Classifying Images in MNIST Data

Kaustav Khatua

Roll No. 181075

In my project work I have tried to classify the images in MNIST dataset by building a Neural Network from scratch in R. I have used only the functions available in base R and have written codes for all the algorithms on my own. Using this network I am able to correctly classify more than 90% of the test cases.

Abot MNIST Dataset

MNIST dataset comes in two parts; training data and test data. For both the dataset independent variable is an image of a hand-written digit (0 - 9) and dependent variable is the correct label of the image. Training data consists 60000 and test data consists 10000 such data points.

Regressor is an image of 28 X 28 pixels, not a numeric vector and we can not fit model without nmbers. So, we have to convert this image into numbers. The solution is that every image is in black and white; so, each pixel of the image has a certain grey-scale value (a number between 0 - 255, denoting how Black the pixel is. '0' means White, 255 means fully Black and value between 0 and 255 means Grey). So, we can describe an image using a 28 X 28 matrix whose (i,j)th entry is the grey-scale value of (i,j)th pixel of the image. Now dropping the row structure of the matrix we will get a 28 X 28 = 784 dimensional vector. We will train our Neural Network using such 60000 vector from training data. We can skip this step as the data already available in CSV format here (obtained from original data applying the previous method).

How Neural Networks Work

Neural Network is build using many concepts but the main parameters which define a network - is its weights and biases. Problem is that we do not know any of the parameter value in advance, we have to determine these. The way we do that is, we define a cost function and choose those parameter values for which the value of the cost function is minimum.

Structure of the Network

I have used a feed forward network, ie. output from a layer is set as the input of the next layer. The input (regressor) is 784 dimensional, so input layer contains 784 neurons. We have to classify an image in one among the 10 digits, so the output layer contains 10 neurons. The first neuron in output layer represents 0, second neuron represents 1, third neuron 2,.., tenth neuron represents 9. We say that our network is classifying an image as d if (d+1)th neuron has the highest activation value among the neurons in output layer. Number of hidden layers and number of neurons in it will be decided later.

Our cost function is of the form,

$$C = \frac{1}{2n} \sum_{i=1}^{n} ||y_i - a_i^L||^2$$

where,

n = number of training data points (in our case 60000)

a_i^L = a 10 dimensional vector output from the network corresponding to ith input, which is activation values of the neurons in output layer (last layer, L).

y_i = a 10 dimensional vector representing correct label of the ith input. If the image is of 7 then the 8th entry is 1 and other entries are 0. We have to write code to create such 10 dimensional label, in the data it is just a number.

$||\cdot||$ is l_2 norm

To calculate the minima I have used **Stochastic Gradient Descent**, which calculates gradients (derivative) by **Backpropagation** method.

Building the Network

Starting Weights and Biases

Every neuron in one layer is connected with every neuron in the previous layer. Each connection has a unique weight associated with it.So, if there are N_l and N_{l-1} neurons in layer l and layer l-1, then between these two layers there will be N_l * N_{l-1}unique weights. We can arrange these in a matrix W_l of dimension N_l by N_{l-1} where (i,j)th element is the weight associated with the connection betweeen i th neuron in lth layer and jth neuron in (l-1)th layer. Also every neuron in a layer has a bias associated with it. So in total layer l has N_l biases which can be arranged in a column vector B_l of dimension N_l. One thing may be noted that input layer does not have weights and biases.

