```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
from google.colab import files
files=files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving mnist_train.csv to mnist_train.csv
```

Problem 1

Loading the data using a function

```
def load_data(file):
  df = pd.read_csv(file)
  df.head()
  Y = df['5'].to_numpy()
  del df['5']
  X=df.to_numpy()
  return X, Y

X, y = load_data("mnist_train.csv")
X_train, X_test, y_train, y_test = train_test_split(\
                X, y, test_size=0.3, random_state=42)
#One hot encoding of training labels
Labels=pd.get_dummies(y_train)
```

Problem 2

Implement the backpropagation algorithm in a zero hidden layer neural network

```
# Problem 2
class Perceptron():
    def __init__(self,x,y):
        """
        x is 2d array of input images
        y are one hot encoded labels
        """
        self.x=x/255   # Divide by 255 to normalise the pixel values (0-255)
        self.y=y
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
        self.activations=[]

    def connect(self,layer1,layer2):
        """layer 2 of shape 1xn"""
        #Initialise weights,derivatives and activation lists
        self.derivatives.append(np.random.uniform(0,0.1,size=(layer1.shape[1]+1,layer2.shape[1
        self.weights.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[1])))
        self.bias.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[1])))


    def softmax(self,z):
        e=np.exp(z)
        return e/np.sum(e,axis=1).reshape(-1,1)

    def max_log_likelihood(self,y_pred,y):
        """cross entropy"""
        return y*np.log(y_pred)

    def delta_mll(self,y,y_pred):
        """derivative of cross entropy"""
        #return y*(y_pred-1)
        return y_pred-y

    def forward_pass(self,x,y,weights,bias):
        cost=0
        self.outputs=[]
        for i in range(len(weights)):
            samples=len(x)
            ones_array=np.ones(samples).reshape(samples,1)
            self.outputs.append(x) #append without adding ones array
            z=np.dot(np.append(ones_array,x,axis=1),weights[i]+bias[i])
            x=self.softmax(z)
        self.outputs.append(x)
        self.y_pred=x
        temp=-self.max_log_likelihood(self.y_pred,y)
        cost=np.mean(np.sum(temp,axis=1))
        return cost


    def backward_pass(self,y,lr):
        for i in range(len(self.weights)-1,-1,-1):
            ones_array=np.ones(len(n.outputs[i])).reshape(len(n.outputs[i]),1)
            prev_term=self.delta_mll(y,self.y_pred)
            # derivatives follow specific order,last three terms added new,rest from previous
            self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],axis=1
            self.weights[i]=self.weights[i]-lr*((self.derivatives[i].T)/len(y))
            self.bias[i]=self.bias[i]-lr*((self.derivatives[i].T)/len(y))

    def train(self,batches,lr=1e-3,epoch=10):
        """number of batches to split data in,Learning rate and epochs"""
        for epochs in range(epoch):
            samples=len(self.x)
            c=0
            for i in range(batches):
              x_batch=self.x[int((samples/batches)*i):int((samples/batches)*(i+1))]
              y_batch=self.y.loc[int((samples/batches)*i):int((samples/batches)*(i+1))-1]

              c=self.forward_pass(x_batch,y_batch,self.weights,self.bias)
              self.backward_pass(y_batch,lr)
            print(epochs,c/batches)

    def predict(self,x):
        """input: x_test values"""
        x=x/255
        for i in range(len(self.weights)):
```

```
            samples=len(x)
            ones_array=np.ones(samples).reshape(samples,1)
            z=np.dot(np.append(ones_array,x,axis=1),self.weights[i]+self.bias[i])
            x=self.softmax(z)
        return np.argmax(x,axis=1)
```

```
n=Perceptron(X_train,Labels)
n.connect(X_train,Labels)
n.train(batches=1000,lr=0.2,epoch=30)
```

```
0 0.0012326529173191784
1 0.0013336506192253598
2 0.0015425379057302125
3 0.001396635631347119
4 0.0011803980604108617
5 0.0010028776738607646
6 0.0008725671510958899
7 0.0007480063461397748
8 0.0005808357707951654
9 0.0004986331727568831
10 0.00041343828990402415
11 0.0003055680959857993
12 0.00019444419404161708
13 0.0001384401247720068
14 0.00011494320736217149
15 0.00012188409156764916
16 0.00011268670557574174
17 0.00011168953317027843
18 0.00012739842722061018
19 0.000138678977919056
20 0.00013785635015165252
21 0.00013057284954740364
22 0.00011959558884569431
23 0.00010785513221998618
24 9.809299833863174e-05
25 9.303297447185356e-05
26 9.075831025306053e-05
27 8.768241620848049e-05
28 8.401814413370632e-05
29 8.073393794466543e-05
```

```
pred=n.predict(X_test)
np.bincount(n.predict(X_test)),np.bincount(y_test)
```

```
(array([311, 333, 294, 346, 327, 272, 292, 323, 227, 275]),
 array([296, 327, 305, 326, 305, 283, 282, 336, 252, 288]))
```

```
print(f"accuracy is {np.bincount(np.abs(y_test-pred))[0]*100/len(y_test)} %")
```

```
accuracy is 88.53333333333333 %
```

