IN THIS CHAPTER

- Upgrading the perceptron to the interconnection paradigm
- Structuring neural architectures made of nodes and connections
- Getting a glimpse of the backpropagation algorithm
- Understanding what deep learning is and what it can achieve

Chapter 16
Hitting Complexity with Neural Networks

As you journey in the world of machine learning, you often see metaphors from the natural world to explain the details of algorithms. This chapter presents a family of learning algorithms that directly derives inspiration from how the brain works. They are neural networks, the core algorithms of the connectionists' tribe.

Starting with the idea of reverse-engineering how a brain processes signals, the connectionists base neural networks on biological analogies and their components, using brain terms such as neurons and axons as names. However, you'll discover that neural networks resemble nothing more than a sophisticated kind of linear regression when you check their math formulations. Yet, these algorithms are extraordinarily effective against complex problems such as image and sound recognition, or machine language translation. They also execute quickly when predicting.

Well-devised neural networks use the name *deep learning* and are behind such power tools as Siri and other digital assistants. They are behind the more astonishing machine learning applications as well. For instance, you see them at work in this incredible demonstration by Microsoft CEO Rick Rashid, who is

speaking in English while being simultaneously translated into Chinese: `https://www.youtube.com/watch?v=Nu-nlQqFCKg`. If an AI revolution is about to happen, the increased learning capabilities of neural networks will likely drive it.

Learning and Imitating from Nature

The core neural network algorithm is the *neuron* (also called a *unit*). Many neurons arranged in an interconnected structure make up a neural network, with each neuron linking to the inputs and outputs of other neurons. Thus, a neuron can input features from examples or the results of other neurons, depending on its location in the neural network.

Something similar to the neuron, the perceptron, appears earlier in this book, although it uses a simpler structure and function. When the psychologist Rosenblatt conceived the perceptron, he thought of it as a simplified mathematical version of a brain neuron. A perceptron takes values as inputs from the nearby environment (the dataset), weights them (as brain cells do, based on the strength of the in-bound connections), sums all the weighted values, and activates when the sum exceeds a threshold. This threshold outputs a value of 1; otherwise, its prediction is 0. Unfortunately, a perceptron can't learn when the classes it tries to process aren't linearly separable. However, scholars discovered that even though a single perceptron couldn't learn the logical operation XOR shown in Figure 16-1 (the exclusive or, which is true only when the inputs are dissimilar), two perceptrons working together could.

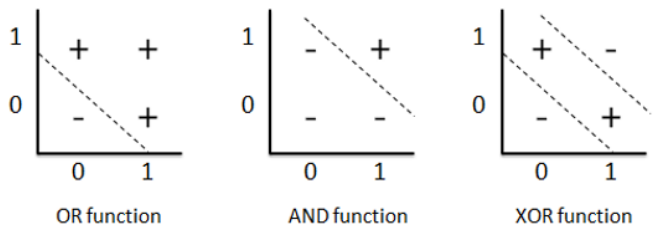

FIGURE 16-1: Learning logical XOR using a single separating line isn't possible.

Neurons in a neural network are a further evolution of the perceptron: they take many weighted values as inputs, sum them, and provide the summation as the result, just as a perceptron does. However, they also provide a more sophisticated transformation of the summation, something that the perceptron can't do. In observing nature, scientists noticed that neurons receive signals but don't always release a signal of their own. It depends on the amount of signal received. When a neuron acquires enough stimuli, it fires an answer; otherwise it remains silent. In a similar fashion, algorithmic neurons, after receiving weighted values, sum them

and use an *activation function* to evaluate the result, which transforms it in a non-linear way. For instance, the activation function can release a zero value unless the input achieves a certain threshold, or it can dampen or enhance a value by nonlinearly rescaling it, thus transmitting a rescaled signal.

A neural network has different activation functions, as shown in Figure 16-2. The linear function doesn't apply any transformation, and it's seldom used because it reduces a neural network to a regression with polynomial transformations. Neural networks commonly use the sigmoid or the hyperbolic tan.

