Kaggle ML Challenge 3

2023-04-06

Introduction

As the getting started notebook had a great overview on getting started with the Multilayer perceptron. We jump started our kaggle challenge by first understanding the algorithm of the multilayer perceptron using the relu activation and sigmoid activation. At the same time, we also started researching on implementing the softmax activation function and understanding its algorithm.

Initially, we started implementing the code as provided by Professor to run through with the model to understand it better.

Importing the data

Firstly, we imported the data using the below function where the data is read and is then converted to a matrix form.

```r
x <- as.matrix(read.csv("mnist_train.csv"))
y <- as.matrix(read.csv("mnist_train_targets.csv"))
x_test <- as.matrix(read.csv("mnist_test.csv"))
```

Normalizing the data

The normalization of the data is performed so that it will help in improving the performance of the model. Here, the apply function is used to apply function to each column of the matrix x and x_test. Here, the function takes all the column of x and x_test which are divided by 255 to scale them between 0 and 1.

```r
x <- apply(x, 2, function(y) y / 255)
x_test <- apply(x_test, 2, function(y) y / 255)
```

We also tried implementing other normalization technique such as:

```r
x <- scale(x)
x_test <- scale(x_test)
```

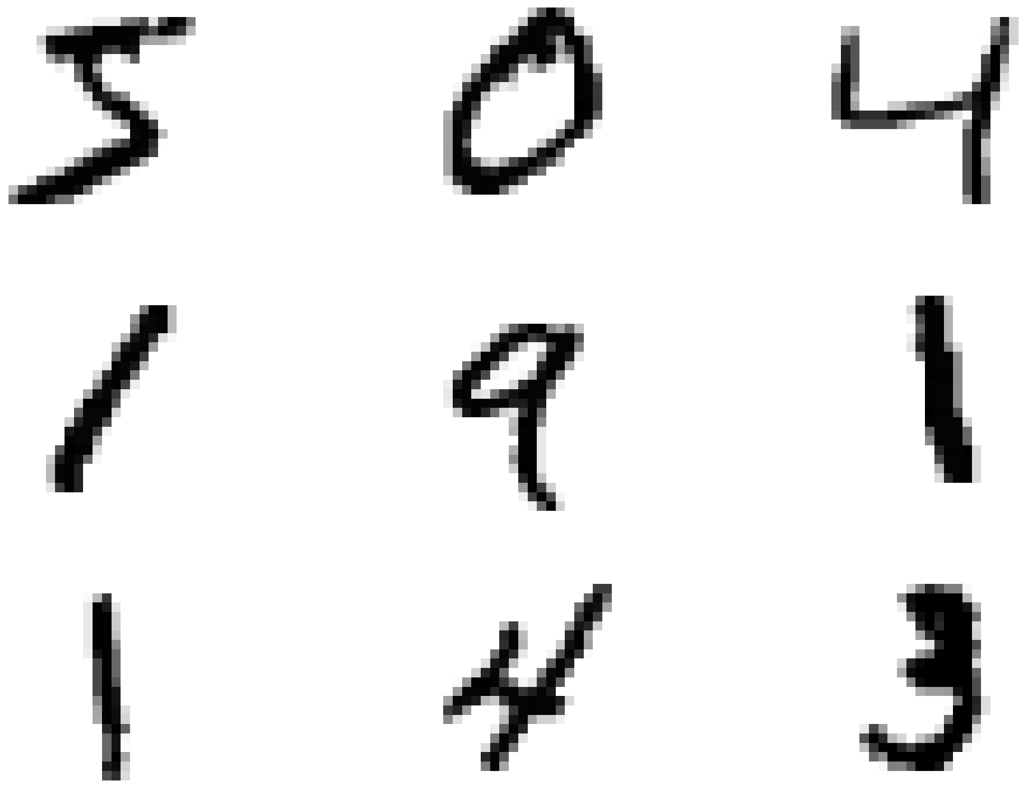
The scale function is used to standardize the vector of matrix where it divides each value in the vector with its standard deviation. The scale function calculates the mean and standard deviation for each column in the x. The use if scale function assures that all the columns in both x and x_test have same scale and applying normalization helps to remove the biases in the data. However, this normalization didn't provided good accuracy score, so we opted with Professor's code on normalization.

