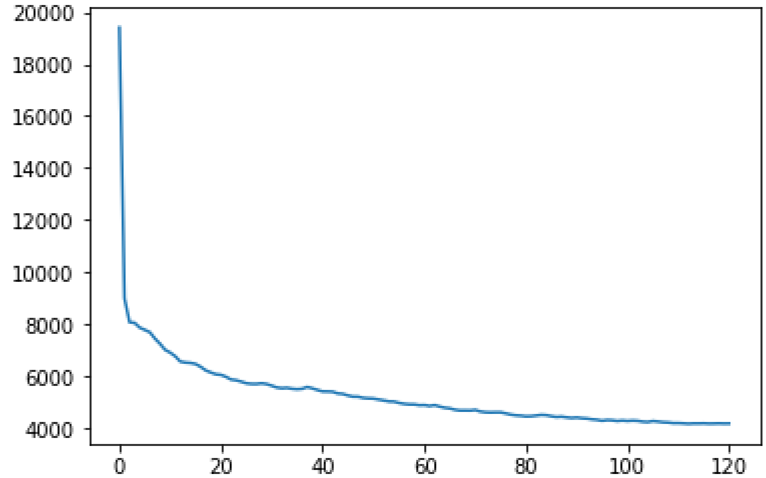
```python
import numpy as np
import matplotlib.pyplot as plt
import struct as st

X_train = read_files('train-images-idx3-ubyte/train-images.idx3-ubyte') #REading the files
y_train = read_files('train-labels-idx1-ubyte/train-labels.idx1-ubyte')
X_test = read_files('t10k-images-idx3-ubyte/t10k-images.idx3-ubyte')
y_test = read_files('t10k-labels-idx1-ubyte/t10k-labels.idx1-ubyte')

y_train_original_form = y_train
y_test_original_form = y_test

def read_files(file): #Function to read files
    with open(file, 'rb') as file:
        zero, data_type, dims = st.unpack('>HBB', file.read(4))
        shape = tuple(st.unpack('>I', file.read(4))[0] for d in range(dims))
        return np.frombuffer(file.read(), dtype=np.uint8).reshape(shape)

y_train = convert_onehot_vectors(y_train)
y_test = convert_onehot_vectors(y_test)

def convert_onehot_vectors(labels): #Function to convert to one hot notation
    vct = np.zeros((labels.size, labels.max()+1))
    vct[np.arange(labels.size), labels] = 1
    return vct
nodes0 = 784            #DEfines the network architecture
nodes1 = 128
nodes2 = 128
nodes3 = 10
def sigmoid_act(val):   # function to implement activation function
    return 1 / (1 + np.exp(-val))
W1 = np.random.normal(0, 1 , (nodes1, nodes0 + 1))# Randomly initilizing the weights
W2 = np.random.normal(0, 1 , (nodes2, nodes1 + 1))
W3 = np.random.normal(0, 1 , (nodes3, nodes2 + 1))
V1 = np.zeros(W1.shape) #Randomly initializing the local fields
V2 = np.zeros(W2.shape)
V3 = np.zeros(W3.shape)
learning_rate = 0.1
train_errors = []   #Containers for training and tesing stats
test_errors = []
energy_train = []
energy_test = []
MSE = []
MSE_test = []
epoch = 0
n = 60000
n_test = 10000
unit_vector = np.array([1]).reshape(1, 1)
while (True):       #Loop for implimenting the training of neural network
    y_3s = []
    y_3s_test = []
    train_accuracy = 0
    Energy_train_total = 0
    for i in range(n):
        y0 = np.array(X_train[i]).reshape(784, 1)
        int_y0 = np.vstack((unit_vector, y0))
        v_1 = W1 @ int_y0
```

```
        y_1 = sigmoid_act(v_1)
        int_y1 = np.vstack((unit_vector, y_1))
        v_2 = W2 @ int_y1
        y_2 = sigmoid_act(v_2)
        int_y2 = np.vstack((unit_vector, y_2))
