gradient descent via back-propagation until we've converged upon an optimal solution. One of the most popular use cases for RBMs has been to populate missing values within a data set, specifically in the case of collaborative filtering. Chapter 11 looks at a simple example of performing collaborative filtering. If you're interested in reading about performing this with RBMs, search for the paper by Salakhutdinov et al. on using RBMs for collaborative filtering (http://www.machinelearning.org/proceedings/icml2007/papers/407.pdf).

With respect to implementations of RBMs, there are a few packages that you may feel free to explore, such as deepnet, darch, and other implementations online. If you feel advanced enough, you may also seek to create your own implementation. In the meantime, you should check for updates to deep learning frameworks to see if/when they add RBM implementations.

Contrastive Divergence (CD) Learning

Developed by Hinton, *contrasting divergence* (CD) learning is a standard method of training restricted Boltzmann machines. It's based on the idea of using a Gibbs sampling, run for k steps, where it is initialized with a training example of the training set and yields the sample after k steps. It has broader applications as a training method for undirected

graph models, but its most popular use case is the training of RBMs. I'll begin this discussion by defining the gradient of the log likelihood:

$$\begin{aligned}\sum_h p(h|v)\frac{\partial E(v,h)}{\partial w_{i,j}} &= \sum_h p(h|v)h_i v_j = \sum_h \prod_{k=1}^{n} p(h_k|v)h_i v_j \\ &= \sum_{h_i}\sum_{h_{-i}} p(h_i|v)p(h_i|v)h_i v_j \\ &= \sum_{h_i} p(h_i|v)h_i v_j \sum_{h_{-i}} p(h_{-i}|v) = p(H_i=1|v)v_j \\ &= sig\left(\sum_{j=1}^{m} w_{i,j}v_j + c_i\right)\end{aligned}$$

Intuitively, we define the log-likelihood as the probability of a parameter having some value. Above, we define the sig() function as the signum function, which returns the sign of a input.

We define the gradient of the log-likelihood of training pattern v with the following equation:

$$\begin{aligned}\frac{\partial ln\mathcal{L}(\theta|v)}{\partial w_{i,j}} &= -\sum_h p(h|v)\frac{\partial E(v,h)}{\partial w_{i,j}} + \sum_{v,h} p(v,h)\frac{\partial E(v,h)}{\partial w_{i,j}} \\ &= \sum_{h_i} p(h_i|v)h_i v_j - \sum_v p(v)\sum_h p(h|v)h_i v_j \\ &= p(H_i=1|v)v_j - \sum_v p(v)p(H_i=1|v)v_j\end{aligned}$$

The mean of the gradient over training set $S=\{v_1,\ldots,v_\ell\}$ is given as

$$\begin{aligned}\frac{1}{\ell}\sum_{v\in S}\frac{\partial ln\mathcal{L}(\theta|v)}{\partial w_{i,j}} &= \frac{1}{\ell}\sum_{v\in S}\left[-\mathbb{E}_{p(h|v)}\left[\frac{\partial E(v,h)}{\partial w_{i,j}}\right] + \mathbb{E}_{p(\mathbf{h},v)}\left[\frac{\partial E(v,h)}{w_{i,j}}\right]\right] \\ &= \frac{1}{\ell}\sum_{v\in S}\left[\mathbb{E}_{p(h|v)}\left[v_i h_j\right] - \mathbb{E}_{p(h,v)}\left[v_i h_j\right]\right] \\ &= \left\langle v_i h_j\right\rangle_{p(h|v)q(v)} - \left\langle v_i h_j\right\rangle_{p(h,v)}\end{aligned}$$

where

$$\sum_{v \in S} \frac{\partial ln\mathcal{L}(\theta \mid v)}{\partial w_{i,j}} \propto \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}$$

$$\frac{\partial ln\mathcal{L}(\theta \mid v)}{\partial b_j} = v_j - \sum_v p(v) v_j,$$

$$\frac{\partial ln\mathcal{L}(\theta \mid v)}{\partial c_j} = p(H = 1 \mid v) - \sum_v p(v) p(H_i = 1 \mid v)$$

Now, returning to our initial discussion, we approximate the gradient of the log-likelihood of training pattern v as the following:

$$CD_k(\theta, v^0) = -\sum_h p(h \mid v^0) \frac{\partial E(v^0, h)}{\partial \theta} + \sum_h p(h \mid v^k) \frac{\partial E(v^k, h)}{\partial \theta}$$

The derivatives of each single parameter are calculated from the approximation just given with respect to the expectations over $p(v)$. In *batch learning*, we compute the gradient over the entirety of the training set. However, there are instances where it would be computationally more efficient to run this approximation over a subset of the training data set, which we denote as a *mini-batch*. If we evaluate a single element of the training set when performing this approximation, it's known as *online learning*. In RBMs, we refer to the *reconstruction error* as the difference between the actual input and the predicted input, which falls drastically from the beginning of training moving forward. It is suggested that this metric be used, but proceed with caution. CD learning is approximately optimizing the KL divergence between the training data and the data produced by the RBM and the Gibbs chain's mixing rate. That said, the reconstruction error often can be deceptively small if the mixing rate is also small. As the weights within the RBM increase, typically we observe the mixing rate to move inversely. But a lower mixing rate doesn't always necessarily mean a model is superior to one in which there is a higher mixing rate.

