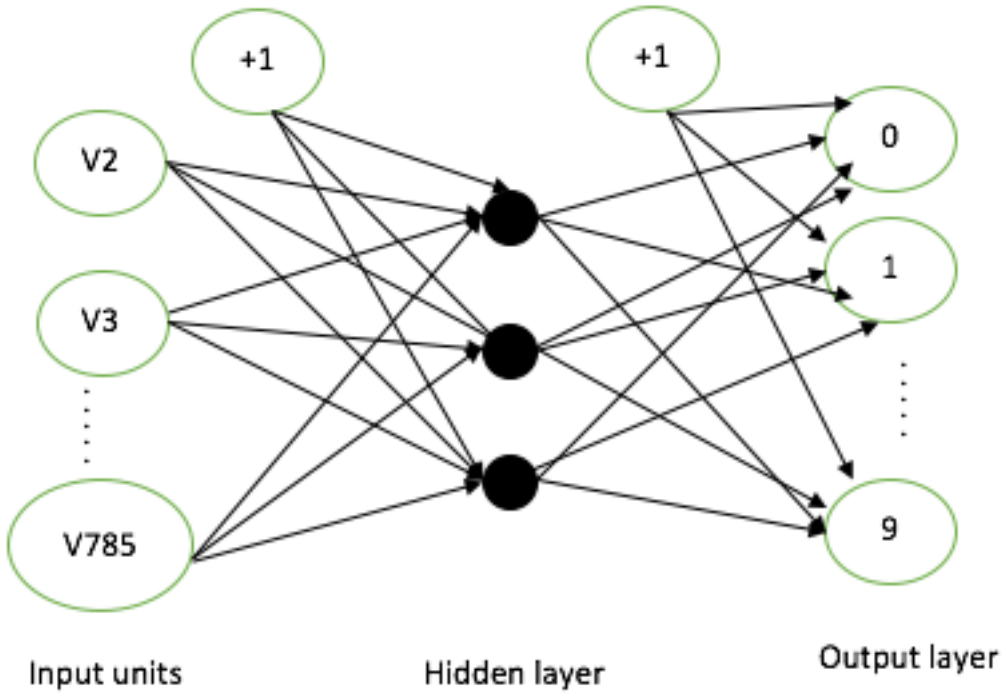
CSI 873/MATH 689
Midterm
Ajay Kulkarni (G01024139)

Artificial Neural Network implementation in R

Abstract: The neural network in R is implemented having one hidden layer with two, three and four hidden units for classifying handwritten digits. For training the neural network, we used first 500 rows from every input file. The same training data is then used for preparing validation data set. We have implemented neural network using Forward propagation as well as Backpropagation. The output from the Forward propagation is used in a Cost function for calculating the cost and then Backpropagation used for finding gradient. After implementing this, we have used an advanced optimizer "L-BFGS-B" for optimizing the weights which then further used for predicting validation as well test data sets.

Introduction

The neural network with one hidden layer and two, three and four hidden units were implemented in R. The data consists of 785 columns, and the first column represents the class labels. So the neural network will have 784 input units and 10 output units. The reason behind 10 output units is that there are 10 classes from 0 to 9. The artificial neural network with one hidden layer and 3 hidden units is shown below.



In the given model +1 is a bias, which is added in every unit. The network has two layers so there will be two matrices for weights w1 and w2. Similarly, we can also build models for two hidden units and four hidden units.

This report is divided into four parts. The first part explains the data preprocessing, second part explains about the implementation of Neural Network. The third part will be results as well as discussion of results, and the fourth part contains the implemented R code.

Part -1: Data Preprocessing

Data preprocessing is performed in R by extracting first 500 records from training files and test files. The steps for preprocessing of the training data are as follows,

1) First 500 rows from every training data file are extracted and stored them into different variables.

2) Every variable is converted into a data frame, and all the data frames are concatenated. The concatenated data is then stored in a variable called training_data and the dimension of training_data is 5000 x 785.

3) Scaling is performed on the concatenated data by dividing maximum value from each column to every value in the same column. This result is again stored in variable training_data. The results generated from scaling was in between 0 and 1.

4) After processing data using all the above steps, we stored it in one variable and then checked for NaN values. After that, we saved the data into a train.csv file, which will be the training data for our model.

