PURVANG LAPSIWALA

UIN = 662689378

ANSWER (1):

Pseudo Code

In this computer project, we will use back propagation method of neural network to update weights for curve fitting problem.

- There are 300 inputs generated uniformly at random on [0, 1]. So we have 300 inputs for our network.
- There is also variable v is generated uniformly at random between [-0.1, 0.1], which will be used to computing desired output.
- Desired output is given by the equation: $d_i = \sin(20x_i) + 3x_i + v_i$. (i = 1, . . . , n), which is nonlinear function.
- By plotting desired output to the input variable x, we will get idealize output which our network should predict, which is non-linear sinusoid curve.

Our aim of this project is to train the network such that we should get curve fit means actual output equal to desired output. Ideally when mean squared error term goes to nearly zero, we will get out ideal curve fit.

The algorithm for this phase may thus be as follows:

- Choose eta and Initialize weights uniformly.
- For our network, 1 hidden layer consisting 24 neurons is chosen. So our network consist of input layer consisting 300 neurons, 1 hidden layer consisting 24 neurons and 1 output neuron. Hence network will have 3N+1 weights where N stands for number of weights.

Clarification on 3N+1 weights,

- N weights for N neurons for input layer connecting input x and induced local field.
- N weights for N neurons for bias of those N neurons.
- N weights after induced local field.
- 1 neuron as bias for our output neuron. So we have total 3N+1 weights in our network.
- Do
  - for i = 1 to n do
    - Calculate the induced local fields with the current training sample and weights: $v = Wx_i$, where $x_i \in R^{300\times1}$ is the i^{th} training.
    - This result of induced local field is passed through activation function which is tangent hyperbolic function, which will give output of [-1,1].

      $\phi(v) = \tanh(v)$. for induced local field.

    - Then Output of tangent hyperbolic function is then passed through identity activation function.
      $\phi(v) = v$. for output

    - If y is not the same as the desired output d, then
      errors(epoch) ← errors(epoch) + 1.
      epoch ← epoch + 1. – 3.1.3)

Weights update and mse:

Back propagation of error:

In multilayer neural network, calculation of gradient to update weight is done by backpropogation algorithm.

Squared function is used as the energy function or cost function to calculate the error

For j in range N, N = 1 to Number of Neurons in hidden Layer

- Calculate the derivative of cost function, which will use to calculate derivatives with respect to weights. Which is -((d[i] - y[i])*2) for squared function.
- Now Derivative of this error function is calculated with respect to each weight associated with network.
- This is how we compute the gradient of our network and weights are updated as per the gradient descent algorithm, which is given by following equation.
- As we applying back propagation algorithm, we also have to consider derivative of activation function.
  W[i+1] ← W[i] - η*(del W[i]) (In General)

```
e = - ((d[i] - y[i])*2)/n
delta_u = e * u[i][j]
w_output_grad.append(delta_u)
delta_w = e  * x[i] * w_output[j] * derv_act_fun(alphas[i][j])
w_input_grad.append(delta_w)
delta_bias = e * w_output[j] * derv_act_fun(alphas[i][j])
w_bias_grad.append(delta_bias)
```

weight update

```
w_input = np.subtract(w_input, np.asarray(w_input_grad))
w_output = np.subtract(w_output, np.asarray(w_output_grad))
w_bias = np.subtract(w_bias, np.asarray(w_bias_grad))
w_final = np.subtract(w_final, np.asarray(w_final_grad))
```

For Mean Square Error

Calculate the mean square error by mse = mse + ((d[i] - y[i])**2), equation, where d is desired output and y is actual output predicted by network.

- If mse becomes greater than previous mse, change eeta to 0.9*eeta.
- If mse <0.01, then stop algorithm and this condition means we achieved best fit. For this condition to be satisfied, network takes around 1850 epochs.

Testing of Network

- Network is tested with updated final weights which satisfies the criteria for mse<0.01. By testing network with updated weights and for new input, network gives exact fit like desired output as shown in result. Hence Given curve fit problem is satisfied.
- We now have good weights that we have obtained via back propagation above. We now test the corresponding network on the test set images and test set labels.
- Given W obtained from the back propagation algorithm
  - Initialize errors = 0.
  - for i = 1 to 300
  - Calculate the induced local fields with the current test sample and weights: v = W*xi , where xi ∈ R 300*24 is the ith test.
  - If actual output is not the same as the input label,then
    errors ← errors + 1.

Plotting

Compare the plot between actual output curve and desired output curve with respect to input to analyze the best fit.