Visual Representation

The below code is from Professor DeBruine's getting started notebook which is used to plot the first 9 numbers in 3*3 grid.

```r
library(ggplot2)
plot_number <- function(fill){
  df <- data.frame("fill" = fill,
                   "y" = do.call(c, lapply(28:1, function(i) rep(i, 28))),
                   "x" = rep(1:28, 28))
  ggplot(df, aes(x, y, fill = fill)) + geom_tile() + scale_fill_gradient(low = "#FFFFFF", high = "#00000
}

# plot first nine numbers
cowplot::plot_grid(plotlist = lapply(1:9, function(i) plot_number(x[,i])), nrow = 3, ncol = 3)
```



Building a Model

We have used Multilayer perceptron to build the model where we initially started with the relu activation and sigmoid activation function. If the input value is positive, the relu activation function returns the value otherwise it returns 0. Similarly, the sigmoid function yields the value between 0 and 1.

Our multilayer_perceptron function had the input data x, the target output y, the number of epochs, learning rate, two hidden layers and the regularization parameter lambda. Our function initialized the weight and bias following the Glorot initialization.

Glorot Initialization is one of the used initialization method for initializing the weight matrices in neural networks. Through this method, it sets the initial weights to a random values and ensure that the initialized weights are neither too small nor too large.

Forward Propagation is used for making prediction for the input data whereas through **backward propagation**, it is used to calculate the error and update the weights and bias to minimize the error during the training process.

The forward and backward propagation are performed in the two loops that propagates the input data forward and compute the output. Similarly, then propagating the error backward through the network to update the weight and biases.

```
sigmoid <- function(x) 1 / (1 + exp(-x))
sigmoid_prime <- function(x) {
  sigmoid_x <- sigmoid(x)
  sigmoid_x * (1 - sigmoid_x)
}

relu <- function(x){
  x[x < 0] <- 0
  x
}
relu_prime <- function(x) x > 0


multilayer_perceptron <- function(x, y, epochs, lr, h1, h2, lambda){
  # weight and bias initialization (Glorot initialization)
  w1 <- matrix(runif(nrow(x) * h1, min = -sqrt(6 / (nrow(x) + h1)), max = sqrt(6 / (nrow(x) + h1))), nr
  b1 <- rnorm(h1, sd = 0.1)
  w2 <- matrix(runif(h1 * h2, min = -sqrt(6 / (h1 + h2)), max = sqrt(6 / (h1 + h2))), h1, h2)
  b2 <- rnorm(h2, sd = 0.1)
  w3 <- matrix(runif(h2 * 1, min = -sqrt(6 / (h2 + 1)), max = sqrt(6 / (h2 + 1))), h2, 1)
  b3 <- rnorm(1, sd = 0.1)

  for(epoch in 1:epochs){
    cat('trying sigmoid ',epoch,'\n')
    total_error <- 0
    for(i in 1:ncol(x)){
      # forward propagation
      z1 <- x[,i] %*% w1 + b1
      a1 <- relu(z1)
      z2 <- a1 %*% w2 + b2
      a2 <- relu(z2)
      z3 <- a2 %*% w3 + b3
      a3 <- sigmoid(z3)

      # backpropagation
      error <- y[[i]] - a3
      total_error <- total_error + abs(round(error))
      a3_delta <- error * sigmoid_prime(a3)
      a2_delta <- (a3_delta %*% t(w3)) * relu_prime(z2)
      a1_delta <- (a2_delta %*% t(w2)) * relu_prime(z1)

      # update weights and biases with L2 regularization
      w3 <- w3 + lr * (t(a2) %*% a3_delta - lambda * w3)
      b3 <- b3 + lr * a3_delta
      w2 <- w2 + lr * (t(a1) %*% a2_delta - lambda * w2)
      b2 <- b2 + lr * a2_delta
      w1 <- w1 + lr * (x[,i] %*% a1_delta - lambda * w1)
      b1 <- b1 + lr * a1_delta
    }
    # calculate regularization term and add to error
    reg_term <- (lambda / 2) * (sum(w1^2) + sum(w2^2) + sum(w3^2))
    total_error <- total_error + reg_term
    accuracy <- round((1 - total_error / ncol(x)) * 100, 2)
    cat(paste0("epoch: ", epoch, ", classification accuracy: ", accuracy, "%\n"))
  }
  list(w1 = w1, b1 = b1, w2 = w2, b2 = b2, w3 = w3, b3 = b3)
}
```

```
# set.seed(123)
# models <- lapply(0:9, function(digit){
#   multilayer_perceptron(x, as.numeric(y == digit), epochs = 10, lr = 0.001, h1 = 128, h2 = 64, lambda
# })
```

