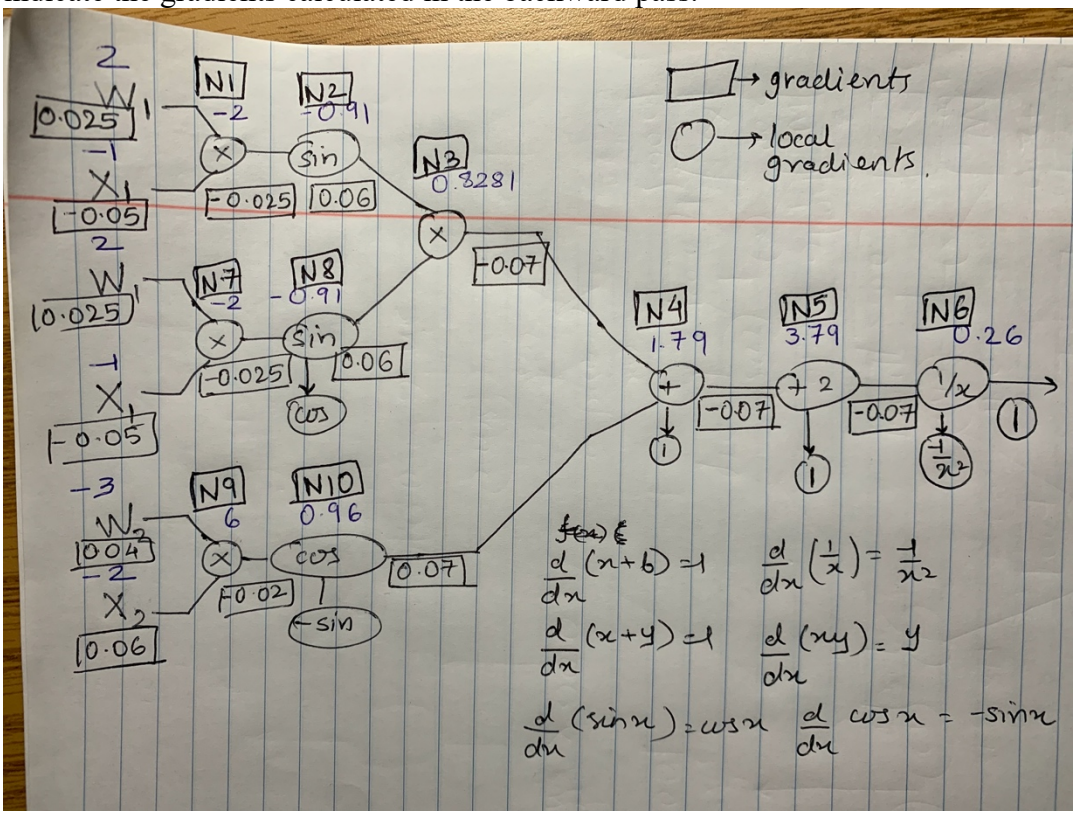
Intro to Deep Learning ECE 579
HW-2
Name: Shounak Rangwala
Netid: snr85

Q1
Part a)
I took the inputs as :
W1=2
X1=-1
W2=-3
X2=-2

With those I made the computation graph as shown in the photo below.
The blue numbers indicate the result of the forward pass and the black numbers in the boxes indicate the gradients calculated in the backward pass.



Part b)

Adopting the OOP model, I thought since the nodes were where the forward and backward propagation computation s were being held, I can make each different type of Node a class and then call an object of that class in my final implementation. I have provided both the python notebook file and the .py script but for your understanding I am also attaching the photo of the same below.

Each node will take input in some form (depending the number of operands needed for the operation). It will compute the "`output`" which is the result for forward propagation. The "`localGradient`" method will compute the local gradient of the node with the input given to the node. The "`downstream`" method is same in all Node classes which is basically multiplying the upstream gradient to the `localGradient` to get the downstream gradient which is used for backward propagation.

In [8]:
```
class NodeCosine ():
    input = 1
    def __init__(self,num):
        self.input=num
    def output(self):
        result = math.cos(self.input)
        return result
    def localGradient(self):
        return -math.sin(self.input)
    def downstream(self,upstream):
        return self.localGradient()*upstream
```

In [9]:
```
class NodeMultiply ():
    input1 = 1
    input2 =1
    def __init__(self,num1,num2):
        self.input1=num1
        self.input2=num2
    def output(self):
        result = self.input1*self.input2
        return result
    def localGradient1(self):
        return self.input2
    def localGradient2(self):
        return self.input1
    def downstream1(self,upstream):
        return self.localGradient1()*upstream
    def downstream2(self,upstream):
        return self.localGradient2()*upstream
```