Python Code:

```
import numpy as np
import matplotlib.pyplot as plt


# input data
n = 300
x = np.random.uniform(low=0.0, high=1.0, size=n)
v = np.random.uniform(low=-0.1, high=0.1, size=n)


# desired output
d = []
for i in range(n):
   d.append(np.sin(20*x[i]) + (3*x[i]) + v[i])


fig, ax = plt.subplots(figsize=(10,10))
plt.xlabel('x')
plt.ylabel('d')
plt.scatter(x,d, c = 'green', label = 'Actual')
plt.legend(loc = 'best')
plt.show()


# feed-forward activation functions
def act_fun(v):
   return np.tanh(v)


def act_op(v):
   return v
```

```
# feedback activation functions
def derv_act_fun(v):
    return (1 - np.tanh(v)**2)

def derv_act_op(v):
    return 1

# weight initialization
N = 24
w_input = np.random.uniform(low=-5, high=5, size=N)
w_bias = np.random.uniform(low=-1, high=1, size=N)
w_output = np.random.uniform(low=-5, high=5, size=N)
w_final = np.random.uniform(low=-1, high=1, size=1)
eta = 6

list_mse = []
z = 0
while(True):
    # feed-forward network
    u = []
    y = []
    alphas = []
    betas = []
    for i in range(n):
        v = []
        temp = []
        for j in range(N):
            alpha = (x[i]*w_input[j]) + w_bias[j]
            temp.append(alpha)
            v.append(act_fun(alpha))
        alphas.append(temp)
        u.append(v)
```

```
    beta = np.matmul(np.array(u[i]),w_output) + w_final
    betas.append(beta[0])
    y.append(act_op(beta[0]))

    # backpropagation
    e = -((d[i] - y[i])*eta*2)/n
    w_output_grad = []
    w_input_grad = []
    w_bias_grad = []
    w_final_grad = []
    delta_final = - e
    w_final_grad.append(delta_final)
    for j in range(N):
        delta_u = e * u[i][j]
        w_output_grad.append(delta_u)
        delta_w = e  * x[i] * w_output[j] * derv_act_fun(alphas[i][j])
        w_input_grad.append(delta_w)
        delta_bias = e * w_output[j] * derv_act_fun(alphas[i][j])
        w_bias_grad.append(delta_bias)
    # weight update
    w_input = np.subtract(w_input, np.asarray(w_input_grad))
    w_output = np.subtract(w_output, np.asarray(w_output_grad))
    w_bias = np.subtract(w_bias, np.asarray(w_bias_grad))
    w_final = np.subtract(w_final, np.asarray(w_final_grad))
# mean square error
mse = 0
for i in range(n):
    mse += (d[i] - y[i])**2
mse = mse/n
list_mse.append(mse)

print (mse, eta, z)
```

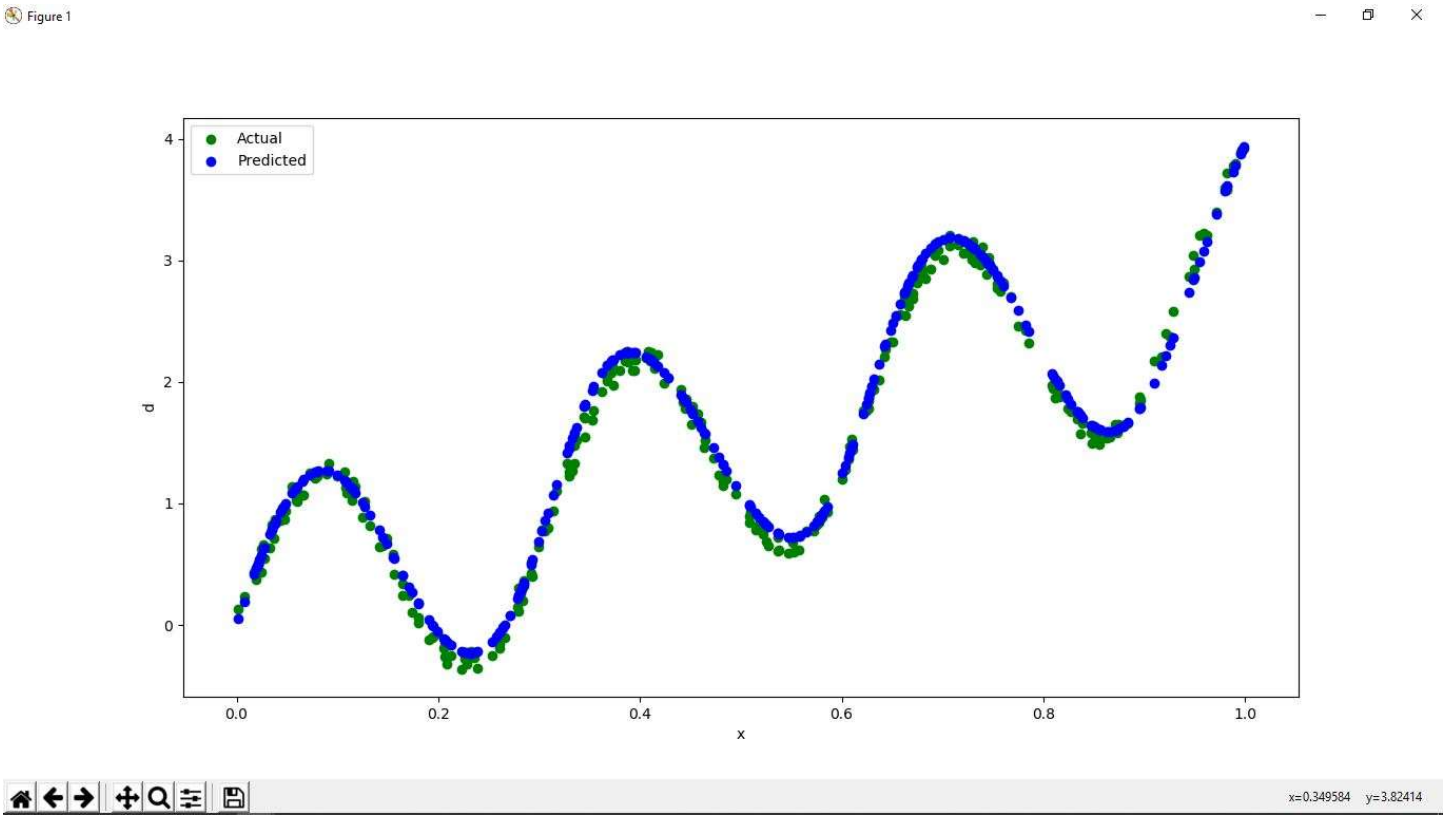
```
    if list_mse[z] > list_mse[z-1]:
        eta = 0.9*eta
    if list_mse[-1]<0.01:
        break
    z += 1

fig, ax = plt.subplots(figsize=(10,10))
plt.ylabel('Mean Square Error')
plt.xlabel('Number of Epochs')
plt.scatter(range(len(list_mse)), list_mse, c = 'green', label = 'MSE')
plt.legend(loc = 'best')
plt.show()

u = []
y = []
alphas = []
betas = []
for j in range(n):
    v = []
    temp = []
    for i in range(N):
        alpha = x[j]*w_input[i] + w_bias[i]
        temp.append(alpha)
        v.append(act_fun(alpha))
    alphas.append(temp)
    u.append(v)
    beta = np.matmul(np.array(u[j]),w_output)+w_final
    betas.append(beta)
    y.append(act_op(beta[0]))

fig, ax = plt.subplots(figsize=(10,10))
```

```
plt.ylabel('d')
plt.xlabel('x')
plt.scatter(x,d, c = 'green', label = 'Actual')
plt.scatter(x,y, c = 'blue', label = 'Predicted')
plt.legend(loc = 'best')
plt.show()
```



