7 Black Box Methods – Neural Networks and Support Vector Machines

The late science fiction author *Arthur C. Clarke* once wrote that "any sufficiently advanced technology is indistinguishable from magic." This chapter covers a pair of machine learning methods that may, likewise, appear at first glance to be magic. As two of the most powerful machine learning algorithms, they are applied to tasks across many domains. However, their inner workings can be difficult to understand.

In engineering, these are referred to as **black box** processes because the mechanism that transforms the input into the output is obfuscated by a figurative box. The reasons for the opacity can vary; for instance, black box closed source software intentionally conceals proprietary algorithms, the black box of sausage-making involves a bit of purposeful (but tasty) ignorance, and the black box of political lawmaking is rooted in bureaucratic processes. In the case of machine learning, the black box is because the underlying models are based on complex mathematical systems and the results are difficult to interpret.

Although it may not be feasible to interpret black box models, it is dangerous to apply the methods blindly. Therefore, in this chapter, we'll peek behind the curtain and investigate the statistical sausage-making involved in fitting such models. You'll discover that:

- Neural networks use concepts borrowed from an understanding of human brains in order to model arbitrary functions
- Support Vector Machines use multidimensional surfaces to define the relationship between features and outcomes
- In spite of their complexity, these models can be easily applied to real-world problems such as modeling the strength of concrete or reading printed text

With any luck, you'll realize that you don't need a black belt in statistics to tackle black box machine learning methods—there's no need to be intimidated!

Understanding neural networks

An **Artificial Neural Network** (**ANN**) models the relationship between a set of input signals and an output signal using a model derived from our understanding of how a biological brain responds to stimuli from sensory inputs. Just as a brain uses a network of interconnected cells called **neurons** to create a massive parallel processor, the ANN uses a network of artificial neurons or **nodes** to solve learning problems.

The human brain is made up of about 85 billion neurons, resulting in a network capable of storing a tremendous amount of knowledge. As you might expect, this dwarfs the brains of other living creatures. For instance, a cat has roughly a billion neurons, a mouse has about 75 million neurons, and a cockroach has only about a million neurons. In contrast, many ANNs contain far fewer neurons, typically only several hundred, so we're in no danger of creating an artificial brain anytime in the near future—even a fruit fly brain with 100,000 neurons far exceeds the current ANN state-of-the-art.

Though it may be infeasible to completely model a cockroach's brain, a neural network might provide an adequate heuristic model of its behavior, such as in an algorithm that can mimic how a roach flees when discovered. If the behavior of a roboroach is convincing, does it matter how its brain works? This question is the basis of the controversial **Turing test**, which grades a machine as intelligent if a human being cannot distinguish its behavior from a living creature's.

Rudimentary ANNs have been used for over 50 years to simulate the brain's approach to problem solving. At first, this involved learning simple functions, like the logical AND function or the logical OR. These early exercises were used primarily to construct models of how biological brains might function. However, as computers have become increasingly powerful in recent years, the complexity of ANNs has likewise increased such that they are now frequently applied to more practical problems such as:

- Speech and handwriting recognition programs like those used by voicemail transcription services and postal mail sorting machines
- The automation of smart devices like an office building's environmental controls or self-driving cars and self-piloting drones
- Sophisticated models of weather and climate patterns, tensile strength, fluid dynamics, and many other scientific, social, or economic phenomena

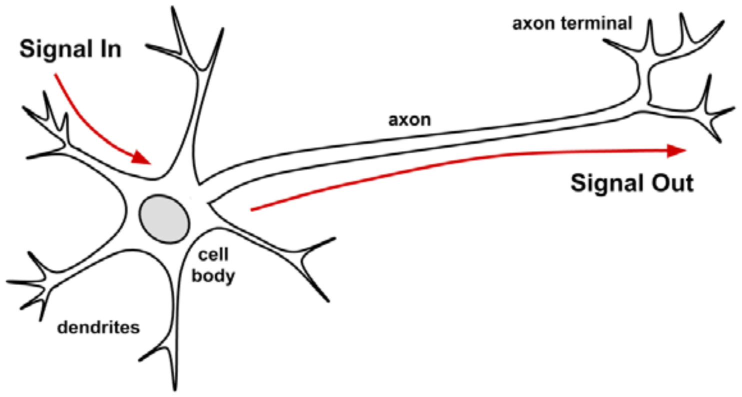
Broadly speaking, ANNs are versatile learners that can be applied to nearly any learning task: classification, numeric prediction, and even unsupervised pattern recognition.

> Whether deserving or not, ANN learners are often reported in the media with great fanfare. For instance, an "artificial brain" developed by Google was recently touted for its ability to identify cat videos on YouTube. Such hype may have less to do with anything unique to ANNs and more to do with the fact that ANNs are captivating because of their similarities to living minds.

