13. Neural Networks

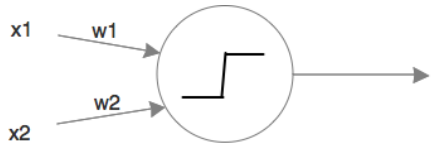

Figure 13.1: Perceptron

The idea for neural networks, also called artificial neural networks (ANNs), was originally inspired by biological neural networks in mammals. Frank Rosenblatt is credited with developing the perceptron for pattern recognition in the late 1950s. A perceptron, as illustrated in Figure 13.1, takes several input values and produces an output signal. In this example, there are two inputs x_1 and x_2 and these are multiplied by their respective weights and sent through the activation function. The activation function, here illustrated as a step function, will output +1 if the combined inputs reaches a certain threshold, and -1 otherwise. The total input to the perceptron is: $\sum_i w_i x_i$

The perceptron was overhyped by its proponents and the press as has often been the case in advances throughout the history of AI. A backlash came in 1969 with a book by Minsky and Papert[1] that described the limitations of the single-layer perceptron, for example, that it could not learn the XOR function. Although it was demonstrated soon thereafter that combining perceptrons in multiple layers could learn XOR, the negative perception persisted for decades. In fact, a network of these perceptrons can learn any function that can be expressed with step functions. Further improvements were made by replacing the step function as the activation function with smoother functions such as the sigmoid. Having a sigmoid activation function allows the network to do regression as

[1]Minsky, Marvin, and Seymour Papert. "Perceptrons: Anlntroduction to Computational Geometry." (1969).

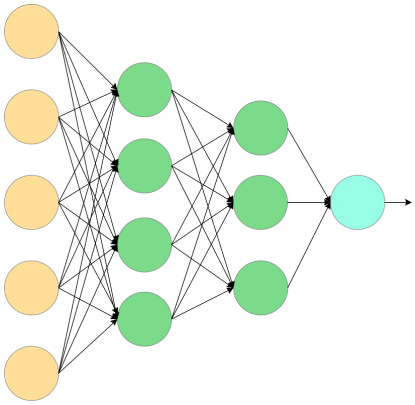

Figure 13.2: Neural Network with 2 Hidden Layers

well as classification. A neural network, then is the evolution of the perceptron to include multiple neurons in a network with the sigmoid or other flexible activation function. This is typically called a feed-forward network because the variables and weights are calculated left to right, going forward through the network.

A neural network is illustrated conceptually in Figure 13.2. Reading forward through the network, left to right, we see 5 input nodes in yellow. There are two hidden layers illustrated with green nodes, 4 in the first hidden layer and 3 in the second hidden layer. Finally the output layer in blue gives the output of the network. In a feed forward network such as this, each node's output is an input to the next layer. Further, each connection in the illustration has a weight value that is multiplied by the output of the source node to compute the input to the destination node. Each of the green and blue nodes will have an activation function such as the sigmoid which determines its output. Each individual node (neuron) in the hidden layers can learn something different, essentially learning different things from the input so that the network is essentially a composition of functions. A "deep" neural network as discussed in the next chapter is a network with many hidden layers.

13.1 Neural Network Regression Example

In this section we build a neural network for regression using the package `neuralnet`. We will use the Boston data set to predict median home value from all predictors. In the online notebook we first divide the data into 75% train 25% test and run the linear regression algorithm as a baseline. Then we scale the data because neural networks perform better on scaled data. Unfortunately, the neuralnet() function has some issue with the tilde-dot formulas. So we have to build the formula before calling the function. We also set a seed. The neural network will initialize the weights randomly (unless we specify initial weights). This random start will produce slightly different results each run. Setting a seed will give us reproducible results. Setting a seed is something that is normally only needed once in an R script. We have noticed that we got different results if we didn't also include it right before the neuralnet() function, so we repeat the set.seed() function.