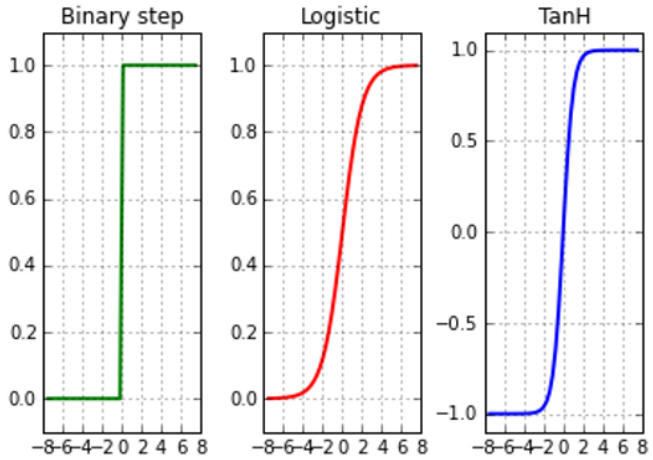

FIGURE 16-2: Plots of different activation functions.

The figure shows how an input (expressed on the horizontal axis) can transform an output into something else (expressed on the vertical axis). The examples show a binary step, a logistic, and a tangent hyperbolic activation function.

TIP

You learn more about activation functions later in the chapter, but note for now that activation functions clearly work well in certain ranges of x values. For this reason, you should always rescale inputs to a neural network using statistical standardization (zero mean and unit variance) or normalize the input in the range from 0 to 1 or from −1 to 1.

Going forth with feed-forward

In a neural network, you have first to consider the architecture, which is how the neural network components are arranged. Contrary to other algorithms, which have a fixed pipeline that determines how algorithms receive and process data, neural networks require you to decide how information flows by fixing the number of units (the neurons) and their distribution in layers, as shown in Figure 16-3.

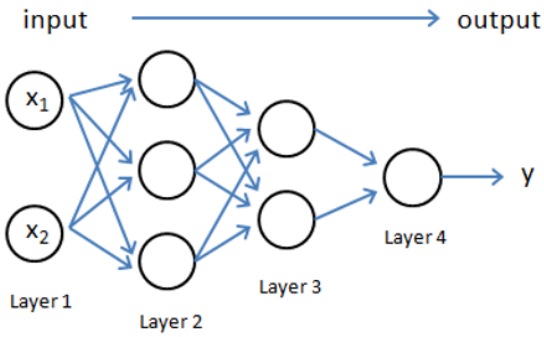

FIGURE 16-3: An example of the architecture of a neural network.

The figure shows a simple neural architecture. Note how the layers filter information in a progressive way. This is a feed-forward input because data feeds one way forward into the network. Connections exclusively link the units in one layer with the units in the following layer (information flow from left to right). No connections exist between units in the same layer or with units outside the next layer. Moreover, the information pushes forward (from the left to the right). Processed data never returns to previous neuron layers.

Using a neural network is like using stratified filtering system for water: You pour the water from above and the water is filtered at the bottom. The water has no way to go back; it just goes forward and straight down, and never laterally. In the same way, neural networks force data features to flow through the network and mix with each other only according to the network's architecture. By using the best architecture to mix features, the neural network creates new composed features at every layer and helps achieve better predictions. Unfortunately, there is no way to determine the best architecture without empirically trying different solutions and testing whether output data helps predict your target values after flowing through the network.

The first and last layers play an important role. The first layer, called the *input layer,* picks up the features from each data example processed by the network. The last layer, called the *output layer,* releases the results.

A neural network can process only numeric, continuous information; it can't be constrained to work with qualitative variables (for example, labels indicating a quality such as red, blue, or green in an image). You can process qualitative variables by transforming them into a continuous numeric value, such as a series of binary values, as discussed in Chapter 9 in the material about working with data. When a neural network processes a binary variable, the neuron treats the variable as a generic number and turns the binary values into other values, even negative ones, by processing across units.

