STATISTICAL METHODS IN AI
ASSIGNMENT3:
Multilayer Neural Networks

Name – Megha Agarwal
Roll No – 201506511
Course – M.Tech CSIS(PG1)
Dated : 26 February, 2016

AIM:

The aim of this assignment is to experiment with Multilayer Feedforward Neural Network (MLFNN) with Backpropagation (BP) learning we learned as part of Chapter 6 on real world problems. Due credit will be given for choosing non-trivial feature extraction and insightful presentation of results.

DATA-SET DESCRIPTION:

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size- normalized and centered in a fixed-size image.

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive. The training data set, has 785 columns. The first column, called "label", is the digit that was drawn by the user. The rest of the columns contain the pixel-values of the associated image.

Each pixel column in the training set has a name like pixelx, where x is an integer between 0 and 783, inclusive. To locate this pixel on the image, suppose that we have decomposed x as x = i * 28 + j, where i and j are integers between 0 and 27, inclusive. Then pixelx is located on row i and column j of a 28 x 28 matrix, (indexing by zero).

The recognizer present in the code read the image data, extract features from it and uses a multilayer feedforward neural network classifier to recognize any test image. The 5-fold cross validation is used for training the data and finally the test data sample is used to find out the accuracy/error rate of the trained Multi Layer Neural Network.

<u>QUESTION-1 :</u> ***Write a detailed report on the different features that you used and the corresponding error rates (reported as percentages). Also give a confusion matrix that shows the kind of errors that your classifier makes. In this problem, your confusion matrix is a 10 x 10 matrix, where the rows represent the true label of a test sample and the columns represent the predicted labels of the NN classifier. Report the average error rate as well as standard deviation of the error rate for each fold along with other metrics such as Precision, Recall/Sensitivity, Specificity and Accuracy.***

