Classification of MNIST Handwritten Digits Dataset using Neural Networks

Harshit Mahapatra
College of Engineering and Technology,Bhubaneswar
harshitmahapatra95@gmail.com

Submitted to:
J Chandrakant Badajena
Assistant Professor
Department of Information Technology
College of Engineering and Technology
chandrakantait@cet.edu.in

September 16, 2016

Abstract

Optical character recognition is an important application of machine learning where an algorithm is trained on a labeled dataset so that it learns to classify unlabeled letters/digits. A variety of algorithms have performed very well on the problem of classification of handwritten digits. In this project we have trained a single hidden layer Artificial Neural Network on the MNIST dataset to classify handwritten digits. The parameters of the neural network were selected using model selection procedure.

This project report is divided into six sections: follows:

- In Section 1 we give a brief introduction to the problem of Optical Character Recognition and Handwritten Digit Recognition.
- In Section 2 we discuss the theory behind neural networks
- In Section 3 we give a brief summary about the dataset used in the project.
- In Section 4 we describe our training and model selection procedure for the classifier.
- In Section 5 we draw conclusions from the observations obtained during the course of the project .
- In Section 6 we present the code used in our project.

I. Introduction

Optical Character Recognition is a technology that allows us to recognize characters through an optical mechanism. For humans,the eyes act as the optical mechanism i.e. the images seen by eyes are inputs to the brain. OCR is a technology that functions like the human reading ability.

Handwritten Digit Recognition is an important application of the OCR technology. One of the most relevant ideas in the field was put forward by Y. Le Cun in the paper titled "Handwritten digit recognition: applications of neural network chips and automatic learning "[5], in the paper he described methods to perform handwritten digit recongnition. It was the followed by another study by Y LeCun B. Boser and others titled "Handwritten Digit Recognition with a Back-Propagation Network " . In

this paper the authors explored the application of back-propagation algorithm in recognition of handwritten zipcodes that appeared on real U.S. Mail[4]. It was followed by studies comparing the performances of different classifiers on the problem. Some notable comparision studies were done by Y LeCun[6] and L Bottou [2].Recent work on the topic include use of deep learning networks to improve the accuracy of classifier further.

II. Neural Networks

The human brain is a collection of more than 10 million interconnected neurons. Each neuron is a cell that uses biochemical reactions to receive, process, and transmit information.Artificial neural networks (ANN) have been developed as generalizations of mathematical models of these biological nervous systems (as described by Ajith Abraham in his article[1]). A typical artifcial neuron and the modeling of a multilayered neural network are illustrated in Figure 1. Referring to Figure 1, the signal flow from inputs $x_1, ..., x_n$ is considered to be unidirectional, which are indicated by arrows, as is a neuron's output signal flow (O). The neuron output signal O is given by the following relationship:

$$O = f(net) = f\left(\sum_{j=1}^{n} w_j x_j\right) \tag{1}$$

where w_j is the weight vector, and the function $f(net)$ is referred to as an activation (transfer) function. The variable net is deïňĄned as a scalar product of the weight and input vectors,

$$contenet = w^T x = w_1 x_1 + ... + w_n x_n \tag{2}$$

where T is the transpose of a matrix, and, in the simplest case, the output value O is computed as

$$O = f(net) = \begin{cases} 1 & \text{if } w^T x \geq \Theta \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where Θ is called the threshold level; and this type of node is called a linear threshold unit.

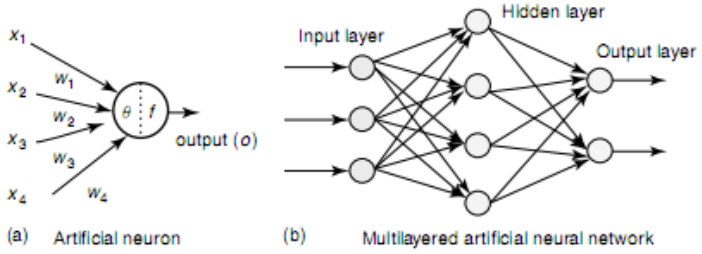

Figure 1: Architecture of an artificial neuron and a multilayered neural network.

