Machine Learning and Neural Networks

Dr. Robert Buscaglia

October 03, 2022

Machine Learning

The most common theme you will find when answering the question 'What is Machine Learning?' is a field of artificial intelligence that attempts to *learn* and *improve* from data. The key here is to be able to do both - learn from data to predict new occurrences - improve as the algorithm sees more data. I give a few links to articles I think that can help us understand this concept.

Useful Machine Learning Links

IBM

IBM has a nice breakdown and were an initial pioneer of 'big-thinking' with the Watson project. It is famous for its appearance on Jeopardy! in 2011.

IBM Machine Learning : **https://www.ibm.com/cloud/learn/machine-learning**

Toward Data Science

Some of you may know of this series of blogs. I give you two different links. The first is an article with excellent details about the field of Machine Learning. The jargon alone is a lot to keep up with. We teach much of what you will find in this article in our STA 478/578 Statistical Computing course.

What is Machine Learning? : **https://towardsdatascience.com/machine-learning-an-introduction-23b84d51e6d0**

If you are interested in trying to keep up or learn how to implement different algorithms, this stream can be very useful.

Toward Data Science Machine Learning: **https://towardsdatascience.com/machine-learning/home**

Data Science Weekly

One last weekly newsletter to find excellent cutting-edge information. There is a huge archive here of information as well. For students, there are often also links to job applications in the field.

Data Science Weekly Sign-up & Archive : **https://www.datascienceweekly.org/**

Kaggle

If you are interesting to putting your skills to the test, kaggle is the place to do it. Kaggle has merged with some very large data hosting websites, creating a platform to learn and test machine learning algorithms. There are even contests that you can compete in for monetary prizes (although they are now very competitive).

Kaggle : **https://www.kaggle.com/**

Building a Simple Neural Network in R

Neural Networks Basics

One of the big themes in machine learning is the success of neural networks and deep learning. This introduction cannot give all the details of neural networks; that is what the sites above can help with, as well as several courses offered here at NAU. Today we will 'hand-code' a simple **feed-forward neural network**. The common setup of such a network includes

- Input layer and objective
  - What data do we have and what are we trying to learn?
- Architecture
  - Neurons and Layers - how the network 'learns' from data.
- Forward- and Back-propagation
  - The model is fit using forward propagation to obtain a 'loss'
  - Back-propagation uses the loss function to determine how to adjust weights
  - Repeating forward and back-propagation is often called an 'epoch'
  - High numbers of iterations can lead to improved learning
- Activation Functions
  - Play a key role in different prediction types
  - Must 'transform' layers to adapt to what we are trying to learn.
- Validation
  - We must always ensure our models are robust to bias
  - Bias-variance trade-off and generalization of models

Here is a diagram of the architecture we will build today.

