Kaggle Challenge 3

2023-04-10

Introduction

I was extremely excited for this project as I was going to implement the concept of neural network. The first week I spend my time mostly researching and understanding the concept of the multilayer perceptron and its algorithm. The getting started notebook is always a great help for me to refer on and start tweaking and understanding. We started working around with relu and sigmoid activation function initially and slowly progressed towards executing the softmax activation function.

Data Import

We imported the data first and the below function reads the data and coverts it to matrix.

```
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"))
```

Normalization

We started the normalization as provided in the getting started notebook but gradually did tried some other normalization techniques too such as scaling the normalization. However, the scaling normalization didn't provided good accuracy score so we proceeded with the below code for normalizing the data.

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

The apply function is used in the above normalization where the function divides each element of the columns of matrices x and x_test with 255 to rescale the value to be between 0 and 1.

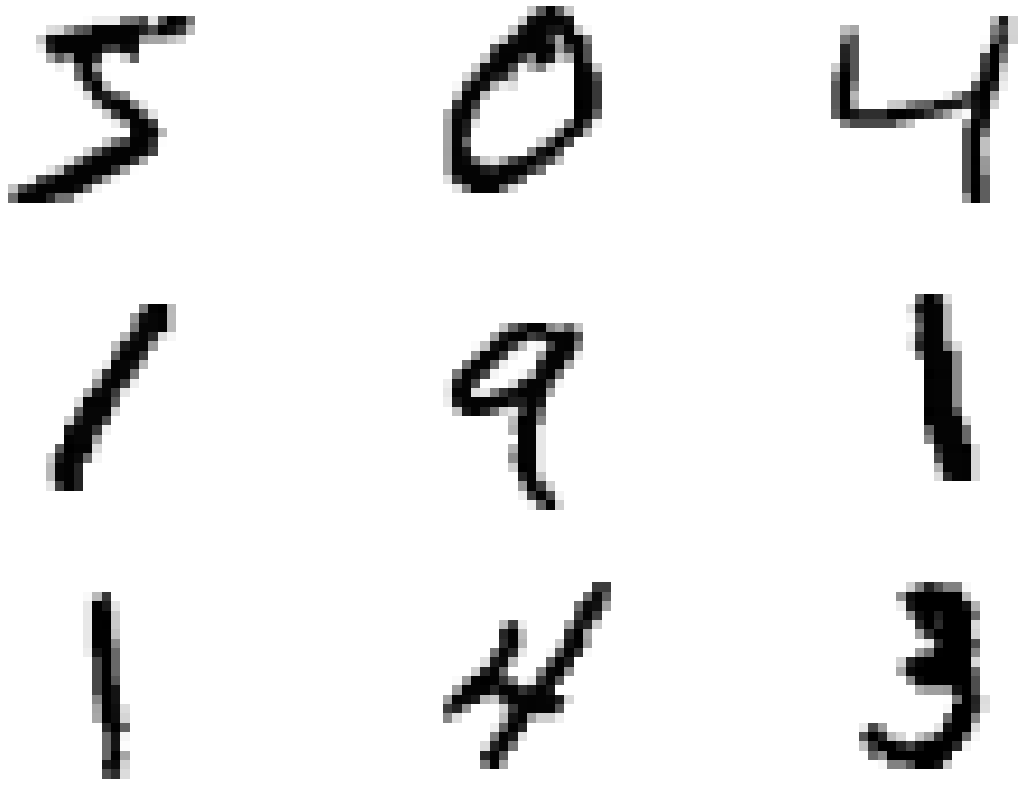
Visual Representation

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

```
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 = "#0000
}

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



Model

Initially, we worked on implementing relu and sigmoid activation function in our multilayer_perceptron function. Our multilayer_perceptron function contained two hidden layers, the regularization parameter lambda, the number of epochs, input data x and the target output y.

We used **glorot initalization** here where it ensures that the initialized weights are neither too little nor large by setting them to random numbers. The weights for each layers are initialized using the 'runif' function generating a random numbers from a uniform distribution.

Similarly, during **forward propagation**, the input data x is passed through the layers of network and the output y-pred is computed. Here, we have three hidden layers with two hidden layers with relu activation function and one output layer with sigmoid activation function. Prior to applying an activation function, each layer applies a weighted sum of the input from the layer before it.

The error between the expected output y_pred and the actual output y is determined during backward propagation, and the error is returned through the network to adjust weights and biases. Furthermore, during the updating weights and biases step, the weight is managed and the overfitting is avoided using the regularization parameter and learning rate.