Problem 3

Extend previous question by adding n hidden layer in single neural network and use sigmoid function

```
# Problem 3
class Layer():
    """
    size: Number of nodes in the hidden layer
    activation: name of activation function for the layer
    """
    def __init__(self,size,activation='sigmoid'):
        self.shape=(1,size)
        self.activation=activation

class SingleLayerNeuralNetwork():
    def __init__(self,x,y):
        """
        x is 2d array of input images
        y are one hot encoded labels
        """
        self.x=x/255   # Divide by 255 to normalise the pixel values (0-255)
        self.y=y
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
        self.activations=[]

    def connect(self,layer1,layer2):
        """layer 2 of shape 1xn"""
        #Initialise weights,derivatives and activation lists
        self.derivatives.append(np.random.uniform(0,0.1,size=(layer1.shape[1]+1,layer2.shape[1
        self.weights.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[1])))
        self.bias.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[1])))
        if isinstance(layer2,Layer):
            self.activations.append(layer2.activation)

    def activation(self,name,z,derivative=False):

        #implementation of various activation functions and their derivatives
        if name=='sigmoid':
            if derivative==False:
                return 1/(1+np.exp(-z))
            else:
                return z*(1-z)

    def softmax(self,z):
        e=np.exp(z)
        return e/np.sum(e,axis=1).reshape(-1,1)

    def max_log_likelihood(self,y_pred,y):
        """cross entropy"""
        return y*np.log(y_pred)

    def delta_mll(self,y,y_pred):
        """derivative of cross entropy"""
        #return y*(y_pred-1)
        return y_pred-y

    def forward_pass(self,x,y,weights,bias):
        cost=0
        self.outputs=[]
        for i in range(len(weights)):
            samples=len(x)
            ones_array=np.ones(samples).reshape(samples,1)
```

```
            self.outputs.append(x) #append without adding ones array
            z=np.dot(np.append(ones_array,x,axis=1),weights[i]+bias[i])
            if i==len(weights)-1:
                x=self.softmax(z)
            else:
                x=self.activation(self.activations[i],z)
        self.outputs.append(x)
        self.y_pred=x

        temp=-self.max_log_likelihood(self.y_pred,y)
        cost=np.mean(np.sum(temp,axis=1))
        return cost


    def backward_pass(self,y,lr):
        for i in range(len(self.weights)-1,-1,-1):
            ones_array=np.ones(len(n.outputs[i])).reshape(len(n.outputs[i]),1)
            if i==len(self.weights)-1:
                prev_term=self.delta_mll(y,self.y_pred)
                # derivatives follow specific order,last three terms added new,rest from previ
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],ax
            else:
                prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(self.activ
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],ax
            self.weights[i]=self.weights[i]-lr*((self.derivatives[i].T)/len(y))
            self.bias[i]=self.bias[i]-lr*((self.derivatives[i].T)/len(y))


    def train(self,batches,lr=1e-3,epoch=10):
        """number of batches to split data in,Learning rate and epochs"""
        for epochs in range(epoch):
            samples=len(self.x)
            c=0
            for i in range(batches):
              x_batch=self.x[int((samples/batches)*i):int((samples/batches)*(i+1))]
              y_batch=self.y.loc[int((samples/batches)*i):int((samples/batches)*(i+1))-1]

              c=self.forward_pass(x_batch,y_batch,self.weights,self.bias)
              self.backward_pass(y_batch,lr)
            print(epochs,c/batches)

    def predict(self,x):
        """input: x_test values"""
        x=x/255
        for i in range(len(self.weights)):
            samples=len(x)
            ones_array=np.ones(samples).reshape(samples,1)
            z=np.dot(np.append(ones_array,x,axis=1),self.weights[i]+self.bias[i])
            if i==len(self.weights)-1:
                x=self.softmax(z)
            else:
                x=self.activation(self.activations[i],z)

        return np.argmax(x,axis=1)
```

```
n=SingleLayerNeuralNetwork(X_train,Labels)
l1=Layer(100)
n.connect(X_train,l1)
n.connect(l1,Labels)
n.train(batches=1000,lr=0.1,epoch=50)
```

```
0 0.0006035243988934736
1 0.00041134807666813215
2 0.00025004604928999664
3 0.00015344188379584208
4 0.0001085434044838894
5 8.462870944148152e-05
6 6.522447563306008e-05
7 4.915096888172114e-05
8 3.8709964792677185e-05
9 3.380620939708583e-05
10 3.27218138542811e-05
11 3.26943012414242e-05
12 3.1846218339490266e-05
13 2.9902041164847622e-05
14 2.738413476589643e-05
15 2.493759863510678e-05
16 2.273898201477572e-05
17 2.0790256628280322e-05
18 1.910321029834869e-05
19 1.767449997213585e-05
20 1.648753655366667e-05
21 1.5523191567753722e-05
22 1.4736452323225292e-05
23 1.4046845797231222e-05
24 1.339274790435979e-05
25 1.2744911427337953e-05
26 1.2098006407511657e-05
27 1.146466304114171e-05
28 1.0863538840910613e-05
29 1.0307695201345296e-05
30 9.801396834431731e-06
31 9.34264291140285e-06
32 8.926448554617388e-06
33 8.547014788027178e-06
34 8.198831107632344e-06
35 7.87712223159411e-06
36 7.577950562262442e-06
37 7.298157947725572e-06
38 7.035246677148885e-06
39 6.787250441698833e-06
40 6.552617798618389e-06
41 6.33011538902039e-06
42 6.118750834257133e-06
43 5.917712519132449e-06
44 5.726323065908623e-06
45 5.544003769806203e-06
46 5.370247851667153e-06
47 5.2046008143323265e-06
48 5.046646455426947e-06
49 4.895997268265909e-06
```

```
pred=n.predict(X_test)
np.bincount(n.predict(X_test)),np.bincount(y_test)
```

```
print(f"accuracy is {np.bincount(np.abs(y_test-pred))[0]*100/len(y_test)} %")
```

