# 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.

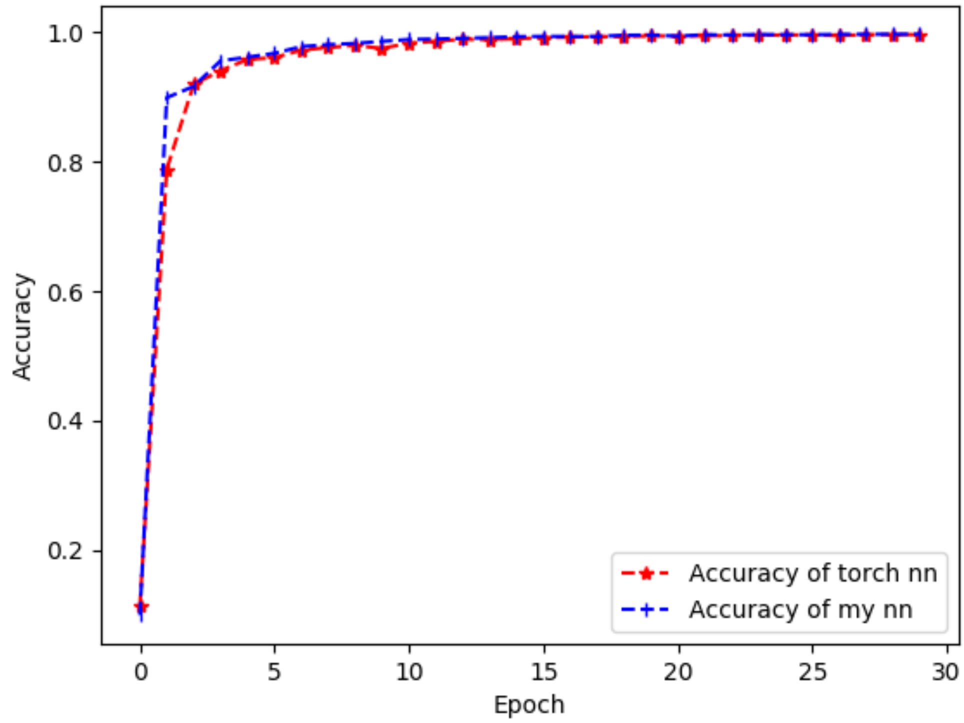


*Figure 1 Accuracy in Both Implementations*

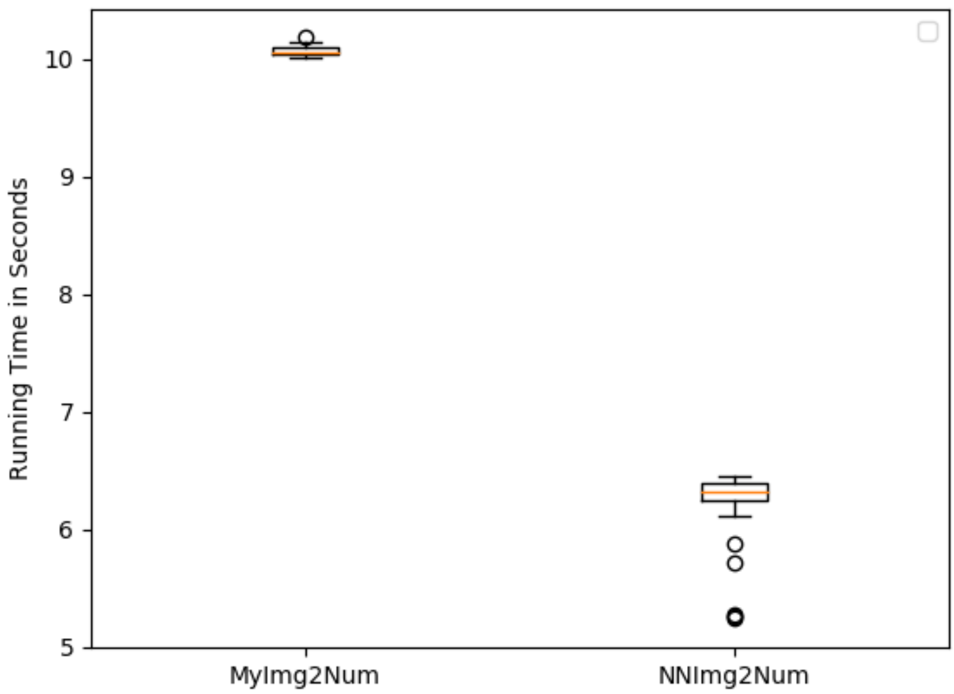


*Figure 2 Running Time for Both Implementations*

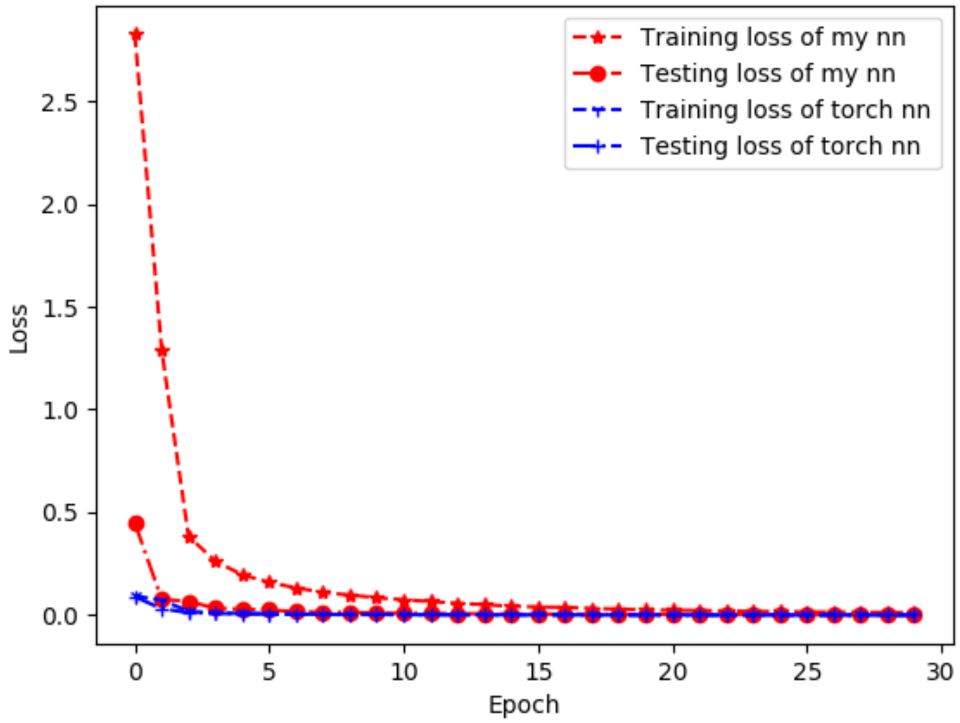


*Figure 3 The change of Training and Testing Loss*

# Appendix

my_img2num.py

```python
from pprint import pprint as pp
from neural_network import NeuralNetwork
import torch
from torchvision import datasets, transforms
from time import time
import matplotlib.pyplot as plt
plt.ioff()
class MyImg2Num:
    def __init__(self):
        self.train_batch_size = 60
        self.epoch = 30
        self.labels = 10
        self.rate = 0.1
        self.input_size = 28 * 28
        self.test_batch_size = 10 * self.train_batch_size
        self.test_loader = torch.utils.data.DataLoader(
            datasets.MNIST('./data',
                train=True,
                download=True,
```

```python
                transform=transforms.Compose([transforms.ToTensor()])),
                batch_size=self.test_batch_size, shuffle=True)

        self.train_loader = torch.utils.data.DataLoader(
            datasets.MNIST('./data',
                train=True,
                download=True,
                transform=transforms.Compose([transforms.ToTensor()])),
                batch_size=self.train_batch_size, shuffle=True)

        # input image is 28 * 28 so convert to 1D matrix
        # output labels are 10 [0 - 9]
        self.nn = NeuralNetwork([self.input_size, 512, 256, 64, self.labels])

    def train(self, plot=False):
        print('training')
        def onehot_training(target, batch_size):
                output = torch.zeros(batch_size, self.labels)
                for i in range(batch_size):
                    output[i][int(target[i])] = 1.0
                return output