Similar steps are executed for test data, and then test data is stored in csv file named test.csv. Thus, from data preprocessing we processed the original data and made training as well as test data available for the implementing Artificial Neural Network. The data for cross-validation set is prepared from training set by extracting first 100 records for every digit. So the size of the cross-validation set is 1000 x 785. Also, we have made a small adjustment by replacing class label ‘0’ with class label ‘10’ to make the calculations work better

Part 1: Artificial Neural Network implementation

The implementation of Artificial Neural Network model is divided into four parts, and the details about that are given on the next page,

1) Parameter initialization and weights allocation

In this part we initialized the basic parameters like a number of input units (784), a number of output units (10) and a number of hidden units (2, 3 or 4). Also, the total number of connections are dependent on bias unit, number of input units, number of hidden units and number of output units. So, we have defined a parameter 'n' which represents a total number of connections in the neural network.

$$n = \big((number\ of\ input\ units + 1) * number\ of\ hidden\ units\big) + ((number\ of\ hidden\ units + 1) * number\ of\ output\ units)$$

From input layer to hidden layer there will be $\big((number\ of\ input\ units + 1) * number\ of\ hidden\ units\big)$ number of connections while from hidden layer to output layer number of connections will be $((number\ of\ hidden\ units + 1) * number\ of\ output\ units)$. The weight for every connection is generated using 'n' normally distributed random numbers and then stored in variable 'wt'.

2) Forward propagation and cost function

In this section, we have implemented Forward propagation, and the activation function which we have used is the sigmoid function. The Forward propagation algorithm which we have implemented is given below,

1) $Add\ bias\ {a_0}^{(1)} = 1$
2) $Z^{(2)} = w^1 a^{(1)}$
3) $a^{(2)} = g(z^{(2)})$
4) $Add\ bias\ {a_0}^{(2)} = 1$
5) $Z^{(3)} = w^2 a^{(2)}$
6) $a^{(3)} = g(Z^{(3)})$

In the above algorithm w^1 and w^2 are the weight matrices, ${a_0}^{(1)}$ and ${a_0}^{(2)}$ are the bias units which are added as input in every layer, $Z^{(2)}$ is the raw output from the hidden layer and $a^{(2)}$ is the final output from the hidden layer after implementing sigmoid function $g(z)$. Similarly, we performed the same calculations from hidden layer to output layer and then we stored our final output. The output which we calculated using Forward propagation is then used for finding the cost. The regularized cost function which we have used is given below,

$$J(w) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{k=1}^{K} {y_k}^{(i)} \log\left(h_w\left(x^{(i)}\right)\right)_k + \left(1 - {y_k}^{(i)}\right)\log\left(1 - \log\left(h_w\left(x^{(i)}\right)\right)_k\right]\right.$$
$$+ \frac{\lambda}{2n}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}({w_{ji}}^{(l)})^2$$

In the above regularized cost function n represents the number of rows present in the training data, K is the number of output units, L is the total number of layers in the network and s_l shows the number of units i the layer l.

3) Backpropagation implementation and calculation of partial derivative of cost function

The purpose of the Backpropagation algorithm is to find the error ${\delta_j}^{(l)}$ which represents an error of node j in layer l. It will help us to find out how much that node was responsible for any error in the output and based on calculated error we can modify the weights accordingly. In this part first we are implementing Forward propagation and finding outputs. After finding outputs, we are implementing the Backpropagation algorithm using following two formulas,

$$\delta^{(3)} = {a_j}^{(3)} - y_j$$

where ${a_j}^{(3)}$ is the final output obtained from the backpropagation and y_j is the actual output.

$$\delta^{(2)} = (w^{(2)})^T \delta^{(3)} * g'(z^{(2)})$$

Here, $g'(z^{(2)})$ is the implementation of the gradient of the sigmoid function on the output generated from the hidden layer. So using Backpropagation, we are first calculating the gradient of an unregularized neural network and then we are implementing the gradient for the regularized network. For calculating gradient first we are accumulating the gradient and then we will obtain the gradient for the neural network cost function by dividing accumulated gradient by $\frac{1}{n}$.

4) Training the model and predicting the results

We are ready with a function to minimize and a function to return the gradient. Now, our next step is to use the optimization method which will give us the optimized weights. In R there is one function named “optim” which is a general purpose optimizer. So, we are using the “optim” function with “L-BFGS-B” method and will pass both the functions along with the training data for execution. The reason behind choosing the method “L-BFGS-B” and number of iterations will discuss in the result section.

After executing this function, we will select the first parameter from the output which will give us the optimized weight. Further, these optimized weights we are using first on cross-validation data set and then on test data set. To find the predicted result, we have built a predict function which implements Forward propagation on the data using optimized weights.

Part 3: Results and conclusion

In our implementation, there are three parameters which we have set for getting better results. These three parameters are methods of optimization, a number of iterations and value for λ. Now, we will discuss how we decided to set the values to those parameters and then we will discuss the results for Artificial Neural Network with two, three and four hidden units.

1) Selection of parameters

All the analysis is performed for the Artificial Neural Network having one hidden layer and three hidden units. For Artificial Neural Network with two and four hidden units, we only have selected best value for λ.

- **Method of optimization**

R function 'optim' provides 6 different methods for optimization. These 6 methods are – Nelder-Mead, BFGS, CG, L-BFGS-B, SANN and Brent. To select the best method for optimization we have used all the methods for 50 iterations and for $\lambda = 1$. The percentage accuracy on validation set is calculated and given below,

Method	Percentage accuracy
Nelder-Mead	10%
BFGS	27.4%
CG	10%
L-BFGS-B	**44.7%**
SANN	10%
Brent	10%

It can be observed from the above table that "L-BFGS-B" method is giving maximum accuracy. Thus, we decided to use "L-BFGS-B" method for the optimization.