RBM weights, similar to other deep learning models, are typically initialized using values randomly sampled from a normal distribution or other infinitesimally small values. With respect to the learning rate, the same considerations with gradient methods must be taken into account, particularly being careful not to choose a learning rate that's too large or too small. With that being said, an adaptive learning rate may cause issues as it will give the appearance that the model is improving due to a lower reconstruction error, however, as explained earlier, this may not always be the case. It is recommended that each weight update generally be about 10^{-3} times the current weights. Initial hidden biases and weights typically are initialized by selecting them randomly from a normal distribution, as is standard operating procedure for other neural network models.

Momentum Within RBMs

To increase the speed of learning within an RBM, *momentum* is a recommended method. Imagine a gradient plot such as the one in Figure 7-2. If we can imagine the error represented by a point on one of the circles, the dot gains "momentum" as it moves closer to the minimum—but it loses momentum if it tries to go past that point and upwards along the sphere on the opposite side.

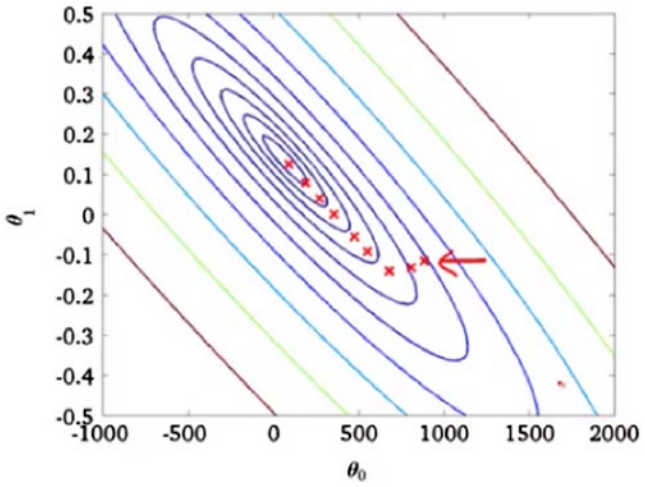

Figure 7-2. *Gradient plot*

Rather than the traditional gradient descent formula, the momentum method incrementally affects the velocity of the parameter update. We define *momentum* as the percentage of the velocity that is still present after a given epoch; we assume that over time the velocity of a parameter decays. In effect, the momentum method causes the update of the parameters to move in a direction that is not the steepest descent, as with a typical gradient method except less intricate. When using the momentum method, it is suggested that the momentum parameter, α, be set to .5. When it becomes more difficult to reduce the reconstruction error any further, the momentum should be increased to .9. If we notice instability in the reconstruction error—typically noted by occasional, incremental increases—we reduce the learning rate by factors of 2 until this phenomenon subsides. We define the momentum method of updating a parameter as follows:

$$\Delta\theta_i(t) = v_i(t) = \alpha v_i(t-1) - \epsilon \frac{dE(t)}{d\theta_i}$$

Weight Decay

Weight decay can be viewed as a form of regularization, similar to that of the parameter regularization seen in ridge regression and/or LASSO. In RMBs, we typically use a Euclidean norm, which we denote as cost of the weights. Commonly, practitioners take the derivative of the penalty term and multiply it by the learning rate. This prevents the learning rate from changing the objective function we are trying to optimize. Weight decay helps reduce overfitting in such a way that the solution achieved doesn't have units with unusually large weights or weights that are either always on or off. It also improves the mixing rate, in reference to the Gibbs sampling we perform, making CD learning more accurate. Geoffrey Hinton suggests that initially a weight cost of 0.0001 be used.

Sparsity

Generally, a good model is one that has hidden units that are active only part of the time. The reason is that models with sparsely active units are considerably easier to interpret compared to models that are densely populated with active units. We can achieve *sparsity* by specifying the probability of a unit being active, performed by using regularization. This probability is denoted by q and is estimated by

$$q_{new} = \lambda q_{ild} + (1-\lambda) q_{current},$$

where $q_{current}$ = mean activiation probability of hidden unit

The natural penalty measure to use is the cross entropy between the desired and actual distributions:

$$Sparsity\ penalty \propto -plogq - (1-p)log(1-q)$$

As suggested by Hinton, we seek to have a sparsity target as low as 0.1[9] and as high as 0.01. We denote the decay rate as λ, which refers to the estimated sparsity value. This should be no higher than 0.99 but higher than 0.9. We should reduce the sparsity cost if the probabilities we calculate are clustering around the target value, and a general suggestion for modeling this is to collect a histogram of mean activities when collecting random samples.

