Lusine Kamikyan
BME 595
Homework 2 Report
Collaborated with Emma Reid

Part A.

In the test.py file I created object *model* of class *NeuralNetwork*.

The NeuralNetwork class has *__init__, getLayer* and *forward function*.

__init__ function accepts a list of sizes of layers (number of neurons in each layer) of the neural network. I create a dictionary of random layer matrices (in the code they are random double torch tensors with mean 0 and the given standard deviation).
getLayer function takes in the layer number and returns the weight matrix by calling the *network* dictionary we created in the *__init__* function.
forward function takes in the input (which is a double tensor). Note, if you have n inputs each having m samples, then the input should be a n by m double tensor, ex, torch.tensor([[1,2,3],[3,4,5]], dtype = torch.double). I attach the bias to the input tensor, then loop through the layers of the neural network. In each loop I call the *getLayer(layer i) to get the weight matrix,* then the weight matrix Θ_i gets multiplied with the input, then we take the sigmoid of the result. From the result we get the new input to the next layer by again attaching the bias. Then we go back to the next loop. After going through all the layers, we get the final output of the forward pass which is returned by the forward() function.

The code works for any number of inputs, outputs, hidden layers and samples.

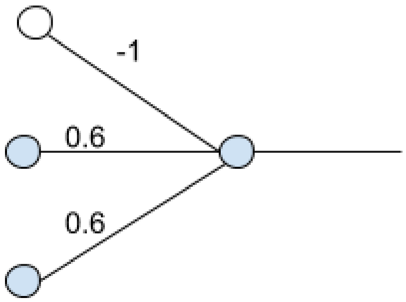
I have not used any helper functions or global variables in neither part A nor B.

Part B.

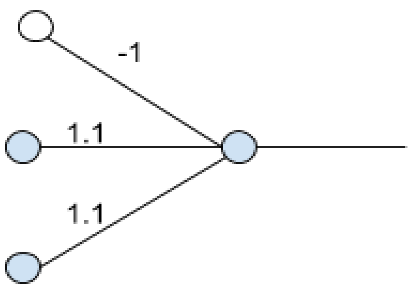
The *logic_gates.py* has 4 classes *AND, OR, XOR,* and *NOT.* All of them have 3 methods: *__init__, __call__, forward*.
__init__ method initializes the *NeuralNetwork* class, and then calls the getLayer to set up the weight matrices manually. The *AND, OR* and *NOT* only need an input and output layer, but the *XOR* also needs 1 hidden layer with 2 neurons in it.
__call__ method calls the forward function of the specific gate. Getting the output from the *forward* we check to see if it is greater than 0.5 (returns True) or less than 0.5 (returns False)
The *forward* method of each gate converts the boolean values into 0 and 1, and makes an input double tensor out of the values and then calls the forward() function of the *NeuralNetwork* class.

The following are the weights I chose: Finding AND, OR, NOT weights is similar to the class example, only using the sigmoid function

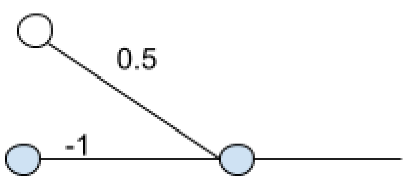
AND: $w_0 = -1,\ w_1 = 0.6,\ w_2 = 0.6$



OR: $w_0 = -1,\ w_1 = 1.1,\ w_2 = 1.1$



NOT: $w_0 = 0.5,\ w_1 = -1$

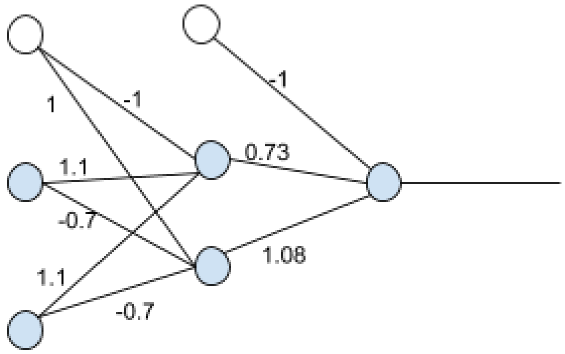


XOR: To figure out the weights for XOR I used that XOR can be represented using the OR, NAND and AND operations. Θ^1 is the weight matrix connecting the input layer to the hidden layer, Θ^2 is the weight matrix connecting the hidden layer to the output layer

$\theta^1{}_{10} = -1,\ \theta^1{}_{11} = 1.1,\ \theta^1{}_{12} = 1.1$ (OR)

$\theta^1{}_{20} = 1,\ \theta^1{}_{21} = -0.7,\ \theta^1{}_{22} = -0.7$ (NAND)

$\theta^2{}_{10} = -1,\ \theta^2{}_{11} = 0.73,\ \theta^2{}_{12} = 1.08$ (AND)



How to run the code:

For part A:
Import and initialize the NeuralNetwork class, define the input double tensor and call the forward function of the NeuralNetwork class, in the following way:

```
from neural_network import NeuralNetwork
import torch
model = NeuralNetwork([3,2,2])
input_tensor= torch.tensor([[1,2,3,3],[1,2,3,3],[2,3,4,4]], dtype = torch.double)
output = model.forward(input_tensor)
```

For part B:
Import the AND, OR, NOT, and XOR classes, then run the following code for each to see the results:

```
And = AND()
print(And(True, False))  #change True or False to try all combinations

Or = OR()
print(Or(False,True))  #change True or False to try all combinations

Not = NOT()
print(Not(False))  #change True or False to try all combinations

Xor = XOR()
print(Xor(False,True))  #change True or False to try all combinations
```