        def training():
            loss = 0
            for batch_id, (data, target) in enumerate(self.train_loader):
                # data.view change the dimension of input to use forward function
                forward_pass_output = self.nn.forward(data.view(self.train_batch_size, self.input_size).type(torch.DoubleTensor))
                onehot_target = onehot_training(target, self.train_batch_size).type(torch.DoubleTensor)
                #print(onehot_target.type())
                self.nn.backward(onehot_target)
                loss += self.nn.total_loss
                self.nn.updateParams(self.rate)
            # loss / number of batches
            avg_loss = loss / (len(self.train_loader.dataset) / self.train_batch_size)
            return avg_loss

        def testing():
            loss = 0
            correct = 0
```

```python
            for batch_id, (data, target) in enumerate(self.test_loader):
                # data.view change the dimension of input to use forward function
                forward_pass_output = self.nn.forward(data.view(self.test_batch_size, self.input_size).type(torch.DoubleTensor))
                onehot_target = onehot_training(target, self.test_batch_size).type(torch.DoubleTensor)
                loss += (onehot_target - forward_pass_output).pow(2).sum() / 2
                #print(forward_pass_output.size())
                #print(onehot_target.size())
                for i in range(self.test_batch_size):
                    val, position = torch.max(forward_pass_output[i], 0)
                 #   print('prediction = {}, actual = {}'.format(int(position), target[i]))
                    if position == target[i]:
                        correct += 1
            # loss / number of batches
            avg_loss = loss / len(self.test_loader.dataset)
            accuracy = correct / len(self.test_loader.dataset)
