Linear MNIST Classifier

Yin-Yu Chang

1. (1 pts) Write the code for downloading and formatting the data.

I use PyTorch to download the data. The original data is a 28x28 matrix. I use .flatten() to make the data 1x184 vector. Then, make all training data into a 60000x784 matrix. For testing data, it is a 10000x784 matrix.

```
[70] ### downloading the data
     ## MNIST dataset(images and labels)
     #use ToTensor() transform to convert images into Pytorch tensors.
     train_data = MNIST(root = 'data/', train = True, download = True, transform = transforms.ToTensor())
     test_data = MNIST(root = 'data/', train = False, download = True, transform = transforms.ToTensor())


     n_train = 60000
     n_test = 10000

     # formatting training data
     flat_train_data = [[0] for i in range(n_train)]
     train_data_label = [0 for i in range(n_train)]
     for i in range(n_train):
       image_tensor, label = train_data[i]
       flat_train_data[i] = image_tensor[:,0:28,0:28].flatten().tolist()
       train_data_label[i] = label

     # formatting testing data
     flat_test_data = [[0] for i in range(n_test)]
     test_data_label = [0 for i in range(n_test)]
     for i in range(n_test):
         image_tensor, label = test_data[i]
         flat_test_data[i] = image_tensor[:,0:28,0:28].flatten().tolist()
         test_data_label[i] = label
```

As for the label, I one hot encode every label and make it a 60000x10 matrix.

```
# initialize Y = one hot encoding label
Y = [[0 for i in range(10)] for j in range(n_train)]
for i in range(n_train):
  Y[i][train_data_label[i]] = 1
```

2. (5 pts) Write the code for minibatch SGD implementation for your linear MNIST classifier.

```
# number of iteration
t = 50
# initialize batch size
b = 1000
# learning rate
# l>0
l = 0.01
# initialize W
W = [[0 for i in range(10)] for j in range(784)]
g = 0
start = time.time()
g = [[0 for i in range(10)] for j in range(784)]
G = [[0 for i in range(10)] for j in range(784)]
x = []
y = []

lost_list = [0 for i in range(t)]
accuracy_list = [0 for i in range(t)]
x_list = [i for i in range(t)]
```

```
for q in range(t):
    print("iteration is {iter}".format(iter = q))
    #g = 0
    g = [[0 for i in range(10)] for j in range(784)]
    for z in range(b):
        print("batch size is {b}".format(b = z))
        temp = random.randrange(60000)
        x.append(flat_train_data[temp])

        x_T = np.array(x).T
        y.append(Y[temp])

        xT_mul_x = x_T.dot(x)
        xT_mul_x_mul_W = xT_mul_x.dot(W)
        xT_mul_y = x_T.dot(y)

        temp = matrix_sub(xT_mul_x_mul_W, xT_mul_y)
        g = matrix_add(g, matrix_sub(xT_mul_x_mul_W, xT_mul_y))

        x = []
        y = []

    G = matrix_div(g, b)
    W = matrix_sub(W, matrix_mul(G, l))

    W_T = np.array(W).T
    training_loss(W_T, flat_train_data, Y)
    lost_list[q] = training_loss(W_T, flat_train_data, Y)
    accuracy_list[q] = accuracy(W_T, flat_test_data, test_data_label)

    print("training loss is {loss}".format(loss = training_loss(W_T, flat_train_data, Y)))
    print("accuracy is {accuracy}".format(accuracy = accuracy(W_T, flat_test_data, test_data_label)))
```

In the inner for loop, the gradient would be calculated. We would renew the weight W for every iteration by subtracting the (learning rate * gradient).

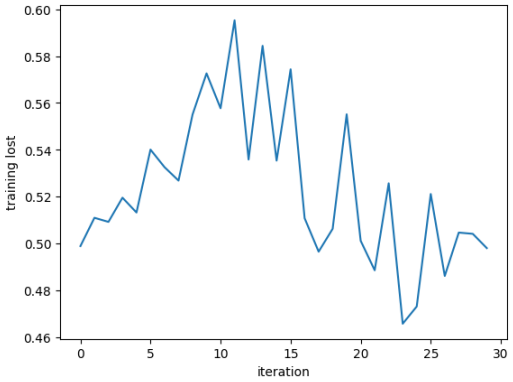
3. (7 pts) The role of batch size: Run your code with batch sizes B = 1, 10, 100, 1000. For each batch size,

