Neural Networks

Hakan Gogtas

11/30/2020

Neural Networks

Step 1 – collecting data

For this analysis, we will utilize data on the compressive strength of concrete donated to the UCI Machine Learning 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.

According to the website, the 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 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)

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("/cloud/project/concrete.csv")
str(concrete)
```

```
## 'data.frame':    1030 obs. of  9 variables:
##  $ cement      : num  540 540 332 332 199 ...
##  $ slag        : num  0 0 142 142 132 ...
##  $ ash         : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ water       : num  162 162 228 228 192 228 228 228 228 228 ...
##  $ superplastic: num  2.5 2.5 0 0 0 0 0 0 0 0 ...
##  $ coarseagg   : num  1040 1055 932 932 978 ...
##  $ fineagg     : num  676 676 594 594 826 ...
##  $ age         : int  28 28 270 365 360 90 365 28 28 28 ...
##  $ strength    : num  80 61.9 40.3 41 44.3 ...
```

```
head(concrete)
```

```
##   cement  slag ash water superplastic coarseagg fineagg age strength
## 1  540.0   0.0   0   162          2.5    1040.0   676.0  28    79.99
## 2  540.0   0.0   0   162          2.5    1055.0   676.0  28    61.89
## 3  332.5 142.5   0   228          0.0     932.0   594.0 270    40.27
## 4  332.5 142.5   0   228          0.0     932.0   594.0 365    41.05
## 5  198.6 132.4   0   192          0.0     978.4   825.5 360    44.30
## 6  266.0 114.0   0   228          0.0     932.0   670.0  90    47.03
```

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 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, 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 zero to one range may be more appropriate. In this case, we'll use the latter.

In Classification Using Nearest Neighbors example, 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 zero and one, respectively:

```
summary(concrete_norm$strength)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.2664  0.4001  0.4172  0.5457  1.0000
```

In comparison, the original minimum and maximum values were 2.33 and 82.60:

```
summary(concrete$strength)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.33   23.71   34.45   35.82   46.13   82.60
```

Following Yeh's precedent 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, ]
```

Step 3 – training a model on the data

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

```
#install.packages("neuralnet")
library(neuralnet)

concrete_model <- neuralnet(strength ~ cement + slag
                            + ash + water + superplastic + coarseagg + fineagg + age,
                            data = concrete_train)
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 labeled with the number 1. The bias terms are numeric constants that allow the value at the indicated nodes to be shifted upward or downward, much like the intercept in a linear equation.

R reports the number of training steps and an error measure called the sum of squared errors (SSE), which, as you might expect, is the sum of the squared differences between the predicted and actual values. The lower the SSE, the more closely the model conforms to the training data, which tells us about performance on the training data but little about how it will perform on unseen data.

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 future data. To generate predictions on the test dataset, we can use the compute() function as follows:

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

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.result, 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'll measure the correlation between our predicted concrete strength and the true value.

If the predicted and actual values are highly correlated, the model is likely to be a useful gauge of concrete strength. Recall that the cor() function is used to obtain a correlation between two numeric vectors:

```
cor(predicted_strength, concrete_test$strength)
```

```
##           [,1]
## [1,] 0.7206086
```

Correlations close to one indicate strong linear relationships between two variables. Therefore, the correlation here of about 0.806 indicates a fairly strong relationship. This implies that our model is doing a fairly good job, even with only a single hidden node.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.

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 5.08 in the previous model to 1.63 here. Additionally, the number of training steps rose from 4,882 to 86,849, which should come as no surprise given how much more complex the model has become. More complex networks take many more iterations to find the optimal weights. Applying the same steps to compare the predicted values to the true values, we now obtain a correlation around 0.92, which is a considerable improvement over the previous result of 0.80 with a single hidden node:

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

```
##           [,1]
## [1,] 0.8486049
```

Recently, an activation function known as a rectifier has become extremely popular due to its success on complex tasks such as image recognition. A node in a neural network that uses the rectifier activation function is known as a rectified linear unit (ReLU). As depicted in the following figure, the rectifier activation function is defined such that it returns x if x is at least zero, and zero otherwise. The significance of this function is due to the fact that it is nonlinear yet has simple mathematical properties that make it both computationally inexpensive and highly efficient for gradient descent.

Unfortunately, its derivative is undefined at x = 0 and therefore cannot be used with the neuralnet() function. Instead, we can use a smooth approximation of the ReLU known as softplus or SmoothReLU, an activation function defined as log(1 + ex).

```
softplus <- function(x) { log(1 + exp(x)) }
```

This activation function can be provided to neuralnet() using the act.fct parameter. Additionally, we will add a second hidden layer of five nodes by supplying the hidden parameter the integer vector c(5, 5). This creates a two-layer network, each having five nodes, all using the softplus activation function:

```
RNGversion("3.5.2")
```

```
## Warning in RNGkind("Mersenne-Twister", "Inversion", "Rounding"): non-uniform
## 'Rounding' sampler used
```

```
set.seed(1234567)
concrete_model3 <- neuralnet(strength ~ cement + slag +
                               ash + water + superplastic +
                               coarseagg + fineagg + age,
                             data = concrete_train,
                             hidden = c(5, 5),
                             act.fct = softplus)
plot(concrete_model3)
```

the correlation between the predicted and actual concrete strength can be computed:

```
model_results3 <- compute(concrete_model3, concrete_test[1:8])
predicted_strength3 <- model_results3$net.result
cor(predicted_strength3, concrete_test$strength)
```

```
##           [,1]
## [1,] 0.8157861
```

The correlation between the predicted and actual strength was 0.935, which is our best performance yet. Interestingly, in the original publication, Yeh reported a correlation of 0.885. This means that with relatively little effort, we were able to match and even exceed the performance of a subject matter expert. Of course, Yeh's results were published in 1998,giving us the benefit of over 20 years of additional neural network research!

One important thing to be aware of is that, because we had normalized the data prior to training the model, the predictions are also on a normalized scale from zero to one. For example, the following code shows a data frame comparing the original dataset's concrete strength values to their corresponding predictions side-by-side:

```
strengths <- data.frame(
  actual = concrete$strength[774:1030],
  pred = predicted_strength3)
head(strengths, n = 3)
```

```
##     actual      pred
## 774  37.42 0.4202910
## 775  11.47 0.2870828
## 776  22.44 0.2513222
```

```
cor(strengths$pred, strengths$actual)
```

```
## [1] 0.8157861
```

With this in mind, we can create an unnormalize() function that reverses the min-max normalization procedure and allow us to convert the normalized predictions to the original scale:

```
unnormalize <- function(x) {return((x * (max(concrete$strength)) -min(concrete$strength)) + min(concrete

strengths$pred_new <- unnormalize(strengths$pred)
strengths$error <- strengths$pred_new - strengths$actual

head(strengths, n = 3)
```

```
##     actual      pred pred_new     error
## 774  37.42 0.4202910 34.71604 -2.703961
## 775  11.47 0.2870828 23.71304 12.243038
## 776  22.44 0.2513222 20.75922 -1.680783
```

```
cor(strengths$pred_new, strengths$actual)
```

```
## [1] 0.8157861
```