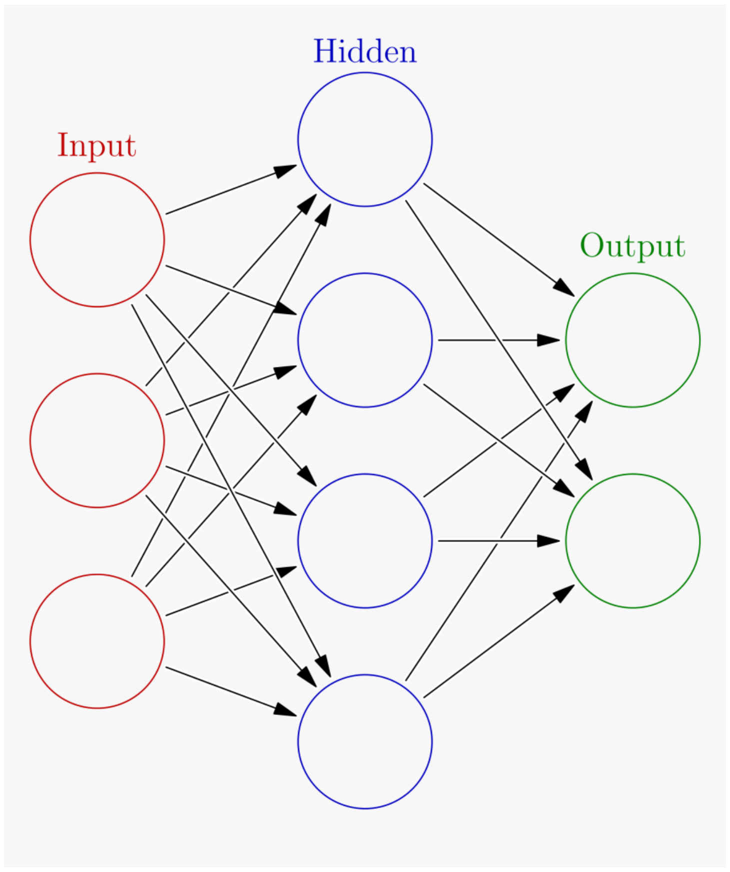


Figure 1: Simple Neural Newtwork Architecture

The red circles represent the inputs. These can sometimes be referred to as predictors or features. The arrows represent simple linear combinations, $x * w$, where each input is multiplied by a weight, then passed forward through an activation function. The blue circles the hidden layer, where information from input it passed and then new activation functions and weights are estimated, continuing the ‘feed-forward’ process. We will be able to observe all layers in the algorithms built below. The hidden layer is where abstract model forms can be represented, and the idea of neurons and activation are modeled after how our own neurons and brain learn from information. We pass the hidden layer forward just like the input layer. A large mesh of simple linear combinations that are transformed by an activation function and then considered outputs. There is some very nice mathematics to work out here, but we’ll focus on the computers today!

Simple Network Function Definitions

Data setup

We will produce a simple feed-forward neural network using sigmoidal activation functions. We will consider being given binary inputs (features) and a binary response. I will try to discuss some details as we move forward. We first set up our 'simple' problem. You can expand the complexity later on your own if you want. I generated 5 samples with binary responses each having 5 different binary 'predictors'.

```
set.seed(10)
### Make my input values
X <- matrix(sample(c(0,1), 25, replace=T), ncol=5)
y <- sample(c(0,1), 5, replace=T)
Example.1 <- cbind(X,y)
Example.1
```

```
##                  y
## [1,] 0 0 0 0 0 0
## [2,] 0 1 1 1 1 0
## [3,] 1 1 0 1 1 1
## [4,] 1 0 0 1 0 1
## [5,] 1 0 1 0 1 0
```

The 5 predictors act as my input layer. We will input these five predictors, of which we look to predict a binary response.

Activation Function

We can next take care of our activation function. Activation functions are how we pass information between layers. For this problem, we are after predicting either a zero or one. The sigmoid function will 'smash' whatever is passed forward into the interval (0, 1). We use the `sigmoid` function created below. The sigmoid function is also a good starting activation function, because we can quickly find its derivative. This is important for back-propagation. I also create `sigmoid.derivative` which calculates the derivative of the `sigmoid` function at x.

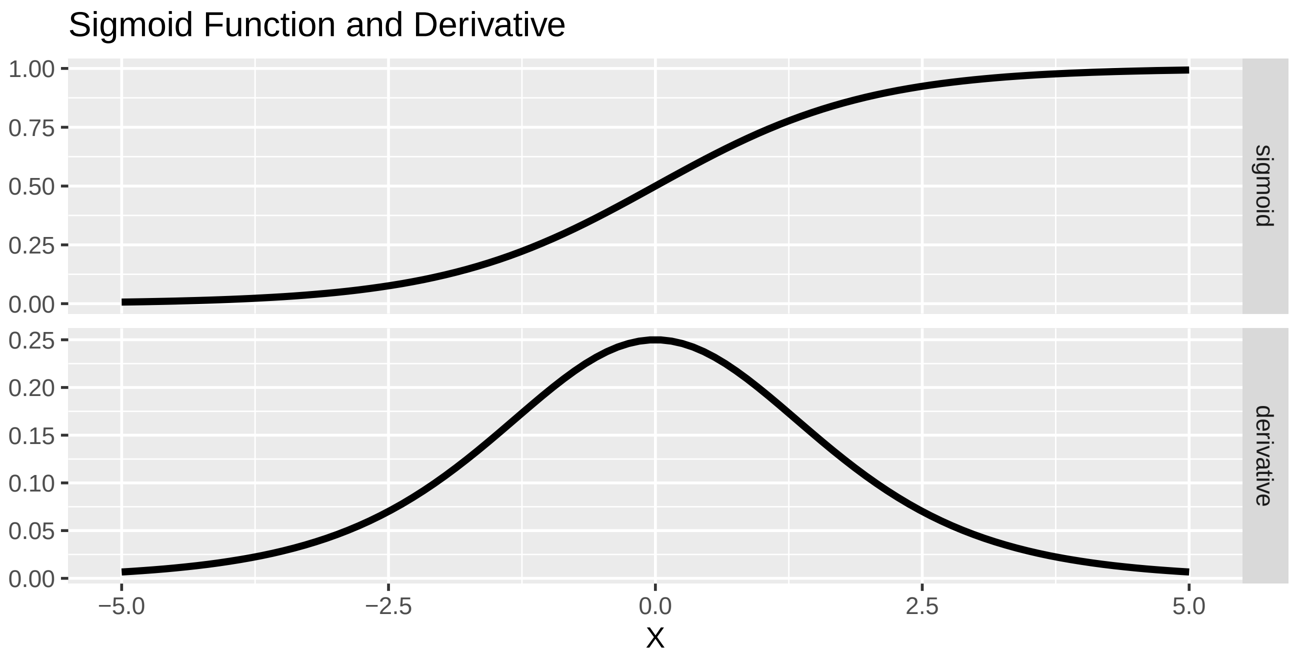
```
### activation function and derivative
sigmoid <- function(x) 1/(1+exp(-x))
sigmoid.derivative <- function(x) sigmoid(x)*(1-sigmoid(x))
```