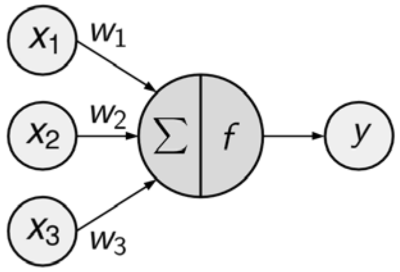
ANNs are best applied to problems where the input data and output data are well-understood or at least fairly simple, yet the process that relates the input to output is extremely complex. As a black box method, they work well for these types of black box problems.

From biological to artificial neurons

Because ANNs were intentionally designed as conceptual models of human brain activity, it is helpful to first understand how biological neurons function. As illustrated in the following figure, incoming signals are received by the cell's **dendrites** through a biochemical process that allows the impulse to be weighted according to its relative importance or frequency. As the cell body begins to accumulate the incoming signals, a threshold is reached at which the cell fires and the output signal is then transmitted via an electrochemical process down the **axon**. At the axon's terminals, the electric signal is again processed as a chemical signal to be passed to the neighboring neurons across a tiny gap known as a **synapse**.



The model of a single artificial neuron can be understood in terms very similar to the biological model. As depicted in the following figure, a directed network diagram defines a relationship between the input signals received by the dendrites (*x* variables) and the output signal (*y* variable). Just as with the biological neuron, each dendrite's signal is weighted (*w* values) according to its importance—ignore for now how these weights are determined. The input signals are summed by the cell body and the signal is passed on according to an **activation function** denoted by *f*.



A typical artificial neuron with *n* input dendrites can be represented by the formula that follows. The *w* weights allow each of the *n* inputs, (*x*), to contribute a greater or lesser amount to the sum of input signals. The net total is used by the activation function *f(x)*, and the resulting signal, *y(x),* is the output axon.

$$y(x) = f\left(\sum_{i=1}^{n} w_i x_i\right)$$

Neural networks use neurons defined in this way as building blocks to construct complex models of data. Although there are numerous variants of neural networks, each can be defined in terms of the following characteristics:

- An **activation function**, which transforms a neuron's net input signal into a single output signal to be broadcasted further in the network
- A **network topology** (or architecture), which describes the number of neurons in the model as well as the number of layers and manner in which they are connected
- The **training algorithm** that specifies how connection weights are set in order to inhibit or excite neurons in proportion to the input signal

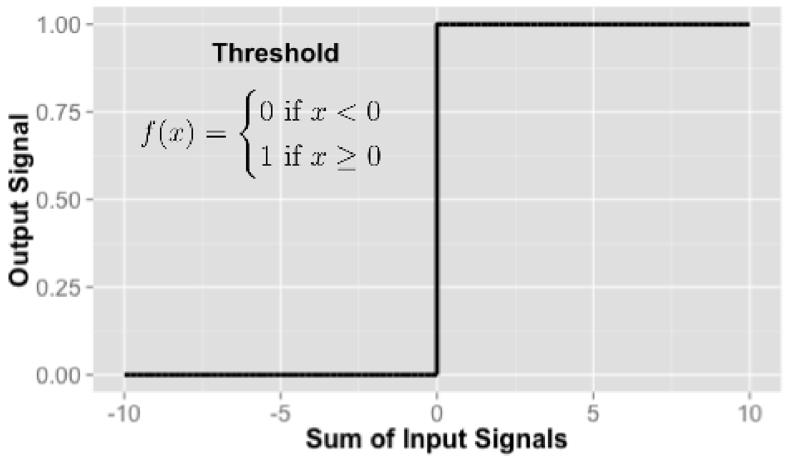
Let's take a look at some of the variations within each of these categories to see how they can be used to construct typical neural network models.

Activation functions

The activation function is the mechanism by which the artificial neuron processes information and passes it throughout the network. Just as the artificial neuron is modeled after the biological version, so too is the activation function modeled after nature's design.

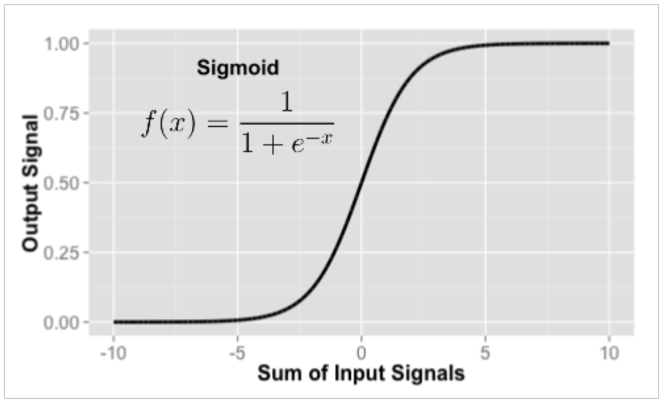
In the biological case, the activation function could be imagined as a process that involves summing the total input signal and determining whether it meets the firing threshold. If so, the neuron passes on the signal; otherwise, it does nothing. In ANN terms, this is known as a **threshold activation function**, as it results in an output signal only once a specified input threshold has been attained.

The following figure depicts a typical threshold function; in this case, the neuron fires when the sum of input signals is at least zero. Because of its shape, it is sometimes called a **unit step activation function**.

