Homework 4 Report

Heng Zhang

0029109755

zhan2614

# Introduction

In this homework we are using the MINST dataset to evaluate the neural network we built for previous homework and the neural network class from torch. The dataset has 60000 pictures showing numbers from 0 to 9 and the task of both neural networks is to check if the neural network can correctly recognize those numbers.

# Setup

I use my\_img2num.py and nn\_img2num.py to denote the use of my own implementation of neural network and the neural network interface provided by torch. In both cases, the size of the neural network is 784 \* 512 \* 256 \* 64 \* 10. The output has ten features indexed from 0 to 9 and final predicted number will be the index of the feature that has the largest value. Each neural network will have 30 epochs.

# Evaluation

To compare both implementations of the neural network, Figure 1 shows the prediction accuracy of both networks. They roughly converge at similar speed and both networks achieve comparable accuracy in the end. Figure 2 shows the running time of each epoch for both implementations. torch.nn runs faster. My own implementation of the neural network runs roughly 10 seconds for each epoch while the torch.nn runs for about 6 seconds. Figure 3 shows the change of loss of my nn and torch.nn. At the initial phase, my nn has larger losses in testing and training but eventually both implementations achieve comparable losses.

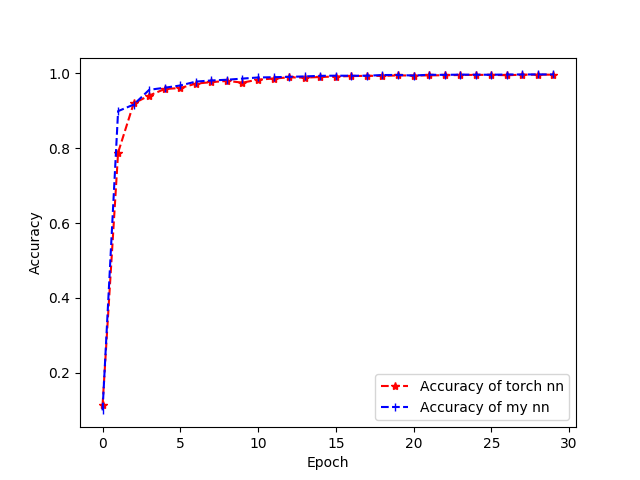


Figure 1 Accuracy in Both Implementations

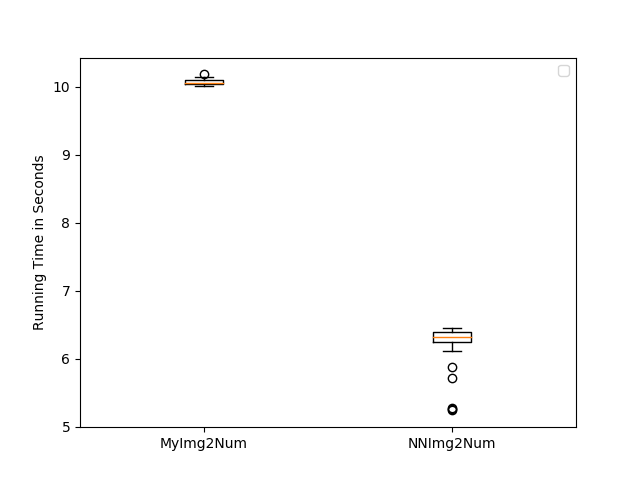


Figure 2 Running Time for Both Implementations

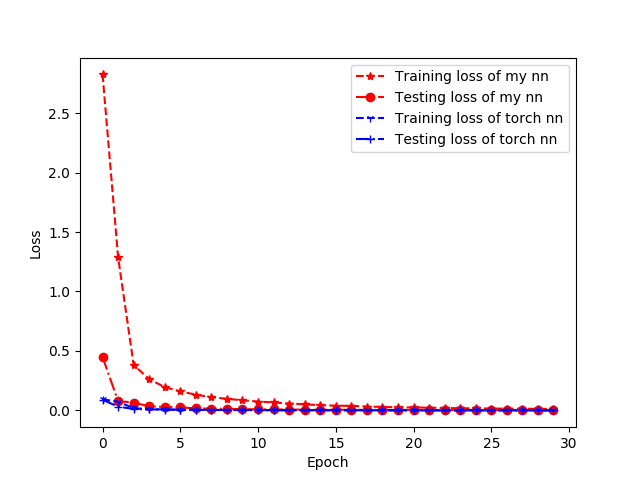


Figure 3 The change of Training and Testing Loss

# Appendix

my\_img2num.py

1. **from** pprint **import** pprint as pp
2. **from** neural\_network **import** NeuralNetwork
3. **import** torch
4. **from** torchvision **import** datasets, transforms
5. **from** time **import** time
6. **import** matplotlib.pyplot as plt
7. plt.ioff()
8. **class** MyImg2Num:
9. **def** \_\_init\_\_(self):
10. self.train\_batch\_size = 60
11. self.epoch = 30
12. self.labels = 10
13. self.rate = 0.1
14. self.input\_size = 28 \* 28
15. self.test\_batch\_size = 10 \* self.train\_batch\_size
16. self.test\_loader = torch.utils.data.DataLoader(
17. datasets.MNIST('./data',
18. train=True,
19. download=True,
20. transform=transforms.Compose([transforms.ToTensor()])),
21. batch\_size=self.test\_batch\_size, shuffle=True)
23. self.train\_loader = torch.utils.data.DataLoader(
24. datasets.MNIST('./data',
25. train=True,
26. download=True,
27. transform=transforms.Compose([transforms.ToTensor()])),
28. batch\_size=self.train\_batch\_size, shuffle=True)
30. # input image is 28 \* 28 so convert to 1D matrix
31. # output labels are 10 [0 - 9]
32. self.nn = NeuralNetwork([self.input\_size, 512, 256, 64, self.labels])
34. **def** train(self, plot=False):
35. **print**('training')
36. **def** onehot\_training(target, batch\_size):
37. output = torch.zeros(batch\_size, self.labels)
38. **for** i **in** range(batch\_size):
39. output[i][int(target[i])] = 1.0
40. **return** output
42. **def** training():
43. loss = 0
44. **for** batch\_id, (data, target) **in** enumerate(self.train\_loader):
45. # data.view change the dimension of input to use forward function
46. forward\_pass\_output = self.nn.forward(data.view(self.train\_batch\_size, self.input\_size).type(torch.DoubleTensor))
47. onehot\_target = onehot\_training(target, self.train\_batch\_size).type(torch.DoubleTensor)
48. #print(onehot\_target.type())
49. self.nn.backward(onehot\_target)
50. loss += self.nn.total\_loss
51. self.nn.updateParams(self.rate)
52. # loss / number of batches
53. avg\_loss = loss / (len(self.train\_loader.dataset) / self.train\_batch\_size)
54. **return** avg\_loss
56. **def** testing():
57. loss = 0
58. correct = 0
59. **for** batch\_id, (data, target) **in** enumerate(self.test\_loader):
60. # data.view change the dimension of input to use forward function
61. forward\_pass\_output = self.nn.forward(data.view(self.test\_batch\_size, self.input\_size).type(torch.DoubleTensor))
62. onehot\_target = onehot\_training(target, self.test\_batch\_size).type(torch.DoubleTensor)
63. loss += (onehot\_target - forward\_pass\_output).pow(2).sum() / 2
64. #print(forward\_pass\_output.size())
65. #print(onehot\_target.size())
66. **for** i **in** range(self.test\_batch\_size):
67. val, position = torch.max(forward\_pass\_output[i], 0)
68. #   print('prediction = {}, actual = {}'.format(int(position), target[i]))
69. **if** position == target[i]:
70. correct += 1
71. # loss / number of batches
72. avg\_loss = loss / len(self.test\_loader.dataset)
73. accuracy = correct / len(self.test\_loader.dataset)
74. **return** avg\_loss, accuracy
75. acc\_list = []
76. train\_loss\_list = []
77. test\_loss\_list = []
78. speed = []
80. **for** i **in** range(self.epoch):
81. s = time()
82. train\_loss = training()
83. e = time()
84. test\_loss,accuracy = testing()
85. **print**('Epoch {}, training\_loss = {}, testing\_loss = {}, accuracy = {}, time = {}'.format(i, train\_loss, test\_loss, accuracy, e - s))
86. acc\_list.append(accuracy)
87. train\_loss\_list.append(train\_loss)
88. test\_loss\_list.append(test\_loss)
89. speed.append(e-s)
90. **if** plot:
91. **return** speed, train\_loss\_list, test\_loss\_list, acc\_list

94. **def** forward(self, img):
96. output = self.nn.forward(img.view(1, self.input\_size))
97. \_, result = torch.max(output, 1)
98. **return** result
99. '''''
100. plt.plot(range(self.epoch), acc\_list, 'r|--', label='Accuracy')
101. plt.plot(range(self.epoch), train\_loss\_list, 'b\*--', label='Training Loss')
102. plt.plot(range(self.epoch), test\_loss\_list, 'yo--', label='Test Loss')
103. plt.xlabel('Epoch')
104. plt.legend()
105. plt.title('My Neural Network Evaluation')
106. plt.savefig('my\_compare.png')
107. plt.clf()
108. '''

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 size', target.size())
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)
90. #print(delta.size())
92. **for** i **in** range(self.L - 1, -1, -1):
93. **if** i != self.L - 1:
94. #indices = torch.LongTensor(list(range(self.a[i].size()[0] - 1)))
95. #print(indices)
96. #indices = torch.LongTensor([0,1])
97. #delta = torch.index\_select(delta, 0, indices)
98. delta = delta.narrow(0, 0, delta.size(0) - 1)
99. # from the layer before the output
100. self.dE\_dTheta[i] = torch.mm(self.a[i], delta.t())
101. delta = torch.mul(torch.mm(self.theta[i], delta), torch.mul(self.a[i], (1 - self.a[i])))
102. #     print('dE\_dTheta', self.dE\_dTheta[i])
103. #     print('theta', self.theta[i])
104. #     print('delta', delta)
105. #     print('diff\_a', torch.mul(self.a[i], (1 - self.a[i])))
106. **elif** loss == 'CE':
107. **pass**
109. **else**:
110. **print**('unrecognized error functino')
112. **def** updateParams(self, rate):
113. **for** i **in** range(len(self.theta)):
114. # print('before update', self.theta[i])
115. self.theta[i] = self.theta[i] - torch.mul(self.dE\_dTheta[i], rate)
116. #  print('after update', self.theta[i])

nn\_img2num.py

1. **from** pprint **import** pprint as pp
2. **from** neural\_network **import** NeuralNetwork
3. **import** torch
4. **import** torch.nn as nn
5. **from** torchvision **import** datasets, transforms
6. **from** time **import** time
7. **from** torch.autograd **import** Variable
8. **import** matplotlib.pyplot as plt
9. plt.ioff()
10. **class** NNImg2Num:
12. **def** \_\_init\_\_(self):
13. self.train\_batch\_size = 60
14. self.epoch = 30
15. self.labels = 10
16. self.rate = 30
17. self.input\_size = 28 \* 28
18. self.test\_batch\_size = 10 \* self.train\_batch\_size
19. self.test\_loader = torch.utils.data.DataLoader(
20. datasets.MNIST('./data',
21. train=True,
22. download=True,
23. transform=transforms.Compose([transforms.ToTensor()])),
24. batch\_size=self.test\_batch\_size, shuffle=True)
26. self.train\_loader = torch.utils.data.DataLoader(
27. datasets.MNIST('./data',
28. train=True,
29. download=True,
30. transform=transforms.Compose([transforms.ToTensor()])),
31. batch\_size=self.train\_batch\_size, shuffle=True)
33. # input image is 28 \* 28 so convert to 1D matrix
34. # output labels are 10 [0 - 9]
35. self.model = nn.Sequential(
36. nn.Linear(self.input\_size, 512), nn.Sigmoid(),
37. nn.Linear(512, 256), nn.Sigmoid(),
38. nn.Linear(256, 64), nn.Sigmoid(),
39. nn.Linear(64, self.labels), nn.Sigmoid(),
40. )
42. self.optimizer = torch.optim.SGD(self.model.parameters(), lr=self.rate)
43. self.loss\_function = nn.MSELoss()
45. **def** forward(self, img):
47. output = self.model.forward(img.view(1, self.input\_size))
48. \_, result = torch.max(output, 1)
49. **return** result
51. **def** train(self, plot=False):
52. **print**('training')
53. **def** onehot\_training(target, batch\_size):
54. output = torch.zeros(batch\_size, self.labels)
55. **for** i **in** range(batch\_size):
56. output[i][int(target[i])] = 1.0
57. **return** output
59. **def** training():
60. loss = 0
61. self.model.train() # set to training mode
62. **for** batch\_id, (data, target) **in** enumerate(self.train\_loader):
63. # data.view change the dimension of input to use forward function
64. self.optimizer.zero\_grad()
65. forward\_pass\_output = self.model(data.view(self.train\_batch\_size, self.input\_size))
66. onehot\_target = onehot\_training(target, self.train\_batch\_size)
67. #print(onehot\_target.type())
68. cur\_loss = self.loss\_function(forward\_pass\_output, onehot\_target)
69. loss += cur\_loss.data
70. cur\_loss.backward()
71. self.optimizer.step()
72. # loss / number of batches
73. avg\_loss = loss / (len(self.train\_loader.dataset) / self.train\_batch\_size)
74. **return** avg\_loss
76. **def** testing():
77. self.model.eval()
78. loss = 0
79. correct = 0
80. **for** batch\_id, (data, target) **in** enumerate(self.test\_loader):
81. # data.view change the dimension of input to use forward function
82. forward\_pass\_output = self.model(data.view(self.test\_batch\_size, self.input\_size))
83. onehot\_target = onehot\_training(target, self.test\_batch\_size)
84. cur\_loss = self.loss\_function(forward\_pass\_output, onehot\_target)
85. loss += cur\_loss.data
86. #print(forward\_pass\_output.size())
87. #print(onehot\_target.size())
88. **for** i **in** range(self.test\_batch\_size):
89. val, position = torch.max(forward\_pass\_output.data[i], 0)
90. #   print('prediction = {}, actual = {}'.format(int(position), target[i]))
91. **if** position == target[i]:
92. correct += 1
93. # loss / number of batches
94. avg\_loss = loss / (len(self.test\_loader.dataset) / self.test\_batch\_size)
95. accuracy = correct / len(self.test\_loader.dataset)
96. **return** avg\_loss, accuracy
97. acc\_list = []
98. train\_loss\_list = []
99. test\_loss\_list = []
100. speed = []
101. **for** i **in** range(self.epoch):
102. s = time()
103. train\_loss = training()
104. e = time()
105. test\_loss,accuracy = testing()
106. **print**('Epoch {}, training\_loss = {}, testing\_loss = {}, accuracy = {}, time = {}'.format(i, train\_loss, test\_loss, accuracy, e - s))
107. acc\_list.append(accuracy)
108. train\_loss\_list.append(train\_loss)
109. test\_loss\_list.append(test\_loss)
110. speed.append(e-s)
111. **if** plot == True:
112. **return** speed, train\_loss\_list, test\_loss\_list, acc\_list
114. '''''
115. plt.plot(range(self.epoch), acc\_list, 'r|--', label='Accuracy')
116. plt.plot(range(self.epoch), train\_loss\_list, 'b\*--', label='Training Loss')
117. plt.plot(range(self.epoch), test\_loss\_list, 'yo--', label='Test Loss')
118. plt.xlabel('Epoch')
119. plt.legend()
120. plt.title('Library Neural Network Evaluation')
121. plt.savefig('nn\_compare.png')
122. plt.clf()
123. '''

test.py

1. **from** my\_img2num **import** MyImg2Num
2. **import** matplotlib.pyplot as plt
3. **from** nn\_img2num **import** NNImg2Num
4. plt.ioff()
5. **print**('running self nn')
6. my\_img\_2num = MyImg2Num()
7. my\_time,my\_train\_loss, my\_test\_loss, my\_accuracy = my\_img\_2num.train(True)
8. **print**(my\_time)
9. **print**('running library nn')
10. nn\_img = NNImg2Num()
11. nn\_time, nn\_train\_loss, nn\_test\_loss, nn\_accuracy = nn\_img.train(True)
12. **print**(nn\_time)
13. data = [my\_time, nn\_time]
14. plt.boxplot(data)
15. plt.xticks(range(1, 3), ['MyImg2Num', 'NNImg2Num'])
16. plt.ylabel('Running Time in Seconds')
17. plt.savefig('efficiency.png')
18. plt.clf()
20. plt.plot(range(30), nn\_accuracy, 'r\*--', label='Accuracy of torch nn')
21. plt.plot(range(30), my\_accuracy, 'b|--', label='Accuracy of my nn')
22. plt.xlabel('Epoch')
23. plt.ylabel('Accuracy')
24. plt.savefig('accuracy.png')
25. plt.clf()
27. plt.plot(range(30), my\_train\_loss, 'r\*--', label='Training loss of my nn')
28. plt.plot(range(30), nn\_train\_loss, 'b|--', label='Training loss of torch nn')
30. plt.plot(range(30), my\_test\_loss, 'ro-.', label='Testing loss of my nn')
31. plt.plot(range(30), nn\_test\_loss, 'n+-.', label='Testing loss of torch nn')
32. plt.xlabel('Epoch')
33. plt.ylabel('Loss')
34. plt.savefig('loss.png')
35. plt.clf()