Answer 2:

Pseudocode:

Network Topology

For the given problem, First task is to convert idx files in array format and convert labels of those training and testing files in to 0 and 1.

As per MNIST dataset, we have 784 input neurons as each image is 28*28 and our network consist of 10 output neurons.

Weight Initialization

- Initialize W1 ∈ R 50 ×784 randomly.
- Initialize W2 ∈ R 10 ×50 randomly.
- Initialize epoch = 0
- Initialize error of epoch = 0
- Do
  - for i = 1 to n do (this loop is where we count the misclassification errors)
  - Calculate the induced local fields with the current training sample and weights: v = Wxi , where xi ∈ R 784×50 is the i^{th} training.
  - This result of induced local field is passed through activation function which is tangent hyperbolic function, which will give output of [-1,1].

    ϕ(v) = tanh(v). for induced local field.

    - Then Output of tangent hyperbolic function is again passed through tangent hyperbolic activation function.

    ϕ(v) = tanh(v). for output

Back propagation of error:

In multilayer neural network, calculation of gradient to update weight is done by backpropogation algorithm.

Squared function is used as the energy function or cost function to calculate the error

For j in range N, N = 1 to Number of Neurons in hidden Layer

- Calculate the derivative of cost function, which will use to calculate derivatives with respect to weights. Which is -((d[i] - y[i])*2) for squared function.
- Now Derivative of this error function is calculated with respect to each weight associated with network, which is nothing but the chain rule for taking partial derivative.
- This is how we compute the gradient of our network and weights are updated as per the gradient descent algorithm, which is given by following equation.
  W[i+1] ← W[i] - η*(del W[i]) (In General).

- Calculate the mean square error by ((d[i] - y[i])**2), equation, where d is desired output and y is actual output predicted by network.
- Calculate the number of successfully classified images.

Testing of Network

Testing of Network is simultaneously done after updation of weight from training and calculate mean squared error and number of successfully classified images.

- Network is tested with updated final weights which satisfies the criteria for mse<0.01. By testing network with updated weights and for new input, network gives exact fit like desired output as shown in result. Hence Given curve fit problem is satisfied.
- We now have good weights that we have obtained via back propagation above. We now test the corresponding network on the test set images and test set labels.
- Given W obtained from the back propagation algorithm
  - Initialize errors = 0.
  - for i = 1 to 10000 (since 10000 test images)
  - Calculate the induced local fields with the current test sample and weights: v = W*xi , where xi ∈ R 784×50 is the ith test sample (the vectorized version of the 28 × 28 image from the test set images).
  - If success rate is greater than 95%, break.

- In this Problem, only one hidden layer with 50 neuron is used to classify the input data.