I have started the weights and biases randomly by drawing samples from **Standard Normal distribution**.

```
Parameters <- function(size){     # size is a vector of length equal to the number of layers.
                                  # ith element in size denotes number of neurons in ith layer.
  l <- length(size)

  w <- list()
  b <- list()
  for(i in 2:l){                                  # Input layer does not have weights and biases.
    w[[i - 1]] <- matrix(rnorm(size[i] * size[i - 1]), nrow = size[i])  # Samples from N(0,1).
    b[[i - 1]] <- matrix(rnorm(size[i]), ncol = 1)
  }

  return(list("Weights" = w, "Biases" = b))
}
```

Feed Forward Method

Feed-Forward method sets activations of layer l neurons in the following way. If w_1, w_2, w_3,..., $w_{N_{l-1}}$ are the weights and b is the bias corresponding to the connections between a neuron in layer l and neurons in layer l-1, then activation value of the layer l neuron is 1 / (1 + exp(-($w_1.x_1$ + $w_2.x_2$ + $w_3.x_3$ + ... + $w_{N_{l-1}}.x_{N_{l-1}}$ + b)). We can calculate activations of all the layer l neurons, at once, by $\sigma(W_l.x + B_l)$, where xs are activation values of layer l-1 and $\sigma((x_1, x_2)) = (\sigma(x_1), \sigma(x_2))$.

We don't set the activations of a layer as $W_l.x + B_l$ as, if we do this then use of extra layer will be meaningless. The reason is as follows. If we do not use any transformation then activations of layer l will be $y = W_l.x + B_l$ (. is dot product) and the activations of the next layer will be $W_{l+1}.y + B_{l+1} = W_{l+1}.(W_l.x + B_l) + B_{l+1} = W.x + B, W = W_{l+1}.W_l, B = W_{l+1}.B_l + B_{l+1}$. So, we need a transformation. We can not use discrete transformation, as we will do derivative to determine correct weights and biases, but discrete function is not derivable. So, we have to use continuous transformation. We use sigmoid transformation as it produces values between 0 and 1, which we typically wish. Also, sigmoid function has derivative of nice form.

```
sigmoid <- function(x){
  return(1 / (1 + exp(- x)))
}

feed_forward <- function(input, w, b){    # input is a column vector, w and b are lists containing
                                          # the weight matrix and bias vector for every layer.
  l <- length(w)
  for(i in 1:l){
    z <- (w[[i]] %*% input) + b[[i]]
    input <- sigmoid(z)
  }

  return(input)
}
```

Stochastic Gradient Descent

In our case minimizing cost function by finding critical points is not easy, as we have to calculate a large Hessian matrix to ensure that the critical point is a point of minimum. So, we use Gradient Descent Method which instead of finding minima directly, iteratively updates the weights and biases until we find a **local minima**. In this method we randomly start the input variables of the target function; calculate negative derivative of the function at that point, which determines, from the point in which direction the function decreases most rapidly; in that direction we move the input variables slightly. We repeat this procedure for a large number of times.
In our case the update equations are:

$$p = p - \eta \frac{\partial C}{\partial p}$$

where, p is any parameter from the set of weights and biases and η is the step size (learning rate).

So, in Gradient Descent we have to calculate derivative many times. But, we have another problem in our case.

$$C = \frac{1}{2n} \sum_{i=1}^{n} ||y_i - a_i^L||^2 = \frac{1}{2n} \sum_{i=1}^{n} C_i \implies \frac{\partial C}{\partial p} = \frac{1}{2n} \sum_{i=1}^{n} \frac{\partial C_i}{\partial p}$$

where, p is any parameter from the set of weights and biases. Here n is large (60000), so for a partial derivative with respect to one parameter we have to calculate derivative 60000 times and we have many such parameters. So number of derivatives will be large. To solve this problem we use Stochastic Gradient Descent. This method tries to approximate $\partial C/\partial p$ by using samples from original data, ie.