Problem 4

Extend Problem 3 and Implement a 2-layer neural network, starting with a simple architecture containing N hidden units in each layer

```
# Problem 4
class Layer():
    """
    size: Number of nodes in the hidden layer
    activation: name of activation function for the layer
    """
    def __init__(self,size,activation='sigmoid'):
        self.shape=(1,size)
        self.activation=activation

class DoubleLayerNeuralNetwork():
    def __init__(self,x,y):
        """
        x is 2d array of input images
        y are one hot encoded labels
        """
        self.x=x/255   # Divide by 255 to normalise the pixel values (0-255)
        self.y=y
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
        self.activations=[]

    def connect(self,layer1,layer2):
        """layer 2 of shape 1xn"""
        #Initialise weights,derivatives and activation lists
        self.derivatives.append(np.random.uniform(0,0.1,size=(layer1.shape[1]+1,layer2.shape[1
        self.weights.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[1])))
        self.bias.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[1])))
        if isinstance(layer2,Layer):
            self.activations.append(layer2.activation)

    def activation(self,name,z,derivative=False):

        #implementation of various activation functions and their derivatives
        if name=='sigmoid':
            if derivative==False:
                return 1/(1+np.exp(-z))
            else:
                return z*(1-z)

    def softmax(self,z):
        e=np.exp(z)
        return e/np.sum(e,axis=1).reshape(-1,1)

    def max_log_likelihood(self,y_pred,y):
        """cross entropy"""
        return y*np.log(y_pred)

    def delta_mll(self,y,y_pred):
        """derivative of cross entropy"""
        #return y*(y_pred-1)
        return y_pred-y

    def forward_pass(self,x,y,weights,bias):
        cost=0
        self.outputs=[]
        for i in range(len(weights)):
            samples=len(x)
            ones_array=np.ones(samples).reshape(samples,1)
            self.outputs.append(x) #append without adding ones array
            z=np.dot(np.append(ones_array,x,axis=1),weights[i]+bias[i])
            if i==len(weights)-1:
                x=self.softmax(z)
            else:
                x=self.activation(self.activations[i],z)
        self.outputs.append(x)
        self.y_pred=x

        temp=-self.max_log_likelihood(self.y_pred,y)
        cost=np.mean(np.sum(temp,axis=1))
        return cost


    def backward_pass(self,y,lr):
        for i in range(len(self.weights)-1,-1,-1):
            ones_array=np.ones(len(n.outputs[i])).reshape(len(n.outputs[i]),1)
            if i==len(self.weights)-1:
                prev_term=self.delta_mll(y,self.y_pred)
                # derivatives follow specific order,last three terms added new,rest from previ
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],ax
            else:
                prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(self.activ
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],ax
            self.weights[i]=self.weights[i]-lr*((self.derivatives[i].T)/len(y))
            self.bias[i]=self.bias[i]-lr*((self.derivatives[i].T)/len(y))


    def train(self,batches,lr=1e-3,epoch=10):
        """number of batches to split data in,Learning rate and epochs"""
        for epochs in range(epoch):
            samples=len(self.x)
            c=0
            for i in range(batches):
              x_batch=self.x[int((samples/batches)*i):int((samples/batches)*(i+1))]
              y_batch=self.y.loc[int((samples/batches)*i):int((samples/batches)*(i+1))-1]

              c=self.forward_pass(x_batch,y_batch,self.weights,self.bias)
              self.backward_pass(y_batch,lr)
            print(epochs,c/batches)

    def predict(self,x):
        """input: x_test values"""
        x=x/255
        for i in range(len(self.weights)):
            samples=len(x)
            ones_array=np.ones(samples).reshape(samples,1)
            z=np.dot(np.append(ones_array,x,axis=1),self.weights[i]+self.bias[i])
```

```
        if i==len(self.weights)-1:
            x=self.softmax(z)
        else:
            x=self.activation(self.activations[i],z)
    return np.argmax(x,axis=1)
```

```
n=DoubleLayerNeuralNetwork(X_train,Labels)
l1=Layer(100)
l2=Layer(100)
n.connect(X_train,l1)
n.connect(l1,l2)
n.connect(l2,Labels)
n.train(batches=1000,lr=0.1,epoch=20)
```

```
pred=n.predict(X_test)
np.bincount(n.predict(X_test)),np.bincount(y_test)
```

```
print(f"accuracy is {np.bincount(np.abs(y_test-pred))[0]*100/len(y_test)} %")
```

