Simple MLP From Scratch

My goal for this project is to create a simple MLP neural network by hand. I'll do this by writing code that manually does the matrix operations and computes the gradients for backpropagation.

The neural network will be designed for species classification on the well-known Iris dataset. After building and testing our model, we'll compare the results with the output of R's nnet() function for a network of the same structure.

Data Preparation

```
# split between training and testing data
set.seed(3)
n <- dim(iris)[1]
rows <- sample(1:n, 0.8 * n)
train <- iris[rows,]
test <- iris[-rows,]

train_species <- train[5]
train_data <- train[1:4]
test_species <- test[5]
test_data <- test[1:4]

## Scale our data by dividing by the global max of that column (max of the entire iris dataset, not jus
maxes <- as.numeric(apply(iris, 2, max)[1:4])
train <- as.data.frame(mapply('/', train_data, maxes))
test <- as.data.frame(mapply('/', test_data, maxes))

train['Species'] <- train_species
test['Species'] <- test_species

## Convert species names into factors
train['Setosa'] <- as.integer(train$Species == "setosa")
train["Versicolor"] <- as.integer(train$Species == "versicolor")
train['Virginica'] <- as.integer(train$Species == "virginica")

test['Setosa'] <- as.integer(test$Species == "setosa")
test["Versicolor"] <- as.integer(test$Species == "versicolor")
test['Virginica'] <- as.integer(test$Species == "virginica")
```

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.

For simplicity, we'll have one hidden layer with three nodes.

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.

J is a matrix of 1s so that the bias values in B can be added to all rows.

Sigmoid Activation function

We will use the sigmoid function as our activation function.

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

Forward Propagation Process

$$\underset{N\times 4}{\mathbf{X}}\ \underset{4\times 3}{\mathbf{W}^{(1)}} + \underset{N\times 1}{\mathbf{J}}\ \underset{1\times 3}{\mathbf{B}^{(1)T}} = \underset{N\times 3}{\mathbf{z}^{(2)}}$$

$$f(\underset{N\times 3}{\mathbf{z}^{(2)}}) = \underset{N\times 3}{\mathbf{a}^{(2)}}$$

$$\underset{N\times 3}{\mathbf{a}^{(2)}}\underset{3\times 3}{\mathbf{W}^{(2)}} + \underset{N\times 1}{\mathbf{J}}\ \underset{1\times 3}{\mathbf{B}^{(2)T}} = \underset{N\times 3}{\mathbf{z}^{(3)}}$$

$$f(\underset{N\times 3}{\mathbf{z}^{(3)}}) = \underset{N\times 3}{\hat{\mathbf{y}}}$$

Forward Propogation Code

```
input_layer_size <- 4
hidden_layer_size <- 3
output_layer_size <- 3
num_points <- length(train$Sepal.Length)

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


## Randomly initialize weight and bias matrices
set.seed(1)

W_1 <- matrix(runif(input_layer_size * hidden_layer_size), nrow = input_layer_size)
B_1 <- matrix(runif(hidden_layer_size), nrow = hidden_layer_size)
W_2 <- matrix(runif(hidden_layer_size * output_layer_size), nrow = hidden_layer_size)
B_2 <- matrix(runif(output_layer_size), nrow = output_layer_size)
```

```
J <- rep(1, num_points)


X <- unname(as.matrix(train[1:4]))

Z_2 <- X %*% W_1 + t(B_1 %*% J)

A_2 <- sigmoid(Z_2)

Z_3 <- A_2 %*% W_2 + t(B_2 %*% J)

Y_hat <- sigmoid(Z_3)
```

Back Propagation

The cost function that we will use to evaluate the performance of our neural network will be the squared error cost function: (add constant of 0.5 to make derivative easier)

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

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

Compute Gradients Recall: $f()$ refers to sigmoid function, $f'()$ refers to derivative of sigmoid function.

$$\frac{\partial C}{\partial W^{(2)}} = A^{(2)T}((\hat{y} - y) \odot f'(z^{(3)}))$$

$$\frac{\partial C}{\partial B^{(2)}} = (\mathbf{J}^T((\hat{y} - y) \odot f'(z^{(3)})))^T$$

$$\frac{\partial C}{\partial W^{(1)}} = X^T((((\hat{y} - y) \odot f'(z^{(3)}))W^{(2)T}) \odot f'(z^{(2)T}))$$

$$\frac{\partial C}{\partial B^{(1)}} = (\mathbf{J}^T((((\hat{y} - y) \odot f'(z^{(3)}))W^{(2)T}) \odot f'(z^{(2)T})))^T$$