Code 13.1.1 — Neural Network. Boston Data.

```
library(neuralnet)
n <- names(train)
f <- as.formula(paste("medv ~", paste(n[!n %in% "medv"],
    collapse = " + ")))
set.seed(1234)
nn1 <- neuralnet(f, data=train_scaled, hidden=c(6, 3),
                 linear.output = TRUE)
plot(nn1)
```

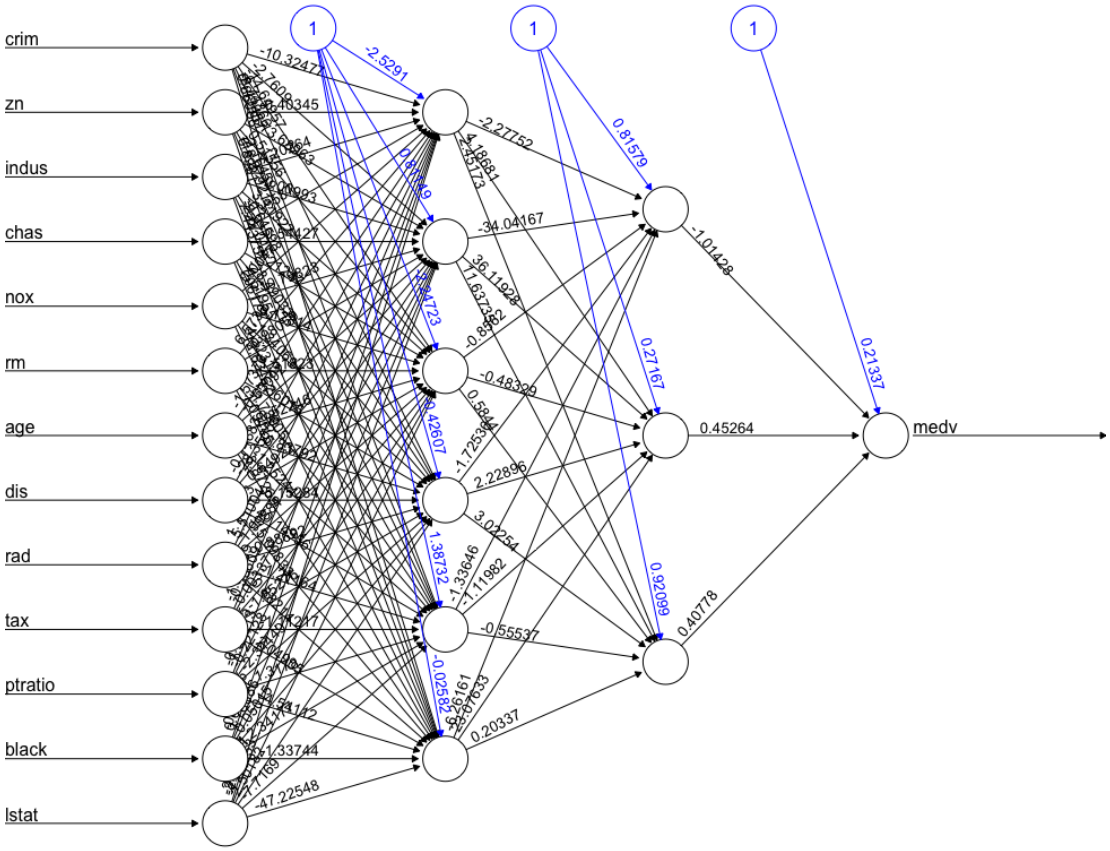

Figure 13.3: Neural Network for the Boston Data

13.1.1 Results

Figure 13.3 shows the plot of the network. We requested two hidden layers, the first with 6 neurons and the second with 3. The plot shows the feed-forward connections and the weights on the black arrows. Notice also the blue arrows representing bias terms. Each hidden node and the output node has its own bias term. The correlation and mse for the linear regression model versus the neural network are as follows:

	Linear Regression	Neural Network
correlation	0.888	0.951
mse	23.094	15.479

Table 13.1: Results on the Boston Housing Data

We got improved results with the neural network over linear regression; however, there is a bit of trial and error and luck involved. On a small data set, linear regression is likely to outperform a neural network. Plus, the linear regression model is easy to fit and highly interpretable. Neural networks can learn complex relationships that simple algorithms like linear regression cannot but a neural network is not necessarily the best choice. Another distinction between the two algorithms is that linear regression will find the optimal parameters whereas a neural network may find a local optima and there is no guarantee that it is the best you can do. Unfortunately there is a lot of trial and error involved in working with neural networks.

In the online notebook you will see that to find the mse we had to reverse the scaling we did earlier to get mse in units of the original data.