- **Number of iterations**

After selecting the optimization method, our next task was to select the number of iterations which will give us the maximum accuracy. So, we used "L-BFGS-B" method for a different number of iterations with $\lambda = 1$. The percentage accuracy for the validation set and number of iterations are shown below,

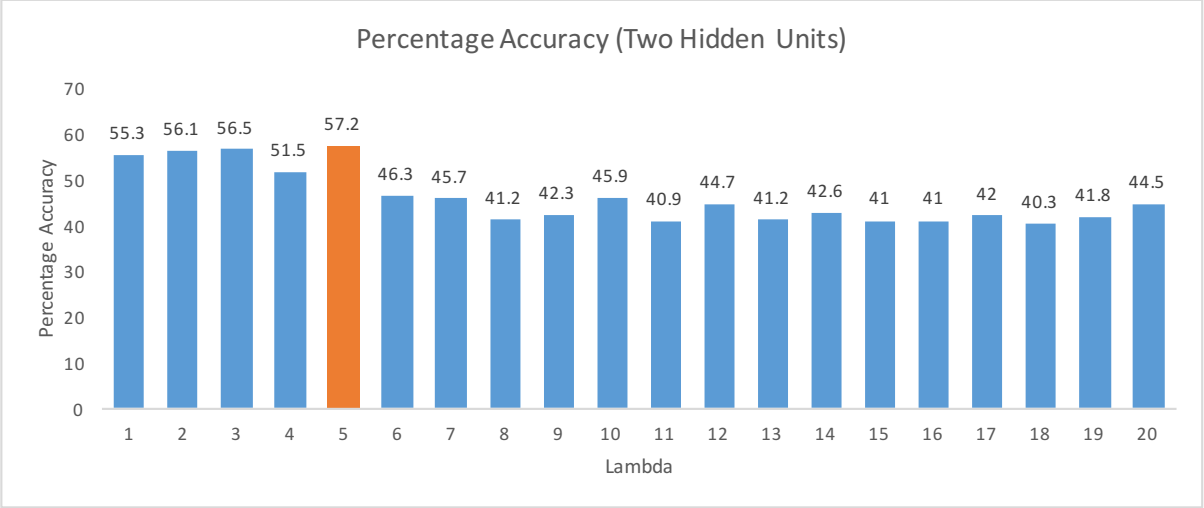
Number of iterations	Percentage accuracy
50	44.7%
100	49.4%
150	49.5%
200	53.6%
300	64.7%
400	73.1%
500	**73.6%**
600	72.9%

So, from the above table, it can observe that as the number of iterations is increasing the percentage accuracy for the validation set is also increasing until 500 iterations. Also, the maximum accuracy is found for the 500 iterations which is 73.6%. So we decided to use 500 number of iterations for our analysis.

- **Selecting value for λ**

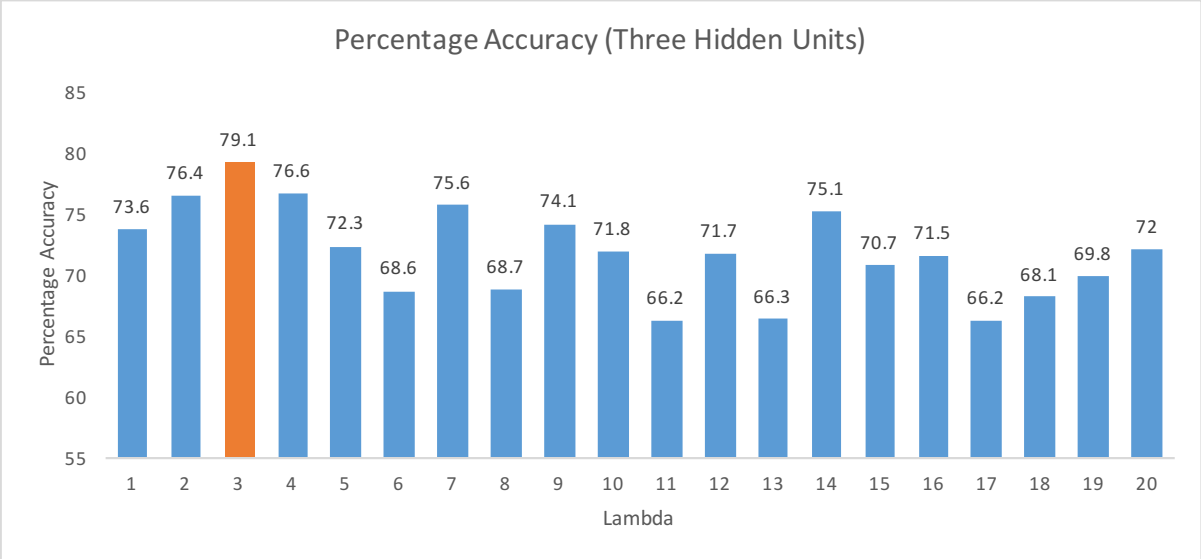
We have calculated percentage accuracy for different values of λ on validation set for "L-BFGS-B" method and 500 iterations. The values which we consider for λ are from 1 to 20. The results for different models are given below,

- **Artificial Neural Network with one hidden layer and two hidden units**



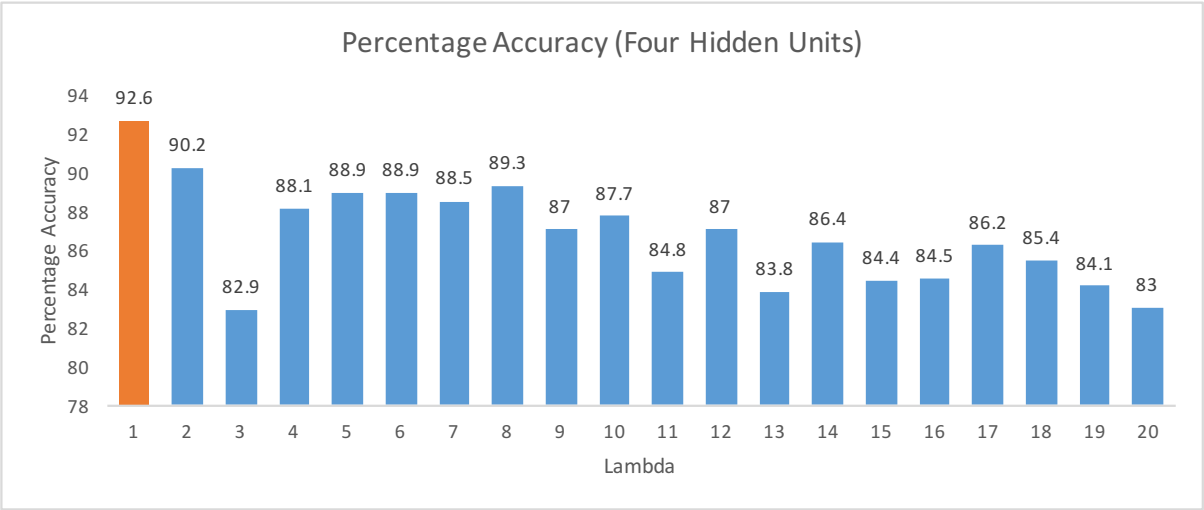
It can be observed from the plot that for $\lambda = 5$ we are getting maximum percentage accuracy for the validation set. So for the Artificial Neural Network with Two hidden units we will be using $\lambda = 5$.

- **Artificial Neural Network with one hidden layer and three hidden units**



It can be observed from the above plot that for $\lambda = 3$ we are getting maximum percentage accuracy for the validation set. So for the Artificial Neural Network with Three hidden units we will be using $\lambda = 3$.

- **Artificial Neural Network with one hidden layer and four hidden units**



It can be observed from the above plot that for $\lambda = 1$ we are getting maximum percentage accuracy for the validation set. So for the Artificial Neural Network with Four hidden units we will be using $\lambda = 1$.

2) Results for Test data

- ANN with one hidden layer and two hidden units

We have selected $\lambda = 5$ for ANN with two hidden units. In addition to that, we have used "L-BFGS-B" optimization method with 500 iterations. The confusion matrix for the test data is given below,

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	**418**	0	13	8	4	305	68	0	98	1
1	0	**383**	26	25	2	5	3	44	18	7
2	1	8	**30**	28	0	13	1	0	38	2
3	5	21	343	**358**	0	21	0	16	124	6
4	1	0	3	1	**255**	6	26	14	8	79
5	28	0	19	18	1	**59**	10	0	63	1
6	26	2	5	0	132	39	**376**	6	10	13
7	0	79	18	13	42	3	2	**392**	6	214
8	21	2	37	48	1	45	4	0	**128**	11
9	0	5	6	1	63	4	10	28	7	**166**

It can be observed from the confusion matrix that digit '0' shows the maximum number of correct classified labels and the digit '5' shows the least number of correct classified labels. Also, we have observed that digit '5' is misclassified as digit '0' in 305 data records. **The overall accuracy** calculated for the model with two hidden units is **51.3%.** A total number of **misclassified labels is 2435,** and from that, we calculated upper and lower bound of 95% confidence interval. **The upper bound calculated as 0.5302956 and lower bound calculated as 0.4437044**.

