

Downloading data:


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

```
[ ] import matplotlib.pyplot as plt
import numpy as np
import torch
import pandas as pd
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
```

```
[ ] # Download training data from open datasets.
train_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)


# Download test data from open datasets.
test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

 Downloading <http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz>
Failed to download (trying next):

```
 # Preprocess the data
def preprocess_data(data):
    X = [] #(60000, 784)
    Y = [] # (60000, 10)
    for image_tensor, label in data:
        X.append( (image_tensor.numpy()).reshape(784)
        yoh = np.zeros(10) # Convert the label into one-hot
        yoh[label] = 1
        Y.append(yoh)
    X_train = np.array(X)
    Y_train = np.array(Y)
    return X_train, Y_train

X_train, Y_train = preprocess_data(train_data)
X_test, Y_test = preprocess_data(test_data)
```

#2. Sample implementation


```
 #Batch = 10
num_classes = 10
input_size = 784
batch_size = 10
num_epochs = 50
learning_rate = 0.005

reg_lambda = 0.001

batch10S = time.time() #batch1 time start

batch10= train_classifier(X_train, Y_train, num_class

batch10E = time.time() #batch1 time end
batch10TD = batch10E - batch10S
```

```
 Epoch: 0 Loss: 0.39796504069167404
Epoch: 1 Loss: 0.3480134629189122
Epoch: 2 Loss: 0.32339620183569323
Epoch: 3 Loss: 0.3083371352366459
Epoch: 4 Loss: 0.2983578316930637
Epoch: 5 Loss: 0.2915059168714281
Epoch: 10 Loss: 0.27802510145672993
Epoch: 20 Loss: 0.27446914741229445
Epoch: 30 Loss: 0.27380851878867674
Epoch: 40 Loss: 0.2734494110083452
accuracy: 0.916
```

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

The Stochastics gradient descent is similar to general gradient descent. The key difference is SGD updates its weight W for each small batch of the total data points. The advantage is reducing the training time. The downside is negative impact on the accuracy since there is more noisy.

I have implemented the model with batch size = 1, 10, 100, 1000. The parameter I chose are num_classes = 10

input_size = 784

batch_size = 100

num_epochs = 50

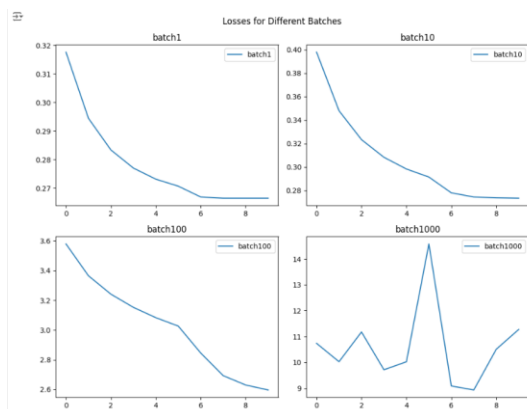
learning_rate = 0.005

reg_lambda = 0.001

I have chosen total number of epoch as 50 since its large enough for most plots to converge. Refer to the losses plot, for the first plot the loss almost flat, implies reached the converged error loss. For the last plot, at Batch = 1000. There is big peak at middle-right of the graph, and the loss start to increasing. I suppose this is due to the relatively larger batch size since by the idea of control variables, batch is the only parameter modified.

The instability can be resolve by pick a even smaller learning rate. I will provide more details at the end.

The following is the plot for losses



Following the running duration for each batch respectively:

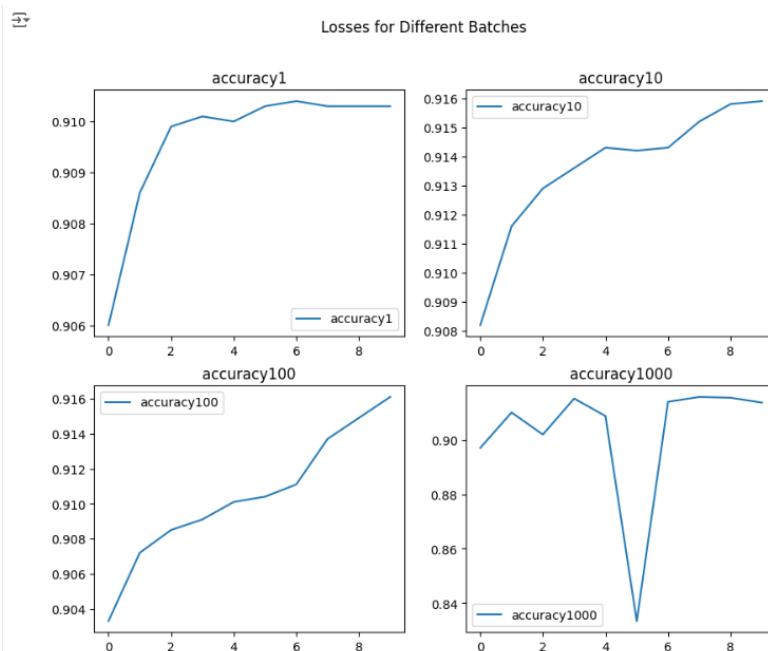
```
times = [batch1TD, batch10TD, batch100TD, batch1000TD]
#mistakely subtract time start and time end. should negate each value
labels = ['batch1', 'batch10', 'batch100', 'batch1000']
for i in range(len(times)):
    print(labels[i], "Running time:", times[i])
```

