Naveen_joon_ML_Lab9

April 24, 2020

```
In [1]: 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: Here you must read an input file. Each line contains 785 numbers (comma delimited): the first values are between 0.0 and 1.0 correspond to the 784 pixel values (black and white images), and the last number denotes the class label: 0 corresponds to digit 0, 1 corresponds to digit 1, etc.

0.1 Solution 1:

```
In [2]: 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
```

```
In [3]: 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=4

        Labels=pd.get_dummies(y_train)
```

Problem 2:Implement the backpropagation algorithm in a zero hidden layer neural network

0.2 Solution 2:

```
In [4]: class Perceptron():
            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
```

```python
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
        self.activations=[]

    def connect(self,layer1,layer2):
        """layer 2 of shape 1xn"""
        self.derivatives.append(np.random.uniform(0,0.1,size=(layer1.shape[1]+1,layer2
        self.weights.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape
        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_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)
            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)
            self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outputs[i
            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))
```

```python
    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))-

              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)
```

In [5]:
```python
n=Perceptron(X_train,Labels)
n.connect(X_train,Labels)
n.train(batches=1000,lr=0.2,epoch=30)
```

```
0 0.0007346536093938932
1 0.001333487169224761
2 0.0013481239936291142
3 0.00140891863962916
4 0.0013453292726588928
5 0.0013381109501742542
6 0.0012614553011848957
7 0.001106183056327947
8 0.0009913863734484585
9 0.0008757571024339127
10 0.0007338984410219489
11 0.0006864509257048199
12 0.0005775119638534283
13 0.00048018933575138346
14 0.00037368116723659746
15 0.0002783396432771148
16 0.00021805448702410285
17 0.00018112696097078769
18 0.00014298112213567696
19 0.000120601465357245
```

```
20 0.00011512104828964503
21 0.00010645510225767944
22 9.898418290156401e-05
23 9.471556040883543e-05
24 8.981066362311253e-05
25 8.515503292730198e-05
26 8.086352397337016e-05
27 7.612173743806715e-05
28 7.182886799448558e-05
29 6.86133687423989e-05
```

In [6]:
```
pred=n.predict(X_test)
np.bincount(n.predict(X_test)),np.bincount(y_test)
```

Out[6]:
```
(array([311, 333, 303, 327, 330, 280, 286, 325, 234, 271]),
 array([296, 327, 305, 326, 305, 283, 282, 336, 252, 288]))
```

In [7]:
```
print(f"accuracy: {np.bincount(np.abs(y_test-pred))[0]*100/len(y_test)} %")
```

```
accuracy: 89.0 %
```

Problem 3: Extend your code from problem 2 to support a single layer neural network with N hidden units

0.3 Solution 3:

In [8]:
```
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
        self.y=y
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
```

```python
        self.activations=[]

    def connect(self,layer1,layer2):
        """layer 2 of shape 1xn"""
        self.derivatives.append(np.random.uniform(0,0.1,size=(layer1.shape[1]+1,layer2
        self.weights.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shap
        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):
        """cross entropy"""
        return y*np.log(y_pred)

    def delta_mll(self,y,y_pred):
        """derivative of cross entropy"""
        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)
            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
```

```python
    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 fr
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outpu
            else:
                prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(se
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outpu
            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))-

              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)
```

In [9]:
```python
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 0.00026507369571026614
1 0.00025858462939753183
2 0.0003219614067774773
3 0.0003969718784473034
4 0.00040219823644652093
5 0.0003539709694831452
6 0.0002872648113062959
7 0.00022643033055996613
8 0.00017733474096884785
9 0.00013823525845028798
10 0.00010853908154940441
11 8.745823571803799e-05
12 7.289272699810053e-05
13 6.290169997826579e-05
14 5.5982952283681096e-05
15 5.0701449324372964e-05
16 4.612029222306469e-05
17 4.2153518585174634e-05
18 3.894917880239001e-05
19 3.640450110123102e-05
20 3.4349600142749704e-05
21 3.265893785975352e-05
22 3.1218027519601566e-05
23 2.9930836254492685e-05
24 2.8743384629089865e-05
25 2.7640925347847073e-05
26 2.6626127615626717e-05
27 2.5691004651994144e-05
28 2.481661925753757e-05
29 2.3986358964226792e-05
30 2.3192789990938606e-05
31 2.243617005877505e-05
32 2.17189028243342e-05
33 2.1041594716529147e-05
34 2.0402456071458396e-05
35 1.9798332478703507e-05
36 1.9225746165598018e-05
37 1.8681471640113977e-05
38 1.8162740072012184e-05
39 1.7667251520627377e-05
40 1.7193113843307993e-05
41 1.6738766155371304e-05
42 1.6302909147056948e-05
43 1.5884448022904003e-05
44 1.5482447477912558e-05
45 1.5096096376830381e-05
46 1.472467983609257e-05
47 1.4367556963838973e-05
```

```
48 1.4024143085109682e-05
49 1.3693895694613838e-05
```

In [10]:
```
pred=n.predict(X_test)
np.bincount(n.predict(X_test)),np.bincount(y_test)
```

Out[10]:
```
(array([302, 322, 293, 330, 310, 264, 296, 327, 253, 303]),
 array([296, 327, 305, 326, 305, 283, 282, 336, 252, 288]))
```

In [11]:
```
print(f"accuracy: {np.bincount(np.abs(y_test-pred))[0]*100/len(y_test)} %")
```