13.1.2 Hidden Nodes and Layers

The most important decision when building a neural network is its architecture, or topology. The input and output layers should be fairly intuitive for a given problem but designing the hidden layers is challenging. How many hidden layers? How many nodes in the hidden layers? Having too few hidden nodes may result in underfitting while having two many can result in overfitting. There are a few rules of thumb to find the number of hidden nodes:

- between 1 and the number of predictors
- two-thirds of the input layer size plus the size of the output layer
- < twice the input layer size

Following these very general guidelines for the Boston data with 13 predictors, the suggestions are: (1) 1 - 13, (2) 9, and (3) < 26. We tried 9 hidden nodes and arranged them in two layers.

Are there any advantages in having two layers rather than 1? In this particular problem, the results were dramatically worse than linear regression when we tried one hidden layer with 9 nodes rather than spreading the nodes into two layers. If the mapping of the inputs to the outputs is smooth, one layer should be enough. Having two layers can capture a more complex relationship, at the cost of extra training time. A potential downside of more layers is that you may overfit if you have a small training set. A simpler architecture is usually recommended for small data.

13.2 The Algorithm

In Figure 13.3 you can see that each neuron has its own bias term and weight, drawn in blue. Every neuron in the hidden and output layers will have a bias term, as you can see in the neural net plots above. The bias is like the intercept term in a linear regression or logistic regression model, it is always 1, and multiplied by its weight. In a feed forward neural network, each input including the bias term is multiplied by its weight to get a sum

of inputs. This is just basic matrix multiplication which is optimized in modern computers to be very fast. Once a neuron or node receives its sum of weighted inputs, the activation function kicks in. In a simple activation function, if the sum is over a certain threshold, the output will be 1 otherwise it will be 0.

So far we have not described any learning. We have just described the feed forward mechanism. The weights are initialized randomly so the first output of the feed-forward mechanism is just a random guess. So how does it learn? Like any supervised algorithm, a neural network learns from labeled data. So on the first pass through the network with its random guess, the network will know whether the output is close to the labeled target or not. The network improves by adjusting the weights using *back propagation*, and efficient methods of computing gradients. Weights are adjusted by assigning blame backwards so that these neurons' weights are adjusted. The error is the difference between the output and the true label. Ideally we would like this error to be zero. If we had a single input we could adjust that input's weight using the slope, (the derivative). However we will have many inputs each with their own weights. The derivatives will be in matrix form and give us the gradient matrix. We want to move down that error surface simultaneously for all inputs. Each input node's weights are adjusted according to its own gradient. Visualize this as n skiers, one for each input, skiing down different mountain slopes simultaneously where some slopes are steeper than others. We will update the previous hidden layer, then the next, all the way back to the input layer. Training, then, is just a series of forward and backward passes until convergence, meaning that the error is under some threshold value.

13.3 Mathematical Foundations

Training a neural network is an optimization problem, as is the case for many algorithms we have explored. These problems need a loss function. For both regression and classification neural networks, the mean squared error can be used:

$$\mathcal{L} = \frac{1}{2N}\sum_{i=1}^{n} \|(y_i - f(x_i))\|^2 \tag{13.1}$$

where $\|x\| := \sqrt{x_1^2 + ... + x_n^2}$

The error term helps with the credit assignment problem, determining how much blame for the error to assign to each input weight. We will take the derivative of the cost function to find the gradients, the rate of change of the cost function with respect to the neuron's output.

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_n}\right) \tag{13.2}$$

The gradients are computed for each training example. Then they are aggregated to modify the weights by this difference between the old values and the new values multiplied by a parameter, alpha, the learning rate.

$$\triangle w_i = -\alpha \frac{\partial E}{\partial w_i} \tag{13.3}$$

Each forward and backard pass through the network is called an *epoch*. If the alpha is too small it may take too long to converge, but if it is too large you may overcorrect. The error surface may not be perfectly convex but be bumpy and may have local minima.

The gradient descent approach described above is sometimes called *batch gradient descent* because we update the gradients for all examples in one batch. An alternative is *stochastic gradient descent*, in which the weights are updated after each example. The error surface for different examples will look different so this prevents us from getting stuck in a local minima, at the cost of greater computation time. A good compromise is *mini-batch descent* in which works like batch descent but on a subset of the full data at a time.

13.4 Neural Network Classification Example

Next we look at a classification example on the PimaIndiansDiabetes2 data set in package `mlbench`. As shown in the online notebook, we first divide into 75% train and 25% test. Then we perform logistic regression with diabetes as the target and all other columns as predictors. The logistic regression classifer got .765 accuracy on the test data.