No. and Type Hidden Units

Being that often the main consideration is that we seek to avoid overfitting. As such, we generally will try to use fewer hidden units rather than more. Particularly, if the data across the observations tends to be very homogenous, we also should try and use fewer rather than more hidden units. However, an instance in which it is reasonable to use more hidden units than normal would be if the sparsity target we're trying to achieve happens to fall within a very small range (or is very small itself). As for the type of units,

we can use Gaussian visible (and/or hidden), in addition to sigmoid and softmax units denoted by the following, respectively:

$$E(v,h)=\sum_{i\in v}\frac{(v_i-a_i)^2}{2\sigma_i^2}-\sum_{j\in h}b_j h_j-\sum_{i,j}\frac{v_i}{\sigma_i}h_j w_{i,j},$$

$$E(v,h)=\sum_{i\in v}\frac{(v_i-a_i)^2}{2\sigma_i^2}+\sum_{j\in h}\frac{(h_j-b_j)^2}{2\sigma_j^2}-\sum_{i,j}\frac{v_i}{\sigma_i}h_j w_{i,j},$$

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

$$p_j=\frac{e^{x_j}}{\sum_{i=1}^{K}e^{x_i}}.$$

Deep Belief Networks (DBNs)

The final model I'll address is the deep belief network (DBN), shown in Figure 7-3, another innovation from Geoffrey Hinton. To make a DBN, we stack together restricted Boltzmann machines and train the layers one at a time. Typically, we use DBNs for unsupervised learning problems.

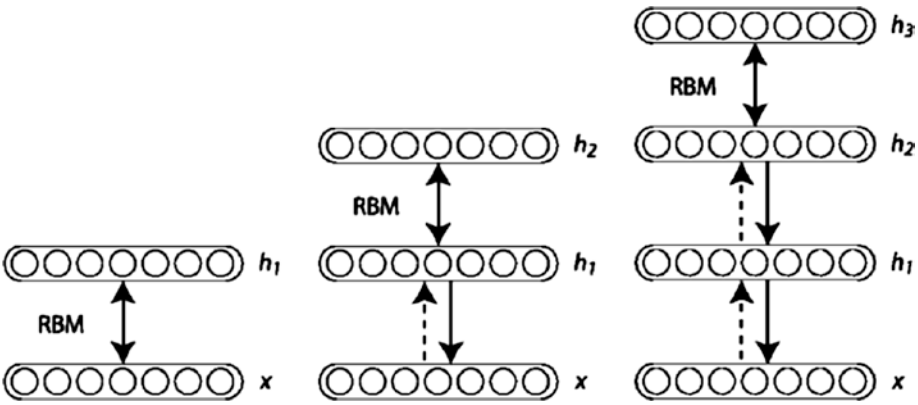

Figure 7-3. *Visualization of a deep belief network*

In a 2006 paper, Geoffrey Hinton and Simon Osindero, both researchers at the University of Toronto, describe an algorithm useful for fast learning. The difficulty posed by training networks with many hidden layers inspired the creation of a hybrid model. The main attraction of this model, in relation to the training problem, is that by design there

are complementary priors that allow us to easily draw from the conditional probability distribution. This is done by starting with a random configuration a layer deep within the network. We then pass through each layer of the network, in which the state of a given layer is determined by a Bernoulli trial. The parameters for the Bernoulli function are derived from the input received from the preceding layer in the initial "top-down" pass.

Fast Learning Algorithm (Hinton and Osindero 2006)

Data is generated from an RBM by taking a random state within a given layer and performing Gibbs sampling over it. Simply stated, *Gibbs sampling* is a type of Monte Carlo method in which we try to obtain a sequence based on a probability distribution that the user specifies, but which the algorithm tries to approximate. Typically, the distribution is multivariate. All units within a chosen layer are updated in a parallel fashion, and this is repeated until we've determined to be sampling from the equilibrium distribution. In Figure 7-4, we can see the visible and the hidden layers of an RBM.