Back Propogation Code

```
sigmoid_deriv <- function(Z){
  exp(-Z) / (1 + exp(-Z))^2
}

Y <- unname(as.matrix(train[6:8]))

delta3 = (Y_hat - Y) * sigmoid_deriv(Z_3)
```

```r
djdw2 = t(A_2) %*% delta3

djdb2 = t(t(J) %*% delta3)

delta2 = (delta3 %*% t(W_2)) * sigmoid_deriv(Z_2)

djdw1 = t(X) %*% delta2

djdb1 = t(t(J) %*% delta2)
```

Sanity-Check

These gradients are pretty complex, so to make sure that they were done correctly, let's compute numeric gradients at the starting point and compare our results.

Since djdw1 uses the previous results, let's just compute that gradient.

```r
e <- 1e-4  # size of perturbation

# an empty placeholder to store our numeric gradient values
num_djdw1 <- matrix(0, nrow = 4 , ncol = 3)

# a loop to perturb each value and store the change in cost function/size of perturbation
for (i in 1:12) {  # calculate the numeric gradient for each value in the w1 matrix

  W_1_perb <- W_1
  W_1_perb[i] <- W_1_perb[i] + e

  Z_2_perb <- X %*% W_1_perb + t(B_1 %*% J)

  A_2_perb <- sigmoid(Z_2_perb)

  Z_3_perb <- A_2_perb %*% W_2 + t(B_2 %*% J)

  Y_hat_perb <- sigmoid(Z_3_perb)



  num_djdw1[i] <- (cost(Y, Y_hat_perb) - cost(Y, Y_hat)) / e # replace with appropriate value
}

num_djdw1
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.6195202 1.2507945 1.4633754
## [2,] 0.6214893 1.3014559 1.3808518
## [3,] 0.4030368 0.7422247 1.0963756
## [4,] 0.3312907 0.5761887 0.9825936
```

This lines up with our code:

```
djdw1
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.6195366 1.2508273 1.4634154
## [2,] 0.6215049 1.3014901 1.3808862
## [3,] 0.4030465 0.7422412 1.0964036
## [4,] 0.3312990 0.5762021 0.9826196
```

Running Gradient Descent

Now, we're ready to train the neural network:

```
X <- unname(as.matrix(train[1:4]))
Y <- unname(as.matrix(train[6:8]))

#### Gradient Descent Algorithm

costs <- rep(NA, 100000)
lambda <- 0.01

## Note: running the algorithm for many more iterations (500,000+) continues to lower the loss,
## but it doesn't make much of a difference and may actually cause overfitting, so we'll stop at 100,00

for(i in 1:100000){
  ## Calculate current loss
  Z_2 <- X %*% W_1 + t(B_1 %*% J)
  A_2 <- sigmoid(Z_2)
  Z_3 <- A_2 %*% W_2 + t(B_2 %*% J)
  Y_hat <- sigmoid(Z_3)
  costs[i] <- cost(Y, Y_hat)


  ## Calculate current gradients
  delta3 = (Y_hat - Y) * sigmoid_deriv(Z_3)
  djdw2 = t(A_2) %*% delta3
  djdb2 = t(t(J) %*% delta3)
  delta2 = (delta3 %*% t(W_2)) * sigmoid_deriv(Z_2)
  djdw1 = t(X) %*% delta2
  djdb1 = t(t(J) %*% delta2)

  ## Update weights based on gradient
  W_2 <- W_2 - lambda * djdw2
  B_2 <- B_2 - lambda * djdb2
  W_1 <- W_1 - lambda * djdw1
  B_1 <- B_1 - lambda * djdb1

  if((i %% 5000 == 0) | ((i < 10000) & (i %% 1000 == 0))){
    cat("Loss at Iteration ", i, ": ", costs[i], "\n")
  }
}
```

```
## Loss at Iteration  1000 :  14.0697
```

```
## Loss at Iteration  2000 :  4.697827
## Loss at Iteration  3000 :  2.812138
## Loss at Iteration  4000 :  2.269847
## Loss at Iteration  5000 :  1.996187
## Loss at Iteration  6000 :  1.818147
## Loss at Iteration  7000 :  1.686307
## Loss at Iteration  8000 :  1.581374
## Loss at Iteration  9000 :  1.494103
## Loss at Iteration  10000 :  1.419365
## Loss at Iteration  15000 :  1.152823
## Loss at Iteration  20000 :  0.9773923
## Loss at Iteration  25000 :  0.8458455
## Loss at Iteration  30000 :  0.7403764
## Loss at Iteration  35000 :  0.6528205
## Loss at Iteration  40000 :  0.578841
## Loss at Iteration  45000 :  0.5157641
## Loss at Iteration  50000 :  0.4617009
## Loss at Iteration  55000 :  0.4151824
## Loss at Iteration  60000 :  0.3750082
## Loss at Iteration  65000 :  0.3401797
## Loss at Iteration  70000 :  0.3098626
## Loss at Iteration  75000 :  0.2833616
## Loss at Iteration  80000 :  0.2600975
## Loss at Iteration  85000 :  0.2395883
## Loss at Iteration  90000 :  0.2214328
## Loss at Iteration  95000 :  0.2052961
## Loss at Iteration  100000 :  0.1908979
```