Here I present a visualization of these two functions.

```
nn.activation.plot <- data.frame(x.grid = seq(-5, 5, length.out=100)) %>%
  mutate(sigmoid = sigmoid(x.grid),
         derivative = sigmoid.derivative(x.grid))

nn.activation.plot <- nn.activation.plot %>%
  pivot_longer(names_to = 'Function', values_to = 'Y', cols=2:3) %>%
  mutate(Function = factor(Function, levels = c('sigmoid', 'derivative')))
```

```r
ggplot(nn.activation.plot, aes(x=x.grid, y=Y)) +
  geom_line(size=1.25) +
  facet_grid(Function~., scales='free') +
  labs(title = 'Sigmoid Function and Derivative', x = 'X', y = '')
```



Loss Function

Setup our loss function. The loss function is how we should score the algorithm. We want to minimize our loss. There are several choices here, and with modern computing very little limitations. However, to do the work explicitly in this problem, I choose the L2-loss, or more commonly, the sum of squared deviations. In statistics, this is related to the MSE.

```r
loss.function <- function(y, y.hat) (y-y.hat)^2 %>% sum()
```

Not much to say about this yet, except this is what we are looking to minimize. It will compare our predictions to the known responses, and we will perform back-propagation to improve this loss.

Neural Network One epoch

With everything defined, lets look at one epoch of our neural network, then we can wrap everything into a function. Here I explicitly work through one forward propagation (allow everything to be computed based on initial random guesses for the weights). We then perform one step of back-propagation (update all the weights in the direction that will improve our loss). After we see how a single iteration (epoch) works, we can try running many of them.

Feedforward

We will generate random weights for the layers, and perform the operations to calculate the feedforward step. Notice the math here is not that complicated. I take my inputs (X) and use matrix multiplication to determine XW_1, where W_1 are randomly chosen weights.

```
set.seed(10) ### same random weights to start
weights1 <- matrix(runif(ncol(X)*nrow(X)), ncol=nrow(X))
X %*% weights1 ### Inputs * Weights*
```

```
##          [,1]      [,2]     [,3]      [,4]     [,5]
## [1,] 0.000000 0.0000000 0.000000 0.0000000 0.000000
## [2,] 1.511914 1.5923364 1.635222 1.5510059 2.151881
## [3,] 1.592485 1.5454680 2.173369 1.7156376 2.241492
## [4,] 1.200580 0.8412659 1.247581 0.8276001 1.220290
## [5,] 1.019522 0.9274132 1.123215 1.5291212 2.045681
```