The perceptron is a single layer neural network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron- learning rule. Perceptrons are especially suited for simple problems in pattern classiïňĄcation.

The simple perceptron is just able to handle linearly separable or linearly independent problems. If we take the negative of this derivative and then proceed to add it to the weight, the error will decrease until it reaches a local minima. This makes sense because if the derivative is positive, this tells us that the error is increasing when the weight is increasing. We then is to add a negative value to the weight and vice versa if the derivative is negative. Be-

cause the taking of these partial derivatives and then applying them to each of the weights takes place, starting from the output layer to hidden layer weights, then the hidden layer to input layer weights this algorithm has been called the backpropagation algorithm. In this project we use backpropagation algorithm to train a single hidden layer neural network using the R language[7] and the package nnet[9].

III. MNIST Dataset

In this project we have used a subset of the MNIST database of handwritten digits[11]. We obtained a CSV format of the dataset Y LeCun's from Joseph Chet Redmon's website[8]. The datset consists of 60000 training examples and 10000 test examples.Each image has dimensions 28x28 and represents a digit between 0 to 9.The digits have been size-normalized and centered in a fixed-size image. It has been made sure that the set of authors of the training set digits and that of the test set digits were disjoint. Figure 2: shows a sample of the dataset.

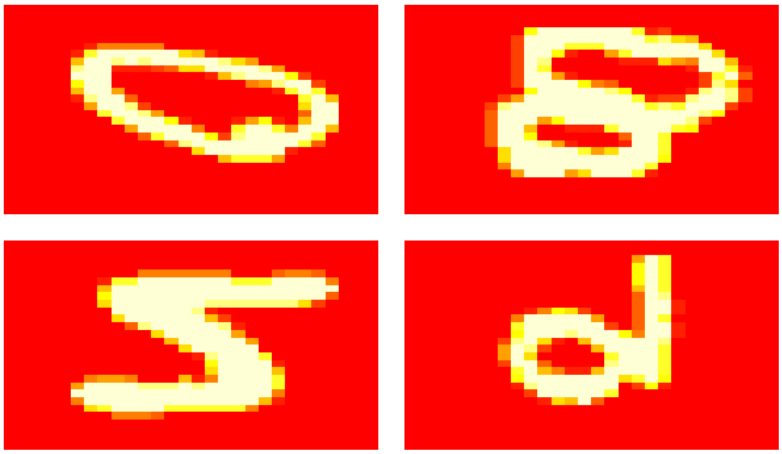

Figure 2: MNIST Dataset

IV. Training and Model Selection

The classifier of this project was trained using the nnet package of R language. The procedure followed is as follows:

1. First the given training data was divided into two sets by random sampling:
    - Training set: The set on which the classifier was trained for various values of the parameters namely the number of nodes in the hidden layer and the maximum number of iterations allowed.
    - Cross Validation Set: The set on which the accuracy of our classifier was tested.
2. The label vectors of both the sets acquired in the above step were converted to class indicator matrices. A class indicator matrix is a matrix in which each row corresponds to an observation and each column corresponds to whether the observation belongs to that class (indicated by 1) or not (indicated by 0).
3. The images were then downsized from 28x28 to 14x14 in both the training and

the cross validation set. This was done to reduce the number of features from 784 to 196.Figure 3 shows the comparision between an image belonging to the original dataset and one belonging to the 'reduced' dataset. The procedure followed for downsizing the datasets is as follows: For each observation in the dataset,

(a) The vector containing 784 pixels was extracted.

(b) Then the pixels were arranged in a 28x28 matrix.

(c) The matrix was then divided into subsection where each subsection was of dimensions 2x2.

(d) The average value of pixels of each subsection was taken and arranged in a new 14x14 matrix.

(e) Finally elements of this new matrix were unrolled to form a vector containing 196 pixels.

(f) This vector was then added as an observation to a new 'reduced' data frame.