```r
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

# leaky_relu <- function(x, alpha = 0.01) {
#   pmax(alpha * x, x)
# }

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
  #     cat('a1_delta = ',dim(a1_delta),'\n')
  # cat('t(x[,i]) = ',length(x[,i]),'\n')
      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)
}
```

I have commented the leaky_relu in the above code. I tried implementing leaky_relu as well, however, there wasn't a good accuracy score through it. With **leaky relu**, it allows for a small non-zero gradient when the input is negative whereas in **relu** the output for the negative is zero.

```
# models <- multilayer_perceptron(x, y, epochs = 10, lr = 0.1, h1 = 128, h2 = 64, lambda = 0.001)
```

```
# 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)
```

We attempted with using an epoch with a value of 10 which assesses 10 different multilayer perceptrons. The other hyperparameters used was learning rate - the step size used during the optimization process while updating the weights is determined through learning rate which was set at 0.0001. Additionally, L2 regularization was also used to stop the model from being overfit and our model had two hudden layers.

Through this we obtained an accuracy score of **0.11133**.We also 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**.

Implementing Tanh Activation Function

I also tried implementing tanh activation function to see how it performs and if it makes significant changes in the model. The difference with the tanh activation with the above activation functions were that the tanh function maps the output values to be between -1 and 1.

```
# Tanh activation function
tanh <- function(x){
  (exp(x) - exp(-x)) / (exp(x) + exp(-x))
}
```

```r
# Derivative of the tanh activation function
tanh_prime <- function(x){
  1 - tanh(x)^2
}


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 <- tanh(z1)
      z2 <- a1 %*% w2 + b2
      a2 <- tanh(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)) * tanh_prime(z2)
      a1_delta <- (a2_delta %*% t(w2)) * tanh_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)
}
```

There wasn't much improvement in the accuracy score through tanh activation and sigmoid function. So, we started implementing softmax activation function. The reason for using the **softmax activation** function is the output that we get through softmax activation function gives the probabilities that can be easily

interpreted as the likelihood of an input belonging to each class.

```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 function, we applied the relu and softmax activation function while using the Glorot initialization. The glorot initialization is used by the function to initialize the weight and biases. In the output layer, the function utilizes a softmax activation function and in the hidden layers, it uses relu function. The function utilizes both forward and back propagation to deliver the result. Through this method our accuracy score increased from **0.11366** to **0.88733**.

We further worked on implementing various hyperparameter tuning to improve our model. We then implemented **he initialization**. As the selection of the initialization also plays a significant effect in the model, we tried to see if the change in the initialization techniques makes an significant changes to our model.

He initialization sets the initial weights of the layer to a random values that us drawn from Gaussian distribution with mean 0 and variance 2/n. The reason for selecting he initialization was that it tends to perform better with relu activation function.

```
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)
}
```

We had hidden layer set at two then tried with three hidden layers, however, we were able to get a good accuracy score of **0.93966** from one hidden layer. We tried researching on the reason for why our one hidden layer performed better as we were expecting it to be more than one hidden layer and as we researched we found out that improving the hidden layer helps in improving models capacity to deal with complex patterns and data but also gives rise to over fitting.

We further tried tweaking with our function where we tried different hyperparameter tuning and were able to increase our accuracy score to **0.98233**.

As we knew we were on track, we also tried understanding and implementing adam optimization. **Adam Optimization** is one of the effective gradient descent optimization algorithm that is a mixture of AdaGrad and RMSprop. Adam optimization is used to update the weights and biases during the backpropagation.

```
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)
}
```

In the above function, adam uses the first and second moments of the gradient to update the parameters that helps in overcoming the limitation of traditional gradient descent methods. The benefits of adam optimization is that it helps in converging the model faster and to achieve better accuracy as compare to other gradient descent methods. 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 but the accuracy score that we received wasn't as good as expected.

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

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

As the accuracy score through the adam optimization was not as we expected, we again went back to tweak our previous function that we created and tried implementing some new techniques such as selu models, leaky_relu with he initialization. However, executing those techniques didn't improved our model and the accuracy score.

The Final Model

We proceeded with going forward with he initialization, performing relu and softmax activation function and lambda regularization parameter.

```
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 which made our accuracy score to be at **0.98233**. Regularization in the model was primarily used to avoid overfitting and enhance the model performance. In the update stage, the regularization is applied to the neural network's weight. The cost function gains a penalty term as a result of regularization, allowing for training-induced cost function optimization.

Note: The codes has been commented as it takes a lot of time to process the code. So, I have included the outline of how I performed execution of each activation function and the code.

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)
```

Conclusion

This project has been a great learning opportunity for me from understanding the concpets and algorithms to implementing the algorithm from scratch. Although, this project was very challenging, I'm happy with the progress I made from this project as this project provided me a great opportunity to gain an in-depth knowledge about forward and backpropagation, adam optimization, updating weights and bias, and neural network as a whole.

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