To move this information forward, we use the activation function. We will move from the input layer X to the first layer of our network. We need only apply the sigmoid function.

```
layer1 <- sigmoid(X%*%weights1)
layer1
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.5000000 0.5000000 0.5000000 0.5000000 0.5000000
## [2,] 0.8193447 0.8309446 0.8368837 0.8250590 0.8958444
## [3,] 0.8309654 0.8242582 0.8978324 0.8475661 0.9039141
## [4,] 0.7686280 0.6987318 0.7768808 0.6958473 0.7721146
## [5,] 0.7348794 0.7165502 0.7545845 0.8218777 0.8855105
```

This is often referred to as the hidden layer. The matrix above is what the hidden layer looks like for each of our samples after forward propagation. We can now pass this forward to the output layer by applying a second layer of weights and a second activation function.

```
### second set of random starting weights
weights2 <- matrix(runif(nrow(X)),ncol=1)
layer1 %*% weights2
```

```
##          [,1]
## [1,] 1.455596
## [2,] 2.430827
## [3,] 2.468256
## [4,] 2.126152
## [5,] 2.249397
```

Notice that the shape has changed, I am now getting 5 outputs. We could add more hidden layers if we want, or more advanced architecture, but here we move straight to output. However, we know that the output can be either zero or one. Thus, we must smash again to move this to the output layer. We apply our sigmoid function again and obtain.

```
output <- sigmoid(layer1 %*% weights2)
output
```

```
##           [,1]
## [1,] 0.8108582
## [2,] 0.9191480
## [3,] 0.9218863
## [4,] 0.8934192
## [5,] 0.9045985
```

This is our first set of guesses based on random weights. Recall we had 3 responses of zero and 2 responses of one, thus there is some learning that needs to be done here. We can now calculate how well our first approximation did…

```
loss.function(y, output)
```

```
## [1] 2.338084
```

This is an arbitrary score, but this is what we want to improve. We have finished forward propagation, and now we use back-propagation to improve the weights by asking what needs to change to improve the loss (or score) above.

Backpropogation

We now evaluate what updates we should make to the weights. This is where we use the concept of *gradient descent*. The math is not worked out here, but requires us to determine the negative gradient based on our loss function, telling us about what changes in the weights will lead us to improve our loss function. The mathematics here is not beyond graduate level mathematicians, and is something you could work out in a Calculus course, using what we know about the fundamental theorem of calculus. I have chosen easy enough activation and loss functions, many of you could work this out easily by hand.

Determine the changes of the second layer ('weights2'). Remember, we are going backward now… I want to determine how I should change the second set of random weights. We do this by using information related to the loss function and activation functions (notice if you look carefully the derivatives of both functions are present). We then left multiply by the transpose of the $layer1$ values obtained during forward propagation. This is exactly one step of gradient descent.

```
d.weights2 <- t(layer1) %*% as.matrix((2*(y-output)*sigmoid.derivative(output)))
### How we should update our weights for layer 2?
d.weights2
```

```
##            [,1]
## [1,] -0.6922401
## [2,] -0.6930688
## [3,] -0.7036346
## [4,] -0.7293325
## [5,] -0.7743257
```

```
### New weights to be used in epoch 2 for layer 2
weights2+d.weights2
```

```
##             [,1]
## [1,]  0.01440683
## [2,]  0.14521889
## [3,] -0.46404547
## [4,]  0.04143904
## [5,] -0.41842799
```

We next have to back propogate the input layer weights. There is a lot more work to do here, but it is just the chain rule and the work is done in matrix form. There is not enough time for the details of this layer, but notice it is not too much worse to work out than above. There is though a much more complex calculation (and remember, we only used 2 layers, image if you used 3, 4, or more layers).

```
d.weights1 <- as.matrix(2*(y-output)*sigmoid.derivative(output)) %*% t(weights2)
d.weights1 <- d.weights1*sigmoid.derivative(layer1)
d.weights1 <- t(X) %*% d.weights1
### how should we update the weights of layer 1?
d.weights1
```

```
##             [,1]        [,2]        [,3]        [,4]        [,5]
## [1,] -0.04597409 -0.05476140 -0.01550155 -0.04800270 -0.02160254
## [2,] -0.05146965 -0.06074153 -0.01736350 -0.05603796 -0.02513026
## [3,] -0.11366303 -0.13497140 -0.03827079 -0.12185312 -0.05475675
## [4,] -0.04475163 -0.05257396 -0.01509267 -0.04852095 -0.02175111
## [5,] -0.10891239 -0.12932094 -0.03670334 -0.11670540 -0.05243435
```

```
### new weights for layer 1 in epoch 2.
weights1 + d.weights1
```

```
##             [,1]      [,2]      [,3]         [,4]      [,5]
## [1,]  0.46150412 0.1706752 0.6361541  0.380806717 0.8431187
## [2,]  0.25529886 0.2137890 0.5503742 -0.004134642 0.5902222
## [3,]  0.31324464 0.1373337 0.0752382  0.142324544 0.7203531
## [4,]  0.64835045 0.5632553 0.5808326  0.350269777 0.3338176
## [5,] -0.02377642 0.3003506 0.3213466  0.719428744 0.3534156
```