            return avg_loss, accuracy
        acc_list = []
        train_loss_list = []
        test_loss_list = []
        speed = []

        for i in range(self.epoch):
            s = time()
            train_loss = training()
            e = time()
            test_loss,accuracy = testing()
            print('Epoch {}, training_loss = {}, testing_loss = {}, accuracy = {}, time = {}'.format(i, train_loss, test_loss, accuracy, e - s))
            acc_list.append(accuracy)
            train_loss_list.append(train_loss)
            test_loss_list.append(test_loss)
            speed.append(e-s)
        if plot:
            return speed, train_loss_list, test_loss_list, acc_list


    def forward(self, img):

        output = self.nn.forward(img.view(1, self.input_size))
```

```
        _, result = torch.max(output, 1)
        return result
        '''''
         plt.plot(range(self.epoch), acc_list, 'r|--', label='Accuracy')
         plt.plot(range(self.epoch), train_loss_list, 'b*--', label='Training Loss')
         plt.plot(range(self.epoch), test_loss_list, 'yo--', label='Test Loss')
         plt.xlabel('Epoch')
         plt.legend()
         plt.title('My Neural Network Evaluation')
         plt.savefig('my_compare.png')
         plt.clf()
         '''

```

neural_network.py

```
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 size', target.size())
        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)
            #print(delta.size())

            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)))
                    #print(indices)
                    #indices = torch.LongTensor([0,1])
                    #delta = torch.index_select(delta, 0, indices)
                    delta = delta.narrow(0, 0, delta.size(0) - 1)
                # 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])
```