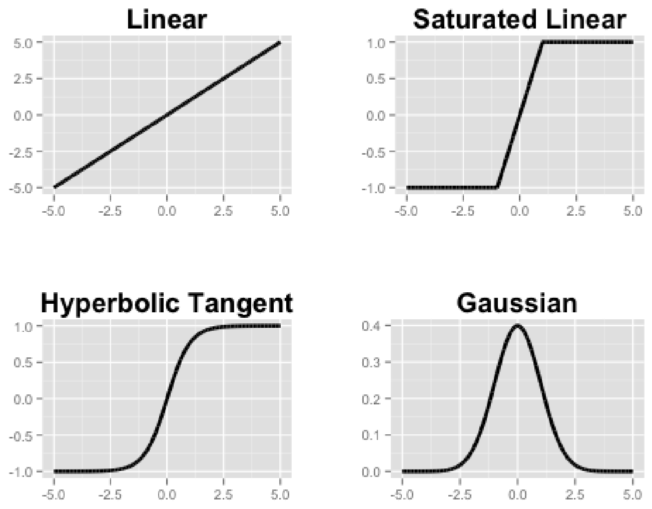


Although the threshold activation function is interesting due to its parallels with biology, it is rarely used in artificial neural networks. Freed from the limitations of biochemistry, ANN activation functions can be chosen based on their ability to demonstrate desirable mathematical characteristics and model relationships among data.

Perhaps the most commonly used alternative is the **sigmoid activation function** (specifically the logistic sigmoid) shown in the following figure, where *e* is the base of natural logarithms (approximately 2.72). Although it shares a similar step or S shape with the threshold activation function, the output signal is no longer binary; output values can fall anywhere in the range from 0 to 1. Additionally, the sigmoid is **differentiable**, which means that it is possible to calculate the derivative across the entire range of inputs. As you will learn later, this feature is crucial for creating efficient ANN optimization algorithms.



Although the sigmoid is perhaps the most commonly used activation function and is often used by default, some neural network algorithms allow a choice of alternatives. A selection of such activation functions is as shown:



The primary detail that differentiates among these activation functions is the output signal range. Typically, this is one of *(0, 1)*, *(-1, +1)*, or *(-inf, +inf)*. The choice of activation function biases the neural network such that it may fit certain types of data more appropriately, allowing the construction of specialized neural networks. For instance, a linear activation function results in a neural network very similar to a linear regression model, while a Gaussian activation function results in a model called a **Radial Basis Function** (**RBF**) **network**.

It's important to recognize that for many of the activation functions, the range of input values that affect the output signal is relatively narrow. For example, in the case of the sigmoid, the output signal is always 0 or always 1 for an input signal below -5 or above +5, respectively. The compression of the signal in this way results in a saturated signal at the high and low ends of very dynamic inputs, just as turning a guitar amplifier up too high results in a distorted sound due to clipping the peaks of sound waves. Because this essentially squeezes the input values into a smaller range of outputs, such activation functions (like the sigmoid) are sometimes called squashing functions.

The solution to the squashing problem is to transform all neural network inputs such that the feature values fall within a small range around 0. Typically, this is done by standardizing or normalizing the features. By limiting the input values, the activation function will have action across the entire range, preventing large-valued features such as household income from dominating small-valued features such as the number of children in the household. A side benefit is that the model may also be faster to train, since the algorithm can iterate more quickly through the actionable range of input values.

Although theoretically a neural network can adapt to a very dynamic feature by adjusting its weight over many iterations, in extreme cases many algorithms will stop iterating long before this occurs. If your model is making predictions that do not make sense, double-check that you've correctly standardized the input data.

Network topology

The capacity of a neural network to learn is rooted in its **topology**, or the patterns and structures of interconnected neurons. Although there are countless forms of network architecture, they can be differentiated by three key characteristics:

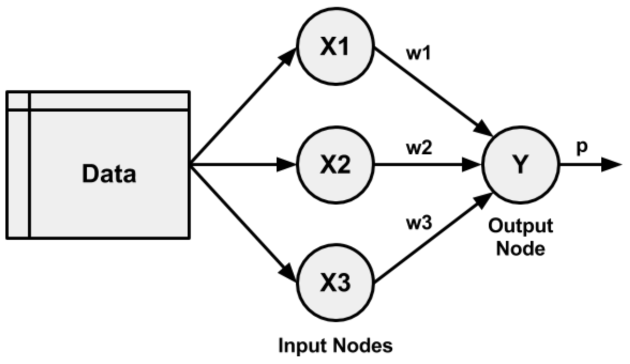
- The number of layers
- Whether information in the network is allowed to travel backward
- The number of nodes within each layer of the network

The topology determines the complexity of tasks that can be learned by the network. Generally, larger and more complex networks are capable of identifying more subtle patterns and complex decision boundaries. However, the power of a network is not only a function of the network size, but also the way units are arranged.