- ANN with one hidden layer and three hidden units

We have selected $\lambda = 3$ for ANN with three hidden units. In addition to that, we have used "L-BFGS-B" optimization method with 500 iterations. The confusion matrix for the test data is given below,

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	**436**	0	37	37	0	55	34	0	23	2
1	0	**471**	23	67	1	17	2	18	109	14
2	18	4	**171**	100	2	11	4	13	7	2
3	4	8	207	**241**	0	17	1	20	18	7
4	0	0	6	0	**394**	27	20	11	12	49

5	12	2	2	19	2	**158**	23	0	59	3
6	28	0	7	1	17	9	**405**	6	1	2
7	1	2	34	21	7	8	5	**389**	5	24
8	1	9	8	10	1	192	5	0	**256**	9
9	0	4	5	4	76	6	1	43	10	**388**

It can be observed from the confusion matrix that digit '0' shows the maximum number of correct classified labels and the digit '5' shows the least number of correct classified labels. Also, we have observed that digit '5' is misclassified as digit '8' in 192 data records. **The overall accuracy** calculated for the model with three hidden units is **66.18%**. A total number of **misclassified labels is 1691,** and from that, we calculated upper and lower bound of 95% confidence interval. **The upper bound calculated as 0.365577 and lower bound calculated as 0.310823**.

- ANN with one hidden layer and four hidden units

We have selected $\lambda = 1$ for ANN with four hidden units. In addition to that, we have used "L-BFGS-B" optimization method with 500 iterations. The confusion matrix for the test data is given below,

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	**411**	0	5	1	16	73	46	1	55	3
1	0	**470**	16	0	8	2	12	24	3	2
2	1	16	**383**	18	0	2	18	7	25	0
3	12	3	36	**374**	0	59	0	19	56	9
4	3	2	3	0	**363**	14	10	18	5	52
5	33	0	7	59	2	**311**	18	2	13	3
6	18	2	24	1	14	8	**396**	0	7	2
7	0	5	5	25	15	14	0	**381**	10	46
8	22	0	19	18	4	14	0	3	**306**	12
9	0	2	2	4	78	3	0	45	20	**371**

It can be observed from the confusion matrix that digit '1' shows the maximum number of correct classified labels and the digit '8' shows the least number of correct classified labels. **The overall accuracy** calculated for the model with four hidden units is **75.32%**. A total number of **misclassified labels is 1234,** and from that, we calculated upper and lower bound of 95% confidence interval. **The upper bound calculated as 0.2655317 and lower bound calculated as 0.2280683.**

Thus, from the implementation and analysis of results, we can say that as the number of hidden units increases percentage accuracy also increases. Also, it is very important to select the proper number of iterations, value for λ and the method for performing optimization. We found that one hidden layer with four hidden units have percentage accuracy 75.32% which is greater as compared to any other models which we built. We also observed that for ANN with 2 and 3 hidden units have maximum classification accuracy for digit '0' and for digit '5' least classification accuracy is observed.

Part 4: R code

In this section, we have included the R code for the artificial neural network with one hidden layer and 3 hidden units. This code can be easily modified for 2 and 4 hidden units by changing the parameter “hidden_layer_size”. The first script is the data preprocessing script and second script is the implementation of the neural network.

- **Data preprocessing script**

```
# Data Preprocessing for Training data
t1 = read.table("train0.txt")
t11 = t1[1:500,]
dim(t11)

t2 = read.table("train1.txt")
t12 = t2[1:500,]
dim(t12)



t3 = read.table("train2.txt")
t13 = t3[1:500,]
dim(t13)

t4 = read.table("train3.txt")
t14 = t4[1:500,]
dim(t14)

t5 = read.table("train4.txt")
t15 = t5[1:500,]
dim(t15)

t6 = read.table("train5.txt")
t16 = t6[1:500,]
dim(t16)

t7 = read.table("train6.txt")
t17 = t7[1:500,]
dim(t17)

t8 = read.table("train7.txt")
t18 = t8[1:500,]
```

```
dim(t18)

t9 = read.table("train8.txt")
t19 = t9[1:500,]
dim(t17)

t10 = read.table("train9.txt")
t10 = t10[1:500,]
dim(t10)

# Converting data into dataframe
t11 = data.frame(t11)
t12 = data.frame(t12)
t13 = data.frame(t13)
t14 = data.frame(t14)
t15 = data.frame(t15)
t16 = data.frame(t16)
t17 = data.frame(t17)
t18 = data.frame(t18)
t19 = data.frame(t19)
t10 = data.frame(t10)

# Combining data into variable training_data
training_data = rbind(t11,t12,t13,t14,t15,t16,t17,t18,t19,t10)
d = dim(training_data)

# Scaling
for (i in 2:785) {
  k = max(training_data[,i])
  if(k!=0) {
    training_data[,i] = training_data[,i]/k
  }
}

# Checking NaN values
sum(is.na(training_data))

# Storing training data in csv file
write.csv(training_data, file = "train_data.csv", row.names = FALSE)

# Data Preprocessing for Test data
```

```
tr1 = read.table("test0.txt")
t21 = tr1[1:500,]
dim(t21)

tr2 = read.table("test1.txt")
t22 = tr2[1:500,]
dim(t22)

tr3 = read.table("test2.txt")
t23 = tr3[1:500,]
dim(t23)

tr4 = read.table("test3.txt")
t24 = tr4[1:500,]
dim(t24)

tr5 = read.table("test4.txt")
t25 = tr5[1:500,]
dim(t25)

tr6 = read.table("test5.txt")
t26 = tr6[1:500,]
dim(t26)

tr7 = read.table("test6.txt")
t27 = tr7[1:500,]
dim(t27)

tr8 = read.table("test7.txt")
t28 = tr8[1:500,]
dim(t28)

tr9 = read.table("test8.txt")
t29 = tr9[1:500,]
dim(t29)

tr10 = read.table("test9.txt")
t20 = tr10[1:500,]
dim(t20)

# Converting data into dataframe
```

```
t21 = data.frame(t21)
t22 = data.frame(t22)
t23 = data.frame(t23)
t24 = data.frame(t24)
t25 = data.frame(t25)
t26 = data.frame(t26)
t27 = data.frame(t27)
t28 = data.frame(t28)
t29 = data.frame(t29)
t20 = data.frame(t20)

# Combining data into variable test_data
test_data = rbind(t21,t22,t23,t24,t25,t26,t27,t28,t29,t20)
d = dim(test_data)

# Scaling
for (i in 2:785) {
  k = max(test_data[,i])
  if(k!=0) {
    test_data[,i] = test_data[,i]/k
  }
}

# Checking NaN values
sum(is.na(test_data))

# Storing test data in csv file
write.csv(test_data, file = "test_data.csv", row.names = FALSE)
```