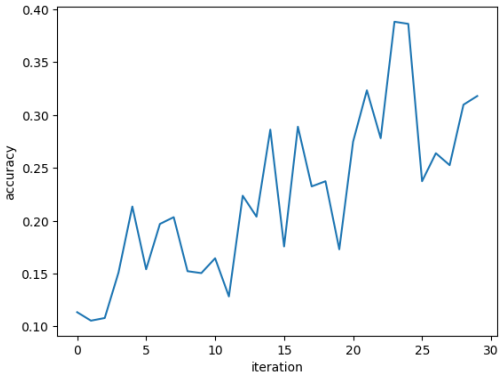
• determine a good choice of learning rate

• pick ITR sufficiently large to ensure the (approximate) convergence of the training loss

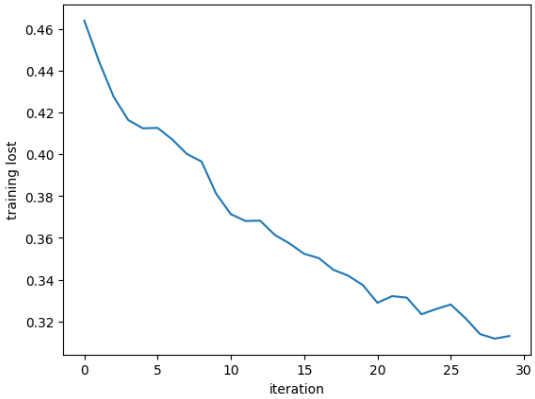
• Plot the progress of training loss (y-axis) as a function of the iteration counter t (x-axis)
• Report how long the training takes (in seconds).
• Plot the progress of the test accuracy (y-axis) as a function of the iteration counter t (x-axis)

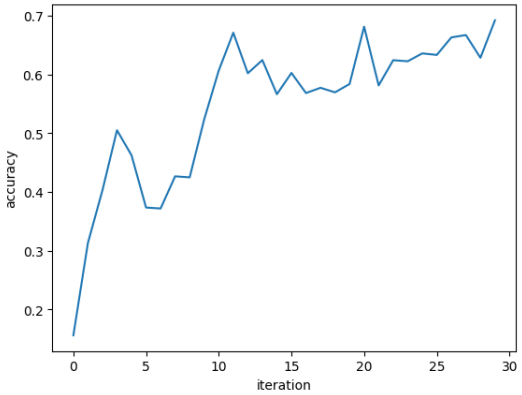
1. B=1, ITR=30,
   Took 420(s)