Note the limitation of dealing only with numeric values, because you can't expect the last layer to output a nonnumeric label prediction. When dealing with a regression problem, the last layer is a single unit. Likewise, when you're working with a classification and you have output that must choose from a number n of classes, you should have n terminal units, each one representing a score linked to the probability of the represented class. Therefore, when classifying a multiclass problem such as iris species (as in the Iris dataset demonstration found in Chapter 14), the final layer has as many units as species. For instance, in the classical iris classification example, created by the famous statistician Fisher, you have three classes: `setosa`, `versicolor`, and `virginica`. In a neural network based on the Iris dataset, you therefore have three units representing one of the three iris species. For each example, the predicted class is the one that gets the higher score at the end.


TIP

In some neural networks, there are special final layers, called a softmax, which can adjust the probability of each class based on the values received from a previous layer.


REMEMBER

In classification, the final layer may represent both a partition of probabilities thanks to softmax (a multiclass problem in which total probabilities sum to 100 percent) or an independent score prediction (because an example can have more classes, which is a multilabel problem in which summed probabilities can be more than 100 percent). When the classification problem is a binary classification, a single node suffices. Also, in regression, you can have multiple output units, each one representing a different regression problem (for instance, in forecasting, you can have different predictions for the next day, week, month, and so on).

Going even deeper down the rabbit hole

Neural networks have different layers, each one having its own weights. Because the neural network segregates computations by layers, knowing the reference layer is important because it means accounting for certain units and connections. Thus you can refer to every layer using a specific number and generically talk about each layer using the letter l.

Each layer can have a different number of units, and the number of units located between two layers dictates the number of connections. By multiplying the number of units in the starting layer with the number in the following layer, you can determine the total number of connections between the two: $number\ of\ connections^{(l)} = units^{(l)} * units^{(l+1)}$.

A matrix of weights, usually named with the uppercase Greek letter theta (Θ), represents the connections. For ease of reading, the book uses the capital letter W, which is a fine choice because it is a matrix. Thus, you can use W^1 to refer to the

connection weights from layer 1 to layer 2, W^2 for the connections from layer 2 to layer 3, and so on.


REMEMBER

You may see references to the layers between the input and the output as *hidden layers* and count layers starting from the first hidden layer. This is just a different convention from the one used in the book. The examples in the book always start counting from the input layer, so the first hidden layer is layer number 2.

Weights represent the strength of the connection between neurons in the network. When the weight of the connection between two layers is small, it means that the network dumps values flowing between them and signals that taking this route won't likely influence the final prediction. On the contrary, a large positive or negative value affects the values that the next layer receives, thus determining certain predictions. This approach is clearly analogous to brain cells, which don't stand alone but are in connection with other cells. As someone grows in experience, connections between neurons tend to weaken or strengthen to active or deactivate certain brain network cell regions, causing other processing or an activity (a reaction to a danger, for instance, if the processed information signals a life-threatening situation).

Now that you know some conventions regarding layers, units, and connections, you can start examining the operations that neural networks execute in detail. First, you can call inputs and outputs in different ways:

- **a:** The result stored in a unit in the neural network after being processed by the activation function (called *g*). This is the final output that is sent further along the network.
- **z:** The multiplication between a and the weights from the W matrix. z represents the signal going through the connections, analogous to water in pipes that flows at a higher or lower pressure depending on the pipe thickness. In the same way, the values received from the previous layer get higher or lower values because of the connection weights used to transmit them.

Each successive layer of units in a neural network progressively processes the values taken from the features, same as in a conveyor belt. As data transmits in the network, it arrives into each unit as a value produced by the summation of the values present in the previous layer and weighted by connections represented in the matrix W. When the data with added bias exceeds a certain threshold, the activation function increases the value stored in the unit; otherwise, it extinguishes the signal by reducing it. After processing by the activation function, the result is ready to push forward to the connection linked to the next layer. These steps repeat for each layer until the values reach the end and you have a result, as shown in Figure 16-4.