- R code for neural network implementation

```
# Implementation of Artificial Neural Network in R for 3 hidden units
# ------------------------------------------------------------------------------

# Importing training data
train_data = read.csv("train_data.csv", check.names = FALSE)

# Check structure and view the data
str(train_data)
View(train_data)
# ------------------------------------------------------------------------------

# ------  Data preparation for training and validation sets ------ #
# Training data without labels
x_train= train_data[-1]
x_train = as.matrix(x_train)
dim(x_train)

# Training data labels
y_train = train_data[,1]
y_train = as.numeric(y_train)
# Replacing label 0 by 10 for calculation
y_train[y_train == 0] = 10

# Validation data
v1 = train_data[1:100,]
v2 = train_data[501:600,]
v3 = train_data[1001:1100,]
v4 = train_data[1501:1600,]
v5 = train_data[2001:2100,]
v6 = train_data[2501:2600,]
v7 = train_data[3001:3100,]
v8 = train_data[3501:3600,]
v9 = train_data[4001:4100,]
v10 = train_data[4501:4600,]

val_data = rbind(v1, v2, v3, v4, v5, v6, v7, v8, v9, v10)
dim(val_data)
```

```
# Validation data without labels
x_val= val_data[-1]
x_val = as.matrix(x_val)
dim(x_val)

# Validation data labels
y_val = val_data[,1]
y_val = as.numeric(y_val)
# Replacing label 0 by 10 for calculation
y_val[y_val == 0] = 10
# ---------------------------------------------------------------------------

# ------ Parameter initialization ------ #
# Total number of input layers.
input_layer_size = dim(x_train)[2]
# In this case total number of inout layers will be 784.

# Total number of output layers
output_layer_size = length(unique(y_train))
# This will be 10 becasue we have 10 classes

# Number of neurons present in the hidden layer
hidden_layer_size = 3

# Generating normally distributed random numbers which will be used as weights.
# Size will depend upon input layer size, bias, and hidden layer size
# So, we will define a variable of the size
# ((input_layer_size + 1) * hidden_layer_size) + ((hidden_layer_size + 1) * output_layer_size)
n = ((input_layer_size + 1) * hidden_layer_size) +
  ((hidden_layer_size + 1) * output_layer_size)

# Setting seed so it will generate same weights in every iteration
set.seed(1)
wt = runif(n)

# ---------------------------------------------------------------------------

# ------ Activation function -> sigmoid function ------ #
sigmoid <- function(x) {1 / (1 + exp(-x))}

# ---------------------------------------------------------------------------
```

```
# ------ Cost function ------ #
cost_fun = function(wt, input_layer_size, hidden_layer_size, output_layer_size, x, y,
lambda) {
  # Defining the weight matrix for the neural network. The first layer will have
  # inputs from all the input nodes and bias. The second layer will have inputs only
  # from hidden units and bias. Here w1 will represent weights for layer 1 and
  # w2 will represent weights for layer 2.
  k = (input_layer_size + 1) * hidden_layer_size
  w1 = matrix(wt[1:k], hidden_layer_size, input_layer_size + 1)
  w2 = matrix(wt[(k + 1):length(wt)],output_layer_size, hidden_layer_size + 1)

  # Total number of rows in data
  n = dim(x)[1]

  # ---- Forward Propagation Algorithm ---- #
  # Adding bias to each example
  b1 = cbind(rep(1,n), x)
  # Multiplying weights and the inputs
  o1 = b1 %*% t(w1)
  # Applying activation function on the calculated raw output
  o1 = sigmoid(o1)
  # Adding bias unit in the input for the next layer
  b2 = cbind(rep(1,n), o1)
  # Multiplying weigths and the inputs
  o2 = b2 %*% t(w2)
  # Applying activation function on the calculated raw output
  final_output = sigmoid(o2)

  # Defining a diagonal matrix of the size of output layer
  diag_matrix = diag(output_layer_size)

  # Implementing cost function to calculate cost and adding regularization term for weights
  cost1 = ((-1/n) * sum (log(final_output) * t(diag_matrix[,y]) +
                    log(1-final_output) * t((1- diag_matrix[,y]))))
  cost = cost1 + lambda/(2*n) * (sum(w1[, -c(1)]^2)
                          + sum(w2[, -c(1)]^2))
  cost
}
```

```
# t1 = cost_fun(wt, input_layer_size, hidden_layer_size, output_layer_size, x_train, y_train,
1)