  First of all, it doesn't mean that more hidden layers means better output. Generally it depends upon the type of problem you are working on. If given inputs are linearly separable, generally one hidden layer is sufficient to classify. If given problem or function is non-linear and non-convex, then we might need more than one layer to extract complex information from inputs and classify the problem correctly. So as given problem is linearly separable, I used only one hidden layer with 50 hidden neurons.

- Number of neurons in hidden layer has been debatable point and there are many papers published on that. But there is no hard and fast rule for this. But the thing I kept in mind while selecting neurons is number of output neurons. As we have total of 10 output so I started with 10 hidden neurons and trained my network and observed output. And then I keep increasing number of neuron by 1 and observed output each time. By doing that, I have analyzed that if number of neuron is less, then network is not able to extract the full information from the input and hence network was getting stuck at local minima even by changing in parameters. So by doing that I have observed that keeping neurons around 50 would give me best result. I also tried with more number of neurons but that approach was taking too much time and over fitting was observed.

- I tried both logistic regression and tangent hyperbolic function. Though the logistic sigmoid has a nice biological interpretation, it turns out that the logistic sigmoid can cause a neural network to get "stuck" during training. This is due in part to the fact that if a strongly-negative input is provided to the logistic sigmoid, it outputs values very near zero. Since neural networks use the feed-forward activations to calculate parameter gradients, this

can result in model parameters that are updated less regularly than we would like, and are thus "stuck" in their current state. An alternative to the logistic sigmoid is the hyperbolic tangent, or tanh function. Thus strongly negative inputs to the tanh will map to negative outputs. Additionally, only zero-valued inputs are mapped to near-zero outputs and hence tangent hyperbolic function is used as an activation function.

- For the output I have selected 10 neurons for representation of 0…9 digits. If network gives the output of 0, then it will give 1 in 0^{th} place and all remaining values will be zero.

- The cost function chosen for the problem is squared function. Mse is used in comparison with mae (mean absolute error) is just because fact that in mae, gradient will be positive only and it will be constant or not much change. So it will take long time to reach to global minimum as compared to mse, which uses squared value between desired and actual value. Also, as we have no outliers in the training set, it is better to use mse as it reaches more rapidly towards global minimum and easy to calculate.

Problems faced during Example:

Even though this was said to be easy question, this is really tricky question and gave me great exposure of tuning different parameters to achieve accuracy.

- First main challenge was to come up with the weights that can satisfy requirement. First I tried my network with random initialization between [-1, 1] for all weights, but it didn't work well for me as network was taking too much time to converge. I tried increasing weight but it also started with very less accuracy and taking so much time to update. Then I tweaked a bit and tried first layer with lower weight and second one with higher weights. Reason why I approached this way because as gradient decreases as we approaches to layers, so to give equal weightage to second layer, I raised the weight of second layer and that approach gave me good set of weights.
- Second problem was to go with batch learning or online learning. But as though in to class, batch learning is bit slower and it can't escape from local minima. I tried with doing batch learning and it was taking lot of time and I opted for online learning.
- One more problem occurred while come up with number of layers and number of neurons in each layer. First I was thinking to use 2 hidden layers for my network and I tried with 2 hidden layers each having 100 neurons but I observed that my network was getting stuck at local minima that was around 9 to 10 in terms of mse. Even though I changed parameters with almost all possible variety and also hyper parameter as well, but still I wasn't succeed. Then I read few articles and come to know about complexity and number of layer. I tried with single layer and 300 neurons but algorithm was taking almost hour to converge and by reducing weight I finally come to proper number of neurons which are 50, which took 32 number of epochs. Result could be faster if laptop with NVIDEA graphic card would have been used.
- The main problem faced during this example is the time taken by algorithm to converge and that took lot of time.