$$\frac{\partial}{\partial p} \frac{1}{2n} \sum_{i=1}^{n} ||y_i - a_i^L||^2 \approx \frac{\partial}{\partial p} \frac{1}{2m} \sum_{i=1}^{m} ||y_i - a_i^L||^2$$

where $m < n$. Here we divide the whole data into various groups of same sizes (**Mini Batches**), apply Gradient Descent on every group one after another (ie. applying Gradient Descent on first Mini Batch we update parameters, the parameters are further updated using second Mini Batch,... so on). This procedure is repeated for a number of times(**Epochs**).

```
#
# For every epoch sgd function randomly shuffles the data, create mini batches; for every mini
# batch calls the update function. For every data point in mini batch update function calls
# backpropagation function which calculates derivative for that point. Update function keeps
# track of all the derivatives, adds them and divide the sum by size of the mini batch.
# This sum is the approximate value of the gradient we wanted.

sgd <- function(x, epochs, mini_batch_size, eta, weights, biases){
  n <- nrow(x)
  number_of_mini_batches <- ceiling(n / mini_batch_size)

  for(i in 1:epochs){

    x <- x[sample(1:n), ]          # Randomly shuffling the data.

    for(j in 1:number_of_mini_batches){

      if(((j - 1) * mini_batch_size + mini_batch_size) > n){    # Handling mini batch of small size.
        mini_batch <- x[((j - 1) * mini_batch_size + 1):n, ]
        }else{
        mini_batch <- x[((j-1) * mini_batch_size + 1):
                          ((j-1) * mini_batch_size + mini_batch_size), ]
      }

      output <- update(mini_batch, eta, weights, biases)
      weights <- output$Weights
      biases <- output$Biases
    } }
  return(list("Weights" = weights, "Biases" = biases)) }
```

```
#
# sgd function will use this update function.

update <- function(mini_batch, eta, weights, biases){
  n <- nrow(mini_batch)        # Number of data points in Mini Batch.
  l <- length(weights)         # Number of layers.

  derivative_b = list()
  derivative_w <- list()
  for(i in 1:l){
    derivative_b[[i]] <- matrix(0, nrow = nrow(biases[[i]]), ncol = ncol(biases[[i]]))
    derivative_w[[i]] <- matrix(0, nrow = nrow(weights[[i]]), ncol = ncol(weights[[i]]))
  }

  x <- mini_batch[, 2:ncol(mini_batch)]   # Getting the data matrix.
  labels <- mini_batch[, 1]               # Getting the labels.

  y <- matrix(0, nrow = 10, ncol = n)
  for(i in 1:n){
    y[labels[i] + 1, i] <- 1              # Creating the 10 dimensional label.
  }

  for(i in 1:n){
    output <- backpropagation(x[i, ], y[, i], weights, biases)    # Derivatives for one data point.
    o_b <- output$Biases
    o_w <- output$Weights

    for(j in 1:l){
      derivative_b[[j]] <- derivative_b[[j]] + o_b[[j]]      # Summing the derivatives.
      derivative_w[[j]] <- derivative_w[[j]] + o_w[[j]]
    }
  }

  for(i in 1:l){
    weights[[i]] <- weights[[i]] - ((eta / n) * derivative_w[[i]])   # Update equations of GD.
    biases[[i]] <- biases[[i]] - ((eta / n) * derivative_b[[i]])
  }

  return(list("Weights" = weights, "Biases" = biases))
}
```

Backpropagation

We have to follow systmatic approach to find derivatives, as the cost function is not a simple function of parameters distributed over different layers. The systematic approach is known as Backpropagation. Here we first obtain derivatives with respect to weights and biases of the output layer and then using these we calculate derivatives with respect to parameters of the previous layer, so on. Backpropagation equations are, Explanation is here.

$$\delta^L = \nabla_a C \circ \sigma'(z^L) \qquad \cdots (1)$$

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l) \qquad \cdots (2)$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \qquad \cdots (3)$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \cdots (4)$$

where, $\circ$ is Hadamard Product $(1, 2) \circ (3, 4) = (1 * 3, 2 * 4) = (3, 8)$, $\nabla_{(x_1,x_2)} f(x_1, x_2) = (\partial f/\partial x_1, \partial f/\partial x_2)$, L stands for last layer and l represents lth layer.

