BackPropagation

Loading data

In [148]:
```
import pickle
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt


with open("C:/Users/91888/Desktop/Assignment/BackPropagation Assignment/data.pkl", 'rb') as f:
    data = pickle.load(f)
print(data.shape)
X = data[:, :5]
y = data[:, -1]
print(X.shape, y.shape)
```

```
(506, 6)
(506, 5) (506,)
```

Task 1: Implementing backpropagation and Gradient checking

In [149]:
```
def sigmoid(z):
    return(1/(1 + np.exp(-z)))
```

In [150]:
```
#weight initialization
w=np.ones(9)*0.1
```

In [151]:
```
def forward_propagation(X, y, w):

    #part1
    exp=((w[0]*X[0]+w[1]*X[1])*(w[0]*X[0]+w[1]*X[1]))+w[5]
    exp=np.exp(exp)

    #part2
    tanh=(np.tanh(exp+w[6]))

    #part3
    sig=np.sin(w[2]*X[2])*(w[3]*X[3]+w[4]*X[4])+w[7]
    sig=(sigmoid(sig))

    #compute Y'
    Y1=(sig*w[8]+tanh)

    #compute Loss

    L=(y-Y1)*(y-Y1)

    #derivative of L w.r.t. Y1
    dl=(-2*y)+2*Y1

    #storing values in dict
    dict={}


    dict['dy_pr']=dl
    dict['loss']=L
    dict['exp']=exp
    dict['tanh']=tanh
    dict['sigmoid']=sig



    return dict
```

Grader function - 1

In [152]:
```
def grader_sigmoid(z):
  val=sigmoid(z)
  assert(val==0.8807970779778823)
  return True
grader_sigmoid(2)
```

Out[152]: True

Grader function - 2

In [153]:
```
def grader_forwardprop(data):
    dl = (data['dy_pr']==-1.9285278284819143)
    loss=(data['loss']==0.9298048963072919)
    part1=(data['exp']==1.1272967040973583)
    part2=(data['tanh']==0.8417934192562146)
    part3=(data['sigmoid']==0.5279179387419721)
    assert(dl and loss and part1 and part2 and part3)
    return True
w=np.ones(9)*0.1
d1=forward_propagation(X[0],y[0],w)
grader_forwardprop(d1)
```

Out[153]: True

Backward propagation

In [154]:
```
def backward_propagation(X,W,dict):

    dw={}

    dw['dw9']=dict['dy_pr']*dict['sigmoid']
    dw['dw8']=dict['dy_pr']*w[8]*(dict['sigmoid']*(1-dict['sigmoid']))
    dw['dw7']=dict['dy_pr']*(1-(np.tanh(dict['exp']+w[6])**2))
    dw['dw6']=dw['dw7']*dict['exp']
    dw['dw5']=dw['dw8']*(np.sin(X[2]*w[2]))*X[4]
    dw['dw4']=dw['dw8']*(np.sin(X[2]*w[2]))*X[3]
    dw['dw3']=dw['dw8']*X[2]*(X[3]*w[3]+X[4]*w[4])*(np.cos(X[2]*w[2]))
    dw['dw2']=2*(((w[0]*X[0])*X[1]*dw['dw6'])+((w[1]*X[1])*X[1]*dw['dw6']))
    dw['dw1']=2*(((w[0]*X[0])*X[0]*dw['dw6'])+((w[1]*X[1])*X[0]*dw['dw6']))


    return dw
```

Grader function - 3

In [155]:
```
def grader_backprop(data):
    dw1=(data['dw1']==-0.22973323498702003)
    dw2=(data['dw2']==-0.021407614717752925)
    dw3=(data['dw3']==-0.005625405580266319)
    dw4=(data['dw4']==-0.004657941222712423)
    dw5=(data['dw5']==-0.0010077228498574246)
    dw6=(data['dw6']==-0.6334751873437471)
    dw7=(data['dw7']==-0.561941842854033)
    dw8=(data['dw8']==-0.04806288407316516)
    dw9=(data['dw9']==-1.0181044360187037)
    assert(dw1 and dw2 and dw3 and dw4 and dw5 and dw6 and dw7 and dw8 and dw9)
    return True
w=np.ones(9)*0.1
d1=forward_propagation(X[0],y[0],w)
d1=backward_propagation(X[0],w,d1)
grader_backprop(d1)
```