That's it! One epoch done. With the updated weights, we would now perform the feedforward step and evaluate how we have improved our results (change in loss). We can then continue this process for a desired number of iterations or until we reach a desired threshold for our loss function.

A Simple Neural Network

We can put this all of this together into a straight-forward algorithm. The best step is to prepare an object that can control everything we are generating during the different operations. Thus, we can make a list containing all of the important quantities for the neural network. I produce a list called `my.nn` which keeps track of how the network changes as we perform multiple iterations. This matches exactly what was done above by controlling the seed.

```
set.seed(10)
my.nn <- list(
  input = X,
  weights1 = matrix(runif(ncol(X)*nrow(X)), ncol=nrow(X)),
  weights2 = matrix(runif(nrow(X)),ncol=1),
  y = y,
  output = matrix(ncol=1)
)
```

We can now take the operations from above and wrap them into function. Here we do feedforward, I show the output of one step, to demonstrate it matches the above. I leave if to you to confirm this reproduces everything above, but now by just a simple function call.

```
feedforward.step <- function(nn)
{
  nn$layer1 <- sigmoid(nn$input %*% nn$weights1)
  nn$output <- sigmoid(nn$layer1 %*% nn$weights2)
  nn$loss <- loss.function(nn$y, nn$output)
  return(nn)
}
### Produce one step of feedforward.
my.nn<-feedforward.step(my.nn)
my.nn
```

```
## $input
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    1    1    1    1
## [3,]    1    1    0    1    1
## [4,]    1    0    0    1    0
## [5,]    1    0    1    0    1
##
## $weights1
##            [,1]      [,2]      [,3]       [,4]      [,5]
## [1,] 0.50747820 0.2254366 0.6516557 0.42880942 0.8647212
## [2,] 0.30676851 0.2745305 0.5677378 0.05190332 0.6153524
## [3,] 0.42690767 0.2723051 0.1135090 0.26417767 0.7751099
## [4,] 0.69310208 0.6158293 0.5959253 0.39879073 0.3555687
## [5,] 0.08513597 0.4296715 0.3580500 0.83613414 0.4058500
##
## $weights2
##           [,1]
## [1,] 0.7066469
## [2,] 0.8382877
## [3,] 0.2395891
## [4,] 0.7707715
## [5,] 0.3558977
##
## $y
## [1] 0 0 1 1 0
##
## $output
##           [,1]
## [1,] 0.8108582
## [2,] 0.9191480
## [3,] 0.9218863
## [4,] 0.8934192
## [5,] 0.9045985
##
## $layer1
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.5000000 0.5000000 0.5000000 0.5000000 0.5000000
## [2,] 0.8193447 0.8309446 0.8368837 0.8250590 0.8958444
## [3,] 0.8309654 0.8242582 0.8978324 0.8475661 0.9039141
## [4,] 0.7686280 0.6987318 0.7768808 0.6958473 0.7721146
## [5,] 0.7348794 0.7165502 0.7545845 0.8218777 0.8855105
##
## $loss
## [1] 2.338084
```

Just to give you some comfort this really is doing the same work as above, here we compare layer1 after the first forward iteration. They agree perfectly.

```
layer1
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.5000000 0.5000000 0.5000000 0.5000000 0.5000000
## [2,] 0.8193447 0.8309446 0.8368837 0.8250590 0.8958444
## [3,] 0.8309654 0.8242582 0.8978324 0.8475661 0.9039141
## [4,] 0.7686280 0.6987318 0.7768808 0.6958473 0.7721146
## [5,] 0.7348794 0.7165502 0.7545845 0.8218777 0.8855105
```

```
my.nn$layer1
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.5000000 0.5000000 0.5000000 0.5000000 0.5000000
## [2,] 0.8193447 0.8309446 0.8368837 0.8250590 0.8958444
## [3,] 0.8309654 0.8242582 0.8978324 0.8475661 0.9039141
## [4,] 0.7686280 0.6987318 0.7768808 0.6958473 0.7721146
## [5,] 0.7348794 0.7165502 0.7545845 0.8218777 0.8855105
```