        v_3 = W3 @ int_y2
        y_3 = sigmoid_act(v_3)
        y_3s.append(y_3)
        check = (np.argmax(y_3) == y_train_original_form[i])
        if check == True:
            train_accuracy += 1
        Energy_train_total += np.sum((y_train[i].reshape(y_3.shape) - y_3) ** 2) / (2 * n)
        d_3 = np.multiply((y_train[i].reshape(y_3.shape) - y_3), (sigmoid_act(v_3) * (1 - sigmoid_a
        d_2 = np.multiply((np.transpose(W3) @ d_3)[1:, :], (sigmoid_act(v_2) * (1 - sigmoid_act(v_2
        d_1 = np.multiply((np.transpose(W2) @ d_2)[1:, :], (sigmoid_act(v_1) * (1 - sigmoid_act(v_1
        de_dW1 = -d_1 @ int_y0.transpose()
        de_dW2 = -d_2 @ int_y1.transpose()
        de_dW3 = -d_3 @ int_y2.transpose()
        V1 = np.subtract(np.multiply(0.15, V1), np.multiply(learning_rate, de_dW1))
        V2 = np.subtract(np.multiply(0.15, V2), np.multiply(learning_rate, de_dW2))
        V3 = np.subtract(np.multiply(0.15, V3), np.multiply(learning_rate, de_dW3))
        W1 = W1 + V1
        W2 = W2 + V2
        W3 = W3 + V3
    energy_train.append(Energy_train_total)
    train_errors.append(n - train_accuracy)
    Energy_test_total = 0
    test_accuracy = 0
    for i in range(n_test):
        y0 = np.array(X_test[i]).reshape(784, 1)
        int_y0 = np.vstack((unit_vector, y0))
        v_1 = W1 @ int_y0
        y_1 = sigmoid_act(v_1)
        int_y1 = np.vstack((unit_vector, y_1))
        v_2 = W2 @ int_y1
        y_2 = sigmoid_act(v_2)
        int_y2 = np.vstack((unit_vector, y_2))
        v_3 = W3 @ int_y2
        y_3 = sigmoid_act(v_3)
        y_3s_test.append(y_3)
        check = (np.argmax(y_3) == y_test_original_form[i])
        if check == True:
            test_accuracy += 1
        Energy_test_total += np.sum((y_test[i].reshape(y_3.shape) - y_3) ** 2) / (2 * n)
    energy_test.append(Energy_test_total)
    test_errors.append(n_test - test_accuracy)
    sum = 0
    for i in range(n):
        sum += np.square(np.subtract(y_train[i].reshape(y_3.shape),y_3s[i])).mean()
    mse = (sum / (n))
    MSE.append(mse)
    sum = 0
    for i in range(n_test):
        sum += np.square(np.subtract(y_test[i].reshape(y_3.shape),y_3s_test[i])).mean()
    mse_test = (sum / (n_test))
    MSE_test.append(mse_test)
    if epoch != 0:
```

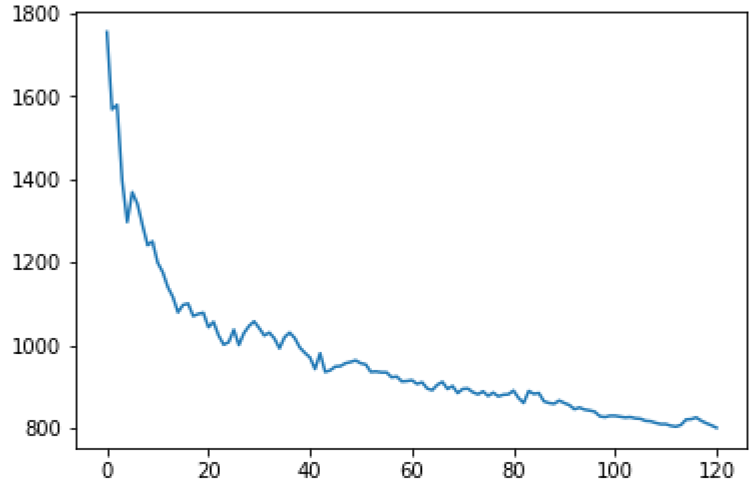
```
        if MSE[epoch] > MSE[epoch - 1]:
            learning_rate = learning_rate * 0.9
        if (test_accuracy / n_test) >= 0.95 or (train_accuracy/n) >0.96:
            break
    print ("Test accuracy ",test_accuracy/n_test," at epoch", epoch)
    epoch += 1
range_epoch = [i for i in range(0,epoch+1)] #plots the required graph for the evaluation of model
print("Following is a graph showing the relation between epochs and train errors")
plt.plot(range_epoch,train_errors)
plt.show()
print("Following is a graph showing the relation between epochs and test errors")
plt.plot(range_epoch,test_errors)
plt.show()
print("Following is a graph showing the relation between epochs and train energy")
plt.plot(range_epoch,energy_train)
plt.show()
print("Following is a graph showing the relation between epochs and train energy")
plt.plot(range_epoch,energy_test)
plt.show()
# Printing model stats
print('Accuracy on training set ', train_accuracy / n)
print('Accuracy on testing set ', test_accuracy / n_test)
print('Mean Squared error of training dataset is ', MSE[-1])
print('Mean Squared error of testing dataset is ', MSE_test[-1])
print('learning_rate used is', learning_rate)
print('Total number if epochs ', epoch)
```

```
Python 3.7.4 (default, Aug  9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.8.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/kaush/Downloads/NN/test.py', wdir='C:/Users/kaush/Downloads/NN')
Test accuracy  0.8245  at epoch 0
Test accuracy  0.8432  at epoch 1
Test accuracy  0.8421  at epoch 2
Test accuracy  0.8603  at epoch 3
Test accuracy  0.8704  at epoch 4
Test accuracy  0.8632  at epoch 5
Test accuracy  0.866  at epoch 6
Test accuracy  0.8711  at epoch 7
Test accuracy  0.8759  at epoch 8
Test accuracy  0.8751  at epoch 9
Test accuracy  0.8801  at epoch 10
Test accuracy  0.8824  at epoch 11
Test accuracy  0.886  at epoch 12
Test accuracy  0.8884  at epoch 13
Test accuracy  0.8921  at epoch 14
Test accuracy  0.8903  at epoch 15
Test accuracy  0.89  at epoch 16
Test accuracy  0.893  at epoch 17
Test accuracy  0.8925  at epoch 18
Test accuracy  0.8922  at epoch 19
Test accuracy  0.8957  at epoch 20
Test accuracy  0.8944  at epoch 21
Test accuracy  0.8977  at epoch 22
Test accuracy  0.8999  at epoch 23
Test accuracy  0.8993  at epoch 24
Test accuracy  0.8963  at epoch 25
Test accuracy  0.9  at epoch 26
Test accuracy  0.8971  at epoch 27
Test accuracy  0.8954  at epoch 28
Test accuracy  0.8943  at epoch 29
Test accuracy  0.8959  at epoch 30
Test accuracy  0.8977  at epoch 31
Test accuracy  0.897  at epoch 32
Test accuracy  0.8984  at epoch 33
Test accuracy  0.9008  at epoch 34
Test accuracy  0.8981  at epoch 35
Test accuracy  0.897  at epoch 36
Test accuracy  0.8983  at epoch 37
Test accuracy  0.9006  at epoch 38
Test accuracy  0.9019  at epoch 39
Test accuracy  0.903  at epoch 40
Test accuracy  0.9058  at epoch 41
Test accuracy  0.902  at epoch 42
Test accuracy  0.9065  at epoch 43
Test accuracy  0.9061  at epoch 44
Test accuracy  0.9052  at epoch 45
Test accuracy  0.9051  at epoch 46
Test accuracy  0.9044  at epoch 47
Test accuracy  0.9041  at epoch 48
Test accuracy  0.9037  at epoch 49
Test accuracy  0.9044  at epoch 50
Test accuracy  0.9047  at epoch 51
Test accuracy  0.9065  at epoch 52
Test accuracy  0.9064  at epoch 53
Test accuracy  0.9066  at epoch 54
Test accuracy  0.9066  at epoch 55
Test accuracy  0.9078  at epoch 56
Test accuracy  0.9076  at epoch 57
```

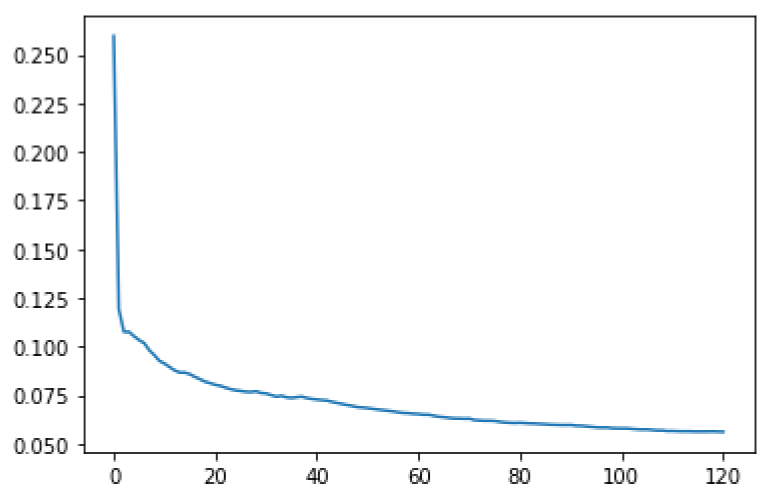
```
Test accuracy  0.9088  at epoch 58
Test accuracy  0.9087  at epoch 59
Test accuracy  0.9085  at epoch 60
Test accuracy  0.9094  at epoch 61
Test accuracy  0.909  at epoch 62
Test accuracy  0.9105  at epoch 63
Test accuracy  0.9109  at epoch 64
Test accuracy  0.9096  at epoch 65
Test accuracy  0.9089  at epoch 66
Test accuracy  0.9106  at epoch 67
Test accuracy  0.9099  at epoch 68
Test accuracy  0.9116  at epoch 69
Test accuracy  0.9106  at epoch 70
Test accuracy  0.9105  at epoch 71
Test accuracy  0.9114  at epoch 72
Test accuracy  0.9119  at epoch 73
Test accuracy  0.9112  at epoch 74
Test accuracy  0.9123  at epoch 75
Test accuracy  0.9115  at epoch 76
Test accuracy  0.9124  at epoch 77
Test accuracy  0.912  at epoch 78
Test accuracy  0.9119  at epoch 79
Test accuracy  0.911  at epoch 80
Test accuracy  0.9128  at epoch 81
Test accuracy  0.914  at epoch 82
Test accuracy  0.9111  at epoch 83
Test accuracy  0.9118  at epoch 84
Test accuracy  0.9116  at epoch 85
Test accuracy  0.9136  at epoch 86
Test accuracy  0.914  at epoch 87
Test accuracy  0.9142  at epoch 88
Test accuracy  0.9134  at epoch 89
Test accuracy  0.914  at epoch 90
Test accuracy  0.9145  at epoch 91
Test accuracy  0.9155  at epoch 92
Test accuracy  0.9151  at epoch 93
Test accuracy  0.9156  at epoch 94
Test accuracy  0.9158  at epoch 95
Test accuracy  0.9161  at epoch 96
Test accuracy  0.9172  at epoch 97
Test accuracy  0.9174  at epoch 98
Test accuracy  0.9171  at epoch 99
Test accuracy  0.9171  at epoch 100
Test accuracy  0.9173  at epoch 101
Test accuracy  0.9175  at epoch 102
Test accuracy  0.9174  at epoch 103
Test accuracy  0.9177  at epoch 104
Test accuracy  0.9178  at epoch 105
Test accuracy  0.9183  at epoch 106
Test accuracy  0.9184  at epoch 107
Test accuracy  0.9188  at epoch 108
Test accuracy  0.9191  at epoch 109
Test accuracy  0.9191  at epoch 110
Test accuracy  0.9195  at epoch 111
Test accuracy  0.9197  at epoch 112
Test accuracy  0.9193  at epoch 113
Test accuracy  0.918  at epoch 114
Test accuracy  0.9179  at epoch 115
Test accuracy  0.9175  at epoch 116
Test accuracy  0.9183  at epoch 117
Test accuracy  0.9189  at epoch 118
Test accuracy  0.9194  at epoch 119
Following is a graph showing the relation between epochs and train errors
```