# 19.87897

# ---------------------------------------------------------------------------

# ------ Derivative of sigmoid function ------ #

siggrad =  function(z) {
  temp <- 1 / (1 + exp(-z))
  temp * (1 - temp)
}

# ------ Partial derivatives of the cost function with respect to w1 and w2 ------ #

p_grad = function(wt, input_layer_size, hidden_layer_size, output_layer_size,
                  x, y, lambda) {
  # Initializing weights for layer1 and layer 2
  k <- (input_layer_size + 1) * hidden_layer_size
  w1 <- matrix(wt[1:k], hidden_layer_size, input_layer_size + 1)
  w2 <- matrix(wt[(k + 1):length(wt)], output_layer_size, hidden_layer_size + 1)

  # Total number of rows in data
  n = dim(x)[1]

  # ---- Forward Propagation Algorithm ---- #
  # Adding bias to each example
  b1 = cbind(rep(1,n), x)
  # Multiplying weights and the inputs
  o1 = b1 %*% t(w1)
  # Applying activation function on the calculated raw output
  o1_1 = sigmoid(o1)
  # Adding bias unit in the input for the next layer
  b2 = cbind(rep(1,n), o1_1)
  # Multiplying weights and the inputs
  o2 = b2 %*% t(w2)
  # Applying activation function on the calculated raw output
  final_output = sigmoid(o2)
  # Defining a diagonal matrix of the size of output layer
  diag_matrix = diag(output_layer_size)
```

```
 # ----- Backpropagation Algorithm ----- #
 # Matrix of actual output
 y_matrix = diag_matrix[y,]
 # Calculating error in output layer by subtracting actual output and
 # predicted output
 d3 = final_output - y_matrix
 # Calculating error in the second layer
 d2 <- (d3 %*% w2[, -c(1)]) * siggrad(o1)
 # Calculating delta values for each layer
 delta1 = t(d2) %*% b1
 delta2 = t(d3) %*% b2

 # ----- Gradient regularization ----- #
 # Calculating regularization terms
 reg1 = lambda/n * w1
 reg1[, 1] = 0
 reg2 = lambda/n * w2
 reg2[,1] = 0
 # Adding calculated regularization terms into delta matrices
 w1_grad = 1/n * delta1 + reg1
 w2_grad = 1/n * delta2 + reg2
 # Calculated gradient
 grad = c(as.vector(w1_grad), as.vector(w2_grad))
 grad
}

#kk = p_grad(wt, input_layer_size, hidden_layer_size, output_layer_size,
#  x_train, y_train, 1)

# ------------------------------------------------------------------------

# ------ Predict function ------ #

predict <- function(w1, w2, x) {

 # Number of rows present in the test data
 n <- dim(x)[1]

 # Adding bias
 p1_1 = cbind(rep(1,n),x)
 # Multiplying inputs with calculated weights
```

```
  p1_2 = p1_1 %*% t(w1)
  # Applying sigmoid function on the output
  p1 = sigmoid(p1_2)

  # Adding bias
  p2_1 = cbind(rep(1,n),p1)
  # Multiplying inputs with calculated weights
  p2_2 = p2_1 %*% t(w2)
  # Applying sigmoid function on the output
  p2 = sigmoid(p2_2)
  # Returning the columns having maximum predicted output
  max.col((p2))
}

# ------------------------------------------------------------------------

# ------- Model execution ------- #

# Intialzing value for lambda
lambda = 1
# Optimizing cost function and calculating optimum weights for
# neural network. Options for optimization - "Nelder-Mead", "BFGS",
# "CG", "L-BFGS-B", "SANN","Brent".
op_out <- optim(wt,
            function(p) cost_fun(p,
                        input_layer_size,
                        hidden_layer_size,
                        output_layer_size,
                        x_train,
                        y_train,
                        lambda),
            function(p) p_grad(p,
                        input_layer_size,
                        hidden_layer_size,
                        output_layer_size,
                        x_train,
                        y_train,
                        lambda),
            method = "L-BFGS-B",
            control = list(maxit = 500))
```

```
# Storing optimized weights in new variable which will be used
# for prediction
new_weight <- op_out[[1]]

# Updating weight matrices which are calculated from training model
k = (input_layer_size + 1) * hidden_layer_size
w1 = matrix(new_weight[1:k], hidden_layer_size, input_layer_size + 1)
w2 = matrix(new_weight[(k + 1):length(new_weight)], output_layer_size, hidden_layer_size
+ 1)

