***For this Assignment  - do the following:***

1) Start with a NN that has three inputs and 3 rows (so X is 3 by 3), and one hidden layer H with two units, h1 and h2, and one output, y^.

n = 3

X = Diagram, text

Description automatically generated with medium confidence

W = Graphical user interface, application

Description automatically generated

n by 2 (c)

2) Use Word or PowerPoint (not paper ;) to draw the network and label it with the appropriate w's. Use b1 for the h1 bias, b2 for the h2 bias, and c for the output bias.

Diagram

Description automatically generated

3) Use a "z" for linear combinations (with the appropriate indices).

z11 = w11x1 + w21x2 + w31x3 + b1

z12 = w12x1 + w22x2 + w32x3 + b2

4) Use the sigmoid as the activation function.

def sigmoid(value, dS=False):

if (dS == True):

return value \* (1 - value)

return 1 / (1 + np.exp(-value))

5) As a first step - create all the general matrices: W1, W2, Z1, H1, Z2, y^

h1 = S(z11), where S is the sigmoid activation function

h2 = S(z12)

H1 = S(Z1)

Diagram, text

Description automatically generated with medium confidence @ Graphical user interface, application

Description automatically generated + b1 b2

The matrix of Z1:

zi1     zi2   
x11w11+x12w21+x13w31+b1  x11w12+x12w22+x13w32+b2    
x21w11+x22w21+x23w31+b1  x21w12+x22w22+x23w32+b2    
x31w11+x32w21+x33w31+b1  x31w12+x32w22+x33w32+b2 

Z2 = S(Z1) @ W2 = S(Z1) @ [w1 w2] = w1h1 + w2h2 + c

y^ = S(Z2)

**Step 2**

Once the network is labeled, create all of the matrices.

Use **X** = [ [1, 2, 3], [-1, -2, -3], [3, 4, -2]  ]      (3 rows and 3 columns)

**Use Y**= [  [1], [0], [0]  ]

**For W1  (from X to H)**

For w11, w21, and w31, use 1, 2, and 3.

For w12, w22, and w32, use -1, -2, and -3

For **W2**  (from H to y^) use the values 4 and 5

For b1 and b2, **use 1 and 2**,  and for c **use 3**.

You will also need to create the Z's and H's, etc.

Activation is the sigmoid.

## Set up

The input is:

[[ 1 2 3]

[-1 -2 -3]

[ 3 4 -2]]

The target value are:

[[1]

[0]

[0]]

The W1 weights for the x's are:

[[ 1 -1]

[ 2 -2]

[ 3 -3]]

The b are:

[[1 2]]

The W2\_h are:

[[4]

[5]]

The c are:

3

##

**Feed the above forward through the network for all input values.** Clearly show all steps, math, and labels so it is easy to see what was done.

You will generate  3 "y^" output values. One for each input vector.

**The Loss Function will be   1/2(y^ - y)2**

 - Calculate the **average** loss/cost for all 3 values and

 - Calculate the **total**loss**/**cost for all 3 inputs.

Write this out clearly.

## Output

The Z1 are: (Z1 = X @ W1\_x + bs)

[[ 15 -12]

[-13 16]

[ 6 -3]]

The H1 are: sigmoid(Z1)

[[9.99999694e-01 6.14417460e-06]

[2.26032430e-06 9.99999887e-01]

[9.97527377e-01 4.74258732e-02]]

The output is: Z2 = (A\_Z1 @ W2\_h) + c; sigmoid(Z2)

[[0.99908898]

[0.99966465]

[0.999274 ]]

The **average** error is: .5 \* (np.mean(np.square((output - y)))

0.3329727527221232

The **total** loss is: .5 \* (np.sum(np.square(output - y)))

0.9989182581663696

**Step 3 -**

Now you have a labeled NN, all the related matrices, and the cost of the network with the current weights and biases, etc. you can create the gradient.

The gradient is the vector of partial derivatives for all weights and biases.

dL/dw11 ... dL/c

Show all derivatives in the chain for each calculation, do the calculation, and get a number for each derivative.

Hint: your will have 8 weight derivatives and 3 bias derivatives in your gradient. Please make sure everything is clearly labeled.

The W1 gradient is: XT (H)(1 - H) (Y^ - Y)(Y^)(1-Y^)(W2) = self.X\_H\_D\_Error\_W2

[[ 2.20153681e-05 4.81847576e-04]

[ 2.93517921e-05 6.42463292e-04]

[-1.46880878e-05 -3.21232502e-04]]

The W2 gradient is: (H)T (Y^ - Y)(Y^)(1-Y^) = self.h\_output\_delta

[[0.00074098]

[0.00037117]]

The biases b gradient is: (H)(1 - H) (Y^ - Y)(Y^)(1-Y^)(W2) = self.H\_D\_Error\_W2

[[-1.02802121e-12 -2.56205382e-11]

[ 3.04690309e-09 1.88228263e-10]

[ 7.33947200e-06 1.60615930e-04]]

The bias c gradient is: (Y^ - Y)(Y^)(1-Y^) = self.output\_delta

[[-8.41389066e-07]

[ 3.37494095e-04]

[ 7.43666151e-04]]

**Step 4**

Using a learning rate of .01 (you can also experiment with LRs), update all weights and biases to the \_new value by subtracting the learning rate times the derivative from the original weight or bias value.

