Neural Networks and Gradient Descent

Andus Kong

February 25, 2016

For this homework assignment, we will build and train a simple neural network, using the famous "iris" dataset. We will take the four variables `Sepal.Length`, `Sepal.Width`, `Petal.Length`, and `Petal.Width` to create a prediction for the species.

We will train the network using gradient descent, a commonly used tool in machine learning.

Task 0:

Split the iris data into a training and testing dataset. Scale the data so the numeric variables are all between 0 and 1.

```r
# split between training and testing data
set.seed(1)
n <- dim(iris)[1]
rows <- sample(1:n, 0.8*n)
train <- iris[rows,]
test <- iris[-rows,]

# write your code here
colmax <- apply(train[,1:4], MARGIN = 2, max)
Train <- t(t(train[,1:4]) / colmax)

colmax <- apply(test[,1:4], MARGIN = 2, max)
Test <- t(t(test[,1:4]) / colmax)
```

Setting up our network

Our neural network will have four neurons in the input layer - one for each numeric variable in the dataset. Our output layer will have three outputs - one for each species. There will be a `Setosa`, `Versicolor`, and `Virginica` node. When the neural network is provided 4 input values, it will produce an output where one of the output nodes has a value of 1, and the other two nodes have a value of 0. This is a similar classification strategy we used for the classification of handwriting digits.

I have arbitrarily chosen to have two nodes in our hidden layer.

We will add bias values before applying the activation function at each of our nodes in the hidden and output layers.

Task 1:

How many parameters are present in our model? List how many are present in: weight matrix 1, bias values for the hidden layer, weight matrix 2, and bias values for output layer.

Your answer: Input Layer: 4 variables - Sepal Length, Sepal Width, Petal Length, Petal Width
Output Layer: 3 variables (in vector form i.e. [1,0,0]) - Setosa, Versicolor, Virginica
Hidden Layer: 2 variables (arbitrarily chosen)

Weight Matrix 1 ($W^{(1)}$) is a 4x2 matrix with 8 values representing going from our input layer to our hidden layer
Bias Values Hidden Layer ($B^{(1)}$) is vector of 2 values where we add bias before applying sigmoid function in the hidden layer
Weight Matrix 2 ($W^{(2)}$) is a 2x3 matrix with 6 values representing going from our hidden layer to our input layer
Bias Values Output Layer ($B^{(2)}$) is a vector of 10 values where we add bias after multiplying weights

Notation

We will define each matrix of values as follows:

$W^{(1)}$ the weights applied to the input layer.

$B^{(1)}$ are the bias values added before activation in the hidden layer.

$W^{(2)}$ the weights applied to the values coming from the hidden layer.

$B^{(2)}$ are the bias values added before the activation function in the output layer.

Task 2:

To express the categories correctly, we need to turn the factor labels in species column into vectors of 0s and 1s. For example, an iris of species *setosa* should be expressed as `1 0 0`. Write some code that will do this. Hint: you can use `as.integer()` to turn a factor into numbers, and then use a bit of creativity to turn those values into vectors of 1s and 0s.

```r
# your code goes here
y <- train$Species
y <- as.integer(y)
vectorize <- function(x){
  k <- rep(0,3)
  k[x] <- 1
  k
}

y <- t(apply(matrix(y), 1, vectorize))
```

Forward Propagation

Sigmoid Activation function We will use the sigmoid function as our activation function.

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

Task 3:

Write the output ($\hat{y}$) of the neural network as a function or series of functions of the parameters ($W^{(1)}, B^{(1)}, W^{(2)}, B^{(2)}$).

In the language of neural networks, this step is called forward propagation. It's the idea of taking your input values and propagating the changes forward until you get your predictions.

You can vist https://github.com/stephencwelch/Neural-Networks-Demystified/blob/master/Part%202%20Forward%20Propagation.ipynb to see how the series of functions would be written if we did not use bias values in our calculations.