Python code:

```
import struct
import numpy.random as random
import numpy as np
import matplotlib.pylab as plt
import math

hidden=50
w1N=[]
w2N=[]
dataset = 60000
dataset1 = 10000
eta = 0.01

def weights(HN,N,L,U):
    W = np.zeros((HN,N))
    for i in range(HN):
        temp = (U-L)*random.sample(N) + L
        W[i] = temp
    W_mat=np.matrix(np.array(W))
    return W_mat
w1N = weights(hidden,784,-0.3,0.3)
w2N = weights(hidden,10,-0.5,0.5)

epoch=0
epoch2 = 0
thres1=0
thres=0
epoch_list1=[]
error_list1=[]
mss_list1=[]
epoch_list2=[]
```

```
error_list2=[]
mss_list2=[]
while(True):
    train_data = open('train-labels.idx1-ubyte', 'rb')
    for i in range(2):
        train_data.read(4)
    d_train1=[]
    des1 = np.zeros((dataset,10))
    for i in range (dataset):
        index1=struct.unpack('>B', train_data.read(1))[0]
        d_train1.append(index1)
        des1[i][index1]=1

    for j in range(1):
        train_img1 = open('train-images.idx3-ubyte', 'rb')
        for i in range(4):
            train_img1.read(4)
        error1 = 0
        error2 = 0
        mse1 = 0
        mse2 = 0
        for i in range(dataset):
            x1=[]
            for p in range(784):
                x1.append((struct.unpack('>B', train_img1.read(1))[0])/255.0)
            x1=np.matrix(np.array(x1))
            x_T1=np.transpose(x1)

            v11 = np.dot(w1N,x_T1)
            z1=np.tanh(v11)
            z_T1=np.transpose(z1)
            v21 = np.transpose(np.dot(z_T1,w2N))
```

```
    y1=np.tanh(v21)

    ymax1=np.argmax(y1)

    delta2= np.zeros((10,1))
    des_T=np.transpose(np.matrix(np.array(des1[i])))

    sub= -2*np.subtract(des_T,y1)

    for h in range(10):
        delta2[h]= sub[h]*(1-math.pow(np.tanh(v21[h]),2))
    delta2_T=np.transpose(delta2)

    delta1= np.zeros((hidden,1))
    mul = np.dot(w2N,delta2)
    for j in range(hidden):
        delta1[j]=mul[j]*(1-math.pow(np.tanh(v11[j]),2))

    w2N = w2N - eta*np.dot(z1,delta2_T)
    w1N = w1N - eta*np.dot(delta1,x1)
    mse1 = mse1 + math.pow((d_train1[i] - ymax1), 2)
    if d_train1[i]!=ymax1:
        error1 = error1 +1
thres1=error1/dataset
msee1=mse1/dataset
epoch=epoch+1
epoch_list1.append(epoch)
error_list1.append(error1)
mss_list1.append(msee1)
suc1=100-(thres1*100)
print("Epoch",epoch,"Error",error1,"Success",suc1)
```

```
train_lbl = open('t10k-labels.idx1-ubyte', 'rb')
for i in range(2):
    train_lbl.read(4)
suct = []
d_list=[]
des = np.zeros((dataset1,10))
for i in range (dataset1):
    index=struct.unpack('>B', train_lbl.read(1))[0]
    d_list.append(index)
    des[i][index]=1

for j in range(1):
    train_img = open('t10k-images.idx3-ubyte', 'rb')
    for i in range(4):
        train_img.read(4)
    error = 0
    for i in range(dataset1):
        x=[]
        for p in range(784):
            x.append((struct.unpack('>B', train_img.read(1))[0])/255.0)
        x=np.matrix(np.array(x))
        x_T=np.transpose(x)
        v1 = np.dot(w1N,x_T)
        z=np.tanh(v1)
        z_T=np.transpose(z)
        v2 = np.transpose(np.dot(z_T,w2N))
        y=np.tanh(v2)
        ymax=np.argmax(y)
        mse2 = mse2 + math.pow((d_list[i] - ymax), 2)
        if d_list[i]!=ymax:
```

```
            error2 = error2 +1
        thres=error2/dataset1
        msee2=mse2/dataset1
        epoch2=epoch2+1
        epoch_list2.append(epoch2)
        error_list2.append(error2)
        mss_list2.append(msee2)
        suc2=100-(thres*100)
        suct.append(suc2)
        print("Error",error2,"Success",suc2)
    if suc2 > 95:
      break

plt.title("Missclassification vs Epoch Training")
plt.xlim(0,epoch+1)
plt.plot(epoch_list1,error_list1,'o-')
plt.show()

plt.title("MSE vs Epoch Trainig")
plt.xlim(0,epoch+1)
plt.plot(epoch_list1,mss_list1,'o-')
plt.show()

plt.title("Missclassification vs Epoch Test")
plt.xlim(0,epoch+1)
plt.plot(epoch_list2,error_list2,'o-')
plt.show()

plt.title("MSE vs Epoch Test")
plt.xlim(0,epoch+1)
plt.plot(epoch_list2,mss_list2,'o-')
plt.show()
```