We can do the same for the back-propagation. Here we wrap all the steps of back-prorogation for our neural network up into a single function call. Again, it produces equivalent results to the above discussion.

```
backprop.step <- function(nn)
{
  nn$d.weight2 <- t(nn$layer1) %*%
    as.matrix((2*(nn$y-nn$output)*sigmoid.derivative(nn$output)))

  nn$d.weight1 <- as.matrix(2*(nn$y-nn$output)*sigmoid.derivative(nn$output)) %*%
    t(nn$weights2)
  nn$d.weight1 <- nn$d.weight1*sigmoid.derivative(nn$layer1)
  nn$d.weight1 <- t(nn$input) %*% nn$d.weight1

  nn$weights2 <- nn$weights2 + nn$d.weight2
  nn$weights1 <- nn$weights1 + nn$d.weight1

  return(nn)
}

my.nn<-backprop.step(my.nn)
my.nn
```

```
## $input
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    1    1    1    1
## [3,]    1    1    0    1    1
## [4,]    1    0    0    1    0
## [5,]    1    0    1    0    1
##
## $weights1
##             [,1]      [,2]      [,3]         [,4]      [,5]
## [1,]  0.46150412 0.1706752 0.6361541  0.380806717 0.8431187
## [2,]  0.25529886 0.2137890 0.5503742 -0.004134642 0.5902222
## [3,]  0.31324464 0.1373337 0.0752382  0.142324544 0.7203531
## [4,]  0.64835045 0.5632553 0.5808326  0.350269777 0.3338176
## [5,] -0.02377642 0.3003506 0.3213466  0.719428744 0.3534156
##
## $weights2
##             [,1]
## [1,]  0.01440683
## [2,]  0.14521889
## [3,] -0.46404547
## [4,]  0.04143904
## [5,] -0.41842799
##
## $y
## [1] 0 0 1 1 0
##
```

```
## $output
##           [,1]
## [1,] 0.8108582
## [2,] 0.9191480
## [3,] 0.9218863
## [4,] 0.8934192
## [5,] 0.9045985
##
## $layer1
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.5000000 0.5000000 0.5000000 0.5000000 0.5000000
## [2,] 0.8193447 0.8309446 0.8368837 0.8250590 0.8958444
## [3,] 0.8309654 0.8242582 0.8978324 0.8475661 0.9039141
## [4,] 0.7686280 0.6987318 0.7768808 0.6958473 0.7721146
## [5,] 0.7348794 0.7165502 0.7545845 0.8218777 0.8855105
##
## $loss
## [1] 2.338084
##
## $d.weight2
##            [,1]
## [1,] -0.6922401
## [2,] -0.6930688
## [3,] -0.7036346
## [4,] -0.7293325
## [5,] -0.7743257
##
## $d.weight1
##             [,1]        [,2]        [,3]        [,4]        [,5]
## [1,] -0.04597409 -0.05476140 -0.01550155 -0.04800270 -0.02160254
## [2,] -0.05146965 -0.06074153 -0.01736350 -0.05603796 -0.02513026
## [3,] -0.11366303 -0.13497140 -0.03827079 -0.12185312 -0.05475675
## [4,] -0.04475163 -0.05257396 -0.01509267 -0.04852095 -0.02175111
## [5,] -0.10891239 -0.12932094 -0.03670334 -0.11670540 -0.05243435
```