Your answer: Write your answer here. Not required, but I recommend the use of latex code to express the mathematics. You can learn about writing mathematics in latex at: https://en.wikibooks.org/wiki/LaTeX/Mathematics. (Don't worry about using bold print to express something as a matrix.)

$$Z^{(2)} = XW^{(1)}$$

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

$$a^{(2)} = f(Z^{(2)} + B^{(1)})$$

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

$$\hat{y} = f(Z^{(3)} + B^{(2)})$$

Task 4:

Express the forward propagation as R code using the training data. For now use random uniform values as temporary starting values for the weights and biases.

```r
# your code goes here
input_layer_size <- 4
hidden_layer_size <- 2
output_layer_size <- 3
set.seed(1)
W_1 <- matrix(runif(input_layer_size * hidden_layer_size)-.5, nrow = input_layer_size, ncol = hidden_la
W_2 <- matrix(runif(hidden_layer_size * output_layer_size)-.5, nrow = hidden_layer_size, ncol = output_
B1 <- matrix(runif(hidden_layer_size), ncol = 1)
B2 <- matrix(runif(output_layer_size), ncol = 1)

X <- Train

sigmoid <- function(Z){
  1/(1 + exp(-Z))
}

Z_2 <- as.matrix(X) %*% W_1
A_2 <- sigmoid(Z_2 + t(B1 %*% rep(1,120)))
Z_3 <- A_2 %*% W_2
Y_hat <- sigmoid(Z_3 + t(B2 %*% rep(1,120)))
```

Back Propagation

The cost function that we will use to evaluate the performance of our neural network will be the squared error cost function:

$$J = 0.5 \sum (y - \hat{y})^2$$

Task 5:

Find the gradient of the cost function with respect to the parameters.

You will create four partial derivatives, one for each of $(W^{(1)}, B^{(1)}, W^{(2)}, B^{(2)})$.

This is known as back propagation. The value of the cost function ultimately depends on the data and our predictions. Our predictions are just a result of a series of operations which you have defined in task 2. Thus, when you calculate the derivative of the cost function, you will be applying the chain rule for derivatives as you take the derivative with respect to an early element.

Your answer:

$$\frac{\partial J}{\partial B^{(2)}} = -(y - \hat{y}) \frac{e^{-(Z^{(3)}+B^{(2)})}}{(1 + e^{-(Z^{(3)}+B^{(2)})})^2}$$

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial J}{\partial B^{(2)}} \frac{1}{1 + e^{-(Z^{(2)}+B^{(1)})}}$$

$$\frac{\partial J}{\partial B^{(1)}} = \frac{\partial J}{\partial B^{(2)}} W^{(2)} \frac{e^{-(Z^{(2)}+B^{(1)})}}{(1 + e^{-(Z^{(2)}+B^{(1)})})^2}$$

$$\frac{\partial J}{\partial W^{(1)}} = X \frac{\partial J}{\partial B^{(1)}}$$

Task 6:

Turn your partial derivatives into R code. This step might require some shuffling around of terms because the elements are all matrices and matrix multiplication requires that the inner dimmensions match. To make sure you have coded it correctly, it is always a good idea to perform numeric gradient checking, which will be your next task.

```r
# your code goes here
Y <- y

sigmoidprime <- function(z){
  exp(-z)/((1+exp(-z))^2)
}

delta_3 <- ( -(Y - Y_hat) * sigmoidprime(Z_3 + t( B2 %*% rep(1, 120))))
djdb2 <- delta_3
djdw2 <- t(A_2) %*% delta_3

delta_2 <- delta_3 %*% t(W_2) * sigmoidprime(Z_2 + t( B1 %*% rep(1,120)  ) )
djdb1 <- delta_2
djdw1 <- t(X) %*% delta_2
```

Numerical gradient checking

Task 7:

Perform numeric gradient checking. For the purpose of this homework assignment, show your numeric gradient checking for just the $W^{(1)}$ matrix. You should do numeric gradient checking for all elements in your neural network, but for the sake of keeping the length of this assignment manageable, show your code and results for the first weight matrix only.

To perform numeric gradient checking, create an initial set of parameter values for all of the values (all weight matricies and all bias values). Calculate the predicted values based on these initial parameters, and calculate the cost associated with them. Store this 'initial' cost value.

You will then perturb one element in the $W^{(1)}$ matrix by a small value, say 1e-4. You will then recalculate the predicted values and associated cost. The difference between the new value of the cost function and the initial cost gives us an idea of the change in J. Divide that change by the size of the perturbation (1e-4), and we now have an idea of the slope (partial derivative). You'll repeat this for all of the elements in the $W^{(1)}$ matrix.