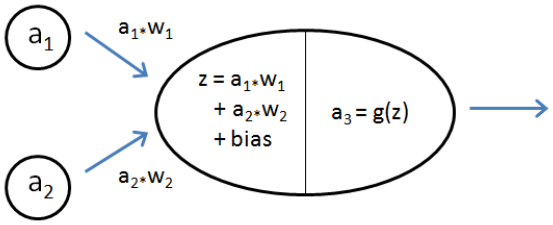

FIGURE 16-4: A detail of the feed-forward process in a neural network.

The figure shows a detail of the process that involves two units pushing their results to another unit. This event happens in every part of the network. When you understand the passage from two neurons to one, you can understand the entire feed-forward process, even when more layers and neurons are involved. For more explanation, here are the seven steps used to produce a prediction in a neural network made of four layers (like the one shown earlier in the chapter in Figure 16-3):

1. The first layer (notice the superscript 1 on a) loads the value of each feature in a different unit:

$$a^{(1)} = X$$

2. The weights of the connections bridging the input layer with the second layer are multiplied by the values of the units in the first layer. A matrix multiplication weights and sums the inputs for the second layer together.

$$z^{(2)} = W^{(1)} a^{(1)}$$

3. The algorithm adds a bias constant to layer two before running the activation function. The activation function transforms the second-layer inputs. The resulting values are ready to pass to the connections.

$$a^{(2)} = g(z^{(2)} + bias^{(2)})$$

4. The third-layer connections weigh and sum the outputs of layer two.

$$z^{(3)} = W^{(2)} a^{(2)}$$

5. The algorithm adds a bias constant to layer three before running the activation function. The activation function transforms the layer-three inputs.

$$a^{(3)} = g(z^{(3)} + bias^{(3)})$$

6. The layer-three outputs are weighted and summed by the connections to the output layer.

$$z^{(4)} = W^{(3)} a^{(3)}$$

7. Finally, the algorithm adds a bias constant to layer four before running the activation function. The output units receive their inputs and transform the input using the activation function. After this final transformation, the output units are ready to release the resulting predictions of the neural network.

$$a^{(4)} = g(z^{(4)} + bias^{(4)})$$

The activation function plays the role of a signal filter, helping to select the relevant signals and avoid the weak and noisy ones (because it discards values below a certain threshold). Activation functions also provide nonlinearity to the output because they enhance or damp the values passing through them in a nonproportional way.


REMEMBER

The weights of the connections provide a way to mix and compose the features in a new way, creating new features in a way not too different from a polynomial expansion. The activation renders nonlinear the resulting recombination of the features by the connections. Both of these neural network components enable the algorithm to learn complex target functions that represent the relationship between the input features and the target outcome.

Getting Back with Backpropagation

From an architectural perspective, a neural network does a great job of mixing signals from examples and turning them into new features to achieve an approximation of complex nonlinear functions (functions that you can't represent as a straight line in the features' space). To create this capability, neural networks work as *universal approximators* (for more details, go to `https://en.wikipedia.org/wiki/Universal_approximation_theorem`), which means that they can guess any target function. However, you have to consider that one aspect of this feature is the capacity to model complex functions *(representation capability),* and another aspect is the capability to learn from data effectively. Learning occurs in a brain because of the formation and modification of synapses between neurons, based on stimuli received by trial-and-error experience. Neural networks provide a way to replicate this process as a mathematical formulation called *backpropagation.*

Since its early appearance in the 1970s, the backpropagation algorithm has been given many fixes. Each neural network learning process improvement resulted in new applications and a renewed interest in the technique. In addition, the current deep learning revolution, a revival of neural networks, which were abandoned at the beginning of the 1990s, is due to key advances in the way neural networks learn from their errors. As seen in other algorithms, the cost function activates the necessity to learn certain examples better (large errors correspond to high costs). When an example with a large error occurs, the cost function outputs a high value that is minimized by changing the parameters in the algorithm.