Using these four equations we calculate derivative of the cost function with respect to weights and biases of all the layers.

```
backpropagation <- function(x, y, weights, biases){
  l <- length(weights)

  derivative_b <- list()
  derivative_w <- list()
  for(i in 1:l){
    derivative_b[[i]] <- matrix(0, nrow = nrow(biases[[i]]), ncol = ncol(biases[[i]]))
    derivative_w[[i]] <- matrix(0, nrow = nrow(weights[[i]]), ncol = ncol(weights[[i]]))
  }

  x <- matrix(x, ncol = 1)
  y <- matrix(y, ncol = 1)

  activation = x
  activations = list(x)
  zs = list()

  for(i in 1:l){
    z = weights[[i]] %*% activation + biases[[i]]
    zs[[i]] <- z
    activation <- sigmoid(z)
    activations[[i + 1]] <- activation
  }

  delta <- (activation - y) * sigmoid_prime(z)
  derivative_b[[l]] <- delta
  derivative_w[[l]] <- delta %*% t(activations[[l]])

  for(i in (l - 1):1){
    z <- zs[[i]]
    sp <- sigmoid_prime(z)
    delta <- (t(weights[[i + 1]]) %*% delta) * sp
    derivative_b[[i]] <- delta
    derivative_w[[i]] <- delta %*% t(activations[[i]])
  }

  return(list("Weights" = derivative_w, "Biases" = derivative_b))
}

# Backpropagation function uses derivative of sigma function, which is calculated by the
# following function.
sigmoid_prime <- function(x){
  return(sigmoid(x) * (1 - sigmoid(x)))
}
```

Codes have been written for all the methods which are used when fitting a neural network. Now, I have to apply this network to classify images in MNIST data.

Applying the Network to Classify Images

Here we have to first determine the number of hidden layers and number of neurons in it. i got an idea and set these accordingly.

Every digit is made up with small structures. Such as, 0 is made up with four small round figures, 1 has a long bar in the middle. If we think properly, we will get more than 30 such patterns. I have used 30 neurons in the hidden layer with the hope that it will detect those patterns and set the activation of the output neurons accordingly. But one thing may be noted that it is not always possible to comprehend what the hidden layers are doing and may not work in the way we expected. Now we have the complete structure of the network. Now we can start classifying images.

```
# Reading Data
train <- read.csv("E:/Artificial Intelligence Sir/mnist_train.csv")
train <- as.matrix(train)

# A glimpse of the data.
first_image <- matrix(train[1, 2:ncol(train)], nrow = 28, byrow = TRUE)
first_label <- train[1, 1]
first_image
first_label
```