I do recommend performing this check for all of the elements.

```r
cost <- function(y,y_hat){
  0.5*sum((y - y_hat)^2)
}

set.seed(1)
W_1 <- matrix(runif(input_layer_size * hidden_layer_size)-.5, nrow = input_layer_size, ncol = hidden_la
W_2 <- matrix(runif(hidden_layer_size * output_layer_size)-.5, nrow = hidden_layer_size, ncol = output_
B1 <- matrix(runif(hidden_layer_size), ncol = 1)
B2 <- matrix(runif(output_layer_size), ncol = 1)

Z_2 <- X %*% W_1
A_2 <- sigmoid(Z_2 + t( B1 %*% rep(1,120) ) )
Z_3 <- A_2 %*% W_2
Y_hat <- sigmoid(Z_3 + t( B2 %*% rep(1,120) ) )
currentcost <- cost(Y,Y_hat)  # Current cost

e <- 1e-4  # size of perturbation



# place holder for our numeric gradients
numgrad_w_1 <- matrix(0, nrow = input_layer_size, ncol = hidden_layer_size)
elements <- input_layer_size * hidden_layer_size

for(i in 1:elements){  # calculate the numeric gradient for each value in the W matrix
  set.seed(1)
  W_1 <- matrix(runif(input_layer_size * hidden_layer_size)-.5, nrow = input_layer_size, ncol = hidden_l
  W_2 <- matrix(runif(hidden_layer_size * output_layer_size)-.5, nrow = hidden_layer_size, ncol = outpu
  B1 <- matrix(runif(hidden_layer_size), ncol = 1)
  B2 <- matrix(runif(output_layer_size), ncol = 1)

  W_1[i] <- W_1[i] + e # apply the perturbation

  Z_2 <- X %*% W_1
  A_2 <- sigmoid(Z_2 + t( B1 %*% rep(1,120) ) )
  Z_3 <- A_2 %*% W_2
  Y_hat <- sigmoid(Z_3 + t( B2 %*% rep(1,120) ) )
  numgrad_w_1[i] <- (cost(Y,Y_hat) - currentcost)/e # change in cost over perturbation = slope
}
```

Now check to make sure that the values produced by the numeric gradient check match the values of the gradient as calculated by the partial derivatives which you calculated in Task 4. The match won't be perfect, but should be pretty good.

```
# your code goes here
numgrad_w_1
```

```
##             [,1]       [,2]
## [1,]  0.04087880 -1.1161995
## [2,] -0.02384859 -0.9788308
## [3,]  0.07314970 -0.9360505
## [4,]  0.04542186 -0.8810974
```

```
djdw1
```

```
##                     [,1]       [,2]
## Sepal.Length  0.04087841 -1.1162204
## Sepal.Width  -0.02384998 -0.9788470
## Petal.Length  0.07314907 -0.9360667
## Petal.Width   0.04542013 -0.8811129
```

Gradient Descent

Task 8:

We will now apply the gradient descent algorithm to train our network. This simply involves repeatedly taking steps in the direction opposite of the gradient.

With each iteration, you will calculate the predictions based on the current values of the model parameters. You will also calculate the values of the gradient at the current values. Take a 'step' by subtracting a scalar multiple of the gradient. And repeat.

I will not specify what size scalar multiple you should use, or how many iterations need to be done. Just try things out. A simple way to see if your model is performing 'well' is to print out the predicted values of y-hat and see if they match closely to the actual values.

```
# your code goes here

set.seed(1)
W_1 <- matrix(runif(input_layer_size * hidden_layer_size)-.5, nrow = input_layer_size, ncol = hidden_la
W_2 <- matrix(runif(hidden_layer_size * output_layer_size)-.5, nrow = hidden_layer_size, ncol = output_
B1 <- matrix(runif(hidden_layer_size), ncol = 1)
B2 <- matrix(runif(output_layer_size), ncol = 1)

for(i in 1:5000){
  scalar <- 0.5
  X <- as.matrix(Train)
  Y <- y


  Z_2 <- X %*% W_1
  A_2 <- sigmoid(Z_2 + t( B1 %*% rep(1, 120)))
  Z_3 <- A_2 %*% W_2
  Y_hat <- sigmoid(Z_3 + t( B2 %*% rep(1,120)))



  delta_3 <- ( -(Y - Y_hat) * sigmoidprime(Z_3 + t( B2 %*% rep(1,120)) ))
  djdb2 <- rep(1, 120) %*% delta_3
  djdw2 <- t(A_2) %*% delta_3

  delta_2 <- delta_3 %*% t(W_2) * sigmoidprime(Z_2 + t( B1 %*% rep(1,120)) )
  djdb1 <- rep(1, 120) %*% delta_2
  djdw1 <- t(X) %*% delta_2


  # update our weights
  W_1 <- W_1 - scalar * djdw1
  B2  <- B2  - scalar * t(djdb2)
  W_2 <- W_2 - scalar * djdw2
  B1  <- B1  - scalar * t(djdb1)

}

head(round(Y_hat))
```

```
##     [,1] [,2] [,3]
## 40     1    0    0
## 56     0    1    0
## 85     0    1    0
## 134    0    1    0
## 30     1    0    0
## 131    0    0    1
```

Testing our trained model

Now that we have performed gradient descent and have effectively trained our model, it is time to test the performance of our network.