In linear regression, finding an update rule to apply to each parameter (the vector of beta coefficients) is straightforward. However, in a neural network, things are a bit more complicated. The architecture is variable and the parameter coefficients (the connections) relate to each other because the connections in a layer depend on how the connections in the previous layers recombined the inputs. The solution to this problem is the backpropagation algorithm. Backpropagation is a smart way to propagate the errors back into the network and make each connection adjust its weights accordingly. If you initially feed-forward propagated information to the network, it's time to go backward and give feedback on what went wrong in the forward phase.

Discovering how backpropagation works isn't complicated, even though demonstrating how it works using formulas and mathematics requires derivatives and the proving of some formulations, which is quite tricky and beyond the scope of this book. To get a sense of how backpropagation operates, start from the end of the network, just at the moment when an example has been processed and you have a prediction as an output. At this point, you can compare it with the real result and, by subtracting the two results, get an offset, which is the error. Now that you know the mismatch of the results at the output layer, you can progress backward in order to distribute it along all the units in the network.

TIP

The cost function of a neural network for classification is based on cross-entropy (as seen in logistic regression):

$$Cost = y * \log\left(h_W(X)\right) + (1-y) * \log\left(1 - h_W(X)\right)$$

This is a formulation involving logarithms. It refers to the prediction produced by the neural network and expressed as $h_W(X)$ (which reads as the result of the network given connections W and X as input). To make things easier, when thinking of the cost, it helps to simply think of the formulation as computing the offset between the expected results and the neural network output.

The first step in transmitting the error back into the network relies on backward multiplication. Because the values fed to the output layer are made of the contributions of all units, proportional to the weight of their connections, you can redistribute the error according to each contribution. For instance, the vector of errors of a layer n in the network, a vector indicated by the Greek letter delta (δ), is the result of the following formulation:

$$\delta^{(n)} = W^{(n)T} * \delta^{(n+1)}$$

This formula says that, starting from the final delta, you can continue redistributing delta going backward in the network and using the weights you used to push forward the value to partition the error to the different units. In this way, you can get the terminal error redistributed to each neural unit, and you can use it to

recalculate a more appropriate weight for each network connection to minimize the error. To update the weights W of layer l, you just apply the following formula:

$$W^{(l)} = W^{(1)} + \eta * \delta^{(1)} * g'\left(z^{(l)}\right) * a^{(1)}$$

It may appear to be a puzzling formula at first sight, but it is a summation, and you can discover how it works by looking at its elements. First, look at the function g'. It's the first derivative of the activation function g, evaluated by the input values z. This book discusses using derivatives in Chapter 10. In fact, this is the gradient descent method. Gradient descent determines how to reduce the error measure by finding, among the possible combinations of values, the weights that most reduce the error.

The Greek letter eta (η), sometimes also called alpha (α) or epsilon (ε) depending on the textbook you consult, is the learning rate. As found in other algorithms, it reduces the effect of the update suggested by the gradient descent derivative. In fact, the direction provided may be only partially correct or just roughly correct. By taking multiple small steps in the descent, the algorithm can take a more precise direction toward the global minimum error, which is the target you want to achieve (that is, a neural network producing the least possible prediction error).

Different methods are available for setting the right eta value, because the optimization largely depends on it. One method sets the eta value starting high and reduces it during the optimization process. Another method variably increases or decreases eta based on the improvements obtained by the algorithm: Large improvements call a larger eta (because the descent is easy and straight); smaller improvements call a smaller eta so that the optimization will move slower, looking for the best opportunities to descend. Think of it as being on a tortuous path in the mountains: You slow down and try not to be struck or thrown off the road as you descend.


TIP

Most implementations offer an automatic setting of the correct eta. You need to note this setting's relevance when training a neural network because it's one of the important parameters to tweak to obtain better predictions, together with the layer architecture.

Weight updates can happen in different ways with respect to the training set of examples:

- **Online mode:** The weight update happens after every example traverses the network. In this way, the algorithm treats the learning examples as a stream from which to learn in real time. This mode is perfect when you have to learn *out-of-core,* that is, when the training set can't fit into RAM memory. However, this method is sensitive to outliers, so you have to keep your

learning rate low. (Consequently, the algorithm is slow to converge to a solution.)