```
batch1 Running time: 246.22294926643372
batch10 Running time: 37.83369469642639
batch100 Running time: 24.11919379234314
batch1000 Running time: 20.562072038650513
```

The processing duration decreases as the batch size increases. Since for batch=1, we will update W each datapoint, with 60k data points and 50 epoch, we update $60k \times 50$ times.

For batch to 1000, we update W per 1000 datapoints. Each epoch update W $60k/1k = 60$ times. Total 60×50 times. Saved 200 seconds, 90% of processing duration.

Following plot is for accuracy



At batch=1, accuracy reaches a peak fastest since it updates W more frequently. With Batch = 1000, there is a big drop which matches the loss plot.

extra case

For the batch =1000 case: If we update the learn rate to 0.001 from 0.005. The loss and accuracy are stabler. I suspect the reason is the smaller learning rate regulates the intensity of gradient(dW 's value), and since regularization term does not operate with learning rate, Thus regularization's intensity does not change, it keeps its intensity on regulate noisy.

```
#Batch = 1000
num_classes = 10
input_size = 784
batch_size = 1000
num_epochs = 50
learning_rate = 0.001
reg_lambda = 0.001

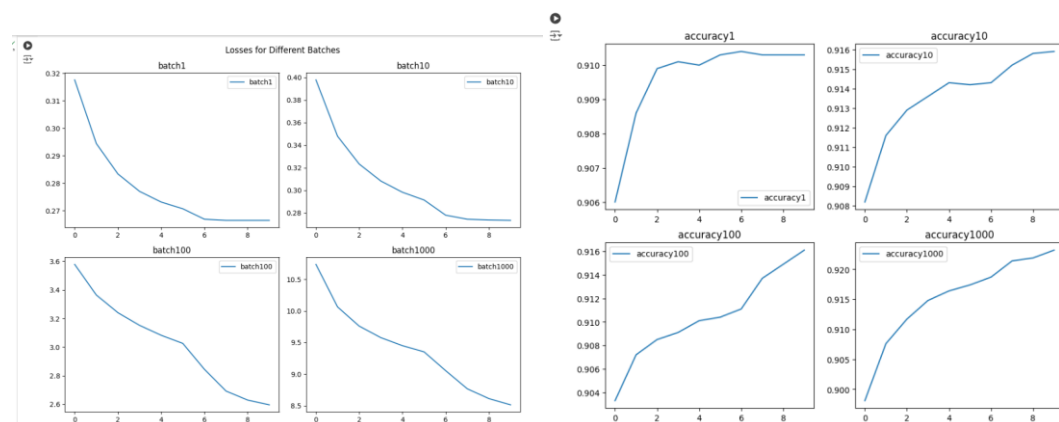
batch1000S = time.time() #batch1 time start