13.4.1 Preprocessing for Classification

Since neural networks converge better with scaled data, we scale all the predictor columns but not the target. Our target is a 2-level factor. The neuralnet() function does not handle factors. If you encode the target as an integer and run the algorithm, it will work, but the accuracy will be less than we got for the logistic regression model. What is typically done instead is to encode the factor with one-hot encoding. In one-hot encoding, you have as many columns as you have levels and each row will be 1 "hot" for one and only one of the columns. In our case, with a two-level factor we will need two extra columns. The code below creates a $nx2$ matrix to represent diabetes pos or neg. This matrix is appended onto the train_scaled data and the diabetes column is removed. The code to do this is shown in the next code block.

Code 13.4.1 — One-Hot Encoding. Diabetes Data.

```
class_column <- train_scaled$diabetes
n <- length(class_column)
x <- matrix(0, n, length(levels(class_column)))
x[(1L:n) + n * (unclass(class_column) - 1L)] <- 1
dimnames(x) <- list(names(class_column), levels(class_column))
train_scaled <- cbind(train_scaled[,1:8], x)
names(train_scaled) <- c(names(train_scaled)[1:8], "neg", "pos")
```

We also have to build the formula:

```
n <- names(train)
f <- as.formula(paste("neg + pos ~", paste(n[!n %in% "diabetes"],
     collapse = " + ")))
```

Finally we are ready to build the neural network. We chose 7 hidden nodes, 5 in the first layer and 2 in the second. We specify `linear.output=FALSE` for classification. The plot is shown in Figure 13.4.

Code 13.4.2 — Build the Neural Network. Diabetes Data.

```
library(neuralnet)
set.seed(1234)
nnet1 <- neuralnet(f, data=train_scaled, hidden=c(5,2),
    threshold=0.1, act.fct = "logistic", linear.output=FALSE,
    lifesign = "minimal")
plot(nnet1)
```

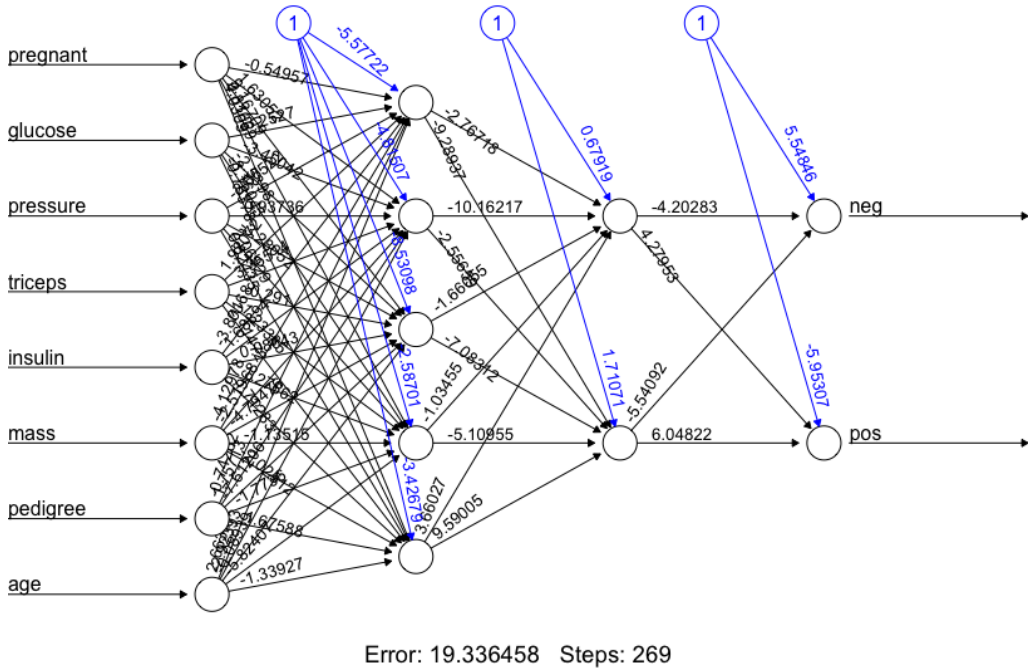

Figure 13.4: Neural Net for Diabetes Prediction

Next we run the test data through the neural network to get predictions. The scaled test data just needs the predictor columns, not diabetes or the the one-hot columns added to the training data. For this package we use its compute() function instead of the usual predict().