Show all the work and steps.

Self.LR = 0.01

self.W1\_x = self.W1\_x - self.LR \* (self.X\_H\_D\_Error\_W2)

New W1:

[[ 0.99978266 -1.00486524]

[ 1.99971023 -2.00648699]

[ 3.000145 -2.9967565 ]]

self.W2\_h = self.W2\_h - self.LR \* (self.h\_output\_delta)

New W2:

[[3.99268368]

[4.99629671]]

self.bs = self.bs - self.LR \* self.H\_D\_Error\_W2

Updated bs are:

[[1. 2. ]

[0.99999997 2. ]

[0.99992754 1.99837825]]

self.c = self.c - self.LR \* self.output\_delta

Updated c's are:

[[3.00000835]

[2.99663694]

[2.99265715]]

**Step 5: Write code for this with 1000 epochs**

Print the critical items and shapes so that you can see if your code is doing what you expect.

*import* numpy *as* np  
*import* matplotlib.pyplot *as* plt  
  
  
*class* neural\_nets(object):  
  
 @staticmethod  
 *def* Sigmoid(value, deriva=*False*):  
 *if* deriva:  
 *return* value \* (1 - value)  
 *return* 1 / (1 + np.exp(-value))  
  
 *def* \_\_init\_\_(*self*, X, y, bs, c, W1\_x, W2\_h):  
 *self*.LR = 0.01  
 *self*.X = X  
 *self*.y = y  
  
 *self*.bs = bs  
 *self*.c = c  
 *self*.W1\_x = W1\_x  
 *self*.W2\_h = W2\_h  
  
 *self*.z = *None  
 self*.h = *None  
 self*.z2 = *None  
  
 self*.GA = *False  
  
 def* train(*self*, X, y):  
 output = *self*.FeedForward(X)  
 *self*.BackProp(X, y, output)  
 *return* output  
  
 *def* FeedForward(*self*, X):  
 print("FeedForward:\n")  
 *self*.z = (np.dot(X, *self*.W1\_x)) + *self*.bs *# X is n by c W1 is c by h --> n by h* print("Z1 is:\n", *self*.z)  
 *self*.h = *self*.Sigmoid(*self*.z) *# activation function shape: n by h* print("H is:\n", *self*.h)  
 *self*.z2 = (np.dot(*self*.h, *self*.W2\_h)) + *self*.c *# n by h @ h by o --> n by o* print("Z2 is:\n", *self*.z2)  
 output = *self*.Sigmoid(*self*.z2)  
 print("y^ is:\n", output)  
 *return* output  
  
 *def* BackProp(*self*, X, y, output):  
 print("\nBackProp:\n")  
 *# Y^ - Y  
 self*.output\_error = output - y  
 *# print("Y^ - Y\n", self.output\_error)  
 # print("SIG Y^\n", self.Sigmoid(output, deriva=True))  
  
 # (Y^ - Y)(Y^)(1-Y^)  
 self*.output\_delta = *self*.output\_error \* *self*.Sigmoid(output, deriva=*True*)  
 *# print("D\_Error (Y^)(1-Y^)(Y^-Y) is:\n", self.output\_delta)  
  
 # (Y^ - Y)(Y^)(1-Y^)(W2)  
 self*.D\_Error\_W2 = *self*.output\_delta.dot(*self*.W2\_h.T) *# D\_Error times W2  
 # print("W2 is\n", self.W2)  
 # print(" D\_Error times W2\n", self.D\_Error\_W2)  
  
 # (H)(1 - H) (Y^ - Y)(Y^)(1-Y^)(W2)  
 self*.H\_D\_Error\_W2 = *self*.D\_Error\_W2 \* *self*.Sigmoid(*self*.h, deriva=*True*)  
 *# Note that \* will multiply respective values together in each matrix  
  
 # print("Derivative sig H is:\n", self.Sigmoid(self.h, deriva=True))  
 # print("self.H\_D\_Error\_W2 is\n", self.H\_D\_Error\_W2)  
  