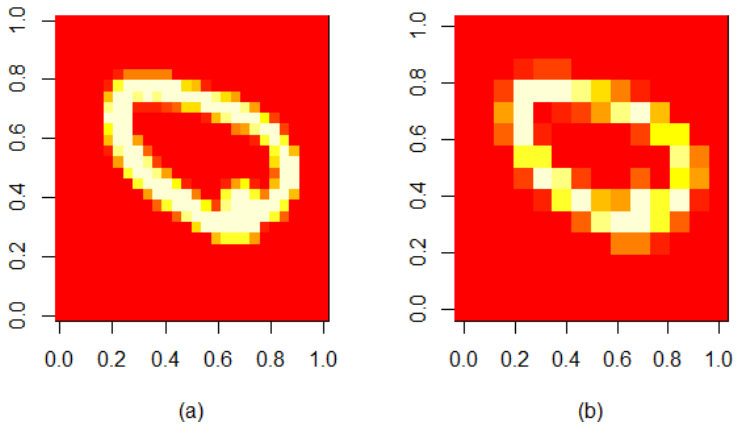

Figure 3: (a) represents an image from the original dataset and (b) represents the same image from the reduced dataset.

4. Both the sets were then feauture normalized i.e. each element in both sets was divided by the range of its column (In this case the range was found to be 255). This was done to ensure that our training algorithm converges in optimal time. the formula used for feature normalization is as follows:

$$x_i = \frac{x_i}{(\text{Range of } x_i)} : \text{for every } x_i \in f$$

where f is the set of features in the dataset.

5. A datatable consisting of the different values of parameters inroder to train our classifier was formed. The table initially consisted of the following columns:

- Number of nodes
- Maximum number of iterations allowed
- Trainset Accuracy
- Testset Accuracy
- Seed (used for reproducible results)

6. The classifier was trained for different values of parameters taken from the datatable and the resultant training set and test set accuracy values were stored in their corresponding columns.

7. The training set error and test set error were calculated for each observation in the datatable and were plotted against the number of nodes in the hidden layer using the R package ggplot2[10]. The datatable is shown in figure 4 and the corresponding plot is shown in figure 5.

	no_of_nodes	maxiter	Accuracytrain	AccuracyCV	Seed	ErrorTrain	ErrorCV
1	10	100	92.6190476190476	90.9833333333333	124	7.38095238095238	9.01666666666667
2	15	100	96.3785714285714	93.8722222222222	125	3.62142857142858	6.12777777777778
3	20	100	97.1142857142857	94.4222222222222	126	2.88571428571429	5.57777777777778
4	25	100	98.4190476190476	95.2833333333333	127	1.58095238095238	4.71666666666667
5	30	100	98.9523809523809	95.6222222222222	128	1.04761904761905	4.37777777777778
6	40	100	99.7023809523809	95.8	129	0.297619047619051	4.2
7	50	100	99.9428571428572	96.3	130	0.0571428571428498	3.7

Figure 4: Datatable containing the values obtained during the training of classifier with various parameters

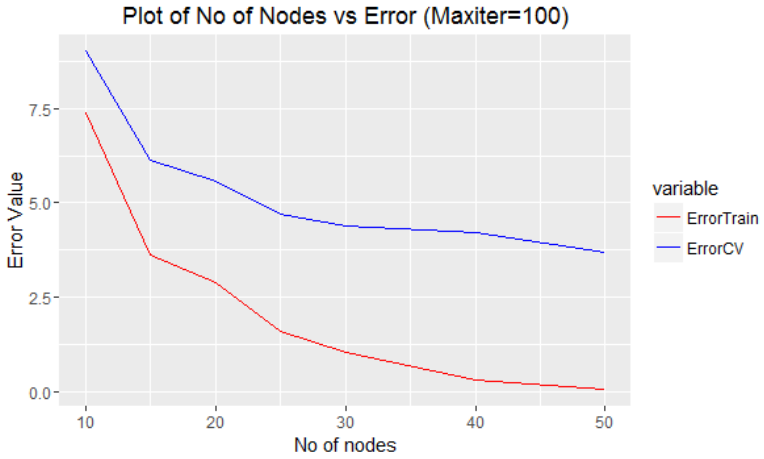

Figure 5: Graph showing the variation of training and test set errors with the no of nodes in the hidden layer.

