Neural Networks

Albert Mata

25/12/2018

Contents

Black Box Methods - Neural Networks

We talk about **black box** processes when the mechanism that transforms the input into the output is obfuscated by an imaginary box (mainly due to the complex mathematics allowing them to function). But it's dangerous to apply black box models blindly, so we'll peek inside the box and discover that neural networks mimic the structure of animal brains to model arbitrary functions.

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. A brain uses a network of interconnected cells called **neurons** to create a massive parallel processor. ANN uses a network of artificial neurons or **nodes** to solve learning problems. ANNs are versatile learners currently used for:

- Speech and handwriting recognition programs.
- Automation of smart devices like self-driving cars and self-piloting drones.
- Scientific, social or economic phenomena such as sophisticated models of weather and climate patterns, tensile strength, fluid dynamics, etc.

ANNs are black box methods best applied to problems where the input data and output data are well-defined or fairly simple, yet the process that relates them is extremely complex.

From biological to artificial neurons

Biological neurons:

1. Incoming signals are received by the cell's **dendrites** and weighted according to relative importance or frequency.
2. As the **cell body** accumulates these signals, at some point it transmits an output signal down the **axon**.
3. At the axon's terminals, the signal is passed to the neighboring neurons across a tiny gap known as a **synapse**.

Artificial neurons:

1. Input signals (x_i) are weighted (w_i) according to importance.
2. Input signals are summed by the cell body and the signal is passed on according to an **activation function** (f) that generates the output signal (y).

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

ANNs use neurons as building blocks to construct complex models of data with these characteristics:

- An **activation function** (f) transforms a neuron's combined input signals into a single output signal to be broadcasted further in the network.
- A **network topology** (or architecture) describes the number of neurons and layers and how they are connected.
- A **training algorithm** specifies how connection weights are set.

Activation functions

A **threshold activation function** is the mechanism by which the artifical neuron processes incoming information and passes it throughout the network only once a specified input threshold has been attained. For example, a neuron that fires only when the sum of input signals is at least zero (**unit step activation function**).

However, the activation functions used in ANN are chosen based on their ability to demonstrate desirable mathematical characteristics and accurately model relationships among data (i.e. the most commonly used S-shaped **sigmoid activation function**):

$$f(x) = \frac{1}{1 + e^{-x}}$$

There are different choices for activation functions: sigmoid, linear, saturated linear, hyperbolic tangent, gaussian… Each of them will result in a different neural network model that has strengths better suited for certain learning tasks.

To avoid **squashing problems** (due to the fact that the range of input values that affect the output signal is relatively narrow) we standardize or normalize all neural network inputs so they fall within a small range around 0. This way we prevent large-valued features from dominating small-valued features.

Network topology

The ability of a neural network to learn more complex tasks is rooted in its topology, defined by:

- **The number of layers**. For example, in a **single-layer network** some neurons called **input nodes** receive unprocessed signals from the input data, process a single feature each and use their activation functions to send signals to the **output node**, which uses its own activation function to generate a final prediction. In this process there is only one set of connection weights (w_1 to w_n), that's why it's a single-layer network. But more sophisticated **multilayer networks** are required for most learning tasks. In these multilayer networks, every node in one layer may or may not be connected to every node in the next layer.
- **Whether information is allowed to travel backward**. Networks in which the input signal is fed continously in one direction from connection to connection until it reaches the output layer

are called **feedforward networks**. They offer a lot of flexibility (changing number of levels and nodes, modeling multiple outcomes, applying multiple hidden layers…). A neural network with multiple hidden layers is called a **Deep Neural Network (DNN)** and its training is referred to as **deep learning**. Current de facto standard ANN topology is the **Multilayer Perceptron (MLP)** (a multilayer feedforward network). *In contrast but still rarely used in practice,* ***recurrent networks*** *(or* ***feedback networks****) allow signals to travel in both directions using loops and works for learning extremely complex patterns. They can add a short-term memory (****delay****) to immensely increase their power.*

- **The number of nodes within each layer of the network**. The number of input nodes is predetermined by the number of features in the input data. The number of output nodes is predetermined by the number of outcomes to be modeled or the number of class levels in the outcome. But the number of hidden nodes is free and depends on many factors. In general, more complex topologies with a greater number of connections allow the learning of more complex problems. A greater number of nodes will result in a model that more closely mirrors the training data (at risk of overfitting and being too expensive and slow). In general, we'll want to use the fewest nodes that give adequate performance in a validation dataset.

Training neural networks with backpropagation

As the neural network processes the input data, connections between the neurons are strengthened or weakened and their weights are adjusted to reflect the patterns observed over time. This is very computationally intensive, which is why we use the errors **backpropagation** algorithm to train ANNs (Rumelhart, Hinton, and Williams 1986).

So, multilayer feedforward networks that use the backpropagation algorithm are now common in the field of data mining.