We tried with epoch set at 10 where it evaluates 10 separate multilayer perceptron. The learning rate is set at 0.001 that determines the step size taken during the optimization process while updating the weights. There are two hidden layers each specifying the number of neurons in the hidden layer. Additionally, we have used the L2 regularization to prevent the over fitting of the model.

```
# predict_mlp <- function(models, x_test){
#   pred <- matrix(0, length(models), ncol(x_test))
#   for(i in 1:ncol(x_test)){
#     for(j in 1:10){
#       z1 <- x[,i] %*% models[[j]]$w1 + models[[j]]$b1
#       a1 <- relu(z1)
#       z2 <- a1 %*% models[[j]]$w2 + models[[j]]$b2
#       a2 <- relu(z2)
#       z3 <- a2 %*% models[[j]]$w3 + models[[j]]$b3
#       pred[j, i] <- sigmoid(z3)
#     }
#   }
#   pred
# }
# pred_test <- apply(predict_mlp(models, x_test), 2, function(y) which.max(y) - 1)
# pred_train <- apply(predict_mlp(models, x), 2, function(y) which.max(y) - 1)
```

Through the predict_mlp function, it performs forward propagation for a single input that is from each of the 10 multilayer perceptron models. It goes in a loop which provides the activation of each layer of the neural network for the input and the output is stored in a matrix called pred. We then get our predicted digit label for each input in the rest and training sets. Through this we obtained an accuracy score of **0.11133**.

We tried implementing various hyper parameter techniques in the above code such as with changing the epochs, the hidden layers, adding different regularization, however, we were only able to make slight improvement from the previous accuracy which was to **0.11366**.