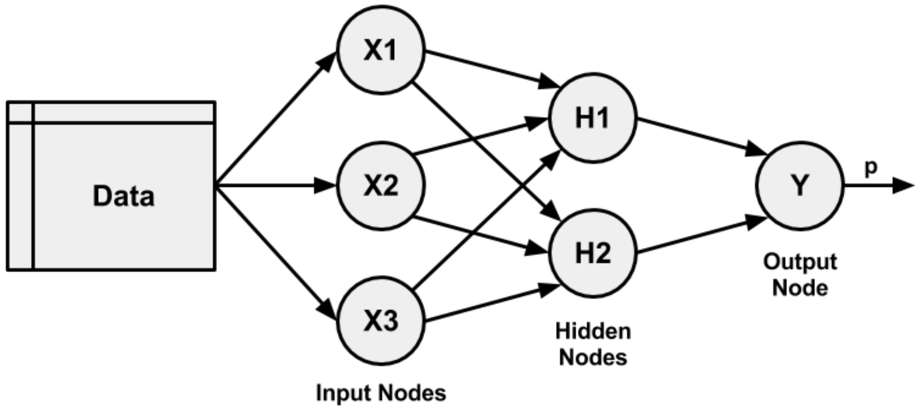
The number of layers

To define topology, we need a terminology that distinguishes artificial neurons based on their position in the network. The figure that follows illustrates the topology of a very simple network. A set of neurons called **Input Nodes** receive unprocessed signals directly from the input data. Each input node is responsible for processing a single feature in the dataset; the feature's value will be transformed by the node's activation function. The signals resulting from the input nodes are received by the **Output Node**, which uses its own activation function to generate a final prediction (denoted here as *p*).

The input and output nodes are arranged in groups known as **layers**. Because the input nodes process the incoming data exactly as received, the network has only one set of connection weights (labeled here as *w1*, *w2*, and *w3*). It is therefore termed a **single-layer network**. Single-layer networks can be used for basic pattern classification, particularly for patterns that are linearly separable, but more sophisticated networks are required for most learning tasks.



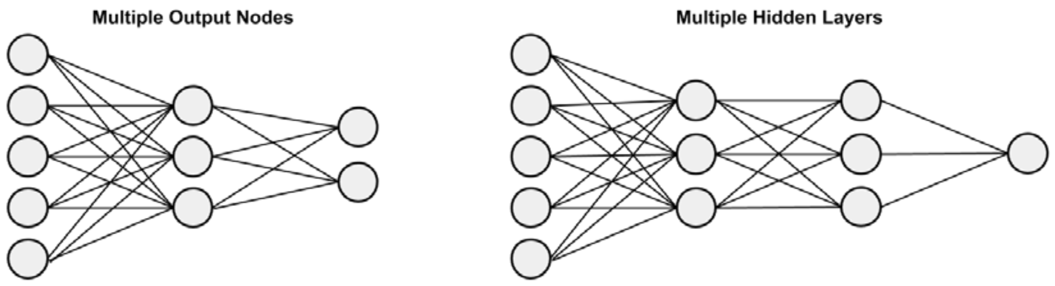
As you might expect, an obvious way to create more complex networks is by adding additional layers. As depicted here, a **multilayer network** adds one or more **hidden layers** that process the signals from the input nodes prior to reaching the output node. Most multilayer networks are **fully connected**, which means that every node in one layer is connected to every node in the next layer, but this is not required.



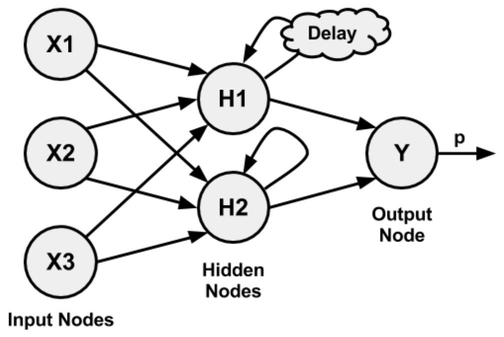
The direction of information travel

You may have noticed that in the prior examples arrowheads were used to indicate signals traveling in only one direction. Networks in which the input signal is fed continuously in one direction from connection-to-connection until reaching the output layer are called **feedforward networks**.

In spite of the restriction on information flow, feedforward networks offer a surprising amount of flexibility. For instance, the number of levels and nodes at each level can be varied, multiple outcomes can be modeled simultaneously, or multiple hidden layers can be applied (a practice that is sometimes referred to as **deep learning**).



In contrast, a **recurrent network** (or feedback network) allows signals to travel in both directions using loops. This property, which more closely mirrors how a biological neural network works, allows extremely complex patterns to be learned. The addition of a short term memory (labeled **Delay** in the following figure) increases the power of recurrent networks immensely. Notably, this includes the capability to understand sequences of events over a period of time. This could be used for stock market prediction, speech comprehension, or weather forecasting. A simple recurrent network is depicted as shown:



In spite of their potential, recurrent networks are still largely theoretical and are rarely used in practice. On the other hand, feedforward networks have been extensively applied to real-world problems. In fact, the multilayer feedforward network (sometimes called the **Multilayer Perceptron** (**MLP**) is the de facto standard ANN topology. If someone mentions that they are fitting a neural network without additional clarification, they are most likely referring to a multilayer feedforward network.