batch1000= train_classifier(X_train, Y_train, num_cl

batch1000E = time.time() #batch1 time end
batch1000TD = batch1000E - batch1000S

print("finished")
```

```
Epoch: 0 Loss: 10.737723106140441
Epoch: 1 Loss: 10.063767196358146
Epoch: 2 Loss: 9.760187117484405
Epoch: 3 Loss: 9.577996074462177
Epoch: 4 Loss: 9.450431815820586
Epoch: 5 Loss: 9.352735855981285
Epoch: 10 Loss: 9.054457012657489
Epoch: 20 Loss: 8.7675637494264
Epoch: 30 Loss: 8.612568328423263
Epoch: 40 Loss: 8.513518691855968
accuracy: 0.9235
```



#6

Implement mnist classification with Tensorflow with learning rate at 0.05 and batch size = 100 yield a similar plot as the hand coded SGD.

Both tensorflow and hand code model converges the loss and increase in their accuracy. However, the TensorFlow seems converged faster. It reached a very low loss in first few epoch (there is a point located almost at x-axis = 0) afterward the loss oscillate since the gradient jumping back and forth at one of the minima. The accuracy matches the loss plot.

```
# code from https://www.tensorflow.org/guide/keras/writing\_a\_training\_loop.html
def train_classifier(X_train, y_train, batch_size, num_epochs, W, X_test, y_test):
    epochLoss = []
    #epochLoss2 = []
    accuracies = []
    for epoch in range(num_epochs):
        for i in range(0, X_train.shape[0], batch_size):
            X_batch = X_train[i:i+batch_size]
            y_batch = y_train[i:i+batch_size]

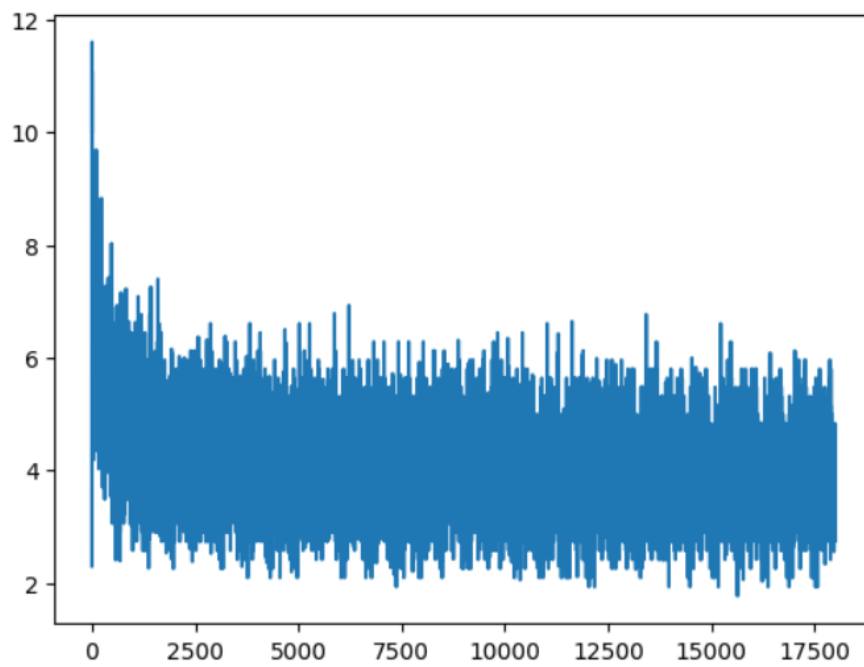
            loss = trainGrad(X_batch, y_batch)
            epochLoss.append(loss.numpy())
            #epochLoss2.append(loss)
        accu = accuracy(W, X_test, y_test)
        accuracies.append(accu)

        if epoch % 10 == 0 or epoch in [0,1,2]:
            print("Epoch: ", epoch, "Loss: ", loss.numpy())
            print("Epoch: ", epoch, "Accuracy: ", accu)

    return epochLoss, accuracies
```

```
lossList = np.array(lossnp[0])  
#plt(lossList)  
type(lossList)  
plt.plot(lossList)
```

[<matplotlib.lines.Line2D at 0x7ccc015258d0>]



✓
0s

```
# accuracy  
accuPlot = np.array(lossnp[1])  
plt.plot(accuPlot)
```

[<matplotlib.lines.Line2D at 0x7ccc0151ae30>]