Out[155]:
```
True
```

Implement gradient checking

In [156]:

```
#weight initialization
w=np.ones(9)*0.1
def gradient_checking(X,y,w):

    approx_gradients = []
    diff=[]
    e=10e-7


    for i in range(0,9):

        wplus=np.copy(w)
        wplus[i]=wplus[i]+e


        wminus=np.copy(w)
        wminus[i]=wminus[i]-e


        Lplus=forward_propagation(X,y,wplus)

        Lminus=forward_propagation(X, y,wminus)


        approx_grad=((Lplus['loss']-Lminus['loss'])/(2*e))


        approx_gradients.append(approx_grad)


        # compute the gradients of W using backword_propagation()
        dict=forward_propagation(X,y,w)
        grad=backward_propagation(X,y,dict)
        #storing all dict values to list
        grad=list(grad.values())
        grad=list(reversed(grad))

        #gradient checking
    for i in range(0,9):
        numerator=np.linalg.norm(grad[i]-(approx_gradients[i]))
        denominator=np.linalg.norm(grad[i])+np.linalg.norm(approx_gradients[i])
```

```
        difference=(numerator/denominator)

        diff.append(difference)

        if difference <1e-7:
            print("The gradient is correct")
        else:
            print("TThe gradient is wrong")



    return diff
```

In [157]:
```
gradient_checking(X[0],y[0],w)
```

```
The gradient is correct
The gradient is correct
The gradient is correct
The gradient is correct
The gradient is correct
The gradient is correct
The gradient is correct
The gradient is correct
The gradient is correct
```

Out[157]:
```
[1.1219604246051178e-10,
 2.1561949116218244e-09,
 4.915159582558029e-10,
 9.416722967098557e-09,
 1.223599362131019e-10,
 5.787853497388374e-11,
 8.271958270840584e-11,
 1.207126897442854e-10,
 7.842408542444892e-12]
```

Task 2 : Optimizers

Algorithm with Vanilla update of weights

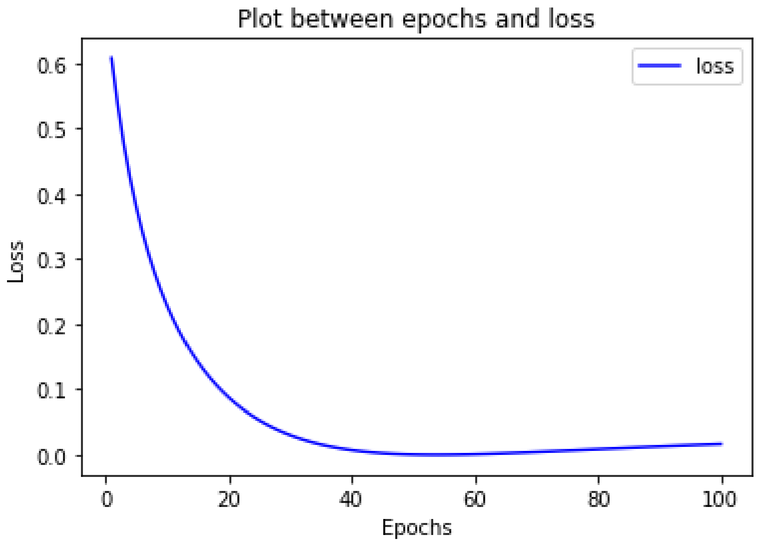
In [158]:
```
#weight Initailization
w2 = np.random.normal(0, 0.01,9)
w2
```

Out[158]:
```
array([ 0.00137162, -0.00015334,  0.01460165,  0.00288842, -0.02071837,
       -0.01384972,  0.02320689, -0.0039296 ,  0.00279488])
```

In [159]:
```
loss=[]
for i in range(1,101):

    for j in range(0,506):

        learning_rate=0.0001
        d1=forward_propagation(X[j],y[j],w2)

        grad=backward_propagation(X[j],w2,d1)
        #storing all dict values to list
        grad=list(grad.values())
        grad=list(reversed(grad))

        #Vanilla update

        for k in range(len(grad)):

            w2[k]=w2[k]-(learning_rate*grad[k])

    #appending loss in list for each epoch
    loss.append((d1['loss']))
```

Plot between epochs and loss

In [160]:
```
loss_val = loss
epochs = range(1,101)
plt.plot(epochs, loss_val, 'b', label='loss')

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title("Plot between epochs and loss")
plt.legend()
plt.show()
```



Algorithm with Momentum update of weights

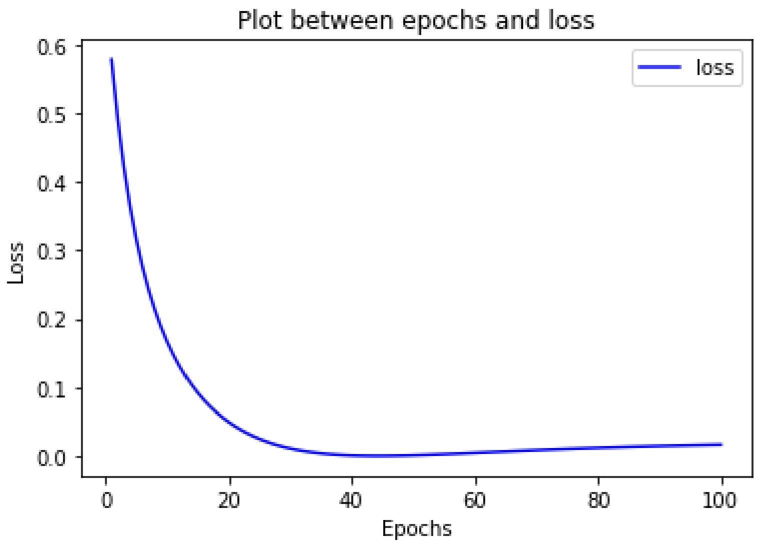
In [161]:
```
#weight Initailization
w3 = np.random.normal(0, 0.01,9)
w3
```

