```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
```

Problem 1

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


X, y = load_data("/content/mnist_train.csv")
X_train, X_test, y_train, y_test = train_test_split(\
                X, y, test_size=0.3, random_state=42)

Labels=pd.get_dummies(y_train)
```

Problem 2

```
class Perceptron():
    def __init__(self,x,y):

        self.x=x/255
        self.y=y
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
        self.activations=[]

    def connect(self,layer1,layer2):

        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):

        return y*np.log(y_pred)

    def delta_mll(self,y,y_pred):

        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 term
        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):

    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):

        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 0.0003401843343197802
1 0.0007090628903846158
2 0.0012230189942198553
3 0.0010462089755032197
4 0.0014541018903940191
5 0.001228932740632526
6 0.0011014676209647803
7 0.0007765437294685917
8 0.0006393105805724627
9 0.0004940354853388284
10 0.00035139907643555316
11 0.0002761884306820409
12 0.00021789949126316225
13 0.00018595681253361898
14 0.0001332720344621988
15 0.00010878347888315236
16 8.434943171791935e-05
17 6.756742731237061e-05
18 5.839900786275416e-05
19 5.3612536730318665e-05
20 5.062935355968479e-05
21 4.990345542544511e-05
22 5.170962563667734e-05
23 5.370091690538339e-05
24 5.5034676400578765e-05
25 5.6537927286404e-05
26 5.799156363136345e-05
27 5.8809197977364746e-05
28 5.8387378452421855e-05
29 5.7143047433452556e-05
```

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

```
(array([313, 331, 293, 333, 326, 282, 284, 320, 231, 287]),
 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 88.86666666666666 %
```

Problem 3

```
class 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
        self.y=y
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
        self.activations=[]

    def connect(self,layer1,layer2):

        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):


        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):

        return y*np.log(y_pred)

    def delta_mll(self,y,y_pred):

        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)
```

```
            self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],axis=1))
        else:
            prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(self.activations[i],self.outputs[i+1],derivative=True)
            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):

    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):

    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 0.0001008144793084795
1 0.00014733920115241376
2 0.0001709062261311298
3 0.00015098365777877843
4 0.00013058410816732739
5 0.00010174485099844704
6 7.916115309203057e-05
7 6.853290951244363e-05
8 6.239210879386298e-05
9 5.625783854557697e-05
10 4.9184838079232994e-05
11 4.250928859506886e-05
12 3.7269593416415614e-05
13 3.3455991400362694e-05
14 3.0750888891067986e-05
15 2.907771442129905e-05
16 2.828417694716069e-05
17 2.8082908886447525e-05
18 2.81205221555986e-05
19 2.8087254599237476e-05
20 2.7797541861294578e-05
21 2.716458892267326e-05
22 2.621850090630246e-05
23 2.508451242412833e-05
24 2.3884129480428268e-05
25 2.2681728054366464e-05
26 2.1500474683325105e-05
27 2.0360755842215393e-05
28 1.9278026220152596e-05
29 1.826256889427404e-05
30 1.731948112188052e-05
31 1.6449158899080232e-05
32 1.5648675068516983e-05
33 1.4913134858713914e-05
34 1.423675532795608e-05
35 1.3613611204080357e-05
36 1.3038073849953853e-05
37 1.2505023265338023e-05
38 1.2009918399151147e-05
39 1.1548790576224833e-05
40 1.1118200563910816e-05
41 1.0715181279467232e-05
42 1.0337176856120494e-05
43 9.981983076879096e-06
44 9.647691759456303e-06
45 9.332640686542568e-06
46 9.035370065490602e-06
47 8.7545859031942e-06
48 8.489130108424058e-06
49 8.237956691764918e-06
```

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

```
(array([309, 332, 291, 314, 309, 267, 288, 330, 260, 300]),
 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 91.73333333333333 %
```

Problem 4

```
class Layer():

    def __init__(self,size,activation='sigmoid'):
        self.shape=(1,size)
        self.activation=activation

class DoubleLayerNeuralNetwork():
    def __init__(self,x,y):

        self.x=x/255
        self.y=y
        self.weights=[]
        self.bias=[]
```

```
    self.outputs=[]
    self.derivatives=[]
    self.activations=[]

def connect(self,layer1,layer2):

    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):

    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):

    return y*np.log(y_pred)

def delta_mll(self,y,y_pred):

    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)

            self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],axis=1))
        else:
            prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(self.activations[i],self.outputs[i+1],derivative=True)
            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):

    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):

    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)
```

```
(array([303, 331, 293, 308, 328, 270, 290, 324, 267, 286]),
 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 90.43333333333334 %
```

Problem 5

```
class Layer():
    def __init__(self,size,activation='sigmoid'):
        self.shape=(1,size)
        self.activation=activation

class NeuralNetworkActivations():
    def __init__(self,x,y):
        self.x=x/255
        self.y=y
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
        self.activations=[]
```

```
def connect(self,layer1,layer2):

    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):


    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):

    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)

            self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],axis=1))
        else:
            prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(self.activations[i],self.outputs[i+1],derivative=True)
            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):

    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):
```

```
        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)
```

```
0 0.0009937734186188378
1 0.00010192100864782828
2 0.0009626608739589575
3 0.0003046631528028078
4 0.0002794191112975886
5 0.0006674692543673785
6 0.000271193293841002
7 0.0004652709235676845
8 1.6752082248823547e-05
9 0.00013189470525134264
10 0.0003312577031024037
11 0.00035803235218961476
12 0.0011818297665540797
13 0.0001859707363231296
14 0.00012244064995101223
15 0.0010419741888859945
16 6.986892700462141e-05
17 3.9227622421434335e-05
18 2.211453099274933e-05
19 7.45718246896994e-05
```

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

```
(array([308, 328, 298, 364, 316, 241, 295, 327, 211, 312]),
 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 90.86666666666666 %
```

Problem 6

```
class Layer():

    def __init__(self,size,activation='sigmoid'):
        self.shape=(1,size)
        self.activation=activation

class NeuralNetworkMomentum():
    def __init__(self,x,y):

        self.x=x/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):

        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):

        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):

        return y*np.log(y_pred)

    def delta_mll(self,y,y_pred):

        #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,momentum=False,beta=0.9):
    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)

            self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],axis=1))
        else:
            prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(self.activations[i],self.outputs[i+1],derivative=True)
            self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i],axis=1))
        if momentum:
            self.delta_weights[i]=beta*self.delta_weights[i]-lr*((self.derivatives[i].T)/len(y))
            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,lr=1e-3,epoch=10,beta):

    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,momentum=True,beta)
        print(epochs,c/batches)

def predict(self,x):

    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,epoch=20,beta=0.5)
```

```
0 0.0020513956891835706
1 0.0008094322496495531
2 0.0010176446990968497
3 0.000998436669465249
4 9.714247948981801e-05
5 0.0001854465172970438 5
6 0.0002444237045170172
7 9.978876765232147e-05
8 6.155397975589383e-05
9 7.194069747141541e-05
10 0.0002123320510529555
11 3.836789826695124e-05
12 0.0001003804854241231 4
13 4.336477090875789e-05
14 9.733941014025356e-05
15 1.2188112752687686e-05
16 9.160967236061117e-06
17 7.784843230345855e-05
18 1.926124806973211e-06
19 2.070498195165521e-06
```

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

```
(array([299, 324, 310, 335, 301, 253, 292, 334, 266, 286]),
 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 92.83333333333333 %
```