Testing the Model

First, let's examine the model's performance facing the unseen test data:

```r
predict_nn <- function(X){
  J <- rep(1, dim(X)[1])
  Z_2 <- X %*% W_1 + t(B_1 %*% J)
  A_2 <- sigmoid(Z_2)
  Z_3 <- A_2 %*% W_2 + t(B_2 %*% J)
  Y_hat <- sigmoid(Z_3)
  Y_hat
}

## Predict testing data

X_test <- unname(as.matrix(test[1:4]))
Y_test <- unname(as.matrix(test[6:8]))

Y_hat_test <- predict_nn(X_test)
predict_test <- round(Y_hat_test, digits = 0)


prediction_df <- data.frame("Predicted" = rep(NA, 30), "Actual" = unname(test[5]))
for(i in 1:30){
  if(predict_test[i, 1] == 1){
```

```
    prediction_df[i, 1] <- "setosa"
  }
  if(predict_test[i, 2] == 1){
    prediction_df[i, 1] <- "versicolor"
    }
  if(predict_test[i, 3] == 1){
    prediction_df[i, 1] <- "virginica"
  }
}

table(prediction_df)
```

```
##             Actual
## Predicted    setosa versicolor virginica
##   setosa          7          0         0
##   versicolor      0         14         2
##   virginica       0          0         7
```

It predicted 28/30 correctly, which makes sense given that there is some overlap in the numerical predictors between the Versicolor and Virginica species.

Now, let's compare these results to R's built in nnet() function:

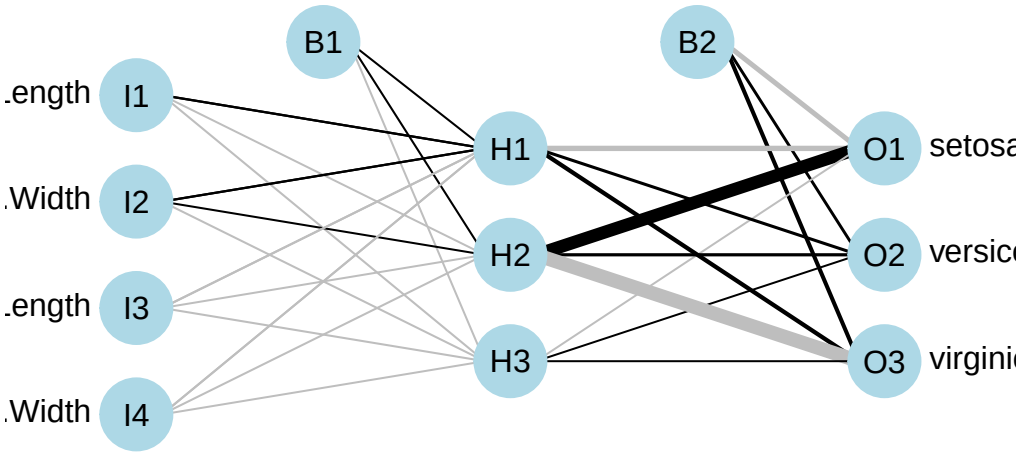
The nnet() function does some more complicated operations under the hood, but since the neural network structure is the same, we should expect to get similar results on the test data, which we do:

```
results <- max.col(predict(irismodel, iris[-rows,]))
results_df <- data.frame(results, actual = as.numeric(iris[-rows, 5]))
table(results_df)
```

```
##        actual
## results  1  2  3
##       1  7  0  0
##       2  0 14  2
##       3  0  0  7
```

We can also see a plot of the neural network here, to see how the different nodes relate:

```
plotnet(irismodel)
```



Conclusion

The project was successful, and the model worked as expected. Learning what neural networks are actually doing while training was valuable, but given the amount of work needed to build a model this simple (only 27 parameters), it's a good thing that Python and R have so many built-in packages when working with larger models.