In the meantime, we were also trying to understand and implement softmax activation function. In the **softmax activation** function, it transforms the output to a vector of probabilities. That is the reason where it requires one-hot encoding to be used. Through one-hot encoding it changes the output in a way that is acceptable with the softmax activation function.

```r
#softmax activation function
softmax <- function(x) {
  e_x <- exp(x - max(x))
  return(e_x / sum(e_x))
}

#softmax prime function
softmax_prime <- function(x){
  s <- softmax(x)
  n <- length(x)
  jacobian <- matrix(0, nrow = n, ncol = n)
  for (i in 1:n){
    for (j in 1:n){
      if(i == j){
        jacobian[i, j] <- s[i] * (1 - s[i])
      }else{
        jacobian[i, j] <- -s[i] * s[j]
      }
    }
  }
  return(jacobian)
}

#relu function
relu <- function(x) ifelse(x > 0, x, 0)

#relu prime activation
relu_prime <- function(x) ifelse(x > 0, 1, 0)

# create one-hot encoded matrix y_mat
y_mat <- matrix(0, nrow(y), length(unique(y)))
for(i in 1:nrow(y_mat)){
  y_mat[i, y[[i]] + 1] <- 1
}

# Define the multilayer perceptron function
multilayer_perceptron <- function(x, y, epochs, lr, h1, h2, lambda){
  # weight and bias initialization (Glorot initialization)
  w1 <- matrix(runif(nrow(x) * h1, min = -sqrt(6 / (nrow(x) + h1)), max = sqrt(6 / (nrow(x) + h1))), nr
  b1 <- rnorm(h1, sd = 0.1)
  w2 <- matrix(runif(h1 * h2, min = -sqrt(6 / (h1 + h2)), max = sqrt(6 / (h1 + h2))), h1, h2)
  b2 <- rnorm(h2, sd = 0.1)
  w3 <- matrix(runif(h2 * 10, min = -sqrt(6 / (h2 + 10)), max = sqrt(6 / (h2 + 10))), h2, 10)
  b3 <- rnorm(10, sd = 0.1)

  for(epoch in 1:epochs){
    cat(' trying \n',epoch)
    total_error <- 0
    for(i in 1:ncol(x)){
      # forward propagation
      z1 <- x[,i] %*% w1 + b1
      a1 <- relu(z1)
      z2 <- a1 %*% w2 + b2
      a2 <- relu(z2)
      z3 <- a2 %*% w3 + b3
      a3 <- softmax(z3)

      # backpropagation
      error <- y_mat[i,] - a3
      total_error <- total_error + sum(abs(round(error)))
      a2_delta <- (a3_delta %*% t(w3)) * relu_prime(z2)
      a1_delta <- (a2_delta %*% t(w2)) * relu_prime(z1)

     # update weights and biases with L2 regularization
      w3 <- w3 + lr * (t(a2) %*% a3_delta - lambda * w3)
      b3 <- b3 + lr * a3_delta
      w2 <- w2 + lr * (t(a1) %*% a2_delta - lambda * w2)
      b2 <- b2 + lr * a2_delta
      w1 <- w1 + lr * (x[,i] %*% a1_delta - lambda * w1)
      b1 <- b1 + lr * a1_delta
      }
    # calculate regularization term and add to error
    reg_term <- (lambda / 2) * (sum(w1^2) + sum(w2^2) + sum(w3^2))
    total_error <- total_error + reg_term
    accuracy <- round((1 - total_error / ncol(x)) * 100, 2)
    cat(paste0("epoch: ", epoch, ", classification accuracy: ", accuracy, "%\n"))
    }
  list(w1 = w1, b1 = b1, w2 = w2, b2 = b2, w3 = w3, b3 = b3)
}
```

In the above method,we have used Glorot initialization and have performed relu and softmax activation function. The softmax activation function provides the probability distribution of the output where the softmax_prime function implements the jacobian matrix of the softmax function. The **jacobian matrix** is used in backpropagation algorithm to calculate the gradient of the loss function with the input.

Through the relu activation function, there is a non-linearity in the hidden layers of the network. Furtheremore,the relu prime provides the derivative of the relu function. We have also computed the one-hot encoding where it creates a matrix that represents the target variable y that can be used for the classification. The funcrion then uses the forward and backward propagation to provide the output as well as update the weights and bias.

```
# set.seed(123)
# models <- multilayer_perceptron(x, y_mat, epochs = 10, lr = 0.5, h1 = 128, h2 = 64, lambda = 0.001)
```

```
# Softmax
# predict_mlp <- function(models, x_test){
#   pred <-rep(0, ncol(x_test))
#   for(i in 1:ncol(x_test)){
#       z1 <- x_test[,i] %*% models[['w1']] + models[['b1']]
#       a1 <- relu(z1)
#       z2 <- a1 %*% models[['w2']] + models[['b2']]
#       a2 <- relu(z2)
#       z3 <- a2 %*% models[['w3']] + models[['b3']]
#       pred[ i] <- which.max(softmax(z3))-1
#   }
#   pred
```

```r
# }
#
# pred_test <- predict_mlp(models, x_test)
# pred_train <- predict_mlp(models, x)
```

We implemented the above softmax activation function, however, at the begining we were having issue on getting a good accuracy score as we missed on indexing for the one-hot encoding which caused our accuracy score to remain at **0.11266**.