Out[161]:
```
array([-0.03102333, -0.01891686,  0.02527777, -0.00698577,  0.00824362,
        0.01028429,  0.00159301, -0.00818121,  0.01060756])
```

In [162]:
```
loss1=[]
v=0
mu=0.5
learning_rate=0.0001
for i in range(1,101):

    for j in range(0,506):


        d1=forward_propagation(X[j],y[j],w3)

        grad=backward_propagation(X[j],w3,d1)
        #storing all dict values to list
        grad=list(grad.values())
        grad=list(reversed(grad))

        #Momentum update

        for k in range(len(grad)):
            v = mu * v - (learning_rate*grad[k])

            w3[k]=w3[k]+v


    loss1.append((d1['loss']))
```

Plot between epochs and loss

In [163]:
```
loss_val = loss1
epochs = range(1,101)
plt.plot(epochs, loss_val, 'b', label='loss')

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title("Plot between epochs and loss")
plt.legend()
plt.show()
```



Algorithm with Adam update of weights

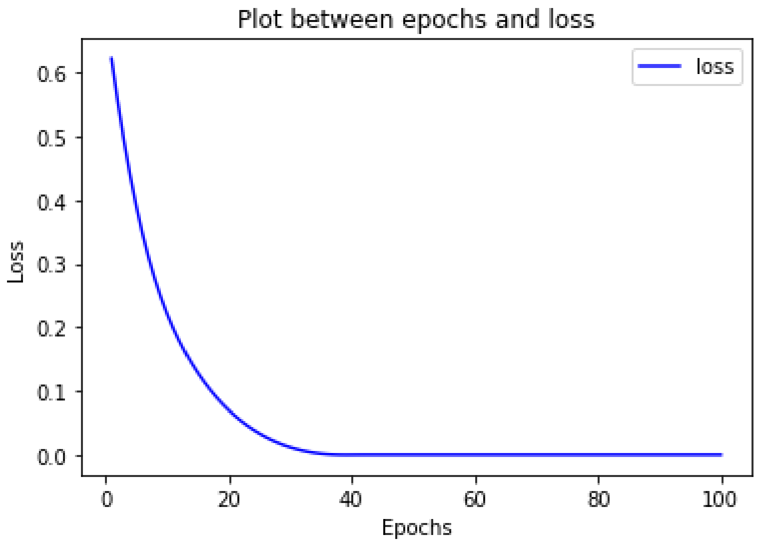
In [164]:
```
#weight Initailization
w4 = np.random.normal(0, 0.01,9)
w4
```

Out[164]:
```
array([-0.01419376, -0.00813039, -0.01426229,  0.00724283, -0.00867423,
       -0.00638041, -0.00550793,  0.00618691, -0.00987836])
```

In [165]:
```
loss2=[]
v=0
m=0
learning_rate=0.0001
eps = 1e-8
beta1 = 0.9
beta2 = 0.999

for i in range(1,101):

    for j in range(0,506):


        d1=forward_propagation(X[j],y[j],w4)

        grad=backward_propagation(X[j],w4,d1)
        #storing all dict values to list
        grad=list(grad.values())
        grad=list(reversed(grad))

        #Momentum update

        for k in range(len(grad)):

            m = beta1*m + (1-beta1)*grad[k]
            v = beta2*v + (1-beta2)*(grad[k]**2)
            w4[k] =w4[k] - learning_rate * m / (np.sqrt(v) + eps)



    loss2.append((d1['loss']))
```

Plot between epochs and loss

In [166]:
```
loss_val = loss2
epochs = range(1,101)
plt.plot(epochs, loss_val, 'b', label='loss')

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title("Plot between epochs and loss")
plt.legend()
plt.show()
```



Comparision plot between epochs and loss with different optimizers

In [167]:
```
epochs = range(1,101)
plt.plot(epochs, loss, 'r', label='Vanilla')
plt.plot(epochs, loss1, 'g', label='Momentum')
plt.plot(epochs, loss2, 'b', label='Adam')
plt.grid()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title("Plot between epochs and loss")
plt.legend()
plt.show()
```