Code:

```
import numpy as np
import operator
import time
import os
import struct
from array import array
import math
import sys
import random
import matplotlib.pyplot as plt
import csv
from tabulate import tabulate
import cPickle
import gzip


def load(path_img, path_lbl):
    with open(path_lbl, 'rb') as file:
        magic, size = struct.unpack(">II", file.read(8))
        if magic != 2049:
            raise ValueError('Magic number mismatch, expected 2049,"got %d' % magic)
        labels = array("B", file.read())

    with open(path_img, 'rb') as file:
        magic, size, rows, cols = struct.unpack(">IIII", file.read(16))
        if magic != 2051:
            raise ValueError('Magic number mismatch, expected 2051,"got %d' % magic)
        image_data = array("B", file.read())

    images = []
    for i in xrange(size):
        images.append([0]*rows*cols)
    for i in xrange(size):
        images[i][:] = image_data[i*rows*cols : (i+1)*rows*cols]
    return images, labels
```

```
def load_data_wrapper(train_data, train_label, test_data, test_labels):

    f = gzip.open('mnist.pkl.gz', 'rb')
    tr_d, va_d, te_d = cPickle.load(f)
    f.close()

    train_data = np.vstack((tr_d[0], va_d[0]))
    training_inputs = [np.reshape(x, (784, 1)) for x in train_data]
    training_results = [vectorized_result(y) for y in train_label]

    training_data = zip(training_inputs, training_results)

    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, test_labels)

    return (training_inputs, training_results, test_inputs, test_labels)


def vectorized_result(j):
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e

def CreateConfusionMatrix(predictions, actual_classes, noOfClasses):
    #Fetching the name of the classes to dictionary and then to the list
    c =[0, 1, 2, 3, 4, 5 ,6, 7, 8, 9]
    length = len(c)

    #Creating confusion matrix as list -> empty list and hence comparing and increasing the 
count
    confusion_matrix=[]
    for i in range(length):
        for j in range(length):
            confusion_matrix.append(0)

    count = 0
    for i in range(len(actual_classes)):
        for j in range(length):
            for k in range(length):
                if actual_classes[i] == c[j] and predictions[i] == c[k]:
                    count = count +1
                    confusion_matrix[j*length+k] = confusion_matrix[j*length+k]+1

    print "\t\t"+'PREDICTED'
    table = []

    #Append Classes name
    L=[]
    L.append('\t')
```

```
    L.append('\t')
    for i in range(length):
        L.append(c[i])
    table.append(L)

    #Create Empty Table
    L=[]
    for i in range(length):
        for j in range(length+2):
            if i==length/2:
                if j==0:
                    L.append('ACTUAL')
                elif j==1:
                    L.append(c[i])
                else:
                    L.append('\t')
            else:
                if j==1:
                    L.append(c[i])
                else:
                    L.append('\t')
        table.append(L)
        L=[]

    #Populate value to the confusion matrix/empty table
    value_index=0
    for i in range(1, length+1):
        for j in range(2, length+2):
            table[i][j] = confusion_matrix[value_index]
            value_index+=1

    print tabulate(table, tablefmt="grid")
    confusionMatrix = [[0 for i in xrange(noOfClasses)] for i in xrange(noOfClasses)]
    for x in range(len(actual_classes)):
        if predictions[x] == actual_classes[x]:
            confusionMatrix[actual_classes[x]][actual_classes[x]] = confusionMatrix[actual_classes[x]][actual_classes[x]] + 1
        else:
            confusionMatrix[actual_classes[x]][predictions[x]] = confusionMatrix[actual_classes[x]][predictions[x]] + 1
    return confusionMatrix

def CalculatePrecisionAndRecall(confusionMatrix, noOfClasses, noOfTestSamples):
    totalRecall = 0.0
    totalPrecision = 0.0
    precision, specificity, recall = [], [], []
    totalSpecificity = 0.0

    for i in range(10):
        classPrecision = 0.0
```

```
        for j in range(10):
            classPrecision = classPrecision + confusionMatrix[j][i]
        if classPrecision != 0.0:
            classPrecision = (confusionMatrix[i][i] / float(classPrecision)) * 100
        else:
            classPrecision = 0.
        precision.append(classPrecision)
        totalPrecision = totalPrecision + classPrecision

    for i in range(10):
        classRecall = 0.0
        for j in range(10):
            classRecall = classRecall + confusionMatrix[i][j]
        if classRecall != 0.0:
            classRecall = (confusionMatrix[i][i] / float(classRecall)) * 100
        else:
            classRecall = 0.0
        recall.append(classRecall)
        totalRecall = totalRecall + classRecall

    for i in range(10):
        numerator = noOfTestSamples - confusionMatrix[i][i]
        denominator = numerator
        for j in range(10):
            if i != j:
                denominator = denominator + confusionMatrix[j][i]
        classSpecificity = (numerator / float(denominator))
        classSpecificity =  classSpecificity * 100
        totalSpecificity = totalSpecificity + classSpecificity
        specificity.append(classSpecificity)

    avgRecall = (totalRecall / float(noOfClasses))
    avgPrecision = (totalPrecision / float(noOfClasses))
    avgSpecificity = (totalSpecificity / float(noOfClasses))
    return avgPrecision, avgRecall, avgSpecificity, precision, recall, specificity


def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))

def PrintResults(confusionMatrix, avgPrecision, avgRecall, avgSpecificity, precision, recall,
specificity):
    for i in range(0, len(precision)):
        print "Class", i
        print "----------------------------------"
        print "Precision :", precision[i]
        print "Recall :", recall[i]
```

```
        print "Specificity :", specificity[i]
        print "\n"
    print "________________________________________"
    print "Average Recall:", avgRecall
    print "Average Precision:", avgPrecision
    print "Average Specificity:", avgSpecificity


def GetAccuracy(testLabels, predictions):
    correct = 0
    for x in range(len(testLabels)):
        if testLabels[x] == predictions[x]:
                correct += 1
    return (correct/float(len(testLabels))) * 100.0


def feedforward(a, biases, weights):
    for b, w in zip(biases, weights):
        a = sigmoid(np.dot(w, a)+b)
    return a

def cross_validation_division(iteration_no, training_data, training_label, testing_label):

    if iteration_no==1:
        train_data = training_data[:50000]
        train_label = training_label[:50000]

        test_inputs = training_data[50000:60000]
        test_labels = testing_label[50000:60000]

    elif iteration_no==2:
        train_data = training_data[10000:60000]
        train_label = training_label[10000:60000]

        test_inputs = training_data[:10000]
        test_labels = testing_label[:10000]


    elif iteration_no==3:
        train_data1 = training_data[:10000]
        train_label1 = training_label[:10000]
        train_data2 = training_data[20000:60000]
        train_label2 = training_label[20000:60000]
        train_data = np.vstack((train_data1, train_data2))
        train_label = np.vstack((train_label1, train_label2))
        test_inputs = training_data[10000:20000]
        test_labels = testing_label[10000:20000]


    elif iteration_no==4:
```

```
        train_data1 = training_data[:20000]
        train_label1 = training_label[:20000]
        train_data2 = training_data[30000:60000]
        train_label2 = training_label[30000:60000]
        train_data = np.vstack((train_data1, train_data2))
        train_label = np.vstack((train_label1, train_label2))

        test_inputs = training_data[20000:30000]
        test_labels = testing_label[20000:30000]

    elif iteration_no==5:
        train_data1 = training_data[:30000]
        train_label1 = training_label[:30000]
        train_data2 = training_data[40000:60000]
        train_label2 = training_label[40000:60000]
        train_data = np.vstack((train_data1, train_data2))
        train_label = np.vstack((train_label1, train_label2))


        test_inputs = training_data[30000:40000]
        test_labels = testing_label[30000:40000]

    elif iteration_no==6:
        train_data1 = training_data[:40000]
        train_label1 = training_label[:40000]
        train_data2 = training_data[50000:60000]
        train_label2 = training_label[50000:60000]
        train_data = np.vstack((train_data1, train_data2))
        train_label = np.vstack((train_label1, train_label2))

        test_inputs = training_data[40000:50000]
        test_labels = testing_label[40000:50000]


    test_data = zip(test_inputs, test_labels)
    training_data = zip(train_data, train_label)
    return training_data, test_data


def SGD(training_inputs, training_label, testing_label, testing_final_inputs, testing_final_labels, epochs, mini_batch_size, eta, noOfClasses, biases, weights):

    iteration_no = 1;
    accuracy_list=[]
    error_list=[]

    epoch_iteration = 0
    for iteration in range(6):
        plot_error_list=[]
        plot_epoch_list=[]
```

```
    print
    print "**********************************"
                                      print    "=============FOLD    NO
{0}=============".format(iteration_no)
      training_data, test_data = cross_validation_division(iteration_no, training_inputs,
training_label, testing_label)
    iteration_no = iteration_no + 1
    n = len(training_data)
    n_test = len(test_data)

    for j in xrange(epochs):
      random.shuffle(training_data)
      epoch_iteration = epoch_iteration + 1
           mini_batches = [training_data[k:k+mini_batch_size] for k in xrange(0, n,
mini_batch_size)]
      for mini_batch in mini_batches:
        biases, weights = update_mini_batch(mini_batch, eta, biases, weights)
      sum,test_results=evaluate(test_data, biases, weights)
      print "Training Iteration {0}: {1} / {2}".format(j, sum, n_test)
      error = (1-(float(sum)/float(n_test)))
      plot_error_list.append(error)
      plot_epoch_list.append(j)


    #print test_results
    predictions = []
    testLabels=[]
    for i in test_results:
      predictions.append(int(i[0]))
      testLabels.append(int(i[1]))

    confusionMatrix = CreateConfusionMatrix(testLabels,predictions, noOfClasses)
          avgPrecision, avgRecall, avgSpecificity, precision, recall, specificity =
CalculatePrecisionAndRecall(confusionMatrix, noOfClasses, len(testLabels))
    PrintResults(confusionMatrix, avgPrecision, avgRecall, avgSpecificity, precision, recall,
specificity)

    accuracy = GetAccuracy(predictions, testLabels)
    accuracy_list.append(accuracy)
    print "Accuracy :", accuracy
    error = 100.00 - accuracy
    error_list.append(error)
    print "Error Rate :", error
    print "__________________________________________"
    plt.plot(plot_epoch_list, plot_error_list)
    plt.show()

  print
  print '*****************************************************'
  print '*************Average Values of FOLDS****************'
```

```
    print '----------------------------------------------------'
    accuracy_list = np.array(accuracy_list)
    print "Average Accuracy :", np.mean(accuracy_list, axis=0)
    error_list = np.array(error_list)
    print "Average Error Rate :", np.mean(error_list, axis=0)
    print "Standard Deviation of Error Rate :", np.std(error_list, axis=0)
    print '*****************************************************'
    print

    print '==========================================================='
    print '****************TESTING the Network********************'
    print '==========================================================='
    print

    test_data = zip(testing_final_inputs, testing_final_labels)
    n_test = len(test_data)
    sum,test_results=evaluate(test_data, biases, weights)
    print "Sum :", sum
    print "/",
    print n_test

    predictions = []
    testLabels=[]
    for i in test_results:
        predictions.append(int(i[0]))
        testLabels.append(int(i[1]))
    confusionMatrix = CreateConfusionMatrix(testLabels,predictions, noOfClasses)
    avgPrecision, avgRecall, avgSpecificity, precision, recall, specificity = CalculatePrecisionAndRecall(confusionMatrix, noOfClasses, len(testLabels))
    PrintResults(confusionMatrix, avgPrecision, avgRecall, avgSpecificity, precision, recall, specificity)
    accuracy = GetAccuracy(predictions, testLabels)
    print "Accuracy :", accuracy
    error = 100.00 - accuracy
    print "Error Rate :", error
    print '*****************************************************'
    print



def update_mini_batch(mini_batch, eta, biases, weights):
    nabla_b = [np.zeros(b.shape) for b in biases]
    nabla_w = [np.zeros(w.shape) for w in weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = backprop(x, y, biases, weights)
```

```
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    weights = [w-(eta/len(mini_batch))*nw for w, nw in zip(weights, nabla_w)]
    biases = [b-(eta/len(mini_batch))*nb
                for b, nb in zip(biases, nabla_b)]

    return biases, weights

def backprop(x, y, biases, weights):
    nabla_b = [np.zeros(b.shape) for b in biases]
    nabla_w = [np.zeros(w.shape) for w in weights]
    activation = x
    activations = [x]
    zs = []
    for b, w in zip(biases, weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    delta = cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    for l in xrange(2, num_layers):
        z = zs[-l]
        spv = sigmoid_prime(z)
        delta = np.dot(weights[-l+1].transpose(), delta) * spv
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)

def evaluate(test_data, biases, weights):
    test_results = [(np.argmax(feedforward(x, biases, weights)), y)
                for (x, y) in test_data]
    #print test_results
    return sum(int(x == y) for (x, y) in test_results),test_results

def cost_derivative(output_activations, y):
    return (output_activations-y)

def binarisation(img):
    for i in range(len(img)):
        if img[i]>0:
            img[i]/= 255.0
        else:
            img[i]=0
    return img

if __name__ == '__main__':
    path='.'
    test_img_fname = 't10k-images.idx3-ubyte'
```

```
    test_lbl_fname = 't10k-labels.idx1-ubyte'
    train_img_fname = 'train-images.idx3-ubyte'
    train_lbl_fname = 'train-labels.idx1-ubyte'

    test_data, test_labels, train_data, train_label = [],[],[],[]
    test_data, test_labels = load(os.path.join(path, test_img_fname),os.path.join(path, test_lbl_fname))
    train_data, train_label = load(os.path.join(path, train_img_fname),os.path.join(path, train_lbl_fname))
    for i in range(len(train_data)):
        train_data[i] = binarisation(train_data[i])
    for i in range(len(test_data)):
        test_data[i] = binarisation(test_data[i])

    test_data = np.array(test_data)
    train_data = np.array(train_data)
    test_data = test_data.astype(float)
    train_data = train_data.astype(float)
    train_label = np.array(train_label)
    test_labels = np.array(test_labels)

    training_data, training_results, test_inputs, test_labels = load_data_wrapper(train_data, train_label, test_data, test_labels)
    sizes = [784, 30, 10]
    num_layers = len(sizes)
    biases = [np.random.randn(y, 1) for y in sizes[1:]]
    weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
    SGD(training_data, training_results, train_label, test_inputs, test_labels, 20, 10, 1.0, 10, biases, weights)
```

Output

```
**********************************
============FOLD NO 1============
Training Iteration 0: 9060 / 10000
Training Iteration 1: 9258 / 10000
Training Iteration 2: 9323 / 10000
Training Iteration 3: 9366 / 10000
Training Iteration 4: 9401 / 10000
Training Iteration 5: 9399 / 10000
Training Iteration 6: 9395 / 10000
Training Iteration 7: 9429 / 10000
Training Iteration 8: 9454 / 10000
Training Iteration 9: 9464 / 10000
Training Iteration 10: 9463 / 10000
Training Iteration 11: 9465 / 10000
```

Training Iteration 12: 9493 / 10000
Training Iteration 13: 9495 / 10000
Training Iteration 14: 9481 / 10000
Training Iteration 15: 9502 / 10000
Training Iteration 16: 9498 / 10000
Training Iteration 17: 9510 / 10000
Training Iteration 18: 9494 / 10000
Training Iteration 19: 9493 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	970	0	6	1	3	7	4	1	3	10
	1	0	1037	2	1	4	1	1	5	8	1
	2	4	7	935	10	1	8	3	12	8	1
	3	1	4	7	980	0	29	0	11	11	10
	4	0	1	6	0	928	7	2	2	2	20
	5	0	1	2	10	1	816	1	1	9	5
	6	7	2	9	3	6	21	950	1	4	0
	7	3	1	6	4	5	3	0	1037	3	11
	8	5	11	14	15	9	21	6	3	954	17
	9	1	0	3	6	26	2	0	17	7	886

Class 0

Precision : 97.8809283552
Recall : 96.5174129353
Specificity : 99.7679814385

Class 1

Precision : 97.462406015
Recall : 97.8301886792
Specificity : 99.6996662959

Class 2

Precision : 94.4444444444
Recall : 94.5399393327

Specificity : 99.3969298246

Class 3

Precision : 95.145631068
Recall : 93.0674264008
Specificity : 99.4487320838

Class 4

Precision : 94.4048830112
Recall : 95.867768595
Specificity : 99.3973923524

Class 5

Precision : 89.1803278689
Recall : 96.4539007092
Specificity : 98.9335344178

Class 6

Precision : 98.2419855222
Recall : 94.7158524427
Specificity : 99.8125068931

Class 7

Precision : 95.1376146789
Recall : 96.6449207829
Specificity : 99.4121561668

Class 8

Precision : 94.5490584737
Recall : 90.4265402844
Specificity : 99.3956708054

Class 9

Precision : 92.1956295525
Recall : 93.4599156118
Specificity : 99.1838067254

Average Recall: 94.9523865774
Average Precision: 94.864290899
Average Specificity: 99.4448377004
Accuracy : 94.93
Error Rate : 5.07

****

=============FOLD NO 2=============
Training Iteration 0: 9655 / 10000
Training Iteration 1: 9629 / 10000
Training Iteration 2: 9615 / 10000
Training Iteration 3: 9588 / 10000
Training Iteration 4: 9588 / 10000
Training Iteration 5: 9628 / 10000
Training Iteration 6: 9622 / 10000
Training Iteration 7: 9631 / 10000
Training Iteration 8: 9605 / 10000
Training Iteration 9: 9608 / 10000
Training Iteration 10: 9620 / 10000
Training Iteration 11: 9616 / 10000
Training Iteration 12: 9625 / 10000
Training Iteration 13: 9589 / 10000
Training Iteration 14: 9589 / 10000
Training Iteration 15: 9597 / 10000
Training Iteration 16: 9583 / 10000
Training Iteration 17: 9604 / 10000
Training Iteration 18: 9602 / 10000
Training Iteration 19: 9591 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	986	0	5	3	1	4	5	0	5	5
	1	0	1095	6	2	1	0	1	0	6	1
	2	0	6	942	15	2	3	3	8	4	3
	3	2	1	5	963	1	7	0	2	7	8
	4	2	2	8	0	939	6	4	6	3	13
A	5	2	4	3	14	2	830	13	0	15	4
	6	4	0	3	1	10	6	986	1	11	1

	7	1	7	7	15	3	1	0	1042	4	15
	8	3	6	11	10	1	4	2	3	884	4
	9	1	6	1	9	20	2	0	8	5	924

Class 0

Precision : 98.5014985015
Recall : 97.2386587771
Specificity : 99.8338686455

Class 1

Precision : 97.1606033718
Recall : 98.4712230216
Specificity : 99.6419380105

Class 2

Precision : 95.0554994955
Recall : 95.537525355
Specificity : 99.4619523444

Class 3

Precision : 93.3139534884
Recall : 96.686746988
Specificity : 99.242257852

Class 4

Precision : 95.8163265306
Recall : 95.523906409
Specificity : 99.5495495495

Class 5

Precision : 96.1761297798
Recall : 93.5738444194
Specificity : 99.6414212757

Class 6

Precision : 97.2386587771
Recall : 96.3831867058
Specificity : 99.6903339969

Class 7

Precision : 97.3831775701
Recall : 95.1598173516
Specificity : 99.6884041843

Class 8

Precision : 93.6440677966
Recall : 95.2586206897
Specificity : 99.3461203139

Class 9

Precision : 94.4785276074
Recall : 94.6721311475
Specificity : 99.408543264

Average Recall: 95.8505660865
Average Precision: 95.8768442919
Average Specificity: 99.5504389437
Accuracy : 95.91
Error Rate : 4.09

****
=============FOLD NO 3=============
Training Iteration 0: 9708 / 10000
Training Iteration 1: 9712 / 10000
Training Iteration 2: 9695 / 10000
Training Iteration 3: 9689 / 10000
Training Iteration 4: 9690 / 10000
Training Iteration 5: 9698 / 10000
Training Iteration 6: 9670 / 10000
Training Iteration 7: 9674 / 10000
Training Iteration 8: 9682 / 10000
Training Iteration 9: 9658 / 10000
Training Iteration 10: 9655 / 10000
Training Iteration 11: 9657 / 10000
Training Iteration 12: 9665 / 10000
Training Iteration 13: 9643 / 10000

Training Iteration 14: 9659 / 10000
Training Iteration 15: 9643 / 10000
Training Iteration 16: 9667 / 10000
Training Iteration 17: 9663 / 10000
Training Iteration 18: 9636 / 10000
Training Iteration 19: 9661 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	974	0	4	0	1	3	7	0	2	5
	1	0	1139	3	5	3	1	1	3	4	1
	2	1	2	883	11	2	3	0	5	8	3
	3	1	4	9	995	0	4	0	4	9	10
	4	0	1	6	1	939	2	1	2	2	15
A	5	5	3	4	10	1	882	2	0	5	9
	6	7	0	5	5	5	3	941	0	5	4
	7	0	1	6	5	1	0	0	1000	2	9
	8	3	3	17	7	1	9	5	1	933	5
	9	2	1	1	5	12	5	0	8	8	975

Class 0

Precision : 98.0866062437
Recall : 97.7911646586
Specificity : 99.7899391929

Class 1

Precision : 98.7001733102
Recall : 98.1896551724
Specificity : 99.8310049572

Class 2

Precision : 94.1364605544
Recall : 96.1873638344
Specificity : 99.4003488879

Class 3

Precision : 95.30651341
Recall : 96.0424710425
Specificity : 99.4588027391

Class 4

Precision : 97.3056994819
Recall : 96.9040247678
Specificity : 99.7138769671

Class 5

Precision : 96.7105263158
Recall : 95.7654723127
Specificity : 99.6720594666

Class 6

Precision : 98.3281086729
Recall : 96.5128205128
Specificity : 99.8236914601

Class 7

Precision : 97.7517106549
Recall : 97.65625
Specificity : 99.7450958661

Class 8

Precision : 95.3987730061
Recall : 94.8170731707
Specificity : 99.5061457419

Class 9

Precision : 94.111969112
Recall : 95.8702064897
Specificity : 99.3286374642

Average Recall: 96.5736501962
Average Precision: 96.5836540762
Average Specificity: 99.6269602743
Accuracy : 96.61
Error Rate : 3.39

****
=============FOLD NO 4=============
Training Iteration 0: 9742 / 10000
Training Iteration 1: 9757 / 10000
Training Iteration 2: 9756 / 10000
Training Iteration 3: 9733 / 10000
Training Iteration 4: 9746 / 10000
Training Iteration 5: 9729 / 10000
Training Iteration 6: 9746 / 10000
Training Iteration 7: 9736 / 10000
Training Iteration 8: 9726 / 10000
Training Iteration 9: 9714 / 10000
Training Iteration 10: 9707 / 10000
Training Iteration 11: 9707 / 10000
Training Iteration 12: 9684 / 10000
Training Iteration 13: 9689 / 10000
Training Iteration 14: 9680 / 10000
Training Iteration 15: 9726 / 10000
Training Iteration 16: 9702 / 10000
Training Iteration 17: 9708 / 10000
Training Iteration 18: 9702 / 10000
Training Iteration 19: 9704 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	952	1	8	5	2	2	3	2	3	2
	1	0	1126	1	4	1	0	1	5	6	1
	2	2	5	983	11	3	2	1	7	4	1
	3	1	4	6	952	0	11	1	3	3	5
	4	1	2	5	2	951	5	3	5	3	11
ACTUAL	5	2	0	1	7	3	900	5	0	7	7
	6	4	0	1	1	4	5	986	1	3	0
	7	0	2	5	4	3	1	0	988	1	8

```
|        | 8 |   5 |    1 |   9 |   8 |   3 |   5 |   4 |   1 | 919 |   7 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        | 9 |   0 |    1 |   0 |   3 |  11 |   3 |   0 |   2 |   4 | 947 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
Class 0
---------------------------------
Precision : 98.4488107549
Recall : 97.1428571429
Specificity : 99.8344918901


Class 1
---------------------------------
Precision : 98.5989492119
Recall : 98.3406113537
Specificity : 99.8200224972


Class 2
---------------------------------
Precision : 96.467124632
Recall : 96.467124632
Specificity : 99.6023417652


Class 3
---------------------------------
Precision : 95.4864593781
Recall : 96.5517241379
Specificity : 99.5051138238


Class 4
---------------------------------
Precision : 96.9418960245
Recall : 96.2550607287
Specificity : 99.6695671329


Class 5
---------------------------------
Precision : 96.3597430407
Recall : 96.5665236052
Specificity : 99.6277643968


Class 6
---------------------------------
Precision : 98.2071713147
Recall : 98.1094527363
```

Specificity : 99.8007085917

Class 7

Precision : 97.4358974359
Recall : 97.628458498
Specificity : 99.7123257358

Class 8

Precision : 96.4323189927
Recall : 95.5301455301
Specificity : 99.6269884805

Class 9

Precision : 95.7532861476
Recall : 97.5283213182
Specificity : 99.5382078065

Average Recall: 97.0120279683
Average Precision: 97.0131656933
Average Specificity: 99.673753212
Accuracy : 97.04
Error Rate : 2.96

****
============FOLD NO 5============
Training Iteration 0: 9758 / 10000
Training Iteration 1: 9768 / 10000
Training Iteration 2: 9749 / 10000
Training Iteration 3: 9736 / 10000
Training Iteration 4: 9759 / 10000
Training Iteration 5: 9749 / 10000
Training Iteration 6: 9736 / 10000
Training Iteration 7: 9755 / 10000
Training Iteration 8: 9766 / 10000
Training Iteration 9: 9726 / 10000
Training Iteration 10: 9748 / 10000
Training Iteration 11: 9736 / 10000
Training Iteration 12: 9711 / 10000
Training Iteration 13: 9731 / 10000
Training Iteration 14: 9728 / 10000
Training Iteration 15: 9731 / 10000

Training Iteration 16: 9730 / 10000
Training Iteration 17: 9735 / 10000
Training Iteration 18: 9744 / 10000
Training Iteration 19: 9723 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	943	0	4	5	3	2	1	2	2	2
	1	0	1125	2	5	1	1	1	4	2	2
	2	2	3	960	11	1	3	1	5	5	1
	3	1	2	0	961	0	3	0	1	2	3
	4	1	3	11	0	956	1	2	5	4	8
ACTUAL	5	3	1	2	9	1	873	7	0	1	2
	6	6	1	4	2	9	2	985	0	5	0
	7	0	4	7	2	1	3	0	986	1	9
	8	6	1	3	8	3	5	3	5	959	11
	9	1	0	2	5	8	2	0	10	4	975

Class 0

Precision : 97.9231568017
Recall : 97.8215767635
Specificity : 99.7796628842

Class 1

Precision : 98.6842105263
Recall : 98.4251968504
Specificity : 99.8312710911

Class 2

Precision : 96.4824120603
Recall : 96.7741935484
Specificity : 99.6143250689

Class 3

Precision : 95.3373015873
Recall : 98.766700925
Specificity : 99.4827206692

Class 4

Precision : 97.2533062055
Recall : 96.4682139253
Specificity : 99.7023481424

Class 5

Precision : 97.5418994413
Recall : 97.1078976641
Specificity : 99.7595365614

Class 6

Precision : 98.5
Recall : 97.1400394477
Specificity : 99.8338870432

Class 7

Precision : 96.8565815324
Recall : 97.3346495558
Specificity : 99.6462524873

Class 8

Precision : 97.3604060914
Recall : 95.5179282869
Specificity : 99.7132458366

Class 9

Precision : 96.2487660415
Recall : 96.82224429
Specificity : 99.5807127883

Average Recall: 97.2178641257

Average Precision: 97.2188040288
Average Specificity: 99.6943962572
Accuracy : 97.23
Error Rate : 2.77

****
=============FOLD NO 6=============
Training Iteration 0: 9768 / 10000
Training Iteration 1: 9753 / 10000
Training Iteration 2: 9779 / 10000
Training Iteration 3: 9772 / 10000
Training Iteration 4: 9766 / 10000
Training Iteration 5: 9752 / 10000
Training Iteration 6: 9763 / 10000
Training Iteration 7: 9761 / 10000
Training Iteration 8: 9744 / 10000
Training Iteration 9: 9755 / 10000
Training Iteration 10: 9760 / 10000
Training Iteration 11: 9743 / 10000
Training Iteration 12: 9753 / 10000
Training Iteration 13: 9743 / 10000
Training Iteration 14: 9742 / 10000
Training Iteration 15: 9753 / 10000
Training Iteration 16: 9749 / 10000
Training Iteration 17: 9740 / 10000
Training Iteration 18: 9738 / 10000
Training Iteration 19: 9745 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	995	0	7	3	3	2	0	3	3	1
	1	1	1102	0	1	0	1	1	3	4	0
	2	0	3	994	19	1	2	3	7	6	1
	3	0	0	2	970	1	4	1	2	8	6
	4	1	1	6	0	925	1	2	5	3	6
ACTUAL	5	2	0	3	4	1	880	2	1	6	3
	6	1	0	3	3	4	4	966	0	4	0
	7	1	3	3	6	1	5	0	1024	3	5
	8	6	4	5	9	1	2	1	0	942	3

```
|        | 9 |   1 |    2 |   2 |   5 |  13 |   1 |   0 |    5 |   3 | 947 |
+--------+---+-----+------+-----+-----+-----+-----+-----+------+-----+-----+
Class 0
--------------------------------
Precision : 98.7103174603
Recall : 97.8367748279
Specificity : 99.8558438678


Class 1
--------------------------------
Precision : 98.8340807175
Recall : 99.0116801438
Specificity : 99.8541128942


Class 2
--------------------------------
Precision : 96.9756097561
Recall : 95.9459459459
Specificity : 99.6569658072


Class 3
--------------------------------
Precision : 95.0980392157
Recall : 97.5855130785
Specificity : 99.449339207


Class 4
--------------------------------
Precision : 97.3684210526
Recall : 97.3684210526
Specificity : 99.7252747253


Class 5
--------------------------------
Precision : 97.5609756098
Recall : 97.5609756098
Specificity : 99.7593524393


Class 6
--------------------------------
Precision : 98.9754098361
Recall : 98.0710659898
Specificity : 99.889429456
```

Class 7

Precision : 97.5238095238
Recall : 97.431018078
Specificity : 99.7111752944

Class 8

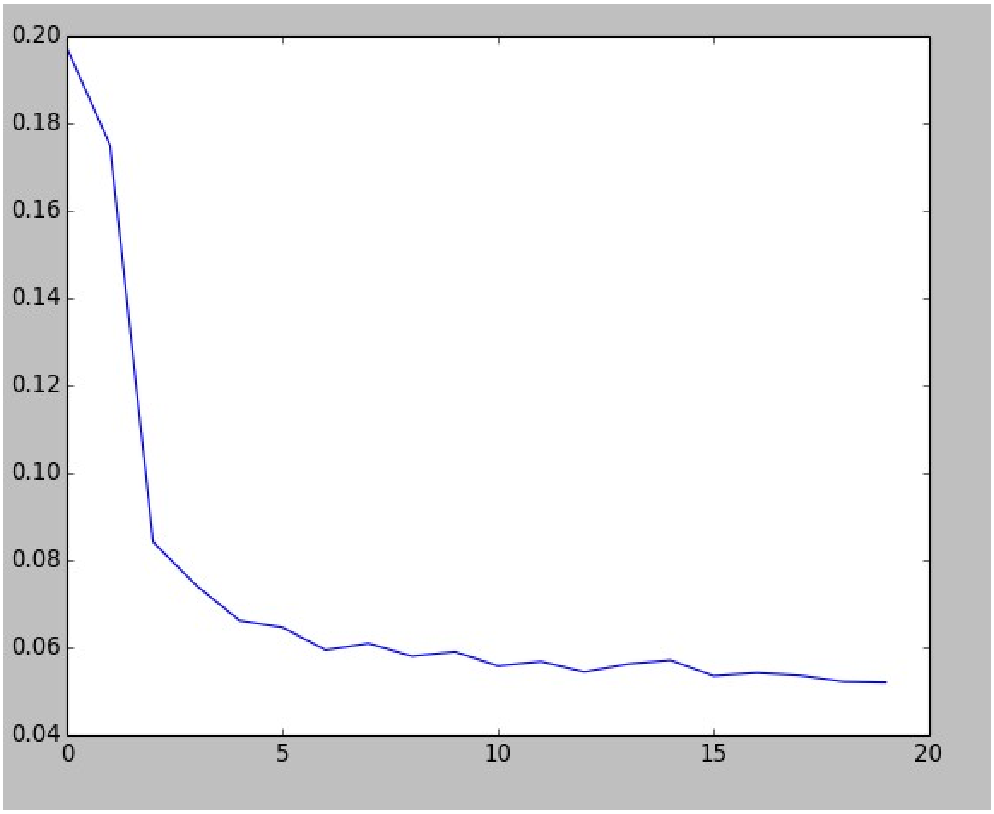
Precision : 95.9266802444
Recall : 96.8139773895
Specificity : 99.5603429325

Class 9

Precision : 97.4279835391
Recall : 96.7313585291
Specificity : 99.7246089447

Average Recall: 97.4356730645
Average Precision: 97.4401326955
Average Specificity: 99.7186445568
Accuracy : 97.45
Error Rate : 2.55

PLOT



****
*************Average Values of FOLDS*****************

Average Accuracy : 96.5283333333
Average Error Rate : 3.47166666667
Standard Deviation of Error Rate : 0.87056335528
****

==
***************TESTING the Network******************
==

Sum : 9525 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	961	0	11	1	3	6	12	2	9	9
	1	0	1121	4	2	1	2	3	9	3	6
	2	2	2	984	15	4	5	6	22	2	0
	3	1	2	1	955	0	23	1	8	9	12
	4	1	0	7	0	936	4	10	1	7	18
ACTUAL	5	4	2	1	9	1	831	8	1	6	4
	6	6	2	3	1	10	6	912	0	11	1
	7	1	1	10	8	2	4	0	971	6	11
	8	3	5	10	9	1	8	6	3	917	11
	9	1	0	1	10	24	3	0	11	4	937

Class 0

Precision : 98.0612244898
Recall : 94.7731755424
Specificity : 99.7902406712

Class 1

Precision : 98.7665198238
Recall : 97.393570808

Specificity : 99.8425728101

Class 2

Precision : 95.3488372093
Recall : 94.43378119
Specificity : 99.4704324801

Class 3

Precision : 94.5544554455
Recall : 94.3675889328
Specificity : 99.3956043956

Class 4

Precision : 95.3156822811
Recall : 95.1219512195
Specificity : 99.4950603732

Class 5

Precision : 93.1614349776
Recall : 95.8477508651
Specificity : 99.3391115926

Class 6

Precision : 95.1983298539
Recall : 95.7983193277
Specificity : 99.496387125

Class 7

Precision : 94.4552529183
Recall : 95.7593688363
Specificity : 99.3726612371

Class 8

Precision : 94.1478439425
Recall : 94.2446043165
Specificity : 99.3763676149

Class 9

Precision : 92.864222002
Recall : 94.5509586276
Specificity : 99.2118226601

Average Recall: 95.2291069666
Average Precision: 95.1873802944
Average Specificity: 99.479026096
Accuracy : 95.25
Error Rate : 4.75
****

QUESTION-2 : ***Compare the results with classification using Euclidean distance based 1-Nearest Neighbor (1NN) Classifier. Present an analysis and discussion of your results.***

Code:

```
import numpy as np
import operator
import time
import os
import struct
from array import array
import math
import sys
import random
from matplotlib.pyplot import *
import csv
from tabulate import tabulate


# majority vote for a little bit optimized worker
def majority_vote(knn, labels):
    knn = [k[0, 0] for k in knn]
    a = {}
    for idx in knn:
        if labels[idx] in a.keys():
            a[labels[idx]] = a[labels[idx]] + 1
        else:
            a[labels[idx]] = 1
    return sorted(a.iteritems(), key=operator.itemgetter(1), reverse=True)[0][0]


def k_nearest_neighbours(train, test, labels):
    k = 7
    train_mat = np.mat(train)
    idx = 0
    size = len(test)
    prediction_list = []

    for test_sample in test:
        idx += 1
        knn = np.argsort(np.sum(np.power(np.subtract(train_mat, test_sample), 2), axis=1), 
axis=0)[:k]
        prediction = majority_vote(knn, labels)
        print prediction
        prediction_list.append(prediction)

    return prediction_list
```

```
def load(path_img, path_lbl):
    with open(path_lbl, 'rb') as file:
        magic, size = struct.unpack(">II", file.read(8))
        if magic != 2049:
            raise ValueError('Magic number mismatch, expected 2049,"got %d' % magic)
        labels = array("B", file.read())

    with open(path_img, 'rb') as file:
        magic, size, rows, cols = struct.unpack(">IIII", file.read(16))
        if magic != 2051:
            raise ValueError('Magic number mismatch, expected 2051,"got %d' % magic)
        image_data = array("B", file.read())

    images = []
    for i in xrange(size):
        images.append([0]*rows*cols)
    for i in xrange(size):
        images[i][:] = image_data[i*rows*cols : (i+1)*rows*cols]
    return images, labels

#Calculate the ratio of the total correct predictions out of all predictions made : classification accuracy
def calculate_accuracy(test_labels, prediction_list):
    correct=0
    length_test_sample = len(test_labels)
    for i in range(len(prediction_list)):
        if test_labels[i] == prediction_list[i]:
            correct = correct+1
    print correct
    accuracy_percentage = (float(correct)/float(len(prediction_list))) * 100
    return accuracy_percentage

def confusion_matrix(predictions, actual_classes):
    #Fetching the name of the classes to dictionary and then to the list
    c =[0, 1, 2, 3, 4, 5 ,6, 7, 8, 9]
    length = len(c)


    #Creating confusion matrix as list -> empty list and hence comparing and increasing the count
    confusion_matrix=[]
    for i in range(length):
        for j in range(length):
            confusion_matrix.append(0)

    count = 0
    for i in range(len(actual_classes)):
        for j in range(length):
            for k in range(length):
                if actual_classes[i] == c[j] and predictions[i] == c[k]:
```

```
                count = count +1
                confusion_matrix[j*length+k] = confusion_matrix[j*length+k]+1

    print "\t\t"+'PREDICTED'
    table = []

    #Append Classes name
    L=[]
    L.append('\t')
    L.append('\t')
    for i in range(length):
        L.append(c[i])
    table.append(L)

    #Create Empty Table
    L=[]
    for i in range(length):
        for j in range(length+2):
            if i==length/2:
                if j==0:
                    L.append('ACTUAL')
                elif j==1:
                    L.append(c[i])
                else:
                    L.append('\t')
            else:
                if j==1:
                    L.append(c[i])
                else:
                    L.append('\t')
        table.append(L)
        L=[]

    #Populate value to the confusion matrix/empty table
    value_index=0
    for i in range(1, length+1):
        for j in range(2, length+2):
            table[i][j] = confusion_matrix[value_index]
            value_index+=1
    print tabulate(table, tablefmt="grid")

if __name__ == '__main__':
    path='.'
    test_img_fname = 't10k-images.idx3-ubyte'
    test_lbl_fname = 't10k-labels.idx1-ubyte'
    train_img_fname = 'train-images.idx3-ubyte'
    train_lbl_fname = 'train-labels.idx1-ubyte'
    test_images, test_labels, train_data, train_label = [],[],[],[]
    test_images, test_labels = load(os.path.join(path, test_img_fname),os.path.join(path,
test_lbl_fname))
```

```
    train_data, train_label = load(os.path.join(path, train_img_fname),os.path.join(path, train_lbl_fname))
    prediction_list = k_nearest_neighbours(train_data, test_images, train_label)
    accuracy = calculate_accuracy(test_labels, prediction_list)
    print "ACCURACY ",
    print accuracy,
    print "%"
    confusion_matrix(prediction_list, test_labels)
```

Output For value K=1:

```
6
9691
ACCURACY  96.91 %
                PREDICTED
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        |   |   0 |    1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        | 0 | 973 |    1 |   1 |   0 |   0 |   1 |   3 |   1 |   0 |   0 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        | 1 |   0 | 1129 |   3 |   0 |   1 |   1 |   1 |   0 |   0 |   0 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        | 2 |   7 |    6 | 992 |   5 |   1 |   0 |   2 |  16 |   3 |   0 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        | 3 |   0 |    1 |   2 | 970 |   1 |  19 |   0 |   7 |   7 |   3 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        | 4 |   0 |    7 |   0 |   0 | 944 |   0 |   3 |   5 |   1 |  22 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
| ACTUAL | 5 |   1 |    1 |   0 |  12 |   2 | 860 |   5 |   1 |   6 |   4 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        | 6 |   4 |    2 |   0 |   0 |   3 |   5 | 944 |   0 |   0 |   0 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        | 7 |   0 |   14 |   6 |   2 |   4 |   0 |   0 | 992 |   0 |  10 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        | 8 |   6 |    1 |   3 |  14 |   5 |  13 |   3 |   4 | 920 |   5 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
|        | 9 |   2 |    5 |   1 |   6 |  10 |   5 |   1 |  11 |   1 | 967 |
+--------+---+-----+------+-----+-----+-----+-----+-----+-----+-----+-----+
megha@megha-HP-Pavilion-g6-Notebook-PC:~/Downloads/SMAI/Assignments/Assignment
$
```

Output For value K=3:

```
6
9714
ACCURACY   97.14 %
                PREDICTED
```

		0	1	2	3	4	5	6	7	8	9
	0	974	1	1	0	0	1	2	1	0	0
	1	0	1133	2	0	0	0	0	0	0	0
	2	10	9	995	2	0	0	0	13	2	1
	3	0	1	4	974	1	12	1	7	4	6
	4	1	6	0	0	947	0	4	2	1	21
ACTUAL	5	6	1	0	11	2	858	5	1	4	4
	6	5	3	0	0	3	3	944	0	0	0
	7	0	21	4	0	1	0	0	991	0	11
	8	7	0	3	11	4	10	3	4	927	5
	9	4	4	1	5	9	2	1	8	4	971

```
megha@megha-HP-Pavilion-g6-Notebook-PC:~/Downloads/SMAI/Assignments/Assignment3$
```

Output For value K=5:

```
9694
ACCURACY   96.94 %
                PREDICTED
```

		0	1	2	3	4	5	6	7	8	9
	0	974	1	1	0	0	1	2	1	0	0
	1	0	1133	2	0	0	0	0	0	0	0
	2	11	8	990	2	1	0	1	15	4	0
	3	0	2	3	974	1	12	1	6	4	7
	4	3	7	0	0	943	0	4	2	1	22
ACTUAL	5	5	0	0	10	2	860	4	1	4	6
	6	5	3	0	0	3	2	945	0	0	0
	7	0	22	4	0	3	0	0	987	0	12
	8	7	3	4	12	4	8	4	5	922	5
	9	5	6	3	7	7	3	1	9	2	966

Analysis:

parameter(lmbda) is passed to the update_mini_batch function. Regulariztion : It's

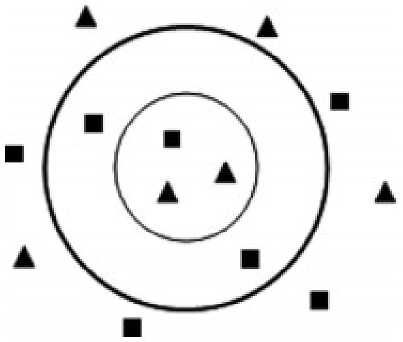


Fig. 3. KNN algorithm for a situation with 2 class and 2 features.

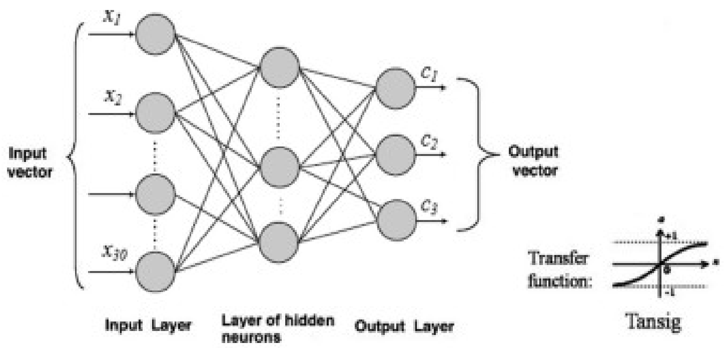


Fig. 4. The architecture of applied neural network.

For the results obtained on the classification on Multi Layer Neural Networks and KNN, the observations are :

- The accuracy of knn was good for the given mnist databse. So KNN gives better results in terms of accuracy than the MLFNN.

- MLFNN results in faster results after training the machine.

- KNN is a lazy learner(unsupervised learning) in slow prediction rate on the big data set line MNIST

- Finally, KNN and ANN were generated by training with training data set and simultaneously simulated by testing data set, we can see that the best performance of KNN on given Data-set was 96% and whereas of MLFNN was 95% with epoch(20) and learning rate 3.0.

- For K values tested with k = 1, 3, 5. The value k = 3 is giving the best accuracy for the mnist dataset

QUESTION-3 : ***Now, try one variation of preprocessing using deskewing, adding noise, etc. and also try one variation with other objective functions / regularization terms such as cross entropy, weight decay, tangentprop, etc. and compare your results from (1) above. Present an analysis and discussion of your results.***

Variation – I (Weight Decay Function):

1) It makes use of new and improved aproach to weight initialisation. The weights input to a neuron are initialized as Gaussian random variables with mean 0 and standard deviation 1 divided by the square root of the number of connections input to the neuron.

2) The real work is done by modifying the gradient descent update rule to include weight decay. Although the modification is tiny, it has a big impact on results! The basic approach is to start with a network with “too many” weights and “decay” all weights during training.

3) A regularization parameter(lmbda) is passed to the update_mini_batch function. Regulariztion : It's conceptually quite subtle and difficult to understand. And yet it was trivial to add to our program! It occurs surprisingly often that sophisticated techniques can be implemented with small changes to code.

Updations:

The following function is updated to update the weight in a different way at each step. It also takes lmbda and n as a parameter which also influence the training of the network.

Code:

```
def update_mini_batch(mini_batch, eta, biases, weights, lmbda, n):
    nabla_b = [np.zeros(b.shape) for b in biases]
    nabla_w = [np.zeros(w.shape) for w in weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = backprop(x, y, biases, weights)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

     weights = [(1-eta*(lmbda/n))*w-(eta/len(mini_batch))*nw for w, nw in zip(weights, nabla_w)]
    biases = [b-(eta/len(mini_batch))*nb for b, nb in zip(biases, nabla_b)]

    return biases, weights
```

Analysis:

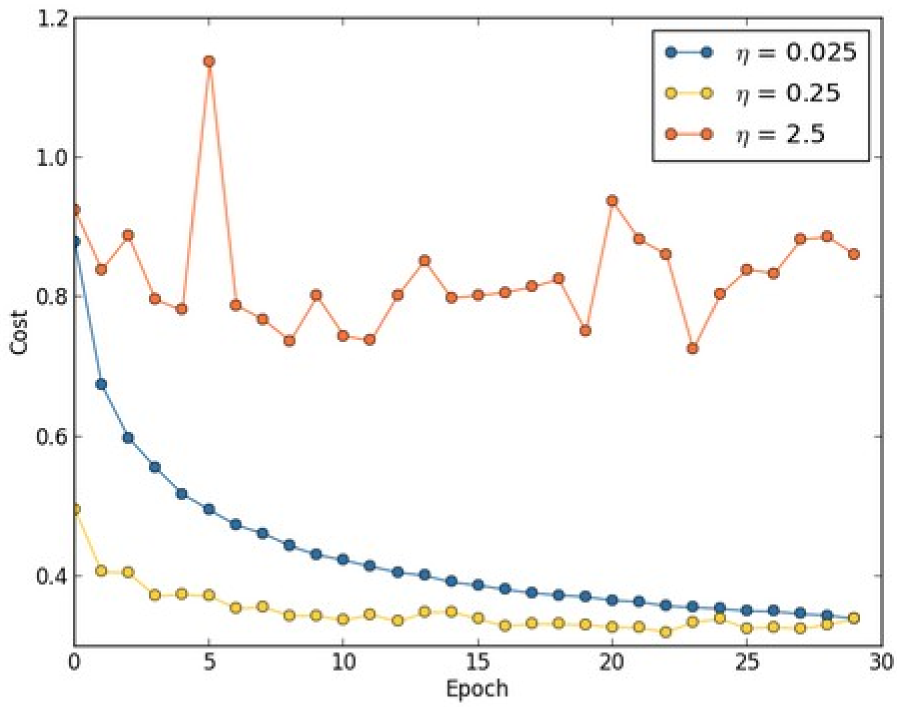
If $\boldsymbol{\eta}$**=10.0 and** $\boldsymbol{\lambda}$**=1000.00**, we hardly gets an accuracy of 9-10%:

Sample Output:
****

======= FOLD NO 1=====
Training Iteration 0: 1064 / 10000
Training Iteration 1: 961 / 10000
Training Iteration 2: 1064 / 10000
Training Iteration 3: 961 / 10000
Training Iteration 4: 1064 / 10000
Training Iteration 5: 1009 / 10000
Training Iteration 6: 1064 / 10000
Training Iteration 7: 983 / 10000
Training Iteration 8: 1064 / 10000
Training Iteration 9: 1064 / 10000
Training Iteration 10: 961 / 10000

"Well, that's easy to fix," -> "just decrease the learning rate and regularization hyper-parameters". Our classification accuracies are no better than chance! Our network is acting as a random noise generator!

Learning rate:
Suppose we run three MNIST networks with three different learning rates, η=0.025, η=0.25 and η=2.5, respectively. We'll set the other hyper-parameters as for the experiments in earlier sections, running over 30 epochs, with a mini-batch size of 10, and with λ=5.0. We'll also return to using the full 50,000 training images.



- With η=0.025 the cost decreases smoothly until the final epoch.
- With η=0.25 the cost initially decreases, but after about 20 epochs it is near saturation, and thereafter most of the changes are merely small and apparently random oscillations.

- With η=2.5 the cost makes large oscillations right from the start.

If learning rate - $\boldsymbol{\eta}$=0.5 and $\boldsymbol{\lambda}$=5.0, We gets an accuracy of more than 90%.
So the regulariztion parameter along with weight decay function is effecting our final accuracy and the converge to a large extent.

****
=============FOLD NO 1=============
Training Iteration 0: 8574 / 10000
Training Iteration 1: 9044 / 10000
Training Iteration 2: 9195 / 10000
Training Iteration 3: 9297 / 10000
Training Iteration 4: 9379 / 10000
Training Iteration 5: 9434 / 10000
Training Iteration 6: 9479 / 10000
Training Iteration 7: 9489 / 10000
Training Iteration 8: 9504 / 10000
Training Iteration 9: 9539 / 10000
Training Iteration 10: 9515 / 10000
Training Iteration 11: 9553 / 10000
Training Iteration 12: 9544 / 10000
Training Iteration 13: 9576 / 10000
Training Iteration 14: 9545 / 10000
Training Iteration 15: 9575 / 10000
Training Iteration 16: 9582 / 10000
Training Iteration 17: 9586 / 10000
Training Iteration 18: 9578 / 10000
Training Iteration 19: 9571 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	973	0	7	4	1	7	3	3	3	6
	1	0	1048	2	2	6	3	0	7	9	2
	2	4	5	939	13	2	7	1	6	7	2
	3	0	2	2	969	1	17	0	2	4	12
	4	0	1	7	0	938	5	3	4	2	15
ACTUAL	5	1	1	2	11	0	840	1	0	3	2
	6	3	0	6	0	5	17	954	0	6	1
	7	1	2	9	4	2	1	0	1055	6	17
	8	6	4	13	20	3	10	5	0	957	6
	9	3	1	3	7	25	8	0	13	12	898

Accuracy : 95.71
Error Rate : 4.29

****
=============FOLD NO 2=============
Training Iteration 0: 9550 / 10000
Training Iteration 1: 9538 / 10000
Training Iteration 2: 9522 / 10000
Training Iteration 3: 9538 / 10000
Training Iteration 4: 9524 / 10000
Training Iteration 5: 9536 / 10000
Training Iteration 6: 9539 / 10000
Training Iteration 7: 9527 / 10000
Training Iteration 8: 9523 / 10000
Training Iteration 9: 9522 / 10000
Training Iteration 10: 9549 / 10000
Training Iteration 11: 9544 / 10000
Training Iteration 12: 9552 / 10000
Training Iteration 13: 9557 / 10000
Training Iteration 14: 9537 / 10000
Training Iteration 15: 9527 / 10000
Training Iteration 16: 9542 / 10000
Training Iteration 17: 9532 / 10000
Training Iteration 18: 9529 / 10000

Training Iteration 19: 9542 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	973	0	7	4	1	7	3	3	3	6
	1	0	1048	2	2	6	3	0	7	9	2
	2	4	5	939	13	2	7	1	6	7	2
	3	0	2	2	969	1	17	0	2	4	12
	4	0	1	7	0	938	5	3	4	2	15
ACTUAL	5	1	1	2	11	0	840	1	0	3	2
	6	3	0	6	0	5	17	954	0	6	1
	7	1	2	9	4	2	1	0	1055	6	17
	8	6	4	13	20	3	10	5	0	957	6
	9	3	1	3	7	25	8	0	13	12	898

Accuracy : 95.71
Error Rate : 4.29

****
============FOLD NO 3============
Training Iteration 0: 9532 / 10000
Training Iteration 1: 9538 / 10000
Training Iteration 2: 9516 / 10000
Training Iteration 3: 9518 / 10000
Training Iteration 4: 9520 / 10000
Training Iteration 5: 9527 / 10000
Training Iteration 6: 9488 / 10000
Training Iteration 7: 9516 / 10000
Training Iteration 8: 9538 / 10000
Training Iteration 9: 9503 / 10000
Training Iteration 10: 9503 / 10000
Training Iteration 11: 9519 / 10000
Training Iteration 12: 9513 / 10000
Training Iteration 13: 9530 / 10000
Training Iteration 14: 9518 / 10000
Training Iteration 15: 9523 / 10000
Training Iteration 16: 9492 / 10000
Training Iteration 17: 9476 / 10000
Training Iteration 18: 9502 / 10000
Training Iteration 19: 9487 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	957	0	6	3	0	3	7	0	4	8
	1	0	1132	5	4	4	2	3	3	9	6
	2	7	6	873	28	5	3	1	8	7	3
	3	0	3	5	962	1	8	0	0	13	17
	4	1	1	12	1	912	1	2	6	2	20
ACTUAL	5	3	3	0	6	0	862	4	0	2	7
	6	13	1	4	4	13	16	934	0	8	2
	7	3	1	12	10	9	4	0	999	2	24
	8	7	7	18	19	7	8	6	3	924	17
	9	2	0	3	7	14	5	0	4	7	932

Accuracy : 94.87
Error Rate : 5.13

****

============FOLD NO 4============

Training Iteration 0: 9554 / 10000
Training Iteration 1: 9540 / 10000
Training Iteration 2: 9534 / 10000
Training Iteration 3: 9523 / 10000
Training Iteration 4: 9528 / 10000
Training Iteration 5: 9531 / 10000
Training Iteration 6: 9505 / 10000
Training Iteration 7: 9509 / 10000
Training Iteration 8: 9510 / 10000
Training Iteration 9: 9522 / 10000
Training Iteration 10: 9522 / 10000
Training Iteration 11: 9523 / 10000
Training Iteration 12: 9508 / 10000
Training Iteration 13: 9522 / 10000
Training Iteration 14: 9516 / 10000
Training Iteration 15: 9517 / 10000
Training Iteration 16: 9506 / 10000
Training Iteration 17: 9511 / 10000
Training Iteration 18: 9530 / 10000
Training Iteration 19: 9515 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	940	1	7	3	2	7	4	2	0	4
	1	1	1120	5	7	3	1	3	7	20	4
	2	2	11	972	21	6	3	2	13	10	1
	3	1	3	4	927	0	12	0	1	8	17
	4	0	2	9	2	938	8	4	6	3	21
ACTUAL	5	1	1	2	18	1	878	15	0	9	3
	6	12	1	5	1	4	16	973	2	16	1
	7	0	2	12	12	4	1	0	967	2	14
	8	10	1	3	5	2	2	3	0	881	5
	9	0	0	0	1	21	6	0	16	4	919

Accuracy : 95.15
Error Rate : 4.85

****
=============FOLD NO 5=============
Training Iteration 0: 9590 / 10000
Training Iteration 1: 9539 / 10000
Training Iteration 2: 9538 / 10000
Training Iteration 3: 9515 / 10000
Training Iteration 4: 9539 / 10000
Training Iteration 5: 9515 / 10000
Training Iteration 6: 9502 / 10000
Training Iteration 7: 9505 / 10000
Training Iteration 8: 9528 / 10000
Training Iteration 9: 9526 / 10000
Training Iteration 10: 9515 / 10000
Training Iteration 11: 9535 / 10000
Training Iteration 12: 9529 / 10000
Training Iteration 13: 9502 / 10000
Training Iteration 14: 9519 / 10000
Training Iteration 15: 9505 / 10000
Training Iteration 16: 9508 / 10000
Training Iteration 17: 9531 / 10000
Training Iteration 18: 9518 / 10000
Training Iteration 19: 9492 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	932	0	5	1	2	6	7	4	4	4
	1	0	1115	1	2	1	5	2	8	7	3
	2	2	10	960	34	7	5	7	6	20	2
	3	1	3	1	906	1	5	0	2	5	9
	4	3	3	7	0	922	4	2	7	3	11
ACTUAL	5	7	1	1	26	0	844	15	1	9	5
	6	9	2	4	3	16	6	960	0	5	0
	7	2	4	10	14	5	2	0	968	0	12
	8	6	1	5	13	2	11	7	2	927	9
	9	1	1	1	9	27	7	0	20	5	958

Accuracy : 94.92
Error Rate : 5.08

****

============FOLD NO 6============

Training Iteration 0: 9531 / 10000
Training Iteration 1: 9527 / 10000
Training Iteration 2: 9492 / 10000
Training Iteration 3: 9497 / 10000
Training Iteration 4: 9506 / 10000
Training Iteration 5: 9517 / 10000
Training Iteration 6: 9484 / 10000
Training Iteration 7: 9515 / 10000
Training Iteration 8: 9524 / 10000
Training Iteration 9: 9462 / 10000
Training Iteration 10: 9500 / 10000
Training Iteration 11: 9495 / 10000
Training Iteration 12: 9486 / 10000
Training Iteration 13: 9506 / 10000
Training Iteration 14: 9482 / 10000
Training Iteration 15: 9479 / 10000
Training Iteration 16: 9482 / 10000
Training Iteration 17: 9496 / 10000
Training Iteration 18: 9476 / 10000
Training Iteration 19: 9484 / 10000

PREDICTED

		0	1	2	3	4	5	6	7	8	9
	0	989	0	20	7	6	6	9	6	14	4
	1	1	1087	4	2	0	1	3	4	8	0
	2	0	9	951	22	0	1	1	12	5	1
	3	0	3	5	913	0	5	0	1	2	11
	4	2	2	10	2	902	1	5	5	1	21
ACTUAL	5	3	2	0	25	0	859	6	2	9	9
	6	2	0	7	6	8	9	944	0	2	0
	7	0	5	13	15	3	6	0	1012	1	29
	8	11	3	14	24	4	8	8	1	937	7
	9	0	4	1	4	27	6	0	7	3	890

Accuracy : 94.84
Error Rate : 5.16

****
**************Average Values of FOLDS*****************

Average Accuracy : 95.1516666667
Average Error Rate : 4.84833333333
Standard Deviation of Error Rate : 0.319865423091
****

===
***************TESTING the Network*******************
===

Sum : 9581 / 10000

		PREDICTED									
		0	1	2	3	4	5	6	7	8	9
	0	969	0	14	3	1	8	12	3	8	11
	1	0	1112	1	1	0	1	3	9	2	4
	2	0	3	976	14	4	1	2	22	1	0
	3	0	3	3	953	0	10	1	3	3	9
	4	0	1	8	2	944	3	4	3	4	14
ACTUAL	5	3	0	0	16	0	846	6	0	5	9
	6	3	4	4	1	11	11	923	0	4	1
	7	1	1	8	11	2	1	1	976	5	11
	8	3	11	18	6	5	6	6	2	941	9
	9	1	0	0	3	15	5	0	10	1	941

Analysis:

Why Weight Decay is Used / Effect of weight decay:-

- To avoid over-fitting, it is required to impose a heuristic that the weights should be small.
- The weight is decreased by a small **weight decay factor** during each epoch.
- Larger weights are needed to accommodate outliers in the data.
- Large weights can hurt generalization in two different ways.
    - Excessively large weights leading to hidden units can cause the output function to be too rough, possibly with near discontinuities.
    - Excessively large weights leading to output units can cause wild outputs far beyond the range of the data if the output activation function is not bounded to the same range as the data. To put it another way, large weights can cause excessive variance of the output
    - The weight decay penalty term causes the weights to converge to smaller absolute values than they otherwise would.

Disadvantage of Weight Decay:-

- Different types of weights in the network will usually require different decay constants for good generalization.
- At the very least, we need three different decay constants for input-to-hidden, hidden-to-hidden, and hidden-to-output weights.
- Adjusting all these decay constants to produce the best estimated generalization error often requires vast amounts of computation.

Code Analysis:-

- The accuracy of all the folds are in range (94.8 to 95.8 %) in contrast to the question 1, where accuracies were varying in a quite a large range from 90 to 97%. This happened becuase of the fundamenetal principle of weight decay i.e. keeping the weights low with a weight decay factor helps to steer the network from overfitting.
- Weight Decay suppresses any irrelevant components of the weight vector by chosing the smallest vector that solves the learning problem.
- Therefore, it leads to almost same accuracy in each fold.

Variation – 2 (Adding Noise to the dataset):

In every instance of the train data, a float value of 0.5 is added.

```
print "ADDING NOISE................"
   for i in range(len(train_data)):
      for j in range(len(train_data[i])):
         train_data[i][j]= train_data[i][j] + 0.5;
   print train_data
```

This has reduced the accuracy of the test data to range of 80-90%. (In my code, increasing with each fold, still reaching a threshold)

```
ADDING NOISE................
[[ 0.5  0.5  0.5 ...,  0.5  0.5  0.5]
 [ 0.5  0.5  0.5 ...,  0.5  0.5  0.5]
 [ 0.5  0.5  0.5 ...,  0.5  0.5  0.5]
 ...,
 [ 0.5  0.5  0.5 ...,  0.5  0.5  0.5]
 [ 0.5  0.5  0.5 ...,  0.5  0.5  0.5]
 [ 0.5  0.5  0.5 ...,  0.5  0.5  0.5]]

***********************************
============FOLD NO 1=============
Training Iteration 0: 8201 / 10000
Training Iteration 1: 8378 / 10000
Training Iteration 2: 8553 / 10000
Training Iteration 3: 8218 / 10000
Training Iteration 4: 8089 / 10000
Training Iteration 5: 8419 / 10000
Training Iteration 6: 8682 / 10000
Training Iteration 7: 8680 / 10000
Training Iteration 8: 8708 / 10000
Training Iteration 9: 8876 / 10000
Training Iteration 10: 8802 / 10000
Training Iteration 11: 8339 / 10000
Training Iteration 12: 8719 / 10000
Training Iteration 13: 8976 / 10000
Training Iteration 14: 8788 / 10000
```

Training Iteration 15: 8980 / 10000
Training Iteration 16: 8964 / 10000
Training Iteration 17: 8904 / 10000
Training Iteration 18: 8953 / 10000
Training Iteration 19: 8864 / 10000
Accuracy : 88.64
Error Rate : 11.36

****
============FOLD NO 2============
Training Iteration 0: 9004 / 10000
Training Iteration 1: 8874 / 10000
Training Iteration 2: 8743 / 10000
Training Iteration 3: 8457 / 10000
Training Iteration 4: 8896 / 10000
Training Iteration 5: 8929 / 10000
Training Iteration 6: 8960 / 10000
Training Iteration 7: 8517 / 10000
Training Iteration 8: 8626 / 10000
Training Iteration 9: 8938 / 10000
Training Iteration 10: 9036 / 10000
Training Iteration 11: 8677 / 10000
Training Iteration 12: 8928 / 10000
Training Iteration 13: 8903 / 10000
Training Iteration 14: 8820 / 10000
Training Iteration 15: 8810 / 10000
Training Iteration 16: 8835 / 10000
Training Iteration 17: 8864 / 10000
Training Iteration 18: 8956 / 10000
Training Iteration 19: 8854 / 10000
Accuracy : 88.54
Error Rate : 11.46

****
============FOLD NO 3============
Training Iteration 0: 8545 / 10000
Training Iteration 1: 8954 / 10000
Training Iteration 2: 8787 / 10000
Training Iteration 3: 8843 / 10000
Training Iteration 4: 8932 / 10000
Training Iteration 5: 8903 / 10000
Training Iteration 6: 8978 / 10000
Training Iteration 7: 8739 / 10000
Training Iteration 8: 9070 / 10000
Training Iteration 9: 8910 / 10000
Training Iteration 10: 8941 / 10000
Training Iteration 11: 8955 / 10000
Training Iteration 12: 8997 / 10000

Training Iteration 13: 8834 / 10000
Training Iteration 14: 8943 / 10000
Training Iteration 15: 9027 / 10000
Training Iteration 16: 9018 / 10000
Training Iteration 17: 8993 / 10000
Training Iteration 18: 8919 / 10000
Training Iteration 19: 9017 / 10000
Accuracy : 90.17
Error Rate : 9.83

****
============FOLD NO 4============
Training Iteration 0: 9053 / 10000
Training Iteration 1: 9026 / 10000
Training Iteration 2: 9009 / 10000
Training Iteration 3: 9075 / 10000
Training Iteration 4: 9051 / 10000
Training Iteration 5: 9125 / 10000
Training Iteration 6: 9079 / 10000
Training Iteration 7: 9022 / 10000
Training Iteration 8: 8979 / 10000
Training Iteration 9: 9066 / 10000
Training Iteration 10: 9081 / 10000
Training Iteration 11: 9029 / 10000
Training Iteration 12: 9062 / 10000
Training Iteration 13: 9061 / 10000
Training Iteration 14: 8934 / 10000
Training Iteration 15: 8991 / 10000
Training Iteration 16: 9022 / 10000
Training Iteration 17: 8978 / 10000
Training Iteration 18: 8999 / 10000
Training Iteration 19: 9063 / 10000
Accuracy : 90.63
Error Rate : 9.37

****
============FOLD NO 5============
Training Iteration 0: 8985 / 10000
Training Iteration 1: 8938 / 10000
Training Iteration 2: 8861 / 10000
Training Iteration 3: 8971 / 10000
Training Iteration 4: 8889 / 10000
Training Iteration 5: 9094 / 10000
Training Iteration 6: 9057 / 10000
Training Iteration 7: 9027 / 10000
Training Iteration 8: 8982 / 10000
Training Iteration 9: 9065 / 10000
Training Iteration 10: 9037 / 10000

Training Iteration 11: 9082 / 10000
Training Iteration 12: 8993 / 10000
Training Iteration 13: 9016 / 10000
Training Iteration 14: 9058 / 10000
Training Iteration 15: 9054 / 10000
Training Iteration 16: 8884 / 10000
Training Iteration 17: 8996 / 10000
Training Iteration 18: 8854 / 10000
Training Iteration 19: 8776 / 10000
Accuracy : 87.76
Error Rate : 12.24

****
============FOLD NO 6============
Training Iteration 0: 9080 / 10000
Training Iteration 1: 8929 / 10000
Training Iteration 2: 8980 / 10000
Training Iteration 3: 9089 / 10000
Training Iteration 4: 9039 / 10000
Training Iteration 5: 9113 / 10000
Training Iteration 6: 9077 / 10000
Training Iteration 7: 9037 / 10000
Training Iteration 8: 9074 / 10000
Training Iteration 9: 9000 / 10000
Training Iteration 10: 9008 / 10000
Training Iteration 11: 9006 / 10000
Training Iteration 12: 9053 / 10000
Training Iteration 13: 9019 / 10000
Training Iteration 14: 9089 / 10000
Training Iteration 15: 9089 / 10000
Training Iteration 16: 9032 / 10000
Training Iteration 17: 9016 / 10000
Training Iteration 18: 9032 / 10000
Training Iteration 19: 9093 / 10000
Accuracy : 90.93
Error Rate : 9.07

****
*************Average Values of FOLDS****************

Average Accuracy : 89.445
Average Error Rate : 10.555
Standard Deviation of Error Rate : 1.18612464213
****

```
================================================
****************TESTING the Network********************
================================================

Sum : 8793/ 10000
Accuracy : 87.93
Error Rate : 12.07
*******************************************************
```