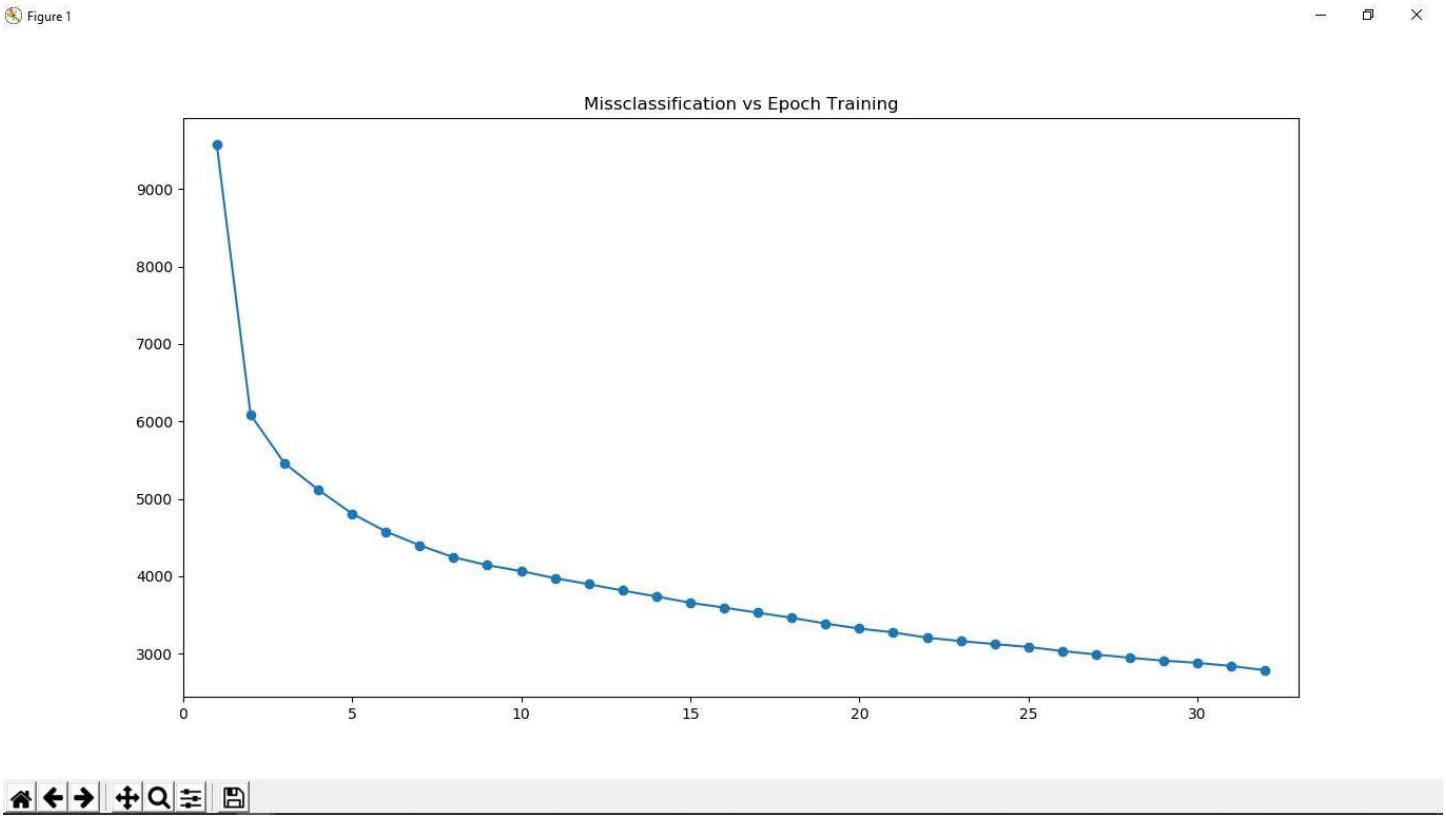


Figure 1
Missclassification vs Epoch Training
9000
8000
7000
6000
5000
4000
3000
0
5
10
15
20
25
30