Code 13.4.3 — Results. Diabetes Data.

```
nnet.results <- compute(nnet1, test_scaled[,1:8])
pred <- max.col(nnet.results$net.result)
mean(test_scaled$diabetes==pred)
```

The mean accuracy was .776 which is higher than the logistic regression model but only by a small amount. The neural network required much more set up time than the logistic regression model and only got a small improvement in accuracy. This small improvement was made after several trial and error runs changing the architecture of the hidden nodes. On a small data set such as this one, simpler models such as logistic regression will learn as much from the data as a neural network.

Check Your Understanding 13.1 — Classification on the Wine Data. The data set "wine_all.csv" is available on the github. It was created by combining a red data set and a white data set from the UCI repository.

Try the following:

- Load the data and examine it with str().
- Divide into 80-20 train and test.
- Normalize the train and test predictors, leaving out the type column, our target.
- Follow the example in the diabetes classification Code 13.4.1 to set up one-hot encoding for train predictors and combine this with the normalized training data.
- Run neuralnet with 16 hidden nodes, act.fct="logistic", linear.output=FALSE.
- What is your accuracy?

In our run of this exercise, we got an accuracy of 63% which is significantly lower than we have achieved with other algorithms such as logistic regression, knn, and decision trees. Trying different numbers of units and layers didn't improve our performance much. As we will see in the next chapter, using Keras gives us a lot more flexibility in designing an architecture and we will use this same data set again with much better results. ▪

13.5 Neural Network Architecture

Next we discuss some intricacies of neural network architecture in general, and the `neuralnet` package, as discussed in a technical report by the package authors.[2] A neural network can be considered an extension of a generalized linear model because it is mapping the input variables through a function approximation to get the output variable(s). The neuralnet() function allows for flexible architectural choices: multiple inputs, multiple outputs, multiple hidden layers and units. There are also algorithmic choices which we will discuss below. The `neuralnet` package is comparable to the `nnet` package but `neuralnet` has more tuning parameters, more algorithm choices, and produces the nice network graphs we've seen earlier in this chapter.

13.5.1 Training

The more complex the architecture (the number of hidden layers and nodes), the longer training will take. The algorithm may reach its predefined maximum number of iteration steps before it converges below the specified threshold. Both the number of steps and the threshold are adjustable parameters. It's unlikely that you will get lucky and specify an architecture that performs optimally by chance. Creating the network is an iterative trial-and-error process. It helps to see how the threshold is diminishing while training. If your data is very large, you might want to go through this trial-and-error phase with a subset of the training data, like 10K observations or less. To watch the progress of the network training, set parameter `lifesign="full"`. The default is "none" which makes it hard to tell if the algorithm is progressing. There is also a lifeline "minimal" option. Using the "full" option for lifeline, you will get a column with the number of iterations and the minimum threshold that was reached at that point. By default you get a printout every 1K steps but you can change this with the lifesign.step parameter. The threshold is set to 0.01

[2] https://journal.r-project.org/archive/2010/RJ-2010-006/RJ-2010-006.pdf

by default, but it can be modified with the threshold parameter. This threshold value is for the partial derivatives of the error function. When the derivatives are below this threshold, the algorithm stops. Recall that the derivatives give the slope and the minimum error will be when the derivates are near 0 in the gradient descent algorithm. If your algorithm fails to converge with threshold=0.01, you could set the threshold to a slightly larger value. The progress in the console should tell you what that value should be. However, setting a larger threshold will probably result in a less accurate model. In the Beijing notebook online, we ran the algorithm with 33K training observations and a threshold of 0.01 for over an hour. The minimum threshold achieved is shown below. The training ended when we got to the default number of max steps. This can be changed with parameter stepmax. We used the default value which is 100,000. As you can see, the threshold never got below 0.04. We could rerun with threhold=0.041 to get a model we can use. You can see the results online.

```
98000 min thresh: 0.0409476277
99000 min thresh: 0.0409476277
stepmax min thresh: 0.0409476277
```

13.5.2 The Backpropagation Algorithm

The package offers variations of the backprop algorithm. Resiliant backpropagation uses a separate learning rate parameter for each weight, and the learning rate can change during training. Additionally, the weights are adjusted by the sign of the partial derivative instead of just the magnitude. The learning rate is increased if the partial derivative keeps its sign, and decreased if the partial derivative changes sign. This is called weight backtracking, and helps prevent skipping over the minimum error. Refer to the paper and the internal R documentation for the complete list of parameters you can modify in training. Generally, for classification you will set linear.output=FALSE, let the err.fct="ce" for cross entropy, and use the default logistic function activation function. For regression, set linear.output=TRUE, use the default error of sse, and use either the logistic or tanh activation function.