Codes:

test.py

```
from neural_network import NeuralNetwork
import torch
from logic_gates import AND
from logic_gates import OR
from logic_gates import XOR
from logic_gates import NOT

##### PART A #####
model = NeuralNetwork([3,2,2])
input_tensor= torch.tensor([[1,2,3,3],[1,2,3,3],[2,3,4,4]], dtype = torch.double)
output = model.forward(input_tensor)
print(output)

##### PART B #####
print("Results for AND:")
And = AND()
print(And(True, False))
print(And(False, True))
print(And(True, True))
print(And(False, False))

print("Results for OR:")
Or = OR()
print(Or(False,True))
print(Or(True,True))
print(Or(False,False))
print(Or(True,False))

print("Results for NOT:")
Not = NOT()
print(Not(False))
print(Not(True))

print("Results for XOR:")
Xor = XOR()
print(Xor(False,True))
print(Xor(False,False))
print(Xor(True,True))
print(Xor(True,False))
```

neural_network.py

```
import torch
class NeuralNetwork:
    def __init__(self,size_list):
        self.__size_list = size_list
        #create an empty dictionary
        network = {}
        # puting random Thetas with mean 0 and std 1/(layer size)
        for i in range(len(self.__size_list)-1):
            layer_matrix =
    torch.normal(torch.zeros(self.__size_list[i+1],self.__size_list[i]+1),1/self.__size_list[i]**.5
    )

        layer_matrix = layer_matrix.type(torch.DoubleTensor)
        network["theta"+str(i+1)] = layer_matrix
        self.__network = network

    def getLayer(self,layer):
        theta_layer = self.__network["theta"+str(layer)]
        return theta_layer  #weight matrix from layer i to layer i+1

    def forward(self, input_tensor): # input_tensor is 1D or 2D tensor that are type double
        bias = torch.ones(1,input_tensor.size(1), dtype = torch.double)
        # attach input tensor and bias
        input_tensor =torch.cat((bias,input_tensor),0)

        for i in range(0,len(self.__size_list)-1):
            # get the theta
            theta_layer = self.getLayer(i+1)
            # theta * x
            y = torch.mm(theta_layer,input_tensor)
            #sigmoid
            output_tensor = torch.sigmoid(y) #1/(1+math.exp(-y))
            # input to the next layer plus bias
            input_tensor = torch.cat((bias,output_tensor),0)

    return output_tensor
```

logic_gates.py

```
from neural_network import NeuralNetwork
import torch

class AND:
    def __init__(self):
        # initialize the NeuralNetwork NN
        self.__NN = NeuralNetwork([2,1])
        #set manual weight
        theta_layer = self.__NN.getLayer(1)
        theta_layer[0,0] = -1
        theta_layer[0,1] = 0.6
        theta_layer[0,2] = 0.6

    def __call__(self,x,y):
        # call self.forward(x, y) and get the output of forward
        # return T or F depending on the output of forward
        output = self.forward(x,y)
        if output > 0.5:
            output = True
        if output <0.5:
            output = False
        return output

    def forward(self,x,y):
        # transfer (x, y) to 0, 1, and call NN.forward() do the computation of forward
        # return the output of NN.forward()
        input_tensor = torch.tensor([[x],[y]])
        input_tensor = input_tensor.type(torch.DoubleTensor)
        output = self.__NN.forward(input_tensor)
        return output

class OR:
    def __init__(self):
        # initialize the NeuralNetwork NN
        self.__NN = NeuralNetwork([2,1])
        theta_layer = self.__NN.getLayer(1)
        theta_layer[0,0] = -1
        theta_layer[0,1] = 1.1
        theta_layer[0,2] = 1.1
```

```
    def __call__(self,x,y):
        output = self.forward(x,y)
        if output > 0.5:
            output = True
        if output < 0.5:
            output = False
        return output

    def forward(self,x,y):
        input_tensor = torch.tensor([[x],[y]])
        input_tensor = input_tensor.type(torch.DoubleTensor)
        output = self.__NN.forward(input_tensor)
        return output

class XOR:
    def __init__(self):
        # initialize the NeuralNetwork NN
        self.__NN = NeuralNetwork([2,2,1])
        theta_layer1 = self.__NN.getLayer(1)
        theta_layer1[0,0] = -1
        theta_layer1[0,1] = 1.1
        theta_layer1[0,2] = 1.1
        theta_layer1[1,0] = 1
        theta_layer1[1,1] = -0.7
        theta_layer1[1,2] = -0.7
        theta_layer2 = self.__NN.getLayer(2)
        theta_layer2[0,0] = -1
        theta_layer2[0,1] = 0.73
        theta_layer2[0,2] = 1.08

    def __call__(self,x,y):
        output = self.forward(x,y)
        if output > 0.5:
            output = True
        if output < 0.5:
            output = False
        return output

    def forward(self,x,y):
        input_tensor = torch.tensor([[x],[y]])
        input_tensor = input_tensor.type(torch.DoubleTensor)
        output = self.__NN.forward(input_tensor)
```

```
        return output


class NOT:
    def __init__(self):
        self.__NN = NeuralNetwork([1,1])
        theta_layer = self.__NN.getLayer(1)
        theta_layer[0,0] = 0.5
        theta_layer[0,1] = -1

    def __call__(self,x):
        output = self.forward(x)
        if output > 0.5:
            output = True
        if output < 0.5:
            output = False
        return output

    def forward(self,x):
        input_tensor = torch.tensor([[x]])
        input_tensor = input_tensor.type(torch.DoubleTensor)
        output = self.__NN.forward(input_tensor)

        return output
```