Homework 2 Report

Heng Zhang
0029109755
zhan2614

Neural Network

This class implements a general neural network with three functions. Taken an input of list specifying the size of each layer, the __init__ function initializes the neural network by randomly choosing weights matrices following normal distribution. The second function getLayer takes an input of the layer index n and returns the weight matrix between neurons in layer and those in layer n + 1. The last function is forward. This function is supposed to work as the forward propagation step in neural network. It takes the input to the neural network as an torch tensor and computes the final output. The inner private function sigmoid is used to normalized the output after each layer.

Logic Gates

Four logical gates are implemented: AND, OR, NOT, XOR. Only NOT gate takes one input and the other three take two inputs. Since in this homework, we do not need to do the training of the weights, we manually modify the weights to achieve the effect of the logic gates. For AND gate, I choose weight 10 for both of the inputs and -15 for the bias so that the only chance for a positive output before applying sigmoid function is when both input are 1. In other three combinations, the output before applying sigmoid function will be negative. After applying sigmoid function and rounding, the positive output changes to 1 and the negative changes to 0. This is in consistent with the logic of AND gate since the only case for a True output is when both the inputs are True. Similar analysis applies to OR gate, the only case for OR gate to output false is when both of the inputs are false. Therefore, I choose 15, 15 for the weights of inputs and -10 for the bias so that it only generates negative output before sigmoid is when both inputs are zero. For the NOT gate, it is simpler by choosing -20 for the input and 10 for the output. The XOR gate is a bit more complicated since it actually consists of a OR gate, an NAND gate and a AND gate as shown in Figure 1. The weights for the OR gate are 15, 15, -10 for x1, x2 and bias respectively. Similarly the weights for the NAND gate are -10, -10 and 15 for x1, x2, and bias respectively. These are just the negative of those weights for AND gates above. For the AND gate in XOR, the weights are 10, 10, -15.

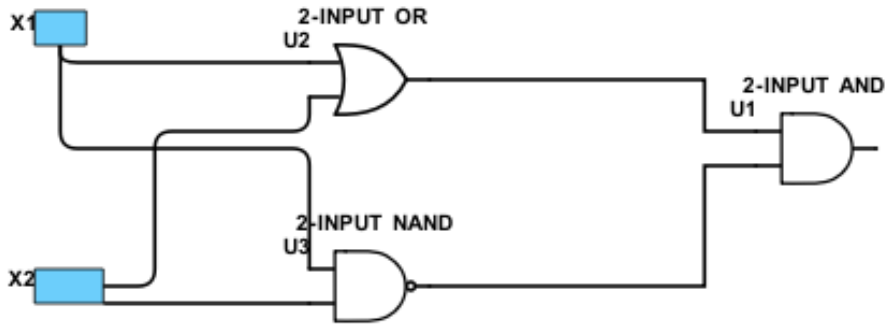

Figure 1 XOR as OR/NAND/AND

Test

There is test script to test the logics for all gates. The test.py tests in the order of AND, OR, NOT, and XOR. Just run it with python3 test.py and put the logic_gates.py and neural_network.py in the same folder then it should outputs the correct results.

Appendix

neural_network.py

```
1.  import torch
2.  from math import sqrt, exp
3.
4.  class NeuralNetwork:
5.      def __init__(self, layers):
6.          # layers is a list of layer sizes
7.          if type(layers) != list:
8.              raise TypeError('Input is not a list')
9.
10.         self.layers = layers
11.         self.theta = {}
12.         # n layers neural network has n-1 weight matrices
13.         for i in range(len(self.layers) - 1):
14.             # the diemension includes one position for biasi
15.             size = (self.layers[i] + 1, self.layers[i+1])
16.             self.theta[i] = torch.normal(
17.                                 torch.zeros(size[0], size[1]),
18.                                 1/sqrt(self.layers[i])
19.                             ).type(torch.DoubleTensor)
20.
21.     def getLayer(self, layer):
22.
```

```
        if layer not in self.theta.keys():
            raise ValueError('Layer index not exists')
        # layer is an integer for the layer index
        # return the corresponding theta matric from that layer to layer + 1

        return self.theta[layer]


    def forward(self, nn_input):
        # the one iteration forward function
        def sigmoid(i):
            if str(i.type()) != 'torch.DoubleTensor':
                raise TypeError('Input of sigmoid is not DoubleTensor')
            return 1 / (1 + torch.pow(exp(1), -i))

        if str(nn_input.type()) != 'torch.DoubleTensor':
            raise TypeError('Input of forward is not DoubleTensor')
        si = [1] + list(nn_input.size()[1:])

        bias = torch.ones(si, dtype=torch.double)
        operation_input = nn_input
        for i in self.theta.keys():
            operation_input =  torch.cat((operation_input, bias), 0)
            operation_input = sigmoid(torch.mm(torch.t(self.theta[i]), operation_input))
            bias = torch.ones([1] + list(operation_input.size()[1:]), dtype=torch.double)
        return operation_input
```

logic_gates.py

```
import torch
from neural_network import NeuralNetwork

class AND:
    def __init__(self):
        self.and_nn = NeuralNetwork([2,1])
        self.layer = self.and_nn.getLayer(0)
        self.layer[:,:] = 0.0
        self.layer += torch.DoubleTensor([[10],[10], [-15]])

    def __call__(self, x, y):
        self.x, self.y = tuple(map(float, (x,y)))
        return bool(self.forward() > 0.5)

```

```
    def forward(self):
        return self.and_nn.forward(torch.DoubleTensor([[self.x], [self.y]]))

class OR:
    def __init__(self):
        self.or_nn = NeuralNetwork([2,1])
        self.layer = self.or_nn.getLayer(0)
        self.layer[:,:] = 0.0
        self.layer += torch.DoubleTensor([[15],[15], [-10]])

    def __call__(self, x, y):
        self.x, self.y = tuple(map(float, (x,y)))
        return bool(self.forward() > 0.5)

    def forward(self):
        return self.or_nn.forward(torch.DoubleTensor([[self.x], [self.y]]))

class NOT:
    def __init__(self):
        self.not_nn = NeuralNetwork([1,1])
        self.layer = self.not_nn.getLayer(0)
        self.layer[:,:] = 0.0
        self.layer += torch.DoubleTensor([[-20], [10]])

    def __call__(self, x):
        self.x = float(x)
        return bool(self.forward() > 0.5)

    def forward(self):
        return self.not_nn.forward(torch.DoubleTensor([[self.x]]))

class XOR:
    def __init__(self):
        self.xor_nn = NeuralNetwork([2,2,1])
        self.layer0 = self.xor_nn.getLayer(0)
        self.layer1 = self.xor_nn.getLayer(1)

        self.layer0[:,:] = 0.0
        self.layer0 += torch.DoubleTensor([[15, -10], [15, -10], [-
10, 15]])
        self.layer1[:, :] = 0.0
        self.layer1 += torch.DoubleTensor([[10],[10],[-15]])
```

```
56.
57.     def __call__(self, x, y):
58.         self.x, self.y = tuple(map(float, (x,y)))
59.         return bool(self.forward() > 0.5)
60.
61.     def forward(self):
62.         return self.xor_nn.forward(torch.DoubleTensor([[self.x], [self.y]]))
```

test.py

```
1.  from logic_gates import AND
2.  from logic_gates import OR
3.  from logic_gates import NOT
4.  from logic_gates import XOR
5.
6.  print("AND Gate test cases")
7.  And = AND()
8.
9.  print("and(False, False) = %r" % And(False, False))
10. print("and(True, False) = %r" % And(True, False))
11. print("and(False, True) = %r" % And(False, True))
12. print("and(True, True) = %r \n" % And(True, True))
13.
14. print("OR Gate test cases")
15. Or = OR()
16. print("or(False, False) = %r" % Or(False, False))
17. print("or(True, False) = %r" % Or(True, False))
18. print("or(False, True) = %r" % Or(False, True))
19. print("or(True, True) = %r\n" % Or(True, True))
20.
21. print("NOT Gate test cases")
22. Not = NOT()
23. print("not(False) = %r" % Not(False))
24. print("not(True) = %r\n" % Not(True))
25.
26. print("XOR Gate test cases")
27. Xor = XOR()
28. print("xor(False, False) = %r" % Xor(False, False))
29. print("xor(True, False) = %r" % Xor(True, False))
30. print("xor(False, True) = %r" % Xor(False, True))
31. print("xor(True, True) = %r" % Xor(True, True))
```