However, implementing the relu and softmax activation increased our accuracy from 0.11366 to **0.88733**. Our model had two hidden layers, learning rate, epochs, and regularization method implemented.

As we now knew that we were on track, we also tried researching and implementing various other normalization method, initialization techniques and hyperparameter tuning.

We then implemented the he initialization technique. The reason for implementing the he initialization was that that it sets the initial values of the weights which prevents from having the vanishing gradient problems during the training. As we were using relu and softmax activation function, the he initialization worked well with these activation function as with relu function, it sets all the negative value to zero which results in sparsity of the activation. The updated code implementing the he initialization is as below:

```r
# define the multilayer perceptron function
multilayer_perceptron <- function(x, y, epochs, lr, h, verbose=TRUE){
  # weight and bias initialization (He initialization)
  w1 <- matrix(rnorm(nrow(x) * h, sd = sqrt(2 / nrow(x))), nrow(x), h)
  b1 <- rnorm(h, sd = 0.1)
  w2 <- matrix(rnorm(h * 10, sd = sqrt(2 / h)), h, 10)
  b2 <- rnorm(10, sd = 0.1)

  for(epoch in 1:epochs){
    if(verbose){
      cat('Epoch:', epoch)
    }
    total_error <- 0
    for(i in 1:ncol(x)){
      # forward propagation
      z1 <- x[,i] %*% w1 + b1
      a1 <- relu(z1)
      z2 <- a1 %*% w2 + b2
      a2 <- softmax(z2)
      # backpropagation
      error <- y_mat[i,] - a2
      total_error <- total_error + sum(abs(round(error)))
      a2_delta <- error  %*%  softmax_prime(a2)
      a1_delta <- (a2_delta %*% t(w2)) * relu_prime(z1)

      # update weights and biases
      w2 <- w2 + lr * (t(a1) %*% a2_delta)
      b2 <- b2 + lr * a2_delta
      w1 <- w1 + lr * (x[,i] %*% a1_delta)
      b1 <- b1 + lr * a1_delta
    }
    accuracy <- round((1 - total_error / ncol(x)) * 100, 2)
    if(verbose){
      cat(paste0(", classification accuracy: ", accuracy, "%\n"))
    }
  }
  list(w1 = w1, b1 = b1, w2 = w2, b2 = b2)
}
```

The implementation of the he initialization improved the score from **0.88733** to **0.93966** with 1 hidden layer. We then tried tweaking around through hyperparameter tuning and tried two and three hidden layers. As we know, increasing the number of hidden layers increases the ability to deal with the complex pattern in the data but at the same time, it will also increase the risk of overfitting. Hence, we tried implementing with three hidden layers, two hidden layers, and one. Among all one hidden layer provided an accuracy score of **0.97233**.