- **Batch mode:** The weight update happens after seeing all the examples in the training set. This technique makes optimization fast and less subject to having variance appear in the example stream. In batch mode, the backpropagation considers the summed gradients of all examples.
- **Mini-batch (or stochastic) mode:** The weight update happens after the network has processed a subsample of randomly selected training set examples. This approach mixes the advantages of online mode (low memory usage) and batch mode (a rapid convergence), while introducing a random element (the subsampling) to avoid having the gradient descent stuck in a local minima.

Struggling with Overfitting

Given the neural network architecture, you can imagine how easily the algorithm could learn almost anything from data, especially if you added too many layers. In fact, the algorithm does so well that its predictions are often affected by a high estimate variance called *overfitting*. Overfitting causes the neural network to learn every detail of the training examples, which makes it possible to replicate them in the prediction phase. But apart from the training set, it won't ever correctly predict anything different. The following sections discuss some of the issues with overfitting in more detail.

Understanding the problem

When you use a neural network for a real problem, you have to take some cautionary steps in a much stricter way than you do with other algorithms. Neural networks are frailer and more prone to relevant errors than other machine learning solutions.

First, you carefully split your data into training, validation, and test sets. Before the algorithm learns from data, you must evaluate the goodness of your parameters: architecture (the number of layers and nodes in them); activation functions; learning parameter; and number of iterations. In particular, the architecture offers great opportunities to create powerful predictive models at a high risk of overfitting. The learning parameter controls how fast a network learns from data, but it may not suffice in preventing overfitting the training data.

You have two possible solutions to this problem. The first is regularization, as in linear and logistic regression. You can sum all connection coefficients, squared or in absolute value, to penalize models with too many coefficients with high values (achieved by L2 regularization) or with values different from zero (achieved by L1 regularization). The second solution is also effective because it controls when overfitting happens. It's called *early-stop* and works by checking the cost function on the validation set as the algorithm learns from the training set.


TIP

You may not realize when your model starts overfitting. The cost function calculated using the training set keeps improving as optimization progresses. However, as soon as you start recording noise from the data and stop learning general rules, you can check the cost function on an out-of-sample (the validation sample). At some point, you'll notice that it stops improving and starts worsening, which means that your model has reached its learning limit.

Opening the black box

The best way to learn how to build a neural network is to build one. Python offers a wealth of possible implementations for neural networks and deep learning. Python has libraries such as Theano (`http://deeplearning.net/software/theano/`), which allows complex computations at an abstract level, and more practical packages, such as Lasagne (`https://github.com/Lasagne/Lasagne`), which allows you to build neural networks, though it still requires some abstractions. For this reason, you need wrappers, such as Nolearn, which is compatible with Scikit-learn (`https://github.com/dnouri/nolearn`), or Keras (`https://github.com/fchollet/keras`), which can also wrap the TensorFlow (`https://github.com/tensorflow/tensorflow`) library released by Google that has the potential to replace Theano as a software library for neural computation.

R provides libraries that are less complicated and more accessible, such as nnet (`https://cran.r-project.org/web/packages/nnet/`), AMORE (`https://cran.r-project.org/web/packages/AMORE/`), and neuralnet (`https://cran.r-project.org/web/packages/neuralnet/`). These brief examples in R show how to train both a classification network (on the Iris dataset) and a regression network (on the Boston dataset). Starting from classification, the following code loads the dataset and splits it into training and test sets:

```
library(MASS)
library("neuralnet")

target <- model.matrix( ~ Species - 1, data=iris )
colnames(target) <- c("setosa", "versicolor", "virginica")
```

```
set.seed(101)
index <- sample(1:nrow(iris), 100)

train_predictors <- iris[index, 1:4]
test_predictors  <- iris[-index, 1:4]
```