Problem 5

Extend your code from problem 4 to implement different activations functions which will be passed as a parameter. In this problem all activations (except the final layer which should remain a softmax) must be changed to the passed activation function.

```
# Problem 5

class Layer():
    """
    size: Number of nodes in the hidden layer
    activation: name of activation function for the layer
    """
    def __init__(self,size,activation='sigmoid'):
        self.shape=(1,size)
        self.activation=activation

class NeuralNetworkActivations():
    def __init__(self,x,y):
        """
        x is 2d array of input images
        y are one hot encoded labels
        """
        self.x=x/255   # Divide by 255 to normalise the pixel values (0-255)
        self.y=y
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
        self.activations=[]

    def connect(self,layer1,layer2):
        """layer 2 of shape 1xn"""
        #Initialise weights,derivatives and activation lists
        self.derivatives.append(np.random.uniform(0,0.1,size=(layer1.shape[1]+1,layer2.shape[1
        self.weights.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[1])))
        self.bias.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[1])))
        if isinstance(layer2,Layer):
            self.activations.append(layer2.activation)

    def activation(self,name,z,derivative=False):

        #implementation of various activation functions and their derivatives
        if name=='sigmoid':
            if derivative==False:
                return 1/(1+np.exp(-z))
            else:
                return z*(1-z)
        elif name=='relu':
            if derivative==False:
                return np.maximum(0.0,z)
            else:
              z[z<=0] = 0.0
              z[z>0] = 1.0
              return z
        elif name=='tanh':
          if derivative==False:
                return np.tanh(z)
          else:
                return 1.0 - (np.tanh(z)) ** 2

    def softmax(self,z):
        e=np.exp(z)
        return e/np.sum(e,axis=1).reshape(-1,1)

    def max_log_likelihood(self,y_pred,y):
        """cross entropy"""
        return y*np.log(y_pred)
```

```
    def delta_mll(self,y,y_pred):
        """derivative of cross entropy"""
        #return y*(y_pred-1)
        return y_pred-y

    def forward_pass(self,x,y,weights,bias):
        cost=0
        self.outputs=[]
        for i in range(len(weights)):
            samples=len(x)
            ones_array=np.ones(samples).reshape(samples,1)
            self.outputs.append(x) #append without adding ones array
            z=np.dot(np.append(ones_array,x,axis=1),weights[i]+bias[i])
            if i==len(weights)-1:
                x=self.softmax(z)
            else:
                x=self.activation(self.activations[i],z)
        self.outputs.append(x)
        self.y_pred=x

        temp=-self.max_log_likelihood(self.y_pred,y)
        cost=np.mean(np.sum(temp,axis=1))
        return cost


    def backward_pass(self,y,lr):
        for i in range(len(self.weights)-1,-1,-1):
            ones_array=np.ones(len(n.outputs[i])).reshape(len(n.outputs[i]),1)
            if i==len(self.weights)-1:
                prev_term=self.delta_mll(y,self.y_pred)
                # derivatives follow specific order,last three terms added new,rest from previ
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],ax
            else:
                prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(self.activ
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],ax
            self.weights[i]=self.weights[i]-lr*((self.derivatives[i].T)/len(y))
            self.bias[i]=self.bias[i]-lr*((self.derivatives[i].T)/len(y))

    def train(self,batches,lr=1e-3,epoch=10):
        """number of batches to split data in,Learning rate and epochs"""
        for epochs in range(epoch):
            samples=len(self.x)
            c=0
            for i in range(batches):
              x_batch=self.x[int((samples/batches)*i):int((samples/batches)*(i+1))]
              y_batch=self.y.loc[int((samples/batches)*i):int((samples/batches)*(i+1))-1]

              c=self.forward_pass(x_batch,y_batch,self.weights,self.bias)
              self.backward_pass(y_batch,lr)
            print(epochs,c/batches)

    def predict(self,x):
        """input: x_test values"""
        x=x/255
        for i in range(len(self.weights)):
            samples=len(x)
            ones_array=np.ones(samples).reshape(samples,1)
            z=np.dot(np.append(ones_array,x,axis=1),self.weights[i]+self.bias[i])
            if i==len(self.weights)-1:
                x=self.softmax(z)
            else:
                x=self.activation(self.activations[i],z)
        return np.argmax(x,axis=1)
```

```
n=NeuralNetworkActivations(X_train,Labels)
l1=Layer(100,'sigmoid')
l2=Layer(50, 'tanh')
n.connect(X_train,l1)
n.connect(l1,l2)
n.connect(l2,Labels)
n.train(batches=1000,lr=0.1,epoch=20)
```

```
pred=n.predict(X_test)
np.bincount(n.predict(X_test)),np.bincount(y_test)
```

```
print(f"accuracy is {np.bincount(np.abs(y_test-pred))[0]*100/len(y_test)} %")
```