 # ------UPDATE weights and biases ------------------  
  
 # XT (H)(1 - H) (Y^ - Y)(Y^)(1-Y^)(W2)  
 self*.X\_H\_D\_Error\_W2 = X.T.dot(*self*.H\_D\_Error\_W2) *# this is dW1  
  
 # (H)T (Y^ - Y)(Y^)(1-Y^)  
 self*.h\_output\_delta = *self*.h.T.dot(*self*.output\_delta) *# this is dW2  
  
 # print("the gradient :\n", self.X\_H\_D\_Error\_W2)  
 # print("the gradient average:\n", self.X\_H\_D\_Error\_W2/self.n)  
  
 if self*.GA == "True":  
 print("Using average gradient........\n")  
 *# self.W1\_x = self.W1\_x - self.LR \* (self.X\_H\_D\_Error\_W2 / 3)  
 # self.W2\_h = self.W2\_h - self.LR \* (self.h\_output\_delta / 3) ## average the gradients  
 # #print("New W1: \n", self.W1)  
 else*:  
 *# print("Using sum gradient........\n")  
 self*.W1\_x = *self*.W1\_x - *self*.LR \* *self*.X\_H\_D\_Error\_W2 *# c by h first set (input -> hidden) weights  
 self*.W2\_h = *self*.W2\_h - *self*.LR \* *self*.h\_output\_delta *# adjusting second set (hidden -> output) weights* print("New W1: \n", *self*.W1\_x)  
 print("New W2: \n", *self*.W2\_h)  
 *# print("The b biases before the update are:\n", self.bs)  
 self*.bs = *self*.bs - *self*.LR \* *self*.H\_D\_Error\_W2  
 *# print("The H\_D\_Error\_W2 is...\n", self.H\_D\_Error\_W2)* print("Updated bs are:\n", *self*.bs)  
  
 *self*.c = *self*.c - *self*.LR \* *self*.output\_delta  
 print("Updated c's are:\n", *self*.c)  
  
 *# print("The W1 is: \n", self.W1\_x)* print("The W1 gradient is: \n", *self*.X\_H\_D\_Error\_W2)  
 *# print("The W1 gradient average is: \n", self.X\_H\_D\_Error\_W2/self.n)* print("The W2 gradient is: \n", *self*.h\_output\_delta)  
 *# print("The W2 gradient average is: \n", self.h\_output\_delta/self.n)* print("The biases b gradient is:\n", *self*.H\_D\_Error\_W2)  
 print("The bias c gradient is: \n", *self*.output\_delta)  
  
  
*# Set up*X = np.array([[1, 2, 3], [-1, -2, -3], [3, 4, -2]])  
  
y = np.array([[1], [0], [0]])  
  
W1\_x = np.array([[1, -1], [2, -2], [3, -3]])  
  
bs = np.array([[1, 2]])  
  
W2\_h = np.array([[4], [5]])  
  
c = 3  
  
*# Initial*NN = neural\_nets(X, y, bs, c, W1\_x, W2\_h)  
  
TotalLoss = []  
AverageLoss = []  
Epochs = 1000  
  
*for* i *in* range(Epochs):  
 print("Iteration ", i + 1)  
 output = NN.train(X, y)  
  
 print("Total Loss:", .5 \* (np.sum(np.square(output - y))))  
 TotalLoss.append(.5 \* (np.sum(np.square(output - y))))  
  
 print("Average Loss:", .5 \* (np.mean(np.square((output - y)))))  
 AverageLoss.append(.5 \* (np.mean(np.square((output - y)))))  
  
*# Plot*fig1 = plt.figure()  
ax = plt.axes()  
x = np.linspace(0, 1000, Epochs)  
ax.plot(x, TotalLoss)  
plt.show()  
  
fig2 = plt.figure()  
ax = plt.axes()  
x = np.linspace(0, 1000, Epochs)  
ax.plot(x, AverageLoss)  
plt.show()

Iteration 1000

FeedForward:

Z1 is:

[[ 14.99963851 -12.00810068]

[-12.99963854 16.00810068]

[ 5.99782865 -3.04860427]]

H is:

[[9.99999694e-01 6.09460394e-06]

[2.26114146e-06 9.99999888e-01]

[9.97522015e-01 4.52777694e-02]]

Z2 is:

[[6.99272866]

[7.99294921]

[7.20168319]]

y^ is:

[[0.99908231]

[0.99966228]

[0.99925522]]

BackProp:

New W1:

[[ 0.99978266 -1.00486524]

[ 1.99971023 -2.00648699]

[ 3.000145 -2.9967565 ]]

New W2:

[[3.99268368]

[4.99629671]]

Updated bs are:

[[1. 2. ]

[0.99999997 2. ]

[0.99992754 1.99837825]]

Updated c's are:

[[3.00000835]

[2.99663694]

[2.99265715]]

The W1 gradient is:

[[ 2.20153681e-05 4.81847576e-04]

[ 2.93517921e-05 6.42463292e-04]

[-1.46880878e-05 -3.21232502e-04]]

The W2 gradient is:

[[0.00074098]

[0.00037117]]

The biases b gradient is:

[[-1.02802121e-12 -2.56205382e-11]

[ 3.04690309e-09 1.88228263e-10]

[ 7.33947200e-06 1.60615930e-04]]

The bias c gradient is:

[[-8.41389066e-07]

[ 3.37494095e-04]

[ 7.43666151e-04]]

Total Loss: 0.9989182581663696

Average Loss: 0.3329727527221232

Total loss plot:

Chart, line chart

Description automatically generated

Average loss plot:

Chart, line chart

Description automatically generated