Homework 2 Report

Heng Zhang
0029109755
zhan2614

Neural Network

This class updates the back propagation and parameter update functions to the class. It calculates the gradient of the loss function and update the weights to learn the proper weights for the network.

Logic Gates

As in homework 2, there are four logic gates AND, OR, NOT, XOR to test the neural network class. But instead of intentionally assigning weights to the neural network, the logic gates will call its train function which runs multiple iterations of forward pass, backward pass and parameter updating until the total loss is lower than 0.1. The 0.1 threshold can be adjusted based on the application.

Test

The change of loss value is plotted in Figure 1. It is clearly shown that the loss value decreases and since the NOT gate has only one input and one output it converges very fast. The XOR gate converges at about 800 iterations and it takes the longest time to converge. This is explained by the more complex structure to implement XOR gate.

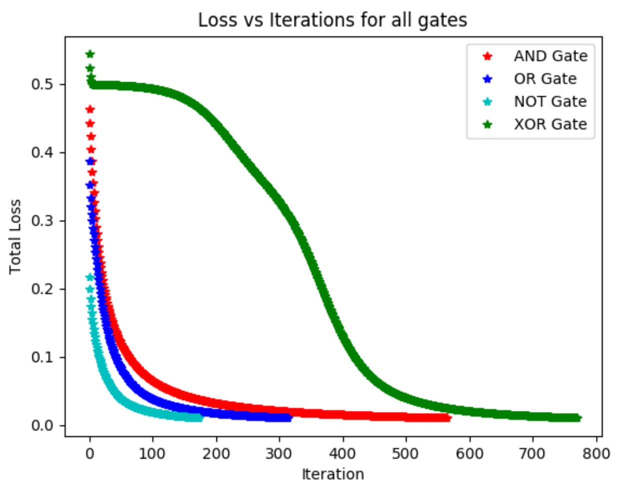

Figure 1 Loss change in different gates

To compare with the weight that I used in homework 2, Table 1 is provided for comparison.

	Homework 2	Homework 3
AND	[10, 10, -15]	[4.7937, 4.7937, -7.2869]
OR	[15, 15, -10]	[4.7852, 4.7854, -2.1282]
NOT	[-20, 10]	[-4.4316, 2.0948]
XOR	Layer 0: [[15, 15, -10], [-10, -10, 15]]; Layer 1: [10, 10, 15]	Layer 0: [[-5.6626, -5.6240, 2.0536], [-3.6809, -3.6742, 5.3859]]; [-7.5530, 7.1618, -3.2250]

Table 1 Weights from Homework 2 and 3

The AND, OR, NOT gate have similar relative weights for x1, x2, and bias but for the XOR gate, since it has different implementations, the weights learned are different. In homework 2, the assigned weights correspond to a OR and a NAND gate at the hidden layer and an AND gate at output layer. In homework 3, the learned weights is no longer the same but the truth table is the correct for XOR. The truth table learned from homework 3 is shown in table 2

X1	X2	Hidden layer upper gate	Hidden layer lower gate	Output layer
0	0	1	1	0
0	1	0	1	1
1	0	0	1	1
1	1	0	0	0

Appendix

neural_network.py

```python
import torch
from math import sqrt, exp

class NeuralNetwork:
    def __init__(self, layers):
        # layers is a list of layer sizes
        if type(layers) != list:
            raise TypeError('Input is not a list')

        self.layers = layers
        self.theta = {}
        self.dE_dTheta = {}
        self.a = {} # the result after applying the sigmoid functino
        self.z = {} # result after weight matrix multiplies the activation
        # self.L is the index of the output layer
        self.L = len(layers) - 1

        # n layers neural network has n-1 weight matrices
        for i in range(len(self.layers) - 1):
            # the diemension includes one position for biasi
            size = (self.layers[i] + 1, self.layers[i+1])
            self.theta[i] = torch.normal(
                                torch.zeros(size[0], size[1]),
                                1/sqrt(self.layers[i])
                            ).type(torch.DoubleTensor)
        self.total_loss = 1

    def getTheta(self):
        return self.theta

    def getLayer(self, layer):

        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):
        # nn_input is mXn where m is the number of samples
        # n is the number of neurons in each sample
        #print('original input', 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, nn_input.size()[0]]
        #print('si', si)

        bias = torch.ones(si, dtype=torch.double)
        #print('bias', bias)
        operation_input = nn_input.t()
        # operation_input has nxm dim
        self.a[0] = nn_input.t()
        #print('a[0]', self.a[0])

        for i in self.theta.keys():
         #   print('i=',i)
          #  print('a[{}]={}'.format(i, self.a[i]))
          #  print('bias=',bias)
            self.a[i] =  torch.cat((self.a[i], bias), 0)
          #  print('cat input', self.a[i])
            theta = torch.t(self.theta[i])
          #  print('theta', theta)
            self.z[i + 1] = torch.mm(theta, self.a[i])
          #  print('z', self.z[i+1])
            self.a[i + 1] = sigmoid(self.z[i + 1])
          #  print('a', self.a[i+1])
            bias = torch.ones([1, self.a[i].t().size()[0]], dtype=torch.double)
          #  print('end bias', bias)
        #print('return from forward', self.a[self.L].t())
        return self.a[self.L].t()


    def backward(self, target, loss='MSE'):
        target = target.t()
        #print('target', target)
        if loss == 'MSE':
            # step 1 calculate the loss function
            self.total_loss = (self.a[self.L] - target).pow(2).sum() / 2 / len(target)
          #  print('output activation:', self.a[self.L])
          #  print('total loss', self.total_loss)
            delta = torch.mul((self.a[self.L] - target), torch.mul(self.a[self.L], (1 - self.a[self.L])))
          #  print('delta', delta)

            for i in range(self.L - 1, -1, -1):
                if i != self.L - 1:
                    indices = torch.LongTensor(list(range(self.a[i].size()[0] - 1)))
                    #indices = torch.LongTensor([0,1])
                    delta = torch.index_select(delta, 0, indices)
                # from the layer before the output
                self.dE_dTheta[i] = torch.mm(self.a[i], delta.t())
                delta = torch.mul(torch.mm(self.theta[i], delta), torch.mul(self.a[i], (1 - self.a[i])))
           #     print('dE_dTheta', self.dE_dTheta[i])
            #     print('theta', self.theta[i])
            #     print('delta', delta)
            #     print('diff_a', torch.mul(self.a[i], (1 - self.a[i])))
         elif loss == 'CE':
             pass

         else:
             print('unrecognized error functino')

     def updateParams(self, rate):
         for i in range(len(self.theta)):
            # print('before update', self.theta[i])
             self.theta[i] = self.theta[i] - torch.mul(self.dE_dTheta[i], rate)
           #  print('after update', self.theta[i])
```

logic_gates.py

```
import torch
from neural_network import NeuralNetwork
import matplotlib.pyplot as plt

class AND:
    def __init__(self):
        self.and_nn = NeuralNetwork([2,1])
        self.iterations = 1000

    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]]))


    def train(self):
        dataset = torch.DoubleTensor([[0, 0],[0, 1],[1, 0],[1, 1]])
        target = torch.zeros(len(dataset), dtype=torch.double)
        target = torch.unsqueeze(target, 1)
        # target now has dimension 4X1