Adam Optimization is one of the popular optimization method in the neural network. We also tried implementing the adam optimization to see how it performs with our model.

```r
multilayer_perceptron <- function(x, y, epochs, lr, h, verbose=TRUE){
  # weight and bias initialization (He initialization)
  w1 <- matrix(rnorm(nrow(x) * h, sd = sqrt(2 / nrow(x))), nrow(x), h)
  b1 <- rnorm(h, sd = 0.1)
  w2 <- matrix(rnorm(h * 10, sd = sqrt(2 / h)), h, 10)
  b2 <- rnorm(10, sd = 0.1)

  # Adam parameters
  beta1 <- 0.9
  beta2 <- 0.999
  epsilon <- 1e-8
  m_w1 <- 0
  v_w1 <- 0
  m_b1 <- 0
  v_b1 <- 0
  m_w2 <- 0
  v_w2 <- 0
  m_b2 <- 0
  v_b2 <- 0

  for(epoch in 1:epochs){
    if(verbose){
      cat('Epoch:', epoch)
    }
    total_error <- 0
    for(i in 1:ncol(x)){
      # forward propagation
      z1 <- x[,i] %*% w1 + b1
      a1 <- relu(z1)
      z2 <- a1 %*% w2 + b2
      a2 <- softmax(z2)
      # backpropagation
      error <- y_mat[i,] - a2
      total_error <- total_error + sum(abs(round(error)))
      a2_delta <- error  %*%  softmax_prime(a2)
      a1_delta <- (a2_delta %*% t(w2)) * relu_prime(z1)

      # Adam weight and bias update
      dw2 <- t(a1) %*% a2_delta
      db2 <- a2_delta
      dw1 <- x[,i] %*% a1_delta
      db1 <- a1_delta

      m_w2 <- beta1 * m_w2 + (1 - beta1) * dw2
      v_w2 <- beta2 * v_w2 + (1 - beta2) * (dw2^2)
      m_hat_w2 <- m_w2 / (1 - beta1^epoch)
      v_hat_w2 <- v_w2 / (1 - beta2^epoch)
      w2 <- w2 + lr * m_hat_w2 / (sqrt(v_hat_w2) + epsilon)

      m_b2 <- beta1 * m_b2 + (1 - beta1) * db2
      v_b2 <- beta2 * v_b2 + (1 - beta2) * (db2^2)
      m_hat_b2 <- m_b2 / (1 - beta1^epoch)
      v_hat_b2 <- v_b2 / (1 - beta2^epoch)
      b2 <- b2 + lr * m_hat_b2 / (sqrt(v_hat_b2) + epsilon)

      m_w1 <- beta1 * m_w1 + (1 - beta1) * dw1
      v_w1 <- beta2 * v_w1 + (1 - beta2) * (dw1^2)
      m_hat_w1 <- m_w1 / (1 - beta1^epoch)
      v_hat_w1 <- v_w1 / (1 - beta2^epoch)
      w1 <- w1 + lr * m_hat_w1 / (sqrt(v_hat_w1) + epsilon)
      m_b1 <- beta1 * m_b1 + (1 - beta1) * db1
      v_b1 <- beta2 * v_b1 + (1 - beta2) * (db1^2)
      m_hat_b1 <- m_b1 / (1 - beta1^epoch)
      v_hat_b1 <- v_b1 / (1 - beta2^epoch)
      b1 <- b1 + lr * m_hat_b1 / (sqrt(v_hat_b1) + epsilon)

    }
accuracy <- round((1 - total_error / ncol(x)) * 100, 2)
if(verbose){
cat(paste0(", classification accuracy: ", accuracy, "%\n"))
}
}
list(w1 = w1, b1 = b1, w2 = w2, b2 = b2)
}
```

```r
# set.seed(123)
# models <- multilayer_perceptron(x, y_mat, epochs = 10, lr = 0.5, h = 512)
```

```r
# Softmax
# predict_mlp <- function(models, x_test){
#   pred <-rep(0, ncol(x_test))
#   for(i in 1:ncol(x_test)){
#       z1 <- x_test[,i] %*% models[['w1']] + models[['b1']]
#       a1 <- relu(z1)
#       z2 <- a1 %*% models[['w2']] + models[['b2']]
#       a2 <- relu(z2)
#       z3 <- a2 %*% models[['w3']] + models[['b3']]
#       pred[ i] <- which.max(softmax(z3))-1
#   }
#   pred
# }
#
```

```r
# pred_test <- predict_mlp(models, x_test)
# pred_train <- predict_mlp(models, x)
```

The above code too uses the relu and softmax activation function, however, this time it also uses the adam optimizer. The adam optimizer are set and initialized to zero and the function iterates and performs forward and backpropagation using the adam optimizer. However, the adam optimizer had the accuracy score of **0.96633** which wasn't an improvement with the previous score. We tried tweaking around with adam optimizer adding regularization too.