```
accuracy: 91.93333333333334 %
```

Problem 4: Extend your code from problem 3 (use cross entropy error) and implement a 2-layer neural network, starting with a simple architecture containing N hidden units in each layer

0.4 Solution 4:

In [12]:
```
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
        self.y=y
        self.weights=[]
        self.bias=[]
        self.outputs=[]
        self.derivatives=[]
        self.activations=[]

    def connect(self,layer1,layer2):
        """layer 2 of shape 1xn"""
        self.derivatives.append(np.random.uniform(0,0.1,size=(layer1.shape[1]+1,layer2
        self.weights.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.sha
        self.bias.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[
        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):
        """cross entropy"""
        return y*np.log(y_pred)

    def delta_mll(self,y,y_pred):
        """derivative of cross entropy"""
        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)
            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.outpu
```

```
                else:
                    prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(se
                    self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outpu
                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)

                  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)
```

In [13]:
```
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)
```

```
0 0.0006147752088982143
1 0.00022125939912860426
2 0.00011095514297885743
3 7.204884572833662e-05
4 5.308982882593679e-05
5 4.081481599292317e-05
6 3.72397198080972e-05
```

```
7 3.457702926569818e-05
8 3.6016675373810834e-05
9 4.140543148333343e-05
10 4.646969123463548e-05
11 4.9277103508864534e-05
12 5.152655618035941e-05
13 5.163862117340579e-05
14 4.846590370833118e-05
15 4.41444585177594e-05
16 3.972547920999699e-05
17 3.530803170294093e-05
18 3.119814553556046e-05
19 2.776326929273412e-05
```

In [14]:
```python
pred=n.predict(X_test)
np.bincount(n.predict(X_test)),np.bincount(y_test)
```

Out[14]:
```
(array([306, 335, 294, 320, 327, 288, 283, 332, 238, 277]),
 array([296, 327, 305, 326, 305, 283, 282, 336, 252, 288]))
```

In [15]:
```python
print(f"accuracy: {np.bincount(np.abs(y_test-pred))[0]*100/len(y_test)} %")
```

```
accuracy: 91.36666666666666 %
```

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.

0.5 Solution 5:

In [16]:
```python
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
        self.y=y
```

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

    def connect(self,layer1,layer2):
        """layer 2 of shape 1xn"""
        self.derivatives.append(np.random.uniform(0,0.1,size=(layer1.shape[1]+1,layer2
        self.weights.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shap
        self.bias.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.shape[
        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):
        """derivative of cross entropy"""
        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)
            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.outpu
            else:
                prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activation(se
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.outpu
            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)

              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)
```

In [17]:
```
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 6.393954821616651e-05
1 0.0005118562710481734
2 0.00023085754588267473
3 0.00016826567329880513
4 0.0002573099462506769
5 0.0006350785332999381
6 4.7923176876536626e-05
7 3.840975191833874e-05
8 3.430696135163668e-05
9 1.0051623852615276e-05
10 0.0005127406062526722
11 4.717236913963525e-06
12 0.00034750915286958333
13 2.7764888942601524e-06
14 7.91851544017604e-06
15 5.149784307016578e-07
16 1.7189875833741053e-05
17 1.1657772127244548e-05
18 2.1612809181756667e-05
19 1.0198810562150977e-06
```

In [18]:
```
pred=n.predict(X_test)
np.bincount(n.predict(X_test)),np.bincount(y_test)
```

Out[18]:
```
(array([305, 329, 296, 331, 308, 265, 286, 338, 246, 296]),
 array([296, 327, 305, 326, 305, 283, 282, 336, 252, 288]))
```

In [19]:
```
print(f"accuracy: {np.bincount(np.abs(y_test-pred))[0]*100/len(y_test)} %")
```

```
accuracy: 92.9 %
```

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.

0.6 Solution 6:

In [20]:
```
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 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,la
        self.weights.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2
        self.bias.append(np.random.uniform(-1,1,size=(layer1.shape[1]+1,layer2.sha
        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,momentum=False,beta=0.5):
        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,re
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.c
            else:
                prev_term=np.dot(prev_term,self.weights[i+1][1:].T)*self.activatio
                self.derivatives[i]=np.dot(prev_term.T,np.append(ones_array,self.c
            if momentum:
                self.delta_weights[i]=beta*self.delta_weights[i]-lr*((self.derivat
                self.delta_bias[i]=beta*self.delta_bias[i]-lr*((self.derivatives[i
                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=0.5):
        """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

              c=self.forward_pass(x_batch,y_batch,self.weights,self.bias)
              self.backward_pass(y_batch,lr,momentum=True,beta=0.5)
            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)
```

In [21]:
```
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 0.0007368871858396068
1 4.896638893996719e-05
2 0.00014723111518745942
3 9.195402384843673e-05
4 8.46722106169578e-05
5 0.0004254245940298797
6 0.0004518025915654164
7 0.0005144148604648345
8 0.00034785497903570513
9 6.636344183019607e-05
10 0.00017596769111351266
11 1.2285565898844749e-05
12 9.561366468641029e-06
13 3.946537442020382e-05
14 3.768419621026274e-06
15 1.4724958080051827e-06
16 9.895523345277317e-07
17 1.1006430491781112e-06
18 1.0262939221277626e-06
19 9.465183321892398e-07
```

In [22]:
```
pred=n.predict(X_test)
np.bincount(n.predict(X_test)),np.bincount(y_test)
```

Out[22]:
```
(array([304, 326, 293, 323, 296, 271, 295, 335, 259, 298]),
 array([296, 327, 305, 326, 305, 283, 282, 336, 252, 288]))
```

In [23]:
```
print(f"accuracy: {np.bincount(np.abs(y_test-pred))[0]*100/len(y_test)} %")
```

```
accuracy: 92.6 %
```

In [ ]: