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.

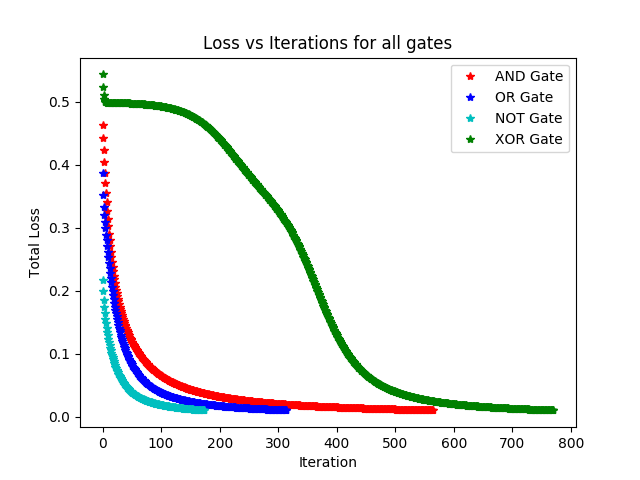


Figure 1 Loss change in different gates

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

1. **import** torch
2. **from** math **import** sqrt, exp
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')
10. self.layers = layers
11. self.theta = {}
12. self.dE\_dTheta = {}
13. self.a = {} # the result after applying the sigmoid functino
14. self.z = {} # result after weight matrix multiplies the activation
15. # self.L is the index of the output layer
16. self.L = len(layers) - 1
18. # n layers neural network has n-1 weight matrices
19. **for** i **in** range(len(self.layers) - 1):
20. # the diemension includes one position for biasi
21. size = (self.layers[i] + 1, self.layers[i+1])
22. self.theta[i] = torch.normal(
23. torch.zeros(size[0], size[1]),
24. 1/sqrt(self.layers[i])
25. ).type(torch.DoubleTensor)
26. self.total\_loss = 1
28. **def** getTheta(self):
29. **return** self.theta
31. **def** getLayer(self, layer):
33. **if** layer **not** **in** self.theta.keys():
34. **raise** ValueError('Layer index not exists')
35. # layer is an integer for the layer index
36. # return the corresponding theta matric from that layer to layer + 1
37. **return** self.theta[layer]

40. **def** forward(self, nn\_input):
41. # nn\_input is mXn where m is the number of samples
42. # n is the number of neurons in each sample
43. #print('original input', nn\_input)
44. # the one iteration forward function
45. **def** sigmoid(i):
46. **if** str(i.type()) != 'torch.DoubleTensor':
47. **raise** TypeError('Input of sigmoid is not DoubleTensor')
48. **return** 1 / (1 + torch.pow(exp(1), -i))
50. **if** str(nn\_input.type()) != 'torch.DoubleTensor':
51. **raise** TypeError('Input of forward is not DoubleTensor')
52. si = [1, nn\_input.size()[0]]
53. #print('si', si)
55. bias = torch.ones(si, dtype=torch.double)
56. #print('bias', bias)
57. operation\_input = nn\_input.t()
58. # operation\_input has nxm dim
59. self.a[0] = nn\_input.t()
60. #print('a[0]', self.a[0])
62. **for** i **in** self.theta.keys():
63. #   print('i=',i)
64. #  print('a[{}]={}'.format(i, self.a[i]))
65. #  print('bias=',bias)
66. self.a[i] =  torch.cat((self.a[i], bias), 0)
67. #  print('cat input', self.a[i])
68. theta = torch.t(self.theta[i])
69. #  print('theta', theta)
70. self.z[i + 1] = torch.mm(theta, self.a[i])
71. #  print('z', self.z[i+1])
72. self.a[i + 1] = sigmoid(self.z[i + 1])
73. #  print('a', self.a[i+1])
74. bias = torch.ones([1, self.a[i].t().size()[0]], dtype=torch.double)
75. #  print('end bias', bias)
76. #print('return from forward', self.a[self.L].t())
77. **return** self.a[self.L].t()

80. **def** backward(self, target, loss='MSE'):
81. target = target.t()
82. #print('target', target)
83. **if** loss == 'MSE':
84. # step 1 calculate the loss function
85. self.total\_loss = (self.a[self.L] - target).pow(2).sum() / 2 / len(target)
86. #  print('output activation:', self.a[self.L])
87. #  print('total loss', self.total\_loss)
88. delta = torch.mul((self.a[self.L] - target), torch.mul(self.a[self.L], (1 - self.a[self.L])))
89. #  print('delta', delta)
91. **for** i **in** range(self.L - 1, -1, -1):
92. **if** i != self.L - 1:
93. indices = torch.LongTensor(list(range(self.a[i].size()[0] - 1)))
94. #indices = torch.LongTensor([0,1])
95. delta = torch.index\_select(delta, 0, indices)
96. # from the layer before the output
97. self.dE\_dTheta[i] = torch.mm(self.a[i], delta.t())
98. delta = torch.mul(torch.mm(self.theta[i], delta), torch.mul(self.a[i], (1 - self.a[i])))
99. #     print('dE\_dTheta', self.dE\_dTheta[i])
100. #     print('theta', self.theta[i])
101. #     print('delta', delta)
102. #     print('diff\_a', torch.mul(self.a[i], (1 - self.a[i])))
103. **elif** loss == 'CE':
104. **pass**
106. **else**:
107. **print**('unrecognized error functino')
109. **def** updateParams(self, rate):
110. **for** i **in** range(len(self.theta)):
111. # print('before update', self.theta[i])
112. self.theta[i] = self.theta[i] - torch.mul(self.dE\_dTheta[i], rate)
113. #  print('after update', self.theta[i])

logic\_gates.py

1. **import** torch
2. **from** neural\_network **import** NeuralNetwork
3. **import** matplotlib.pyplot as plt
5. **class** AND:
6. **def** \_\_init\_\_(self):
7. self.and\_nn = NeuralNetwork([2,1])
8. self.iterations = 1000
10. **def** \_\_call\_\_(self, x, y):
11. self.x, self.y = tuple(map(float, (x,y)))
12. **return** bool(self.forward() > 0.5)
14. **def** forward(self):
15. **return** self.and\_nn.forward(torch.DoubleTensor([[self.x, self.y]]))

18. **def** train(self):
19. dataset = torch.DoubleTensor([[0, 0],[0, 1],[1, 0],[1, 1]])
20. target = torch.zeros(len(dataset), dtype=torch.double)
21. target = torch.unsqueeze(target, 1)
22. # target now has dimension 4X1
24. **for** i **in** range(len(dataset)):
25. target[i, :] = dataset[i, 0] **and** dataset[i, 1]

28. #print('\n\nITERATION', i)
29. i = 0
30. **while** self.and\_nn.total\_loss > 0.01:
31. #print('dataset is', dataset)
32. self.and\_nn.forward(dataset)
33. self.and\_nn.backward(target)
34. self.and\_nn.updateParams(1)
35. line, = plt.plot(i, self.and\_nn.total\_loss, 'r\*')
36. i +=1
38. line.set\_label("AND Gate")
39. plt.xlabel('Iteration')
40. plt.ylabel('Total Loss')
41. plt.grid()

44. **class** OR:
45. **def** \_\_init\_\_(self):
46. self.or\_nn = NeuralNetwork([2,1])
47. self.iterations = 1000
49. **def** \_\_call\_\_(self, x, y):
50. self.x, self.y = tuple(map(float, (x,y)))
51. **return** bool(self.forward() > 0.5)
53. **def** forward(self):
54. **return** self.or\_nn.forward(torch.DoubleTensor([[self.x,self.y]]))
56. **def** train(self):
57. dataset = torch.DoubleTensor([[0, 0],[0, 1],[1, 0],[1, 1]])
58. target = torch.zeros(len(dataset), dtype=torch.double)
59. target = torch.unsqueeze(target, 1)
60. # target now has dimension 4X1
62. **for** i **in** range(len(dataset)):
63. target[i, :] = dataset[i, 0] **or** dataset[i, 1]
64. i = 0
65. **while** self.or\_nn.total\_loss > 0.01:
66. #print('dataset is', dataset)
67. self.or\_nn.forward(dataset)
68. self.or\_nn.backward(target)
69. self.or\_nn.updateParams(1)
70. line, = plt.plot(i, self.or\_nn.total\_loss, 'b\*')
71. i += 1
73. line.set\_label('OR Gate')
75. plt.xlabel('Iteration')
76. plt.ylabel('Total Loss')
77. plt.grid()