13.5.3 Model Output

A lot of information can be retrieved from the model, including the weights. The following shows part of the output from the result matrix for one of neural networks in the Beijing notebook. There were 25788 steps, and the all the weights are displayed.

```
nn2$result.matrix
error                    71.14862965954
reached.threshold         0.07036554366
steps                 25788.00000000000
Intercept.to.1layhid1    -3.35042615102
month.to.1layhid1         4.66639682396
DEWP.to.1layhid1          0.79259916820
TEMP.to.1layhid1         -1.26566024085
PRES.to.1layhid1         -0.51724781133
. . .
```

13.5.4 Visualizing Results

In the previous two examples in this chapter we plotted the network with the plot() command. For larger neural networks, the parameters `dimension` and `radius` can be used to modify the graph size. The package also lets you visualize the weights with gwplot(). An example is shown in the Beijing notebook online.

13.6 Summary

Neural networks are defined by these properties:

- the network architecture or topology - the number of neurons in the model and the number of layers
- the activation function which transforms the inputs to an output; the sigmoid function is commonly used
- the training algorithm

Neural networks can learn complex functions from data. Generally they will not outperform simpler models for small data but truly shine when the amount of data is large and the function to learn is complex.

In this chapter we focused on the `neuralnet` package because of its nice network graph and because working through its parameters provided a nice opportunity to discuss neural network architecture. In the next chapter we focus on Keras, which vastly outperforms neuralnet() in terms of accuracy and speed for neural networks of any depth. Unfortunately for Windows users, Keras can run on Windows but it's not really recommended. If you want to run Keras, it is recommended to first set up a dual-boot Ubuntu or virtual machine, and run Keras from there.

13.6.1 New Terminology

- backpropagation
- epoch
- neuron or node
- perceptron
- activation function

13.6.2 Quick Reference

To build a neural network with the `neuralnet` package, you should first scale the data, then build the formula. When scaling the predictors for classification, we just used the scale() function.

Reference 13.6.1 Scale the Data for Classification

```
# where i is a vector of training indices
scaled <- scale(df[,-9])  # omit target column
df_scaled <- data.frame(cbind(scaled, df$target))
colnames(df_scaled) <- colnames(df)
df_scaled$target <- factor(df_scaled$target)
train_scaled <- df_scaled[i,]
test_scaled <- df_scaled[-i,]
```

Reference 13.6.2 Build a formula

```
f <- as.formula(paste("target ~", paste(n[!n %in% "target"],
    collapse = " + ")))
```

To build a neural network for regression, set `linear.output` to TRUE; for classification set it to FALSE.

Reference 13.6.3 Neural Network with neuralnet()

```
set.seed()
nn1 <- neuralnet(f, data=train, hidden=c(5,3),
    linear.output=TRUE)
plot(nn1)
```

The neuralnet() function cannot handle factors. For classification, a factor target should be recoded as multiple columns in one-hot encoding.

Reference 13.6.4 One-hot Encoding

```
class_column <- train_scaled$target
n <- length(class_column)
x <- matrix(0, n, length(levels(class_column)))
x[(1L:n) + n * (unclass(class_column) - 1L)] <- 1
dimnames(x) <- list(names(class_column), levels(class_column))
train_scaled <- cbind(train_scaled[,1:8], x)
names(train_scaled) <-
    c(names(train_scaled)[1:8], "class1", "class2")
```

Reference 13.6.5 Evaluate Results for Classification

```
# input test data, predictor columns only
nnet.results <- compute(nnet1, test_scaled[,1:8])
pred <- max.col(nnet.results$net.result)
mean(test_scaled$target==pred)
```

13.6.3 Going Further

The heart of neural networks is the propagation algorithm. An entire chapter devoted to backprop can be found here: `http://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf`. The chapter is from Raul Rojas's *Neural Networks - A Systematic Introduction*.