nn_img2num.py

```
from pprint import pprint as pp
from neural_network import NeuralNetwork
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from time import time
from torch.autograd import Variable
import matplotlib.pyplot as plt
plt.ioff()
class NNImg2Num:

    def __init__(self):
        self.train_batch_size = 60
        self.epoch = 30
        self.labels = 10
        self.rate = 30
        self.input_size = 28 * 28
        self.test_batch_size = 10 * self.train_batch_size
        self.test_loader = torch.utils.data.DataLoader(
            datasets.MNIST('./data',
                train=True,
                download=True,
                transform=transforms.Compose([transforms.ToTensor()])),
                batch_size=self.test_batch_size, shuffle=True)

        self.train_loader = torch.utils.data.DataLoader(
            datasets.MNIST('./data',
                train=True,
                download=True,
                transform=transforms.Compose([transforms.ToTensor()])),
                batch_size=self.train_batch_size, shuffle=True)

        # input image is 28 * 28 so convert to 1D matrix
```

```
        # output labels are 10 [0 - 9]
        self.model = nn.Sequential(
                nn.Linear(self.input_size, 512), nn.Sigmoid(),
                nn.Linear(512, 256), nn.Sigmoid(),
                nn.Linear(256, 64), nn.Sigmoid(),
                nn.Linear(64, self.labels), nn.Sigmoid(),
                )

        self.optimizer = torch.optim.SGD(self.model.parameters(), lr=self.rate)
        self.loss_function = nn.MSELoss()

    def forward(self, img):

        output = self.model.forward(img.view(1, self.input_size))
        _, result = torch.max(output, 1)
        return result

    def train(self, plot=False):
        print('training')
        def onehot_training(target, batch_size):
                output = torch.zeros(batch_size, self.labels)
                for i in range(batch_size):
                    output[i][int(target[i])] = 1.0
                return output

        def training():
            loss = 0
            self.model.train() # set to training mode
            for batch_id, (data, target) in enumerate(self.train_loader):
                # data.view change the dimension of input to use forward function
                self.optimizer.zero_grad()
                forward_pass_output = self.model(data.view(self.train_batch_size, self.input_size))
                onehot_target = onehot_training(target, self.train_batch_size)
                #print(onehot_target.type())
                cur_loss = self.loss_function(forward_pass_output, onehot_target)
                loss += cur_loss.data
                cur_loss.backward()
                self.optimizer.step()
            # loss / number of batches
```

```
            avg_loss = loss / (len(self.train_loader.dataset) / self.train_batch_size)
            return avg_loss

        def testing():
            self.model.eval()
            loss = 0
            correct = 0
            for batch_id, (data, target) in enumerate(self.test_loader):
                # data.view change the dimension of input to use forward function
                forward_pass_output = self.model(data.view(self.test_batch_size, self.input_size))
                onehot_target = onehot_training(target, self.test_batch_size)
                cur_loss = self.loss_function(forward_pass_output, onehot_target)
                loss += cur_loss.data
                #print(forward_pass_output.size())
                #print(onehot_target.size())
                for i in range(self.test_batch_size):
                    val, position = torch.max(forward_pass_output.data[i], 0)
                 #   print('prediction = {}, actual = {}'.format(int(position), target[i]))
                    if position == target[i]:
                        correct += 1
            # loss / number of batches
            avg_loss = loss / (len(self.test_loader.dataset) / self.test_batch_size)
            accuracy = correct / len(self.test_loader.dataset)
            return avg_loss, accuracy
        acc_list = []
        train_loss_list = []
        test_loss_list = []
        speed = []
        for i in range(self.epoch):
            s = time()
            train_loss = training()
            e = time()
            test_loss,accuracy = testing()
            print('Epoch {}, training_loss = {}, testing_loss = {}, accuracy = {}, time = {}'.format(i, train_loss, test_loss, accuracy, e - s))
            acc_list.append(accuracy)
```

```
            train_loss_list.append(train_loss)
            test_loss_list.append(test_loss)
            speed.append(e-s)
        if plot == True:
            return speed, train_loss_list, test_loss_list, acc_list

'''''
        plt.plot(range(self.epoch), acc_list, 'r|--', label='Accuracy')
        plt.plot(range(self.epoch), train_loss_list, 'b*--', label='Training Loss')
        plt.plot(range(self.epoch), test_loss_list, 'yo--', label='Test Loss')
        plt.xlabel('Epoch')
        plt.legend()
        plt.title('Library Neural Network Evaluation')
        plt.savefig('nn_compare.png')
        plt.clf()
    '''

```

test.py

```
from my_img2num import MyImg2Num
import matplotlib.pyplot as plt
from nn_img2num import NNImg2Num
plt.ioff()
print('running self nn')
my_img_2num = MyImg2Num()
my_time,my_train_loss, my_test_loss, my_accuracy = my_img_2num.train(True)
print(my_time)
print('running library nn')
nn_img = NNImg2Num()
nn_time, nn_train_loss, nn_test_loss, nn_accuracy = nn_img.train(True)
print(nn_time)
data = [my_time, nn_time]
plt.boxplot(data)
plt.xticks(range(1, 3), ['MyImg2Num', 'NNImg2Num'])
plt.ylabel('Running Time in Seconds')
plt.savefig('efficiency.png')
plt.clf()

plt.plot(range(30), nn_accuracy, 'r*--', label='Accuracy of torch nn')
plt.plot(range(30), my_accuracy, 'b|--', label='Accuracy of my nn')
plt.xlabel('Epoch')
```

```
plt.ylabel('Accuracy')
plt.savefig('accuracy.png')
plt.clf()

plt.plot(range(30), my_train_loss, 'r*--', label='Training loss of my nn')
plt.plot(range(30), nn_train_loss, 'b|--', label='Training loss of torch nn')

plt.plot(range(30), my_test_loss, 'ro-.', label='Testing loss of my nn')
plt.plot(range(30), nn_test_loss, 'n+-.', label='Testing loss of torch nn')

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.savefig('loss.png')
plt.clf()
```