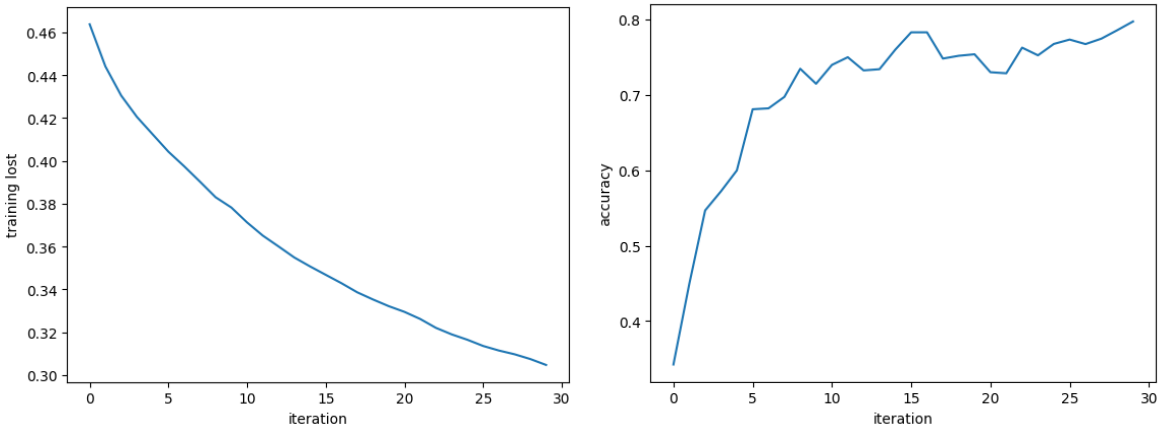


2. B=10, ITR =30,
   Took 481(s)

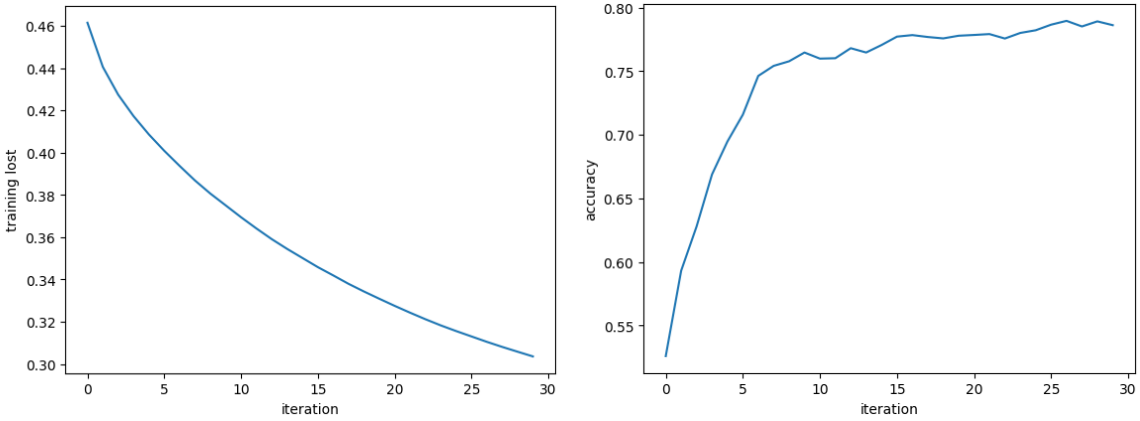




3. B=100, ITR=30,
   Took 602(s)



4. B=1000, ITR=30,
   Took 1775(s)



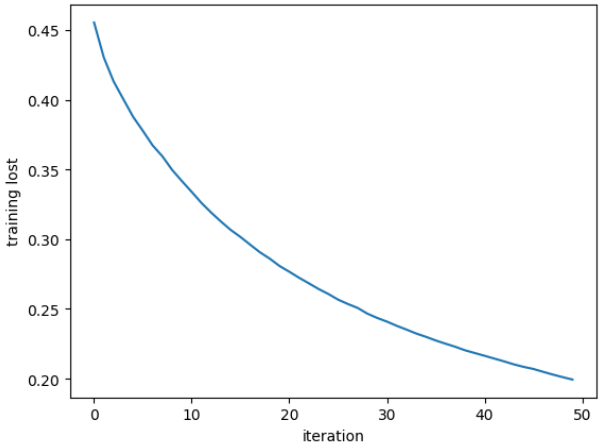
4. (1 pt) Comment on the role of batch size.

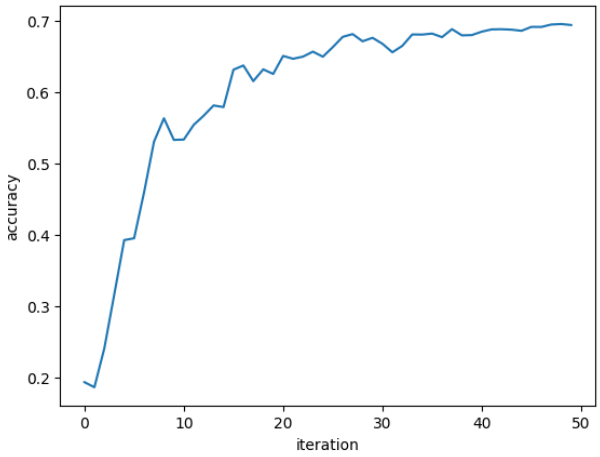
With a larger batch size, the loss would decrease stabler and make the loss chart like a curve.
For accuracy, with a larger batch size, the accuracy would increase quicker and converge faster.

5. (6 pts) The role of training dataset size: Let us reduce the training dataset size. Instead of N = 60,000, let us pick a subset S′ of size N′ from the original dataset without replacement and uniformly at random. Fix batch size to B = 100. Repeat the steps above for N ′ ∈

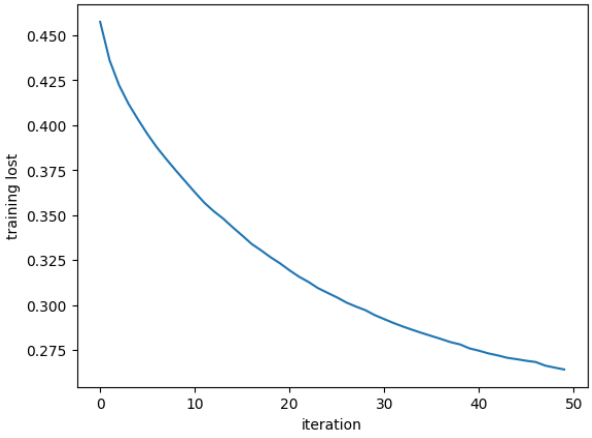
{100, 500, 1000, 10000}. Comment on the accuracy as a function of dataset size.

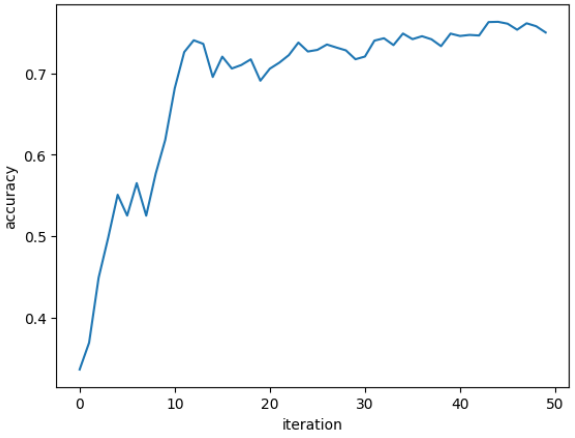
1. N = 100,



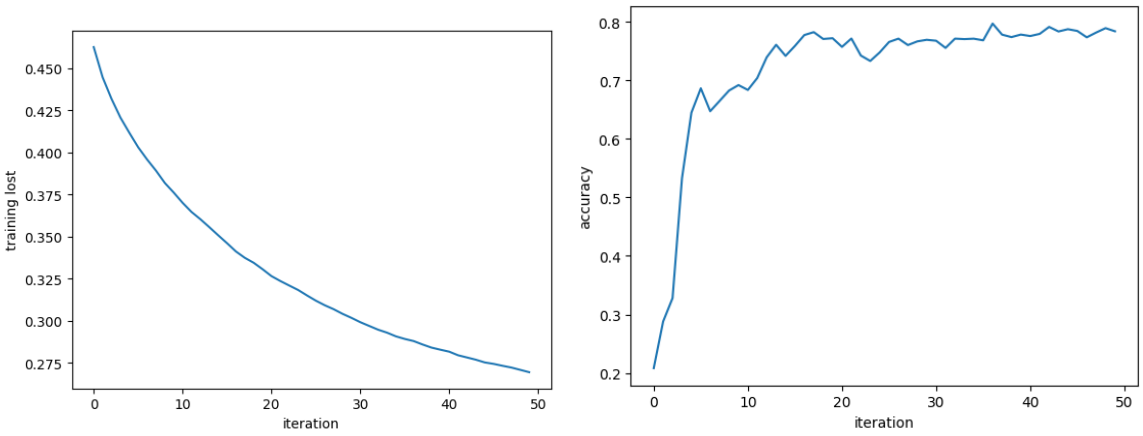


2. N = 500,

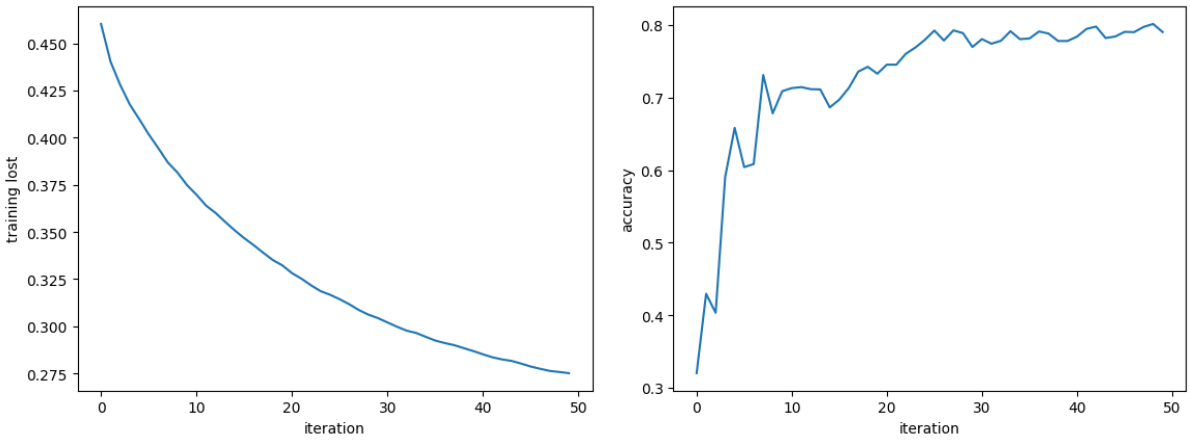




3. N = 1000,



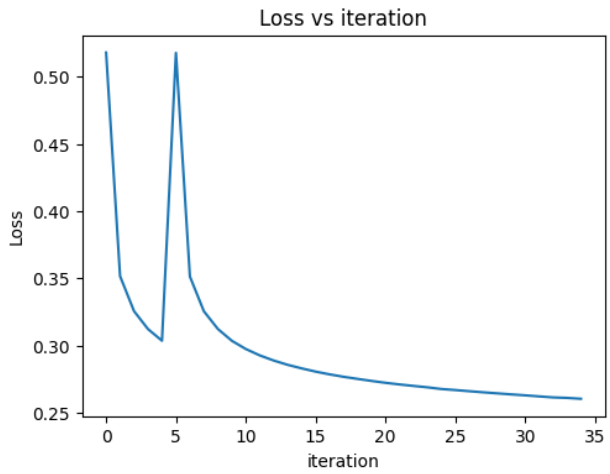
4. N = 10000,

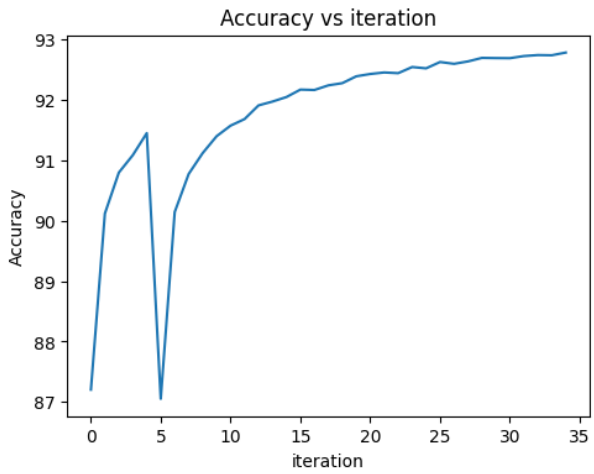


As we can see in the chart, when N is larger, the accuracy would be better and converge earlier.

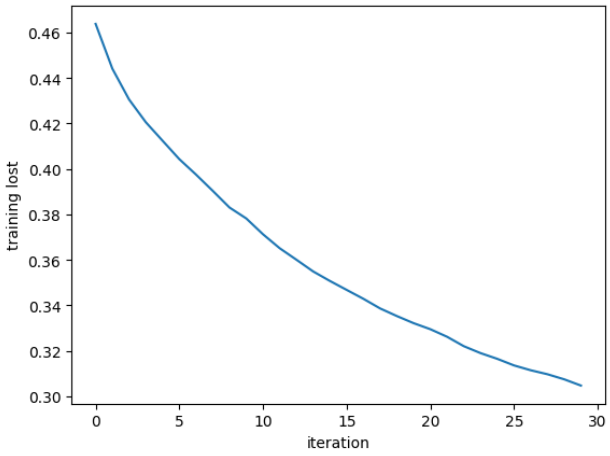
6. (Bonus 5 pts) Simpler Life: Run the linear MNIST classifier with batchsize