The number of nodes in each layer

In addition to variations in the number of layers and the direction of information travel, neural networks can also vary in complexity by the number of nodes in each layer. The number of input nodes is predetermined by the number of features in the input data. Similarly, the number of output nodes is predetermined by the number of outcomes to be modeled or the number of class levels in the outcome. However, the number of hidden nodes is left to the user to decide prior to training the model.

Unfortunately, there is no reliable rule to determine the number of neurons in the hidden layer. The appropriate number depends on the number of input nodes, the amount of training data, the amount of noisy data, and the complexity of the learning task among many other factors.

In general, more complex network topologies with a greater number of network connections allow the learning of more complex problems. A greater number of neurons will result in a model that more closely mirrors the training data, but this runs a risk of overfitting; it may generalize poorly to future data. Large neural networks can also be computationally expensive and slow to train.

A best practice is to use the fewest nodes that result in adequate performance on a validation dataset. In most cases, even with only a small number of hidden nodes—often as few as a handful—the neural network can offer a tremendous amount of learning ability.

It has been proven that a neural network with at least one hidden layer of sufficiently many neurons is a **universal function approximator**. Essentially, this means that such a network can be used to approximate any continuous function to an arbitrary precision over a finite interval.

Training neural networks with backpropagation

The network topology is a blank slate that by itself has not learned anything. Like a newborn child, it must be trained with experience. As the neural network processes the input data, connections between the neurons are strengthened or weakened similar to how a baby's brain develops as he or she experiences the environment. The network's connection weights reflect the patterns observed over time.

Training a neural network by adjusting connection weights is very computationally intensive. Consequently, though they had been studied for decades prior, ANNs were rarely applied to real-world learning tasks until the mid-to-late 1980s, when an efficient method of training an ANN was discovered. The algorithm, which used a strategy of back-propagating errors, is now known simply as **backpropagation**.

Interestingly, several research teams of the era independently discovered the backpropagation algorithm. The seminal paper on backpropagation is arguably *Learning representations by back-propagating errors*, *Nature Vol. 323*, pp. 533-566, by *D.E. Rumelhart, G.E. Hinton*, and *R.J. Williams* (1986).

Although still notoriously slow relative to many other machine learning algorithms, the backpropagation method led to a resurgence of interest in ANNs. As a result, multilayer feedforward networks that use the backpropagation algorithm are now common in the field of data mining. Such models offer the following strengths and weaknesses:

Strengths	Weaknesses
• Can be adapted to classification or numeric prediction problems • Among the most accurate modeling approaches • Makes few assumptions about the data's underlying relationships	• Reputation of being computationally intensive and slow to train, particularly if the network topology is complex • Easy to overfit or underfit training data • Results in a complex black box model that is difficult if not impossible to interpret

In its most general form, the backpropagation algorithm iterates through many cycles of two processes. Each iteration of the algorithm is known as an **epoch**. Because the network contains no *a priori* (existing) knowledge, typically the weights are set randomly prior to beginning. Then, the algorithm cycles through the processes until a stopping criterion is reached. The cycles include:

- A **forward phase** in which the neurons are activated in sequence from the input layer to the output layer, applying each neuron's weights and activation function along the way. Upon reaching the final layer, an output signal is produced.
- A **backward phase** in which the network's output signal resulting from the forward phase is compared to the true target value in the training data. The difference between the network's output signal and the true value results in an error that is propagated backwards in the network to modify the connection weights between neurons and reduce future errors.

Over time, the network uses the information sent backward to reduce the total error of the network. Yet one question remains: because the relationship between each neuron's inputs and outputs is complex, how does the algorithm determine how much (or whether) a weight should be changed?

The answer to this question involves a technique called **gradient descent**. Conceptually, it works similarly to how an explorer trapped in the jungle might find a path to water. By examining the terrain and continually walking in the direction with the greatest downward slope, he or she is likely to eventually reach the lowest valley, which is likely to be a riverbed.

In a similar process, the backpropagation algorithm uses the derivative of each neuron's activation function to identify the gradient in the direction of each of the incoming weights—hence the importance of having a differentiable activation function. The gradient suggests how steeply the error will be reduced or increased for a change in the weight. The algorithm will attempt to change the weights that result in the greatest reduction in error by an amount known as the **learning rate**. The greater the learning rate, the faster the algorithm will attempt to descend down the gradients, which could reduce training time at the risk of overshooting the valley.

Although this process seems complex, it is easy to apply in practice. Let's apply our understanding of multilayer feedforward networks to a real-world problem.

Modeling the strength of concrete with ANNs

In the field of engineering, it is crucial to have accurate estimates of the performance of building materials. These estimates are required in order to develop safety guidelines governing the materials used in the construction of buildings, bridges, and roadways.