Task 9

Using the testing data, create predictions for the 30 observations in the test dataset. Print those results.

```
# your code goes here

Xt <- Test
batch_size <- dim(Xt)[1]
```

```
Z_2 <- Xt %*% W_1
A_2 <- sigmoid(Z_2 + t( B1 %*% rep(1,batch_size) ) )
Z_3 <- A_2 %*% W_2
Y_hat <- sigmoid(Z_3 + t( B2 %*% rep(1,batch_size) ) )
guess <- round(Y_hat)

results <- cbind(guess, test$Species)

results
```

```
##     [,1] [,2] [,3] [,4]
## 5      1    0    0    1
## 11     1    0    0    1
## 12     1    0    0    1
## 17     1    0    0    1
## 19     1    0    0    1
## 23     1    0    0    1
## 32     1    0    0    1
## 35     1    0    0    1
## 36     1    0    0    1
## 45     1    0    0    1
## 52     0    1    0    2
## 63     0    1    0    2
## 74     0    1    0    2
## 76     0    1    0    2
## 89     0    1    0    2
## 92     0    1    0    2
## 93     0    1    0    2
## 100    0    1    0    2
## 104    0    0    1    3
## 106    0    0    1    3
## 112    0    0    1    3
## 115    0    0    1    3
## 120    0    0    1    3
## 121    0    0    1    3
## 124    0    0    1    3
## 126    0    0    1    3
## 133    0    0    1    3
## 135    0    0    1    3
## 144    0    0    1    3
## 147    0    0    1    3
```

How many errors did your network make?

Answer: My network made zero errors. The right most column refers to which row the column the 1's should be.

Using package `nnet`

While instructive, the manual creation of a neural network is seldom done in production environments. (TBH, Python is more commonly used for neural networks in production environments, but that's for another time.)

[Install the `nnet` package and] Read the documentation for the function `nnet()`. Create a neural network for predicting the iris species based on the four numeric variables. Use the same training data to train the network. The function `nnet()` is smart enough to recognize that the values in the species column are a factor and will need to expressed in 0s and 1s as we did in our manually created network.

```r
# your code goes here
library("nnet")
Train_net <- data.frame(Train, train$Species)
colnames(Train_net) <- colnames(train)
m1 <- nnet(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width, data = Train_net, size =
```

```
## # weights:  19
## initial  value 136.688435
## iter  10 value 85.093587
## iter  20 value 7.506465
## iter  30 value 5.896692
## iter  40 value 5.853697
## iter  50 value 5.816343
## iter  60 value 5.795005
## iter  70 value 5.744981
## iter  80 value 5.714736
## iter  90 value 5.700597
## iter 100 value 5.698128
## final  value 5.698128
## stopped after 100 iterations
```

Once you have created the network with nnet, use the predict function to make predictions for the test data.

```r
# your code goes here
round(predict(m1,newdata = Test))
```

```
##     setosa versicolor virginica
## 5        1          0         0
## 11       1          0         0
## 12       1          0         0
## 17       1          0         0
## 19       1          0         0
## 23       1          0         0
## 32       1          0         0
## 35       1          0         0
## 36       1          0         0
## 45       1          0         0
## 52       0          1         0
## 63       0          1         0
## 74       0          1         0
## 76       0          1         0
## 89       0          1         0
## 92       0          1         0
## 93       0          1         0
## 100      0          1         0
## 104      0          0         1
## 106      0          0         1
## 112      0          0         1
## 115       0           0          1
## 120       0           0          1
## 121       0           0          1
## 124       0           0          1
## 126       0           0          1
## 133       0           0          1
## 135       0           0          1
## 144       0           0          1
## 147       0           0          1
```

How does `nnet` perform?

Answer: `nnet` was perfect in its prediction of the testing set