The Final Model

As the accuracy score from the adam optimizer was not as expected, we started tweaking relu and softmax activation with he initialization and the regularization.

```r
multilayer_perceptron <- function(x, y, epochs, lr, h1, lambda){

   # weight and bias initialization (He initialization)
   w1 <- matrix(rnorm(nrow(x) * h1, sd = sqrt(2 / nrow(x))), nrow(x), h1)
   b1 <- rnorm(h1, sd = 0.1)
   w2 <- matrix(rnorm(h1 * 10, sd = sqrt(2 / h1)), h1, 10)
   b2 <- rnorm(10, sd = 0.1)

   for(epoch in 1:epochs){
     cat(' trying \n',epoch)
     total_error <- 0
     for(i in 1:ncol(x)){
       # forward propagation
       z1 <- x[,i] %*% w1 + b1
       a1 <- relu(z1)
       z2 <- a1 %*% w2 + b2
       a2 <- softmax(z2)
       # backpropagation
       error <- y_mat[i,] - a2
       total_error <- total_error + sum(abs(round(error)))
       a2_delta <- error  %*% softmax_prime(a2)
       a1_delta <- (a2_delta %*% t(w2)) * relu_prime(z1)

       # update weights and biases with L2 regularization
       w2 <- w2 + lr * (t(a1) %*% a2_delta - lambda * w2)
       b2 <- b2 + lr * a2_delta
       w1 <- w1 + lr * (x[,i] %*% a1_delta - lambda * w1)
       b1 <- b1 + lr * a1_delta
     }
     # calculate regularization term and add to error
     reg_term <- (lambda / 2) * (sum(w1^2) + sum(w2^2))
     total_error <- total_error + reg_term
     accuracy <- round((1 - total_error / ncol(x)) * 100, 2)
     cat(paste0("epoch: ", epoch, ", classification accuracy: ", accuracy, "%\n"))
   }
   list(w1 = w1, b1 = b1, w2 = w2, b2 = b2)
 }
```

```
# set.seed(123)
# models <- multilayer_perceptron(x ,y_mat, epochs = 40, lr = 0.1, h = 512, lambda = 0.000001)
```

We set the epoch at 40, learning rate at 0.1, one hidden layer with 512 neurons and lambda set at 0.000001. The main use of regularization in the model was to prevent from overfitting and to improve our performance of the model. The regularization is implemented to the weights of the neural network in the update step. The regularization adds a penalty term to the cost function so that it is optimized during the training. Through this our accuracy score increased to **0.98233**.

Accuracy of training set predictions:

```
# sum(pred_train == y) / length(pred_train)
```

We had a accuracy of 100% for the training set predictions.

Prepare Kaggle submission:

We used the code given by Professor DeBruine to download it to csv which is as per the kaggle submission format.

```
# df <- data.frame("Id" = 1:10000, "Expected" = pred_test)
# write.csv(df, "example_submission.csv", row.names = FALSE)
```

I have commented out some of the code as it will take hours to execute each code. Hence, I have just outlayed my codes and the processes I implemented.

Conclusion

This project was very interesting and challenging at the same time. In this project, we were able to implement multilayer perceptron by executing the concept of neural network. Throughout this project, I learned alot about neural networks, back propagation, and different regularization techniques. This project helped a lot in understanding the algorithm and mathematics behind the neural network and training the model. All in all, this project was a great learning opportunity.

Reference

1. https://www.kaggle.com/code/zdebruine/getting-started-with-mnist
2. https://stackoverflow.com/questions/45949141/compute-a-jacobian-matrix-from-scratch-in-python