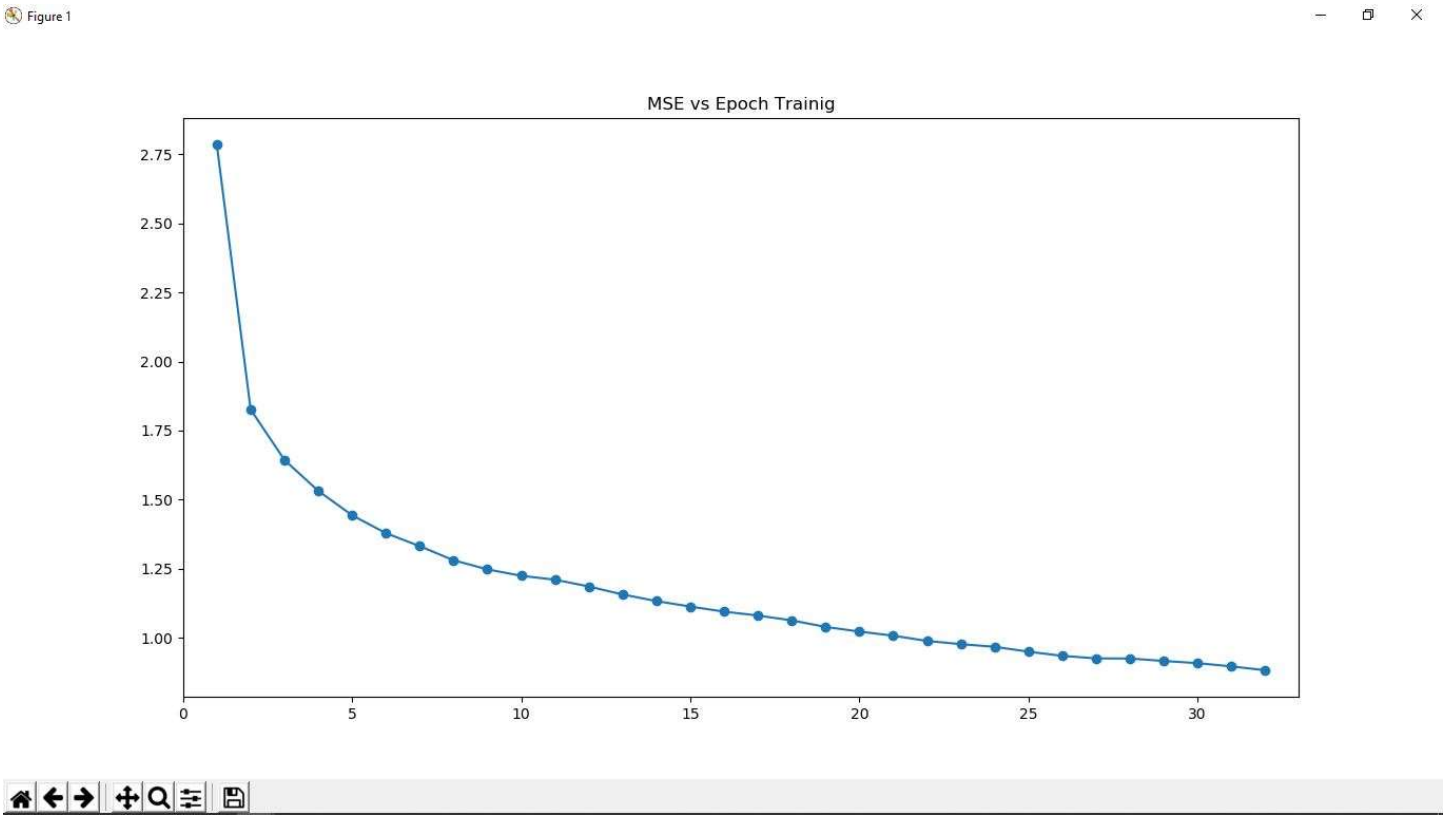


Figure 1
MSE vs Epoch Trainig
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0
5
10
15
20
25
30