Full Example

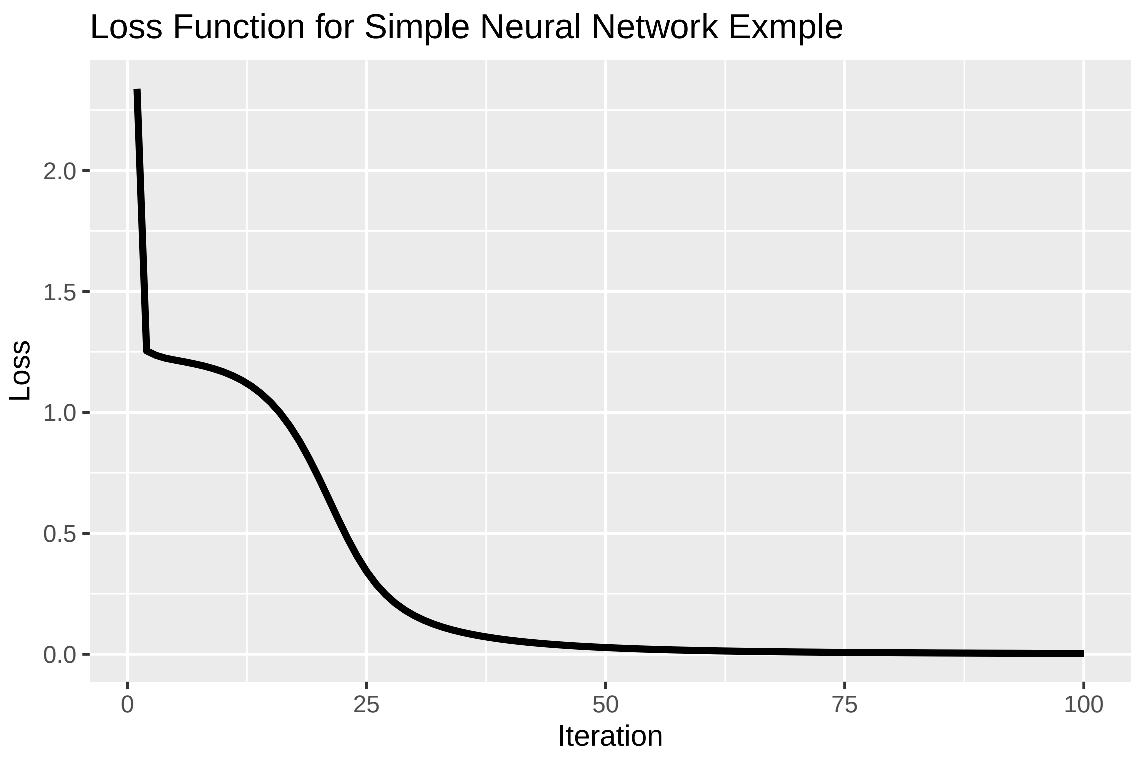
We now have everything we need to let 'tune' the neural network. Here I run 100 epochs, repeating forward propagation to obtain a loss, then back-propagation to update the weights in the direction that improves the loss. After 100 repeats, we evaluate the results of the trained network. If you try this at home, it takes about a half-second to actually compute all this!

```
set.seed(10)
my.nn <- list(
  input = X,
  weights1 = matrix(runif(ncol(X)*nrow(X)), ncol=nrow(X)),
  weights2 = matrix(runif(nrow(X)),ncol=1),
  y = y,
  output = matrix(ncol=1)
)

loss.out <- numeric()
for(j in 1:100)
{
  my.nn <- feedforward.step(my.nn)
  loss.out[j] <- my.nn$loss
  my.nn <- backprop.step(my.nn)
}
```

I create a ggplot graphic to visualize the changes in our loss.

```
loss.plot <- data.frame(x = 1:100, y = loss.out)
ggplot(loss.plot, aes(x=x, y=y)) + geom_line(size=1.25) +
  labs(title='Loss Function for Simple Neural Network Exmple', y='Loss', x='Iteration')
```



Finally, how well are we predicting our 5 training examples?

```r
cbind(my.nn$output, y)
```

```
##                y
## [1,] 0.03205790 0
## [2,] 0.02082247 0
## [3,] 0.96690371 1
## [4,] 0.97208231 1
## [5,] 0.01080376 0
```

Remarkably, this network is now seeing the zeros and ones pretty clearly. This is a good place to stop, but we could certainly keep going even with this basic example. We could predict a new unseen observation, we could consider how to validate this model, there is more to learn even at this introductory level! As for more with neural networks, the sky is the limit at this time. There are always new architectures being developed and new problems to tackle. Computer vision and natural language processing are two vital areas of research at the moment. If this has made you interested, go check out the posts at the top of this document! Good luck, and I hope this gives you a little excitement for neural networks and machine learning!