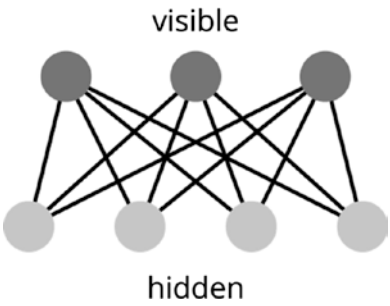

Figure 7-4. *Visualization of restricted Boltzmann machine*

Each weight uses a visible unit, *i*, and a hidden unit, *j*. When a data vector is "clamped" on the visible units, the hidden units are sampled from their conditional distribution, which is factorial. The gradient of the log probability is given by the following:

$$\frac{\partial \log p\left(\boldsymbol{v}^{\mathbf{0}}\right)}{\partial w_{i,j}} = \left\langle v_i^0 h_j^0 \right\rangle - \left\langle v_i^\infty h_j^\infty \right\rangle$$

When we minimize the KL divergence, we in effect maximize the log probability. If you would like to learn complicated models, break up the single model into smaller, simpler models. After this point, these models can be learned sequentially. An example of this sequential learning would be gradient boosting, as discussed in Chapter 3. Reasonable approximations for $\mathbf{W}_0$ are learned based on the assumption that higher layers derive the complimentary prior for $\mathbf{W}_0$. In practice, we can achieve this outcome by assuming that all the weight matrices must be equal to one another. When solving this

constrained optimization problem, learning becomes significantly easier than before, and the problem itself is reduced to learning an RBM, whereupon good approximate solutions are achieved via minimizing contrastive divergence.

Algorithm Steps

1. Under the assumption that all the weight matrices are tied, learn $\mathbf{W_0}$.
2. Use $\mathbf{W_0}^{\mathbf{T}}$ to infer factorial approximate posterior distributions over the states of the variables in the first hidden layer.
3. Learn an RBM model with respect to high-level abstractions of the data generated by $\mathbf{W_0}^{\mathbf{T}}$.
4. Repeat until convergence upon an optimal solution.

If the weight matrices in the higher levels of the model change, we are guaranteed to see improvements in the model. The bound given becomes an equality if $Q(.|\mathbf{v^0})$ is the true posterior of the data. Hinton specifically suggests a greedy learning method, as described in Neal and Hinton (1998). The energy of a given configuration of $\mathbf{v^0}$, $\mathbf{h^0}$ is defined as

$$E\left(\boldsymbol{v}^0, \boldsymbol{h}^0\right) = -\left[log\ p\left(\boldsymbol{h}^0\right) + log\ p\left(\boldsymbol{v}^0 \mid \boldsymbol{h}^0\right)\right],$$

with a bound of

$$log\ p\left(\boldsymbol{v}^0\right) \geq \sum_{\boldsymbol{all}\ h^0} Q\left(\boldsymbol{h}^0 \mid \boldsymbol{v}^0\right)\left[log\ p\left(\boldsymbol{h}^0\right) + log\left(\boldsymbol{v}^0 \mid \boldsymbol{h}^0\right)\right] - \sum_{\boldsymbol{all}\ h^0} Q\left(\boldsymbol{h}^0 \mid \boldsymbol{v}^0\right) log\ Q\left(\boldsymbol{h}^0 \mid \boldsymbol{v}^0\right)$$

where $\mathbf{h}^0$= binary configuration the initial hidden layer units, $p(\mathbf{h}^0)$ = the prior of the current model $\mathbf{h}^0$, and $Q(.|\mathbf{v}^0)$ = probability distribution over the initial hidden layer's binary configurations.

Summary

This brings us to the end of discussing autoencoders, RBMs, and DBNs. This also concludes all the chapters on deep learning models. Now that we've discussed these models, it's time to turn our attention to experimental design and feature selection techniques to help you increase the accuracy of your machine learning models.

CHAPTER 8

Experimental Design and Heuristics

After having reviewed all the machine learning and deep learning models that will be relevant to problem solving that you will encounter, it's finally time to talk about useful methods of structuring your research, both formal and informal.

Beyond just knowing how to properly evaluate the solutions developed, you should be familiar with the concepts associated with the field of *experimental design*. Ronald Fisher, an English statistician prominent in the 20th century, was one of the most influential figures in the field of statistics. His techniques are frequently referenced when performing experimentation and are useful to review even if you don't use them explicitly.

Analysis of Variance (ANOVA)

ANOVA is group of methods that are used to study the variation among groups of observations within data. An extension of the z and t test, and similar to regression, we observe the interaction between the response and explanatory variables. We assume that the observations within the data are independent and identically distributed (IID) normal random variables, that residuals are normally distributed, and that variance is homogenous. Among the multiple ANOVA models are the following ones discussed in the rest of this section.

One-Way ANOVA

Used to compare three or more sample spaces' means/averages to one another. Specifically, it's used in cases where the classification is performed by one variable/factor that has two or more levels.

Two-Way (Multiple-Way) ANOVA

This is similar to one-way ANOVA, except this model can be used where there are two or more explanatory variables.

T. Beysolow II, *Introduction to Deep Learning Using R*, DOI 10.1007/978-1-4842-2734-3_8