```

```python
        for i in range(len(dataset)):
            target[i, :] = dataset[i, 0] and dataset[i, 1]


            #print('\n\nITERATION', i)
        i = 0
        while self.and_nn.total_loss > 0.01:
            #print('dataset is', dataset)
            self.and_nn.forward(dataset)
            self.and_nn.backward(target)
            self.and_nn.updateParams(1)
            line, = plt.plot(i, self.and_nn.total_loss, 'r*')
            i +=1

        line.set_label("AND Gate")
        plt.xlabel('Iteration')
        plt.ylabel('Total Loss')
        plt.grid()


class OR:
    def __init__(self):
        self.or_nn = NeuralNetwork([2,1])
        self.iterations = 1000

    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]]))

    def train(self):
        dataset = torch.DoubleTensor([[0, 0],[0, 1],[1, 0],[1, 1]])
        target = torch.zeros(len(dataset), dtype=torch.double)
        target = torch.unsqueeze(target, 1)
        # target now has dimension 4X1

        for i in range(len(dataset)):
            target[i, :] = dataset[i, 0] or dataset[i, 1]
        i = 0
        while self.or_nn.total_loss > 0.01:
            #print('dataset is', dataset)
            self.or_nn.forward(dataset)
```

```
                self.or_nn.backward(target)
                self.or_nn.updateParams(1)
                line, = plt.plot(i, self.or_nn.total_loss, 'b*')
                i += 1

            line.set_label('OR Gate')

            plt.xlabel('Iteration')
            plt.ylabel('Total Loss')
            plt.grid()




class NOT:
    def __init__(self):
        self.not_nn = NeuralNetwork([1,1])
        self.iterations = 1000

    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]]))


    def train(self):
        dataset = torch.DoubleTensor([[0],[1]])
        target = torch.zeros(len(dataset), dtype=torch.double)
        target = torch.unsqueeze(target, 1)
        # target now has dimension 4X1

         for i in range(len(dataset)):
             target[i, :] = float(not dataset[i, 0])

         i=0
         while self.not_nn.total_loss > 0.01:
                 # print('dataset is', dataset)
             self.not_nn.forward(dataset)
             self.not_nn.backward(target)
             self.not_nn.updateParams(1)
             line, = plt.plot(i, self.not_nn.total_loss, 'c*')
             i+=1
```

```python
        line.set_label('NOT Gate')
        plt.xlabel('Iteration')
        plt.ylabel('Total Loss')
        plt.grid()


class XOR:
    def __init__(self):
        self.xor_nn = NeuralNetwork([2,2,1])
        self.iterations = 1000

    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.xor_nn.forward(torch.DoubleTensor([[self.x, self.y]]))

    def train(self):
        dataset = torch.DoubleTensor([[0, 0],[0, 1],[1, 0],[1, 1]])
        target = torch.zeros(len(dataset), dtype=torch.double)
        target = torch.unsqueeze(target, 1)
        # target now has dimension 4X1

        for i in range(len(dataset)):
            target[i, :] = float((dataset[i, 0] or dataset[i, 1]) and (not
(dataset[i,0] and dataset[i, 1])))

        i = 0
        while self.xor_nn.total_loss > 0.01:
              #  print('dataset is', dataset)
            self.xor_nn.forward(dataset)
            self.xor_nn.backward(target)
            self.xor_nn.updateParams(1)
            line, = plt.plot(i, self.xor_nn.total_loss, 'g*')
            print(self.xor_nn.total_loss)
            i += 1

        line.set_label('XOR Gate')
        plt.xlabel('Iteration')
        plt.ylabel('Total Loss')
        plt.title('Loss vs Iterations for all gates')
        plt.grid()
        plt.legend()
        plt.savefig('loss.png')
```

test.py

```
from logic_gates import AND
from logic_gates import OR
from logic_gates import NOT
from logic_gates import XOR
from pprint import pprint as pp

And = AND()
And.train()
print("AND Gate test cases")
pp(And.and_nn.theta)
print("and(False, False) = %r" % And(False, False))
print("and(True, False) = %r" % And(True, False))
print("and(False, True) = %r" % And(False, True))
print("and(True, True) = %r \n" % And(True, True))
Or = OR()
Or.train()
pp(Or.or_nn.theta)
print("or(False, False) = %r" % Or(False, False))
print("or(True, False) = %r" % Or(True, False))
print("or(False, True) = %r" % Or(False, True))
print("or(True, True) = %r\n" % Or(True, True))

print("NOT Gate test cases")
Not = NOT()
Not.train()
pp(Not.not_nn.theta)
print("not(False) = %r" % Not(False))
print("not(True) = %r\n" % Not(True))
print("XOR Gate test cases")
Xor = XOR()
Xor.train()
pp(Xor.xor_nn.theta)
print("xor(False, False) = %r" % Xor(False, False))
print("xor(True, False) = %r" % Xor(True, False))
print("xor(False, True) = %r" % Xor(False, True))
print("xor(True, True) = %r" % Xor(True, True))
print("OR Gate test cases")
```