82. **class** NOT:
83. **def** \_\_init\_\_(self):
84. self.not\_nn = NeuralNetwork([1,1])
85. self.iterations = 1000
87. **def** \_\_call\_\_(self, x):
88. self.x = float(x)
89. **return** bool(self.forward() > 0.5)
91. **def** forward(self):
92. **return** self.not\_nn.forward(torch.DoubleTensor([[self.x]]))

95. **def** train(self):
96. dataset = torch.DoubleTensor([[0],[1]])
97. target = torch.zeros(len(dataset), dtype=torch.double)
98. target = torch.unsqueeze(target, 1)
99. # target now has dimension 4X1
101. **for** i **in** range(len(dataset)):
102. target[i, :] = float(**not** dataset[i, 0])
104. i=0
105. **while** self.not\_nn.total\_loss > 0.01:
106. # print('dataset is', dataset)
107. self.not\_nn.forward(dataset)
108. self.not\_nn.backward(target)
109. self.not\_nn.updateParams(1)
110. line, = plt.plot(i, self.not\_nn.total\_loss, 'c\*')
111. i+=1
112. line.set\_label('NOT Gate')
113. plt.xlabel('Iteration')
114. plt.ylabel('Total Loss')
115. plt.grid()

118. **class** XOR:
119. **def** \_\_init\_\_(self):
120. self.xor\_nn = NeuralNetwork([2,2,1])
121. self.iterations = 1000
123. **def** \_\_call\_\_(self, x, y):
124. self.x, self.y = tuple(map(float, (x,y)))
125. **return** bool(self.forward() > 0.5)
127. **def** forward(self):
128. **return** self.xor\_nn.forward(torch.DoubleTensor([[self.x, self.y]]))
130. **def** train(self):
131. dataset = torch.DoubleTensor([[0, 0],[0, 1],[1, 0],[1, 1]])
132. target = torch.zeros(len(dataset), dtype=torch.double)
133. target = torch.unsqueeze(target, 1)
134. # target now has dimension 4X1
136. **for** i **in** range(len(dataset)):
137. target[i, :] = float((dataset[i, 0] **or** dataset[i, 1]) **and** (**not** (dataset[i,0] **and** dataset[i, 1])))
139. i = 0
140. **while** self.xor\_nn.total\_loss > 0.01:
141. #  print('dataset is', dataset)
142. self.xor\_nn.forward(dataset)
143. self.xor\_nn.backward(target)
144. self.xor\_nn.updateParams(1)
145. line, = plt.plot(i, self.xor\_nn.total\_loss, 'g\*')
146. **print**(self.xor\_nn.total\_loss)
147. i += 1
149. line.set\_label('XOR Gate')
150. plt.xlabel('Iteration')
151. plt.ylabel('Total Loss')
152. plt.title('Loss vs Iterations for all gates')
153. plt.grid()
154. plt.legend()
155. plt.savefig('loss.png')

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. **from** pprint **import** pprint as pp
7. And = AND()
8. And.train()
9. **print**("AND Gate test cases")
10. pp(And.and\_nn.theta)
11. **print**("and(False, False) = %r" % And(False, False))
12. **print**("and(True, False) = %r" % And(True, False))
13. **print**("and(False, True) = %r" % And(False, True))
14. **print**("and(True, True) = %r \n" % And(True, True))
15. Or = OR()
16. Or.train()
17. pp(Or.or\_nn.theta)
18. **print**("or(False, False) = %r" % Or(False, False))
19. **print**("or(True, False) = %r" % Or(True, False))
20. **print**("or(False, True) = %r" % Or(False, True))
21. **print**("or(True, True) = %r\n" % Or(True, True))
23. **print**("NOT Gate test cases")
24. Not = NOT()
25. Not.train()
26. pp(Not.not\_nn.theta)
27. **print**("not(False) = %r" % Not(False))
28. **print**("not(True) = %r\n" % Not(True))
29. **print**("XOR Gate test cases")
30. Xor = XOR()
31. Xor.train()
32. pp(Xor.xor\_nn.theta)
33. **print**("xor(False, False) = %r" % Xor(False, False))
34. **print**("xor(True, False) = %r" % Xor(True, False))
35. **print**("xor(False, True) = %r" % Xor(False, True))
36. **print**("xor(True, True) = %r" % Xor(True, True))
37. **print**("OR Gate test cases")