8. We then chose the optimal parameters for training our classifier using the plot (in this case, the no of nodes was taken as 50 and the maximum number of allowed iteration was taken as 100).
9. The classifier was then trained on the total training data i.e. training set ∪ cross validation set.
10. The test set was then loaded into memory and was then reduced and normalized just like the training dataset.
11. Finally prediction were made on the test set and were compared with the test set labels inorder to obtain the final test set accuracy.The formula used to calculate accuracy is as follows:

$$Accuracy = \frac{\text{No. of correct predictions}}{\text{Total No. of examples}}$$

The final test set acuuracy was found to be 96.08%.

V. Conclusion

This project explored the performance and implementation of single hidden layer neural networks in handwritten digit classification. It was

found that the classifier performed fairly well given that its parameters were chosen using model selection procedure. It should be noted that the performance of the classifier can be further improved by the use of more complex network architecture and deep learning.[3]

VI. R Code

This section contains the R code we used for our project. The R script can be found in the following url: https://github.com/harshitmahapatra/MNIST-Neural-Net-Digit-Recognizer.git

```
#Reading the data and splitting it into training and cross validation set
train <- read.csv("mnist_train.csv",header= FALSE)
splittingseed <- 123
set.seed(splittingseed )
selsize <- floor(0.70*nrow(train))
sel_ind <- sample(seq_len(nrow(train)),size=selsize)
trainset <- train[sel_ind,]
cvset <- train[-sel_ind,]
X <- trainset[,-1]
Y <- trainset[,1]
trainlabel <- trainset[,1]
cvlabel <- cvset[,1]
library(nnet)
Y <- class.ind(Y)
print(X[1:5,1:5])
print(Y[1:5,])

#Functions to help reduce the dimensions of images
reducematrix <- function(source_matrix, red_ratio) {
if(red_ratio == 1) {
return(source_matrix)
}
red_ratio <- 1/red_ratio #reduction ratio
red_matrix <- matrix(0, nrow = nrow(source_matrix)/red_ratio,
ncol = ncol(source_matrix)/red_ratio) #new matrix with reduced dimensions
for(i in 1:nrow(red_matrix))
{
for(j in 1:ncol(red_matrix))
{
red_matrixrow <- c((red_ratio*i - red_ratio + 1):(red_ratio*i))
#figuring out which pixels get merged
red_matrixcolumn <- c((red_ratio*j - red_ratio + 1):(red_ratio*j))
for(x in red_matrixrow){
for(y in red_matrixcolumn){
red_matrix[i,j] <- red_matrix[i,j] +
source_matrix[x,y]/(red_ratio*red_ratio)
#merging the pixels
}
}}}
```

```
return(red_matrix)
}
reduceDataFrame <- function(frameJob, red_ratio, hasLabels) {
adjust <- 0
if(hasLabels == TRUE) {
adjust <- 1
}
dimension <- sqrt(ncol(frameJob) - adjust) #assume square matrix
newColSize = dimension*dimension*red_ratio*red_ratio + adjust
#new column size with reduced dimensions
newFrame <- data.frame(matrix(0, ncol = newColSize, nrow =
nrow(frameJob))) #new frame with reduced dimensions
for(z in 1:nrow(frameJob)) {
if((z %% 1000) == 0) {
cat(c(z, "\n"))
}
imatrix <- matrix(0, nrow = 2, ncol = 2) #matrix to hold image
#test vs train
if(adjust == 0) {
imatrix <- matrix(frameJob[z, ], ncol = dimension)
}
if(adjust == 1) {
imatrix <- matrix(frameJob[z, -1], ncol = dimension)
}
imatrix <- as.character(imatrix)
imatrix <- as.numeric(imatrix)
imatrix <- matrix(imatrix, ncol = dimension)
imatrix <- reducematrix(imatrix, red_ratio)
newRow <- c(as.vector(imatrix))
if(adjust == 1) {  #add labels back in
newRow <- c(frameJob[z, 1], newRow)
}
newFrame[z, ] <- newRow
}
return(newFrame)
}
#End of functions

#Applying the function to training set and also normalizing it
X_reduced <-reduceDataFrame(X,red_ratio=1/2,hasLabels=FALSE)
X_reducednormalized <- X_reduced/255
print(X_reducednormalized[1:5,1:5])


#Visualizing original and reduced image
attach(X)
par(mfrow=c(2,2),mai=c(0.8,0.1,0.1,0.1))
m <- matrix(unlist(X[3,]),nrow = 28,ncol = 28 )
```

```
image(m,axes=FALSE)
m <- matrix(unlist(X[12,]),nrow = 28,ncol = 28 )
image(m,axes=FALSE)
m <- matrix(unlist(X[48,]),nrow = 28,ncol = 28 )
image(m,axes=FALSE)
m <- matrix(unlist(X[212,]),nrow = 28,ncol = 28 )
image(m,axes=FALSE)
par(mfrow=c(1,2),mai=c(0.5,0.5,0.5,0.5))