Because neural networks rely on gradient descent, you need to standardize or normalize the inputs. Normalizing is better so that the minimum is zero and the maximum is one for every feature. Naturally, you learn how to make the numeric conversion using the training set only in order to avoid any chance of using information from the test out-of-sample.

```
min_vector   <- apply(train_predictors, 2, min)
range_vector <- apply(train_predictors, 2, max) -
  apply(train_predictors, 2, min)

train_scaled <- cbind(scale(train_predictors,
                            min_vector, range_vector),
                            target[index,])
test_scaled  <- cbind(scale(test_predictors,
                            min_vector, range_vector),
                            target[-index,])

summary(train_scaled)
```

When the training set is ready, you can train the model to guess three binary variables, with each one representing a class. The output is a value for each class proportional to its probability of being the real class. You pick a prediction by taking the highest value. You can also visualize the network by using the internal plot and thus seeing the neural network architecture and the assigned weights, as shown in Figure 16-5.

```
set.seed(102)
nn_iris <- neuralnet(setosa + versicolor + virginica ~
                     Sepal.Length + Sepal.Width
                     + Petal.Length + Petal.Width,
                     data=train_scaled, hidden=c(2),
                     linear.output=F)

plot(nn_iris)

predictions <- compute(nn_iris, test_scaled[,1:4])
y_predicted <- apply(predictions$net.result,1,which.max)
y_true <- apply(test_scaled[,5:7],1,which.max)
```

```
confusion_matrix <- table(y_true, y_predicted)
accuracy <- sum(diag(confusion_matrix)) /
  sum(confusion_matrix)
print (confusion_matrix)
print (paste("Accuracy:",accuracy))
```

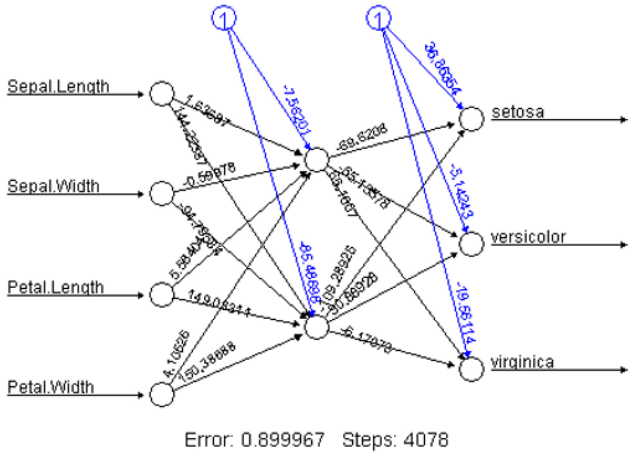

FIGURE 16-5: You can plot a trained neural network.

The following example demonstrates how to predict house values in Boston, using the Boston dataset. The procedure is the same as in the previous classification, but here you have a single output unit. The code also plots the test set's predicted results against the real values to verify the good fit of the model.

```
no_examples <- nrow(Boston)
features <- colnames(Boston)

set.seed(101)
index <- sample(1:no_examples, 400)

train <- Boston[index,]
test  <- Boston[-index,]

min_vector <- apply(train,2,min)
range_vector <- apply(train,2,max) - apply(train,2,min)
scaled_train <- scale(train,min_vector,range_vector)
scaled_test  <- scale(test, min_vector,range_vector)

formula = paste("medv ~", paste(features[1:13],
                                collapse='+'))
```

```
nn_boston <- neuralnet(formula, data=scaled_train,
                       hidden=c(5,3), linear.output=T)
predictions <- compute(nn_boston, scaled_test[,1:13])
predicted_values <- (predictions$net.result *
                       range_vector[14]) + min_vector[14]

RMSE <- sqrt(mean((test[,14] - predicted_values)^2))
print (paste("RMSE:",RMSE))
plot(test[,14],predicted_values, cex=1.5)
abline(0,1,lwd=1)
```

Introducing Deep Learning

After backpropagation, the next improvement in neural networks led to deep learning. Research continued in spite of AI winter and neural networks started to take advantage of the developments in CPUs and GPUs (the graphic processing units better known for their application in gaming but which are actually powerful computing units for matrix and vector calculations). These technologies make training neural networks an achievable task in a shorter time and accessible to more people. Research also opened a world of new applications. Neural networks can learn from huge amounts of data, and because they're more prone to high variance than to bias, they can take advantage of big data, creating models that continuously perform better, depending on the amounts of data you feed them. However, you need large, complex networks for certain applications (to learn complex features, such as the characteristics of a series of images) and thus incur problems like the vanishing gradient.