Strengths:

- Can be adapted to classification or numeric prediction problems.
- Capable of modeling more complex patterns than nearly any algorithm.
- Makes few assumptions about the data's underlying relationships.

Weaknesses:

- Extremely computationally intensive and slow to train, particularly if the network topology is complex.
- Very prone to overfitting training data.
- Results in a complex black box model that is difficult, if not impossible, to interpret.

The backpropagation algorithm iterates through many cycles (or **epochs**) of two processes. As the network has no *a priori* knowledge, starting weights are typically set at random. Then the algorithm iterates through these two phases util a stopping criterion is reached:

1. 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.
2. A **backward phase** in which the output is compared to the true target value in the training data. The difference between the two is the error that is propagated backwards in the network to modify the connection weights between neurons in order to reduce future errors.

In order to know how much the weight for each neuron has to be changed, the algorithm uses a technique called **gradient descent** (always look for the greatest downward slope). To do this, it uses the derivative of each neuron's activation function to identify the gradient in the direction of each of the incoming weights. 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 the training time at the risk of overshooting the valley.

Example - modeling the strength of concrete with ANNs

Step 1 - collecting data

We'll utilize data on the compressive strength of concrete (Yeh 1998).

This dataset includes 1,030 examples of concrete with some features describing the components used in the mixture (i.e. cement, slag, ash, water, superplasticizer, etc) and the aging time in days.

Step 2 - exploring and preparing the data

```
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 ...
```

We see values ranging anywhere from zero up to over a thousand, but we have already said that neural networks work best when the input data are scaled to a narrow range around zero. There exist two

approaches to fix this:

- If the data follow a bell-shaped curve (a normal distribution) we can use R's built-in `scale()` function.
- If the data follow a uniform distribution or are severely nonnormal, normalization to a 0-1 range may be more appropriate.

We'll use the second approach here:

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

concrete_norm <- as.data.frame(lapply(concrete, normalize))
summary(concrete_norm$strength)
```

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

We create a new variable (`concrete_norm`) because 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 this, it is a good idea to keep the original data (`concrete`).

Finally, we will partition the already randomly sorted data into a training set with 75 percent of the examples and a testing set with the remaining 25 percent:

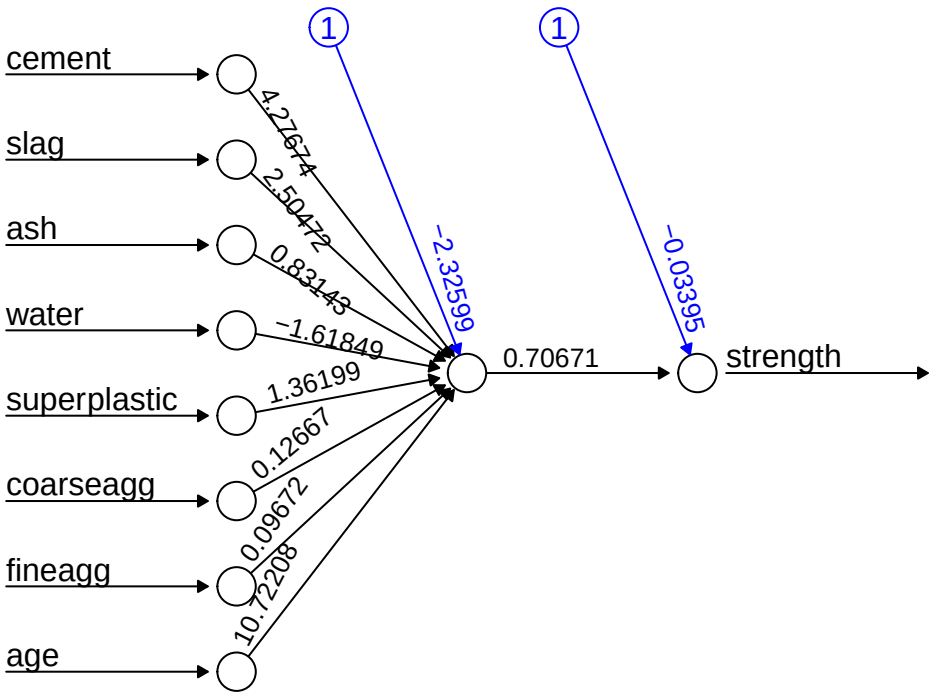
```
concrete_train <- concrete_norm[1:773, ]
concrete_test  <- concrete_norm[774:1030, ]
```

Step 3 - training a model on the data

We will use a multilayer feedforward neural network from the `neuralnet` package to model the relationship between the ingredients used in concrete and the strength of the finished product. Other package with similar options are `nnet` and `RSNNS`.

Initially, we train the simplest multilayer feedforward network with only a single hidden node:

```
library(neuralnet)
set.seed(12345)
concrete_model <- neuralnet(strength ~ cement + slag + ash + water +
                              superplastic + coarseagg + fineagg + age,
                            data = concrete_train)
plot(concrete_model, rep = "best")
```



We can see 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 graph also shows the weights for each of the connections and the **bias terms** (in blue, much like the intercept in a linear equation, as a neural network with a single hidden node is somewhat similar to a linear regression model).

The error measure is the **Sum of Squared Errors (SSE)** (the sum of the squared predicted minus actual values). The lower SSE, the better predictive performance. But this only talks about the model's performance on the training data.

Step 4 - evaluating model performance

We use our model to generate predictions and then compare them to the true values:

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

The `compute()` function 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:

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

We can't use a confusion matrix, as this is not a classification problem but a numeric prediction. What we do is to measure the correlation between our predicted values and the true ones:

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

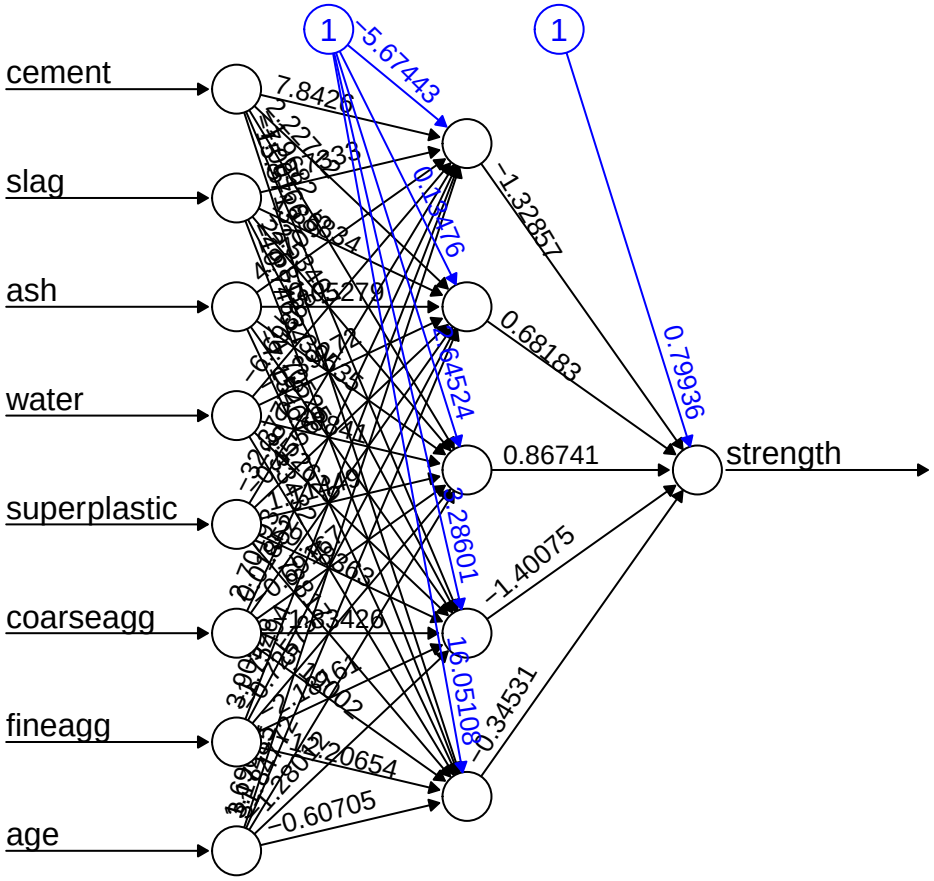
```
##              [,1]
## [1,] 0.8064655576
```

Correlations close to 1 indicate strong linear relationships between two variables. A correlation here of about 0.806 indicates a fairly strong relationship. This implies our model is doing a fairly good job, even with only a single hidden node.

Step 5 - improving model performance

But we can try to improve our results using a more complex topology:

```
set.seed(12345)
concrete_model2 <- neuralnet(strength ~ cement + slag + ash + water +
                               superplastic + coarseagg + fineagg + age,
                             data = concrete_train, hidden = 5)
plot(concrete_model2, rep = "best")
```



The number of steps is much higher now (so computing this model is much more expensive). And we can also see the SSE is much lower now. But let's see how our model performs with unseen data:

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

```
##              [,1]
## [1,] 0.9244533426
```

Correlation has gone from 0.806 to 0.924, which is a considerable improvement. We could try to improve even further trying different number of hidden nodes, applying different activation functions and so on.

References

Lantz, Brett. 2015. *Machine Learning with R*. Packt Publishing Ltd.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323 (6088). Nature Publishing Group: 533.

Yeh, I-C. 1998. "Modeling of Strength of High-Performance Concrete Using Artificial Neural Networks." *Cement and Concrete Research* 28 (12). Elsevier Ltd: 1797–1808.