m <- matrix(unlist(X[3,]),nrow = 28,ncol = 28 )
image(m,xlab='(a)')
m <- matrix(unlist(X_reduced[3,]),nrow = 14,ncol = 14 )
image(m,xlab='(a)')


#Applying the function to cross validation set and normalizing it
cv_reduced <-reduceDataFrame(cvset[,-1],red_ratio=1/2,hasLabels=FALSE)
cv_reducednormalized = cv_reduced/255
print(cv_reducednormalized[5,5])

#Forming a data table to store error for various combinations
of no of nodes and maximum iterations.
datatable_no_of_nodes <-
data.frame(c(10,15,20,25,30,40,50),c(100,100,100,100,100,100,100
),as.numeric(c(NA,NA,NA,NA,NA,NA,NA))
,as.numeric(c(NA,NA,NA,NA,NA,NA,NA)),
as.numeric(c(NA,NA,NA,NA,NA,NA,NA)))
colnames(datatable_no_of_nodes) <-
c("no_of_nodes","maxiter","Accuracytrain","AccuracyCV","Seed")
print(datatable_no_of_nodes)

#Function to train a neural network (where the no_of_nodes and
maxiter are taken from a row of datatable) and the calculated
#accuracies are stored in the same row.

calculate_accuracy<- function(i,nodes,iter)
{
require(nnet)
s <-123+i
set.seed(s)
cat(sprintf("Nodes:%f\n",nodes))
cat(sprintf("Maxiter:%f\n",iter))
model <- nnet(X_reducednormalized,Y,size=nodes,softmax=TRUE,maxit=iter,
MaxNWts =80000)
prediction <- predict(model,X_reducednormalized,type="class")
table(prediction,trainlabel)
correct <- prediction==trainlabel
AccuracyTrain <- (sum(correct)/nrow(X_reducednormalized))*100
cat(sprintf("Accuracytrain:%f\n",AccuracyTrain))
```

```
prediction2 <- predict(model,cv_reducednormalized,type="class")
table(prediction2,cvlabel)
correct2<- prediction2==cvlabel
AccuracyCV <- (sum(correct2)/nrow(cv_reducednormalized))*100
cat(sprintf("Accuracycv:%f\n",AccuracyCV))
return(c(AccuracyTrain,AccuracyCV,s))
}
#End of function

#Applying the function to all the rows of datatable
for(j in 1:7)
{
temp <-calculate_accuracy(j,datatable_no_of_nodes[j,1],
datatable_no_of_nodes[j,2])
datatable_no_of_nodes[j,3]<-temp[1]
datatable_no_of_nodes[j,4]<-temp[2]
datatable_no_of_nodes[j,5]<-temp[3]
}
#Adding columns for errors in data table
datatable_no_of_nodes$ErrorTrain <-
(100-datatable_no_of_nodes$Accuracytrain)
datatable_no_of_nodes$ErrorCV <-
(100-datatable_no_of_nodes$AccuracyCV)
print(datatable_no_of_nodes)

#Storing the datatable in png format for future reference.
library(gridExtra)
png("datatable.png",height = 300,width = 4000)
p <-tableGrob(datatable_no_of_nodes)
grid.arrange(p)
dev.off()

#Ploting the graph of no_of_nodes vs errors
plotdataframe <- datatable_no_of_nodes[,c(1,6,7)]

#melting the dataframe to feed to ggplot() function
library(reshape2)
meltedframe <- melt(plotdataframe,id="no_of_nodes")

#Applying ggplot() function
library(ggplot2)
finalplot<-ggplot(meltedframe,aes(no_of_nodes,value,color=variable))+
geom_line()+scale_colour_manual(values = c("red","blue"))
finalplot <- finalplot+xlab("No␣of␣nodes")+ylab("Error
Value")+ggtitle("Plot␣of␣No␣of␣Nodes␣vs␣Error(Maxiter=100)")
finalplot

#From the plot we choose no_of_nodes=50 and maxiter=100 for
```

```
#optimal error

#Train nnet using optimal parameters on trainingset+cvset to
#form final training set
totaltrain_reducednormalized <-
rbind(X_reducednormalized,cv_reducednormalized)
totaltrainlabel <- c(trainlabel,cvlabel)
totaltrainfinal <- class.ind(totaltrainlabel)

#Training the nnet on totat_training_set
finalseed <- 150
set.seed(finalseed)
model_final <-
nnet(totaltrain_reducednormalized,totaltrainfinal,size=50,softma
x=TRUE,maxit=100,MaxNWts = 80000)


#Reading,reducing and normalizing test data
test<- read.csv("mnist_test.csv",header=FALSE)
test_reduced <-reduceDataFrame(test,red_ratio=1/2,hasLabels=TRUE)
test_reducednormalized <-test_reduced[,-1]/255

#Calculating Final Accuracies
prediction2 <- predict(model_final,test_reducednormalized,type="class")
correct <- prediction2==test_reduced[,1]
FinalAccurracyTest <- sum(correct)/nrow(test_reduced)*100
prediction <-
predict(model_final,totaltrain_reducednormalized,type="class")
correct <- prediction==totaltrainlabel
FinalAccurracyTrain <-
sum(correct)/nrow(totaltrain_reducednormalized) *100
cat(sprintf("Final train accuracy =%f\n",FinalAccuracyTrain))
cat(sprintf("Final test accuracy =%f\n",FinalAccuracyTest))

#Final TestAccuracy=96.08%
#Final TrainAccuracy=99.27%
```