In fact, when training a large network, the error redistributes among the neurons favoring the layers nearest to the output layer. Layers that are further away receive smaller errors, sometimes too small, making training slow if not impossible. Thanks to the studies of scholars such as Geoffrey Hinton, new turnarounds help avoid the problem of the vanishing gradient. The result definitely helps a larger network, but deep learning isn't just about neural networks with more layers and units.

In addition, something inherently qualitative changed in deep learning as compared to shallow neural networks, shifting the paradigm in machine learning from feature creation (features that make learning easier) to feature learning (complex features automatically created on the basis of the actual features). Big players such as Google, Facebook, Microsoft, and IBM spotted the new trend and since 2012 have started acquiring companies and hiring experts (Hinton now works with Google; LeCun leads Facebook AI research) in the new fields of deep

learning. The Google Brain project, run by Andrew Ng and Jeff Dean, put together 16,000 computers to calculate a deep learning network with more than a billion weights, thus enabling unsupervised learning from YouTube videos.

There is a reason why the quality of deep learning is different. Of course, part of the difference is the increased usage of GPUs. Together with parallelism (more computers put in clusters and operating in parallel), GPUs allow you to successfully apply pretraining, new activation functions, convolutional networks, and *drop-out*, a special kind of regularization different from L1 and L2. In fact, it has been estimated that a GPU can perform certain operations 70 times faster than any CPU, allowing a cut in training times for neural networks from weeks to days or even hours (for reference see `http://www.machinelearning.org/archive/icml2009/papers/218.pdf`).

Both pretraining and new activation functions help solve the problem of the vanishing gradient. New activation functions offer better derivative functions, and pretraining helps start a neural network with better initial weights that require just a few adjustments in the latter parts of the network. Advanced pretraining techniques such as Restricted Boltzanman Machines (`https://en.wikipedia.org/wiki/Restricted_Boltzmann_machine`), Autoencoders (`https://en.wikipedia.org/wiki/Autoencoder`), and Deep Belief Networks (`https://en.wikipedia.org/wiki/Deep_belief_network`) elaborate data in an unsupervised fashion by establishing initial weights that don't change much during the training phase of a deep learning network. Moreover, they can produce better features representing the data and thus achieve better predictions.

Given the high reliance on neural networks for image recognition tasks, deep learning has achieved great momentum thanks to a certain type of neural network, the convolutional neural networks. Discovered in the 1980s, such networks now bring about astonishing results because of the many deep learning additions (for reference, see `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`).

To understand the idea behind convolutional neural networks, think about the convolutions as filters that, when applied to a matrix, transform certain parts of the matrix, make other parts disappear, and make other parts stand out. You can use convolution filters for borders or for specific shapes. Such filters are also useful for finding details in images that determine what the image shows. Humans know that a car is a car because it has a certain shape and certain features, not because they have previously seen every type of cars possible. A standard neural network is tied to its input, and if the input is a pixel matrix, it recognizes shapes and features based on their position on the matrix. Convolution neural networks can elaborate images better than a standard neural network because

- The network specializes particular neurons to recognize certain shapes (thanks to convolutions), so that same capability to recognize a shape doesn't need to appear in different parts of the network.
- By sampling parts of an image into a single value (a task called *pooling*), you don't need to strictly tie shapes to a certain position (which would make it impossible to rotate them). The neural network can recognize the shape in every rotation or distortion, thus assuring a high capacity of generalization of the convolutional network.

Finally, *drop-out* is a new type of regularization that is particularly effective with deep convolutional networks, but it also works with all deep learning architectures, which acts by temporarily and randomly removing connections between the neurons. This approach removes connections that collect only noise from data during training. Also, this approach helps the network learn to rely on critical information coming from different units, thus increasing the strength of the correct signals passed along the layers.