B = 100 over the full dataset by using PyTorch or Tensorflow. Use same learning rate and initialization W0 = 0. Verify that it is consistent with your hand-coded algorithm by comparing your results (the accuracy and training loss plots).

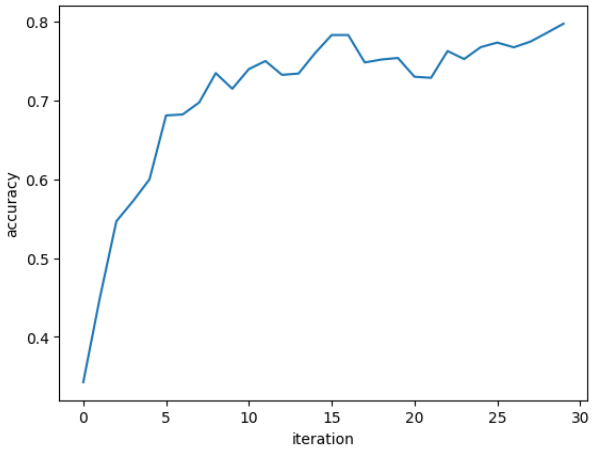
- PyTorch





- Hand code algorithm





I use full training data and set the batch size to 100 for both algorithms.

When using PyTorch, the accuracy is slightly better compared to my hand-code algorithm.