# ------------------------------------------------------------------------

# -------- Predictions on validation set -------- #

# For lambda = 1
# Call to prediction function
preds = predict(w1, w2, x_val)
# Confusion matrix and percentage accuracy
table(preds, y_val)
sum(preds == y_val) * 100 / dim(x_val)[1]
# Accuracy = 73.6%

# Selecting value of lambda
acc = 0
for (i in seq(1,20,1)){
  lambda = i
  opt_out <- optim(weight,
                   function(p) cost_fun(p, input_layer_size,
                                        hidden_layer_size,
                                        output_layer_size,
                                        x_train, y_train, lambda),
                   function(p) p_grad(p, input_layer_size,
                                      hidden_layer_size,
                                      output_layer_size,
                                      x_train, y_train, lambda),
                   method = "L-BFGS-B", control = list(maxit = 500))
  # Storing optimized weights in new variable which will be used
  # for prediction
  new_weight <- opt_out[[1]]
```

```
  # Updating weight matrices which are calculated from training model
  k = (input_layer_size + 1) * hidden_layer_size
  w1 = matrix(new_weight[1:k], hidden_layer_size, input_layer_size + 1)
  w2 = matrix(new_weight[(k + 1):length(new_weight)], output_layer_size,
hidden_layer_size + 1)
  preds = predict(w1, w2, x_val)
  acc[i] = sum(preds == y_val) * 100 / dim(x_val)[1]
}
acc_LBFGS_B = acc
write.csv(acc_LBFGS_B, "acc_LBFGS_B.csv")

# ------------------------------------------------------------------------

# Calculating accuracy for lambda = 3
lambda = 3
op_out <- optim(wt,
            function(p) cost_fun(p,
                         input_layer_size,
                         hidden_layer_size,
                         output_layer_size,
                         x_train,
                         y_train,
                         lambda),
            function(p) p_grad(p,
                        input_layer_size,
                        hidden_layer_size,
                        output_layer_size,
                        x_train,
                        y_train,
                        lambda),
            method = "L-BFGS-B",
            control = list(maxit = 500))
# Storing optimized weights in new variable which will be used
# for prediction
new_weight <- op_out[[1]]

# Updating weight matrices which are calculated from training model
k = (input_layer_size + 1) * hidden_layer_size
w1 = matrix(new_weight[1:k], hidden_layer_size, input_layer_size + 1)
w2 = matrix(new_weight[(k + 1):length(new_weight)], output_layer_size, hidden_layer_size
+ 1)
```

```
preds = predict(w1, w2, x_val)
preds[preds == 10] = 0
y_val[y_val == 10] = 0
table(preds, y_val)
sum(preds == y_val) * 100 / dim(x_val)[1]
# Accuracy = 79.1%

# ------------------------------------------------------------------------

# ------ Testing model on test data ------ #

test_data = read.csv("test_data.csv", check.names = FALSE)

# Training data
x_test= test_data[-1]
x_test = as.matrix(x_test)
dim(x_test)

# Training data labels
y_test = test_data[,1]
y_test = as.numeric(y_test)
y_test[y_test == 0] = 10

preds1 <- predict(w1, w2, x_test)
preds1[preds1 == 10] = 0
y_test[y_test == 10] = 0
table(preds1, y_test)
sum(preds1 == y_train) * 100 / dim(x_test)[1]
# Accuracy = 66.18%

# ------------------------------------------------------------------------

# ----- Confidence Interval  ----- #

# Total misclassified labels
miss_label = length(y_test) - sum(preds1 == y_test)
miss_label
# 1691

m = miss_label/length(y_test)
m
```

```
#  0.3382

v = length(y_test)*m*(1-m)
v
# 1119.104

v1 = sqrt(v)
v1
# 33.45301

s = v1/n
s
# 0.01396785

# Now we need to find 95% confidence interval
u = m + 1.96*s
l = m - 1.96*s
u
# 0.365577
l
# 0.310823
```

References

1) Machine Learning online course, Coursera, Andrew Ng
2) Building a Neural Network from scratch in R, Bath Machine Learning Meetup, Owen Jones
3) R for Deep Learning (I): Build fully connected Neural Network from scratch, Peng Zhao
4) A visual and interactive guide to the basics of Neural Networks, J Alammar
5) https://www.rdocumentation.org/packages/stats/versions/3.4.1/topics/optim
6) Machine Learning, Tom Mitchell