Estimating the strength of concrete is a challenge of particular interest. Although it is used in nearly every construction project, concrete performance varies greatly due to the use of a wide variety of ingredients that interact in complex ways. As a result, it is difficult to accurately predict the strength of the final product. A model that could reliably predict concrete strength given a listing of the composition of the input materials could result in safer construction practices.

Step 1 – collecting data

For this analysis, we will utilize data on the compressive strength of concrete donated to the UCI Machine Learning Data Repository (`http://archive.ics.uci.edu/ml`) by *I-Cheng Yeh*. As he found success using neural networks to model these data, we will attempt to replicate Yeh's work using a simple neural network model in R.

For more information on Yeh's approach to this learning task, refer to: *Modeling of strength of high performance concrete using artificial neural networks, Cement and Concrete Research, Vol. 28,* pp. 1797-1808, by *I-C Yeh* (1998).

According to the website, the concrete dataset contains 1,030 examples of concrete, with eight features describing the components used in the mixture. These features are thought to be related to the final compressive strength, and they include the amount (in kilograms per cubic meter) of cement, slag, ash, water, superplasticizer, coarse aggregate, and fine aggregate used in the product, in addition to the aging time (measured in days).

[To follow along with this example, download the `concrete.csv` file from the Packt Publishing's website and save it to your R working directory.]

Step 2 – exploring and preparing the data

As usual, we'll begin our analysis by loading the data into an R object using the `read.csv()` function and confirming that it matches the expected structure:

```
> concrete <- read.csv("concrete.csv")
> str(concrete)
'data.frame':1030 obs. of  9 variables:
 $ cement      : num  141 169 250 266 155 ...
 $ slag        : num  212 42.2 0 114 183.4 ...
 $ ash         : num  0 124.3 95.7 0 0 ...
 $ water       : num  204 158 187 228 193 ...
 $ superplastic: num  0 10.8 5.5 0 9.1 0 0 6.4 0 9 ...
 $ coarseagg   : num  972 1081 957 932 1047 ...
 $ fineagg     : num  748 796 861 670 697 ...
 $ age         : int  28 14 28 28 28 90 7 56 28 28 ...
 $ strength    : num  29.9 23.5 29.2 45.9 18.3 ...
```

The nine variables in the data frame correspond to the eight features and one outcome we expected, although a problem has become apparent. Neural networks work best when the input data are scaled to a narrow range around zero, and here we see values ranging anywhere from zero up to over a thousand.

Typically, the solution to this problem is to rescale the data with a normalizing or standardization function. If the data follow a bell-shaped curve (a normal distribution as described in *Chapter 2, Managing and Understanding Data*), then it may make sense to use standardization via R's built-in `scale()` function. On the other hand, if the data follow a uniform distribution or are severely non-normal, then normalization to a 0-1 range may be more appropriate. In this case, we'll use the latter.

In *Chapter 3, Lazy Learning – Classification Using Nearest Neighbors*, we defined our own `normalize()` function as:

```
> normalize <- function(x) {
    return((x - min(x)) / (max(x) - min(x)))
  }
```

After executing this code, our `normalize()` function can be applied to every column in the concrete data frame using the `lapply()` function as follows:

```
> concrete_norm <- as.data.frame(lapply(concrete, normalize))
```

To confirm that the normalization worked, we can see that the minimum and maximum strength are now 0 and 1, respectively:

```
> summary(concrete_norm$strength)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
0.0000000 0.2663511 0.4000872 0.4171915 0.5457207 1.0000000
```

In comparison, the original minimum and maximum values were 2.33 and 82.6:

```
> summary(concrete$strength)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
 2.33000 23.71000 34.44500 35.81796 46.13500 82.60000
```

Any transformation applied to the data prior to training the model will have to be applied in reverse later on in order to convert back to the original units of measurement. To facilitate the rescaling, it is wise to save the original data, or at least the summary statistics of the original data.

Following the precedent of *I-Cheng Yeh* in the original publication, we will partition the data into a training set with 75 percent of the examples and a testing set with 25 percent. The CSV file we used was already sorted in random order, so we simply need to divide it into two portions:

```
> concrete_train <- concrete_norm[1:773, ]
> concrete_test <- concrete_norm[774:1030, ]
```

We'll use the training dataset to build the neural network and the testing dataset to evaluate how well the model generalizes to future results. As it is easy to overfit a neural network, this step is very important.

Step 3 – training a model on the data

To model the relationship between the ingredients used in concrete and the strength of the finished product, we will use a multilayer feedforward neural network. The `neuralnet` package by *Stefan Fritsch* and *Frauke Guenther* provides a standard and easy-to-use implementation of such networks. It also offers a function to plot the network topology. For these reasons, the `neuralnet` implementation is a strong choice for learning more about neural networks, though that's not to say that it cannot be used to accomplish real work as well—it's quite a powerful tool, as you will soon see.