The final class I made was for the function itself with 2 methods. The first is called "`forward`" which return the result of forward propagation. The second is called "`backward`" which return the result of backward propagation (df/dW1, df/dX1, df/dW2, df/X2). The nodes have been numbered and initialized as shown in the code for each method.

```
class Fx():
    W1=0
    W2=0
    X1=0
    X2=0
    def __init__(self,a,b,c,d):
        self.W1=a
        self.X1=b
        self.W2=c
        self.X2=d
    def forward (self):
        N1=NodeMultiply(self.W1,self.X1)
        N2=NodeSine(N1.output())
        N7=NodeMultiply(self.W1,self.X1)
        N8=NodeSine(N7.output())
        N3=NodeMultiply(N2.output(),N8.output())
        N9 = NodeMultiply(self.W2,self.X2)
        N10=NodeCosine(N9.output())
        N4 = NodeAdd(N3.output(),N10.output())
        N5 = NodeLinear(N4.output(),2)
        N6 = NodeInverse(N5.output())
        result= N6.output()
        return result
    def backward (self):
        N1=NodeMultiply(self.W1,self.X1)
        N2=NodeSine(N1.output())
        N7=NodeMultiply(self.W1,self.X1)
        N8=NodeSine(N7.output())
        N3=NodeMultiply(N2.output(),N8.output())
        N9 = NodeMultiply(self.W2,self.X2)
        N10=NodeCosine(N9.output())
        N4 = NodeAdd(N3.output(),N10.output())
        N5 = NodeLinear(N4.output(),2)
        N6 = NodeInverse(N5.output())
        dW1= N1.downstream1(N2.downstream(N3.downstream1(N4.downstream(N5.downstream(N6.downstream(1))))))
        dX1= N1.downstream2(N2.downstream(N3.downstream1(N4.downstream(N5.downstream(N6.downstream(1))))))
        dW2= N9.downstream1(N10.downstream(N4.downstream(N5.downstream(N6.downstream(1)))))
        dX2= N9.downstream2(N10.downstream(N4.downstream(N5.downstream(N6.downstream(1)))))
#         N1.downstream1(N2.downstream(N3.downstream1(N4.downstream(N5.downstream(N6.downstream(1))))))
        return [dW1,dX1,dW2,dX2]
```

The final function call and output is as follows:

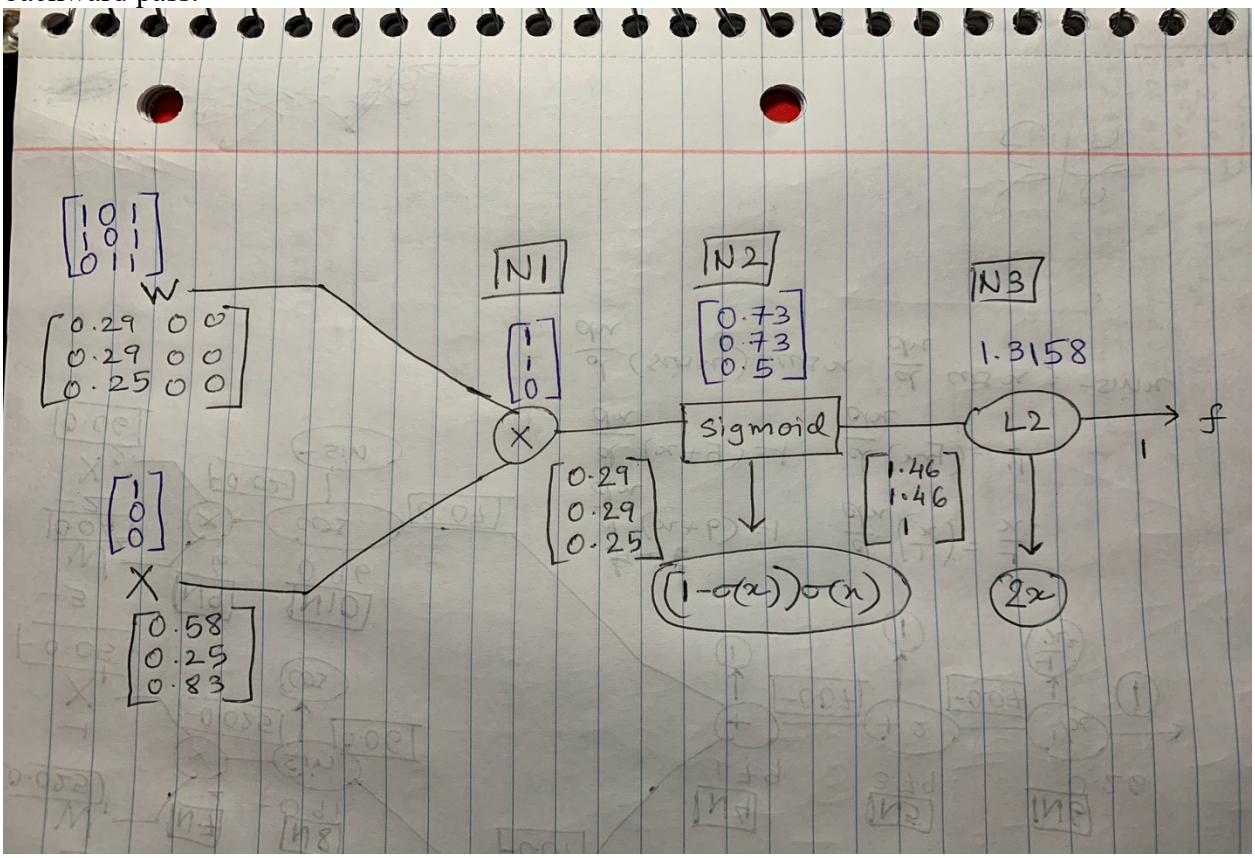
```
In [9]: a=Fx(2,-1,-3,-2)

In [10]: a.backward()

Out[10]: [0.026385404837959904,
          -0.05277080967591981,
          0.03896652605501356,
          0.05844978908252034]

In [11]: a.forward()

Out[11]: 0.26406181327140527
```

This is exactly what was computed on paper and hence the implementation is verified.

Q2

Part a)

Since there was a condition that W is 3x3 and X is 3x1, I took the inputs as shown in the following photo of the computation graph. Again as the first question, the blue matrices are the result of the forward pass and the black matrices in the boxes are the gradients calculated by the backward pass.



Part b)

Adopting the upper method again but this time for matrix inputs, I implemented 3 Node classes, for multiply, sigmoid and L2. I had to use numpy methods to calculate numerous middle values. The following shows how some of the Node classes look in my code.

```
class NodeL2():
    inputArr=np.zeros((3,1))
    def __init__(self,arr):
        self.inputArr=np.copy(arr)
    def output(self):
        return np.sum(np.square(self.inputArr))
    def localGradient(self):
        return np.multiply(2,self.inputArr)
    def downstream(self,upstream):
        return self.localGradient()*upstream

class NodeSigmoid():
    inputArr =np.zeros((3,1))
    def __init__(self,arr):
        self.inputArr=np.copy(arr)
    def sigmoid(self):
        g=1/(1+np.exp(-self.inputArr))
        return g
    def localGradient(self):
        return np.multiply((1-self.sigmoid()),self.sigmoid())
    def downstream(self,upstream):
        return np.multiply(self.localGradient(),upstream)

class NodeMultiply():
    inputArr1=np.zeros((3,3))
    inputArr2=np.zeros((3,1))
    def __init__(self,arr1,arr2):
        self.inputArr1=np.copy(arr1)
        self.inputArr2=np.copy(arr2)
    def output(self):
        return np.dot(self.inputArr1,self.inputArr2)
    def localGradient1(self):
        return np.transpose(self.inputArr2)
    def localGradient2(self):
        return np.transpose(self.inputArr1)
    def downstream1(self,upstream):
        return np.dot(upstream,self.localGradient1())
    def downstream2(self,upstream):
        return np.dot(self.localGradient2(),upstream)
```

The main function class is, as last time, made from 2 methods. The `forward` method contains the result of the forward pass. The `backward` method contains the gradients (df/dW, df/dX) calculated by the backward pass.

```
class Fx():
    W=np.zeros((3,3))
    X=np.zeros((3,1))
    def __init__(self,arr1,arr2):
        self.W = np.copy(arr1)
        self.X = np.copy(arr2)
    def forward (self):
        N1 = NodeMultiply(self.W,self.X)
        N2 = NodeSigmoid(N1.output())
        N3 = NodeL2(N2.sigmoid())
        return N3.output()
    def backward (self):
        N1 = NodeMultiply(self.W,self.X)
        N2 = NodeSigmoid(N1.output())
        N3 = NodeL2(N2.sigmoid())
        dW = N1.downstream1(N2.downstream(N3.downstream(1)))
        dX = N1.downstream2(N2.downstream(N3.downstream(1)))
        return dW,dX
```

The result obtained by this is as follows:

```
n1 =Fx([[1,0,1],[1,0,1],[0,1,1]],[[1],[0],[0]])
```

```
n1.backward()
```

```
(array([[0.28746968, 0.        , 0.        ],
        [0.28746968, 0.        , 0.        ],
        [0.25      , 0.        , 0.        ]]), array([[0.57493936],
        [0.25      ],
        [0.82493936]]))
```

```
n1.forward()
```

```
1.318893290777046
```

This is verifying the result we calculated by hand and thus the code can simulate the function perfectly.