Problem 6

Extend your code from problem 5 to implement momentum with your gradient descent. The momentum value will be passed as a parameter. Your function should perform “epoch” number of epochs and return the resulting weights.

```
# Problem 6

class Layer():
    """
    size: Number of nodes in the hidden layer
```

```
    size: number of nodes in the hidden layer
    activation: name of activation function for the layer
    """
    def __init__(self,size,activation='sigmoid'):
        self.shape=(1,size)
        self.activation=activation

class NeuralNetworkMomentum():
    def __init__(self,x,y):
        """
        x is 2d array of input images
        y are one hot encoded labels
        """
        self.x=x/255   # Divide by 255 to normalise the pixel values (0-255)
        self.y=y
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
        self.activations=[]
        self.delta_weights=[]
        self.delta_bias=[]

    def connect(self,layer1,layer2):
        """layer 2 of shape 1xn"""
        #Initialise weights,derivatives and activation lists
        self.derivatives.append(np.random.uniform(0,0.1,size=(layer1.shape[1]+1,layer2.shape[1
        self.weights.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[1])))
        self.bias.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[1])))
        self.delta_weights.append(np.zeros((layer1.shape[1]+1,layer2.shape[1])))
        self.delta_bias.append(np.zeros((layer1.shape[1]+1,layer2.shape[1])))
        if isinstance(layer2,Layer):
            self.activations.append(layer2.activation)

    def activation(self,name,z,derivative=False):

        #implementation of various activation functions and their derivatives
        if name=='sigmoid':
            if derivative==False:
                return 1/(1+np.exp(-z))
            else:
                return z*(1-z)
        elif name=='relu':
            if derivative==False:
                return np.maximum(0.0,z)
            else:
              z[z<=0] = 0.0
              z[z>0] = 1.0
              return z
        elif name=='tanh':
          if derivative==False:
                return np.tanh(z)
          else:
                return 1.0 - (np.tanh(z)) ** 2

    def softmax(self,z):
        e=np.exp(z)
        return e/np.sum(e,axis=1).reshape(-1,1)

    def max_log_likelihood(self,y_pred,y):
        """cross entropy"""
        return y*np.log(y_pred)

    def delta_mll(self,y,y_pred):
        """derivative of cross entropy"""
        #return y*(y_pred-1)
        return y_pred-y

    def forward_pass(self,x,y,weights,bias):
        cost=0
        self.outputs=[]
        for i in range(len(weights)):
            samples=len(x)
            ones_array=np.ones(samples).reshape(samples,1)
            self.outputs.append(x) #append without adding ones array
            z=np.dot(np.append(ones_array,x,axis=1),weights[i]+bias[i])
            if i==len(weights)-1:
                x=self.softmax(z)
            else:
                x=self.activation(self.activations[i],z)
        self.outputs.append(x)
        self.y_pred=x

        temp=-self.max_log_likelihood(self.y_pred,y)
        cost=np.mean(np.sum(temp,axis=1))
        return cost


    def backward_pass(self,y,lr,beta=0.9,momentum=False):
        for i in range(len(self.weights)-1,-1,-1):
            ones_array=np.ones(len(n.outputs[i])).reshape(len(n.outputs[i]),1)
            if i==len(self.weights)-1:
                prev_term=self.delta_mll(y,self.y_pred)
                # derivatives follow specific order,last three terms added new,rest from previ
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],ax
            else:
                prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(self.activ
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],ax
            if momentum:
                self.delta_weights[i]=beta*self.delta_weights[i]-lr*((self.derivatives[i].T)/l
                self.delta_bias[i]=beta*self.delta_bias[i]-lr*((self.derivatives[i].T)/len(y))
                self.weights[i]=self.weights[i]+self.delta_weights[i]
                self.bias[i]=self.bias[i]+self.delta_bias[i]
            else:
                self.weights[i]=self.weights[i]-lr*((self.derivatives[i].T)/len(y))
                self.bias[i]=self.bias[i]-lr*((self.derivatives[i].T)/len(y))

    def train(self,batches,beta,lr=1e-3,epoch=10):
        """number of batches to split data in,Learning rate and epochs"""
        for epochs in range(epoch):
            samples=len(self.x)
            c=0
            for i in range(batches):
              x_batch=self.x[int((samples/batches)*i):int((samples/batches)*(i+1))]
              y_batch=self.y.loc[int((samples/batches)*i):int((samples/batches)*(i+1))-1]

              c=self.forward_pass(x_batch,y_batch,self.weights,self.bias)
              self.backward_pass(y_batch,lr,beta,momentum=True)
            print(epochs,c/batches)
```

```
    def predict(self,x):
        """input: x_test values"""
        x=x/255
        for i in range(len(self.weights)):
            samples=len(x)
            ones_array=np.ones(samples).reshape(samples,1)
            z=np.dot(np.append(ones_array,x,axis=1),self.weights[i]+self.bias[i])
            if i==len(self.weights)-1:
                x=self.softmax(z)
            else:
                x=self.activation(self.activations[i],z)
        return np.argmax(x,axis=1)
```

```
n=NeuralNetworkMomentum(X_train,Labels)
l1=Layer(100,'sigmoid')
l2=Layer(50, 'tanh')
n.connect(X_train,l1)
n.connect(l1,l2)
n.connect(l2,Labels)
n.train(batches=500,lr=0.1,beta=0.5,epoch=20)
```

```
pred=n.predict(X_test)
np.bincount(n.predict(X_test)),np.bincount(y_test)
```

```
print(f"accuracy is {np.bincount(np.abs(y_test-pred))[0]*100/len(y_test)} %")
```