References

[1] Ajith Abraham. Artificial neural networks. *handbook of measuring system design*, 2005.

[2] Léon Bottou, Corinna Cortes, John S Denker, Harris Drucker, Isabelle Guyon, Lawrence D Jackel, Yann LeCun, Urs A Muller, Edward Sackinger, Patrice Simard, et al. Comparison of classifier methods: a case study in handwritten digit recognition. In *International Conference on Pattern Recognition*, pages 77–77. IEEE Computer Society Press, 1994.

[3] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

[4] Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems*, pages 396–404. Morgan Kaufmann, 1990.

[5] Y Le Cun, LD Jackel, B Boser, JS Denker, HP Graf, I Guyon, D Henderson, RE Howard, and W Hubbard. Handwritten digit recognition: Applications of neural network chips and automatic learning. *Communications Magazine, IEEE*, 27(11):41–46, 1989.

[6] Yann LeCun, LD Jackel, L Bottou, A Brunot, C Cortes, JS Denker, H Drucker, I Guyon, UA Muller, E Sackinger, et al. Comparison of learning algorithms for handwritten digit recognition. In *International conference on artificial neural networks*, volume 60, pages 53–60, 1995.

[7] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. ISBN 3-900051-07-0.

[8] Joseph Chet Redmon. MNIST in CSV. `http://pjreddie.com/projects/mnist-in-csv/`.

[9] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. ISBN 0-387-95457-0.

[10] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009.

[11] Corinna Cortes Y LeCun. MNIST digit database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.