first_image

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25] [,26] [,27] [,28]
##  [1,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
##  [2,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
##  [3,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
##  [4,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
##  [5,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
##  [6,]    0    0    0    0    0    0    0    0    0     0     0     0     0     3    18    18    18   126   136   175    26   166   255   247   127     0     0     0
##  [7,]    0    0    0    0    0    0    0    0   30    36    94   154   170   253   253   253   253   253   225   172   253   242   195    64     0     0     0     0
##  [8,]    0    0    0    0    0    0    0   49   238   253   253   253   253   253   253   253   253   251    93    82    82    56    39     0     0     0     0     0
##  [9,]    0    0    0    0    0    0    0   18   219   253   253   253   253   253   198   182   247   241     0     0     0     0     0     0     0     0     0     0
## [10,]    0    0    0    0    0    0    0    0    80   156   107   253   253   205    11     0    43   154     0     0     0     0     0     0     0     0     0     0
## [11,]    0    0    0    0    0    0    0    0     0    14     1   154   253    90     0     0     0     0     0     0     0     0     0     0     0     0     0     0
## [12,]    0    0    0    0    0    0    0    0     0     0     0   139   253   190     2     0     0     0     0     0     0     0     0     0     0     0     0     0
## [13,]    0    0    0    0    0    0    0    0     0     0     0    11   190   253    70     0     0     0     0     0     0     0     0     0     0     0     0     0
## [14,]    0    0    0    0    0    0    0    0     0     0     0     0    35   241   225   160   108     1     0     0     0     0     0     0     0     0     0     0
## [15,]    0    0    0    0    0    0    0    0     0     0     0     0     0    81   240   253   253   119    25     0     0     0     0     0     0     0     0     0
## [16,]    0    0    0    0    0    0    0    0     0     0     0     0     0     0    45   186   253   253   150    27     0     0     0     0     0     0     0     0
## [17,]    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0    16    93   252   253   187     0     0     0     0     0     0     0     0
## [18,]    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0     0     0   249   253   249    64     0     0     0     0     0     0     0
## [19,]    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0    46   130   183   253   253   207     2     0     0     0     0     0     0
## [20,]    0    0    0    0    0    0    0    0     0     0     0     0     0    39   148   229   253   253   253   250   182     0     0     0     0     0     0     0
## [21,]    0    0    0    0    0    0    0    0     0     0    24   114   221   253   253   253   253   201    78     0     0     0     0     0     0     0     0     0
## [22,]    0    0    0    0    0    0    0    0    23    66   213   253   253   253   253   198    81     2     0     0     0     0     0     0     0     0     0     0
## [23,]    0    0    0    0    0    0   18   171   219   253   253   253   253   195    80     9     0     0     0     0     0     0     0     0     0     0     0     0
## [24,]    0    0    0    0   55   172   226   253   253   253   253   244   133    11     0     0     0     0     0     0     0     0     0     0     0     0     0     0
## [25,]    0    0    0    0   136   253   253   253   212   135   132    16     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
## [26,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
## [27,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
## [28,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
```

```
first_label
```

```
## label
##     5
```

Note This is not original data, original data is an image. This is the numerical representation of the image in matrix form.

```
# We see that values are between 0 - 255. But typically we need values between 0 - 1.
train[, 2:785] <- train[, 2:785] / 255


# Starting the weights and biases of the network.
parameters <- Parameters(c(784, 30, 10))
weights <- parameters$Weights
biases <- parameters$Biases

# Now our data is ready to use. We will apply sgd function on this data. eta is determined after
# variying it a number of times and observing the correct classification rate. For 2 correct
# classification rate is same as 3 but for 4 it is little less than 3.

correct_parameters <- sgd(train, epochs = 20, mini_batch_size = 30,eta =  3, weights, biases)
correct_weights <- correct_parameters$Weights
correct_biases <- correct_parameters$Biases
```

Checking Performance on Test Data

```
#
# First I have to create function which will get output for every data in test set,
# from the network and compare it with the correct label of the data.

evaluate <- function(data, weights, biases){
  n <- nrow(data)

  x <- data[, 2:ncol(data)]
  labels <- data[, 1]

  right <- 0

  for(i in 1:n){
    input <- matrix(x[i, ], ncol = 1)
    output <- feed_forward(input, weights, biases)
    o_l <- which.max(output) - 1
    if(o_l == labels[i]){
      right <- right + 1
    }
  }

  print(right)
}

# Loading test data.
test <- read.csv("E:/Artificial Intelligence Sir/mnist_test.csv")
test <- as.matrix(test)

# Scaling the test data.
test[, 2:785] <- test[, 2:785] / 255

# Getting output for all the test data.
evaluate(test, correct_weights, correct_biases)
```

```
## [1] 9411
```

So the network correctly classifies more than 90% test data correctly, which is good. Now this network's performance can be improved by implementing other concepts, but I stop here.

Way for Further Analysis

From the matrix representation of the data we can see that first and last two rows also first and last two columns are 0. Most of the images have such rows and columns of 0. So ommitting these we can reduce number of parameters to be estimated. Also we can apply PCA to reduce number of regressors (hence reduce number of parameters). Also we can apply SVM to classify images.