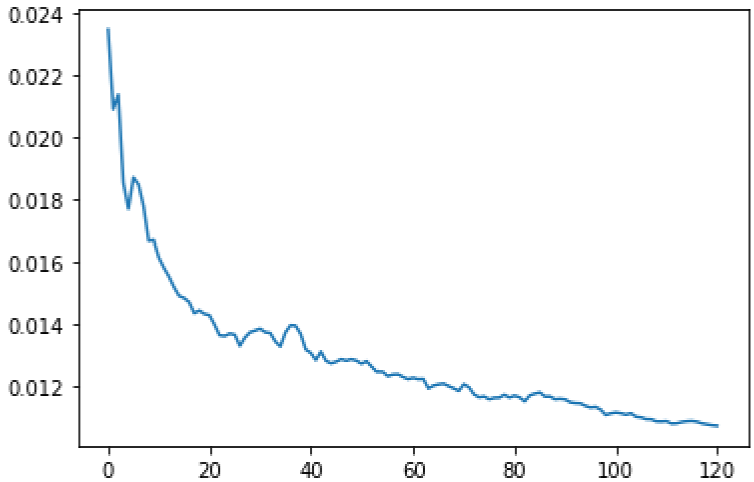
Following is a graph showing the relation between epochs and test errors



Following is a graph showing the relation between epochs and train energy



Following is a graph showing the relation between epochs and train energy




```
Accuracy on training set 96.42
Accuracy on testing set 95.66
Mean Squared error of training dataset is  0.011233402398398368
Mean Squared error of testing dataset is  0.012845487741888737
learning_rate used is 0.01667718169966658
Total number if epochs  120
```

In [2]:

Neural Networks
Assignment 5
Report

1 (i) As per the network topology, I am using one input layer, 2 hidden layers and one output layer. ~~The in~~ In the input layer I have used 784 neurons, in each hidden layer I have used 128 neurons and in the output layer I have used 10 neurons.

(ii) Accordingly the output is represented as 0 for all the neurons, except for the i^{th} neuron it will be 1 where i is equal to the digit in the image.

(iii) The activation functions used in this code are hyperbolic tangent and sigmoid. The initial learning rate used is 0.1 which is dynamically ~~up~~ updated if the error computed in current epoch is greater than previous epoch. This updation is done by multiplying the learning rate with 0.9.

(IV) Energy is calculated by the final output of neuron for each sample.

(V) I have tried to use normalization and dropout method in my code.

2 The reason why I have choosen only 2 hidden layers is because ~~I~~ it was producing a better accuracy and at a faster rate as compared to using one hidden layer.

3 I tried different network configurations to classify these images by using only one hidden layer and two hidden layers, I also tried changing the number of neurons in each to see if my program runs faster, I also implemented normalization in the hope that my program would converge faster but instead ~~my~~ ~~the~~ accuracy started oscillating.

Finally, I tried the ~~conf~~ configuration of 784-128-128-10 which worked best for this case.

4. • First I unzip all the .gz files.

• Read the content of gz files and collect test & training set.

• Then, we randomly initialize the weights, local fields.

• Set epoch to zero.

• Use the ~~desig~~ desired activation function to calculate the output.

• ~~Opa~~ Output is in one hot format and is a vector of 0's & 1's.

• Sigmoid function works better as step function would give the zero gradient which is ~~not~~ not feasible for backpropagation.