There are several other commonly used packages to train ANN models in R, each with unique strengths and weaknesses. Because it ships as part of the standard R installation, the `nnet` package is perhaps the most frequently cited ANN implementation. It uses a slightly more sophisticated algorithm than standard backpropagation. Another strong option is the `RSNNS` package, which offers a complete suite of neural network functionality, with the downside being that it is more difficult to learn.

As `neuralnet` is not included in base R, you will need to install it by typing `install.packages("neuralnet")` and load it with the `library(neuralnet)` command. The included `neuralnet()` function can be used for training neural networks for numeric prediction using the following syntax:

Neural network syntax

using the `neuralnet()` function in the `neuralnet` package

Building the model:

```
m <- neuralnet(target ~ predictors, data = mydata, hidden = 1)
```

- `target` is the outcome in the `mydata` data frame to be modeled
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` specifies the data frame in which the `target` and `predictors` variables can be found
- `hidden` specifies the number of neurons in the hidden layer (by default, 1)

The function will return a neural network object that can be used to make predictions.

Making predictions:

```
p <- compute(m, test)
```

- `m` is a model trained by the `neuralnet()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier

The function will return a list with two components: `$neurons`, which stores the neurons for each layer in the network, and `$net.result`, which stores the model's predicted values.

Example:

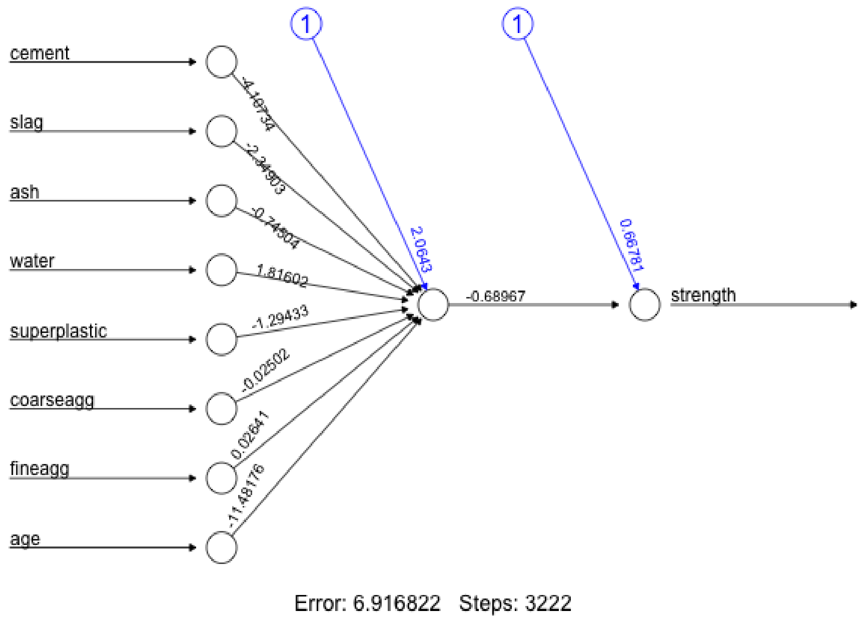
```
concrete_model <- neuralnet(strength ~ cement + slag + ash,
                            data = concrete)
model_results <- compute(concrete_model, concrete_data)
strength_predictions <- model_results$net.result
```

We'll begin by training the simplest multilayer feedforward network with only a single hidden node:

```
> concrete_model <- neuralnet(strength ~ cement + slag +
                              ash + water + superplastic +
                              coarseagg + fineagg + age,
                              data = concrete_train)
```

We can then visualize the network topology using the `plot()` function on the `concrete_model` object:

```
> plot(concrete_model)
```



In this simple model, there is one input node for each of the eight features, followed by a single hidden node and a single output node that predicts the concrete strength. The weights for each of the connections are also depicted, as are the bias terms (indicated by the nodes with a **1**). The plot also reports the number of training steps and a measure called, the **Sum of Squared Errors** (**SSE**). These metrics will be useful when we are evaluating the model performance.

Step 4 – evaluating model performance

The network topology diagram gives us a peek into the black box of the ANN, but it doesn't provide much information about how well the model fits our data. To estimate our model's performance, we can use the `compute()` function to generate predictions on the testing dataset:

```
> model_results <- compute(concrete_model, concrete_test[1:8])
```

Note that the `compute()` function works a bit differently from the `predict()` functions we've used so far. It returns a list with two components: `$neurons`, which stores the neurons for each layer in the network, and `$net.results`, which stores the predicted values. We'll want the latter:

```
> predicted_strength <- model_results$net.result
```

Because this is a numeric prediction problem rather than a classification problem, we cannot use a confusion matrix to examine model accuracy. Instead, we must measure the correlation between our predicted concrete strength and the true value. This provides an insight into the strength of the linear association between the two variables.

Recall that the `cor()` function is used to obtain a correlation between two numeric vectors:

```
> cor(predicted_strength, concrete_test$strength)
             [,1]
[1,] 0.7170368646
```

Don't be alarmed if your result differs. Because the neural network begins with random weights, the predictions can vary from model to model.

Correlations close to 1 indicate strong linear relationships between two variables. Therefore, the correlation here of about 0.72 indicates a fairly strong relationship. This implies that our model is doing a fairly good job, even with only a single hidden node.

A neural network with a single hidden node can be thought of as a distant cousin of the linear regression models we studied in *Chapter 6*, *Forecasting Numeric Data – Regression Methods*. The weight between each input node and the hidden node is similar to the regression coefficients, and the weight for the bias term is similar to the intercept. In fact, if you construct a linear model in the same vein as the previous ANN, the correlation is 0.74.

Given that we only used one hidden node, it is likely that we can improve the performance of our model. Let's try to do a bit better.

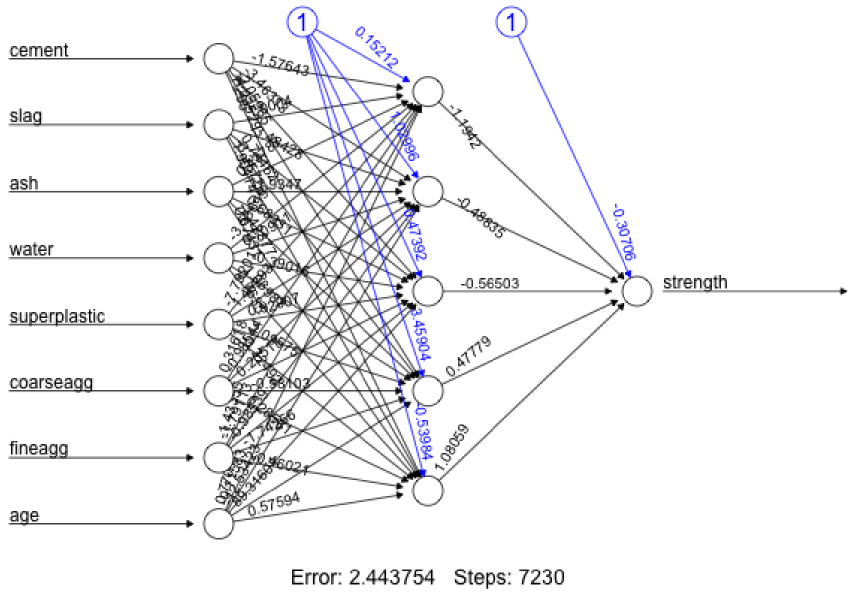
Step 5 – improving model performance

As networks with more complex topologies are capable of learning more difficult concepts, let's see what happens when we increase the number of hidden nodes to five. We use the `neuralnet()` function as before, but add the parameter `hidden = 5`:

```
> concrete_model2 <- neuralnet(strength ~ cement + slag +
                               ash + water + superplastic +
                               coarseagg + fineagg + age,
                               data = concrete_train, hidden = 5)
```

Plotting the network again, we see a drastic increase in the number of connections. How did this impact performance?

```
> plot(concrete_model2)
```



Notice that the reported error (measured again by SSE) has been reduced from 6.92 in the previous model to 2.44 here. Additionally, the number of training steps rose from 3222 to 7230, which is no surprise given how much more complex the model has become.

Applying the same steps to compare the predicted values to the true values, we now obtain a correlation around 0.80, which is a considerable improvement over the previous result:

```
> model_results2 <- compute(concrete_model2, concrete_test[1:8])
> predicted_strength2 <- model_results2$net.result
> cor(predicted_strength2, concrete_test$strength)
            [,1]
[1,] 0.801444583
```

Interestingly, in the original publication, *I-Cheng Yeh* reported a mean correlation of 0.885 using a very similar neural network. For some reason, we fell a bit short. In our defense, he is a civil engineering professor; therefore, he may have applied some subject matter expertise to the data preparation. If you'd like more practice with neural networks, you might try applying the principles learned earlier in this chapter to beat his result, perhaps by using different numbers of hidden nodes, applying different activation functions, and so on. The `?neuralnet` help page provides more information on the various parameters that can be adjusted.

Understanding Support Vector Machines

A **Support Vector Machine** (**SVM**) can be imagined as a surface that defines a boundary between various points of data which represent examples plotted in multidimensional space according to their feature values. The goal of an SVM is to create a flat boundary, called a **hyperplane**, which leads to fairly homogeneous partitions of data on either side. In this way, SVM learning combines aspects of both the instance-based nearest neighbor learning presented in *Chapter 3*, *Lazy Learning – Classification Using Nearest Neighbors*, and the linear regression modeling described in *Chapter 6*, *Forecasting Numeric Data – Regression Methods*. The combination is extremely powerful, allowing SVMs to model highly complex relationships.

Although the basic mathematics that drive SVMs have been around for decades, they have recently exploded in popularity. This is of course rooted in their state-of-the-art performance, but perhaps also due to the fact that award winning SVM algorithms have been implemented in several popular and well-supported libraries across many programming languages, including R. This has led SVMs to be adopted by a much wider audience who previously might have passed it by due to the somewhat complex math involved with SVM implementation. The good news is that although the math may be difficult, the basic concepts are understandable.