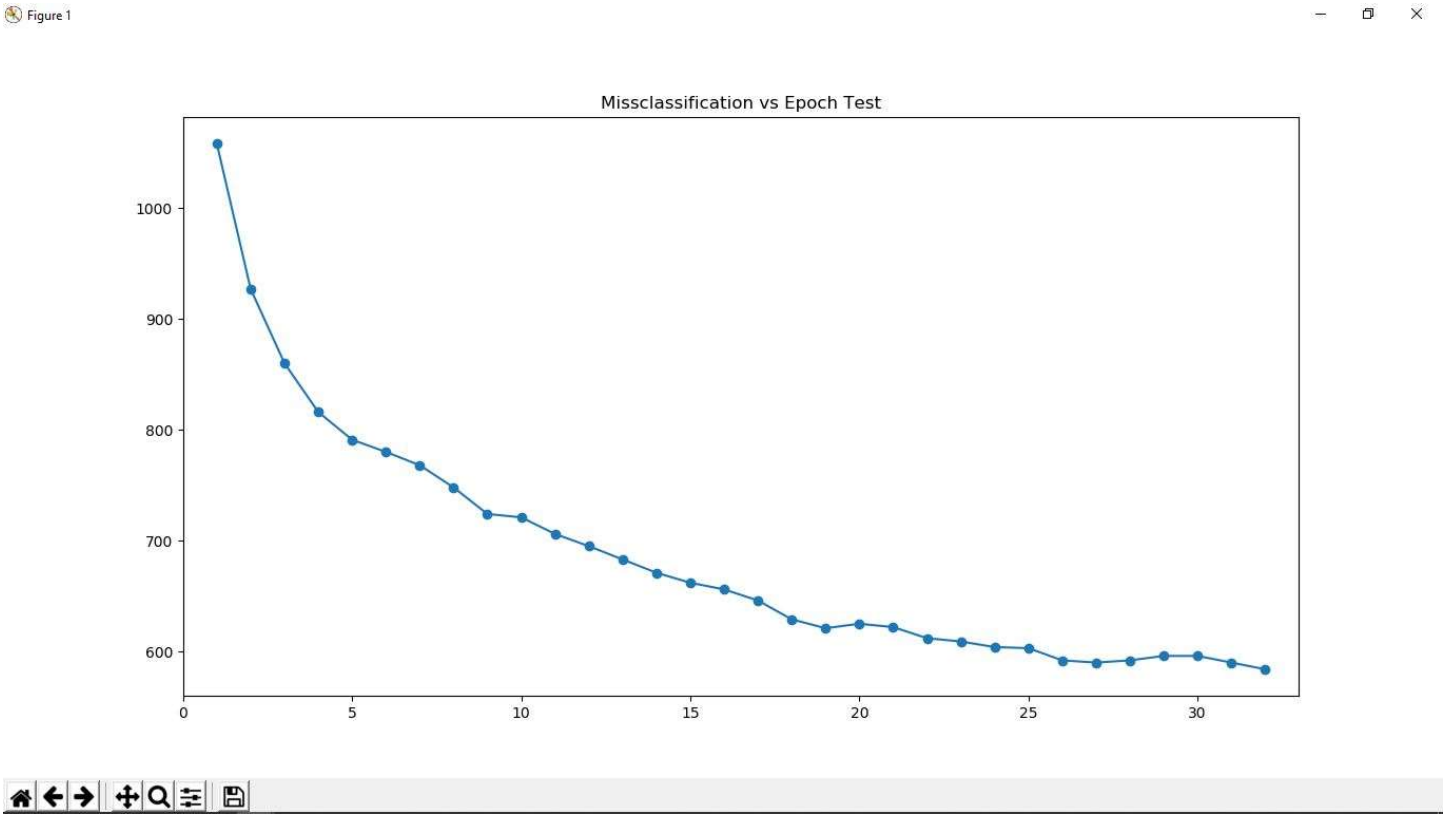


Figure 1
Missclassification vs Epoch Test
1000
900
800
700
600
0
5
10
15
20
25
30

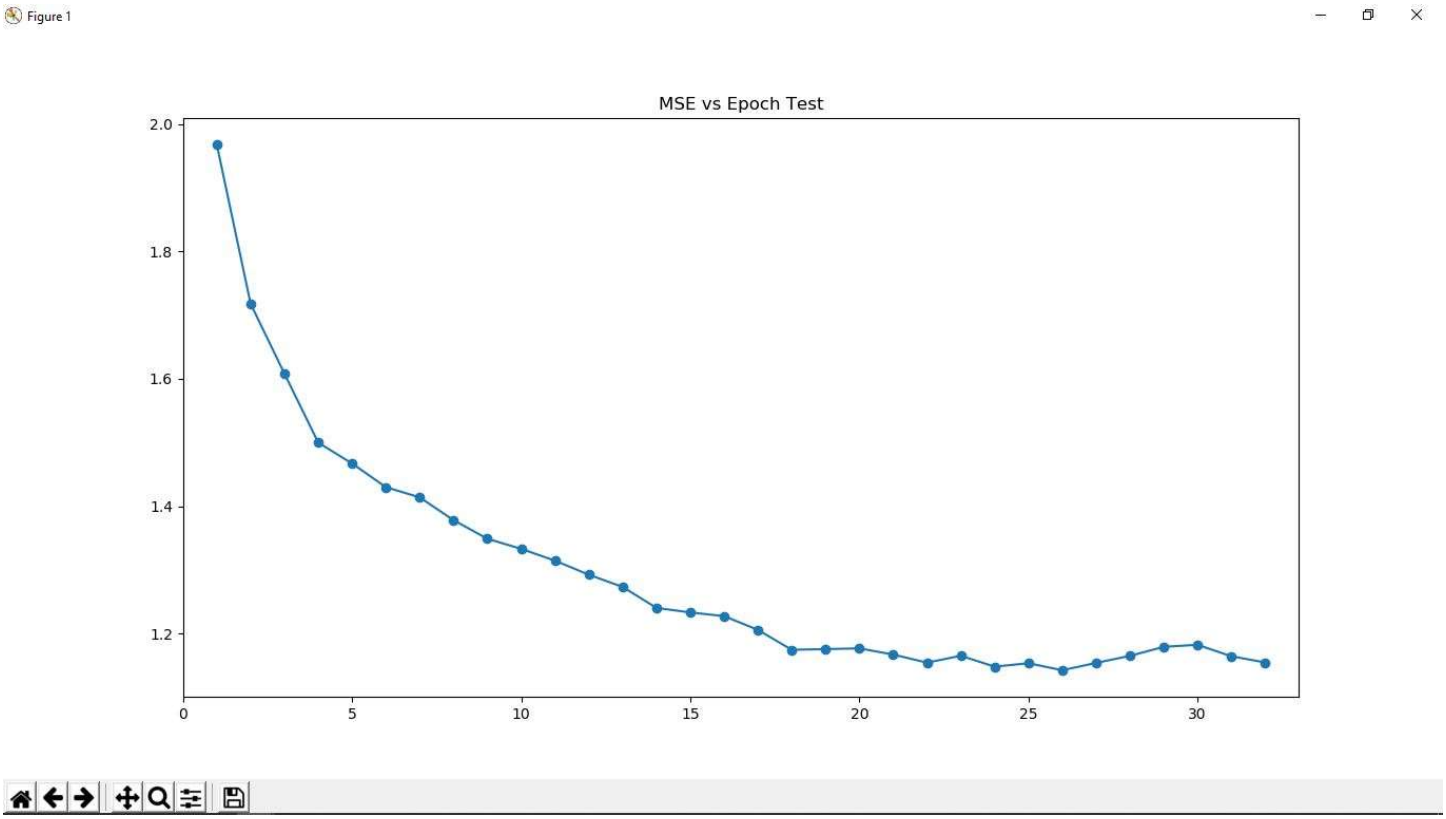


Figure 1
MSE vs Epoch Test
2.0
1.8
1.6
1.4
1.2
0
5
10
15
20
25
30