OK, thus far we have been talking about linear models. All these can be viewed as a single-layer neural net. The next step is to move on to multi-layer nets. Training these is a bit more involved, and implementing from scratch requires time and effort. Instead, we just use well-established libraries. I prefer PyTorch, which is based on an earlier library called Torch (designed for training neural nets via backprop).

NAME : Y.HEMANTH TEJA EMAIL : hy1713@nyu.edu

```
import numpy as np
import torch
import torchvision

import matplotlib.pyplot as plt
import random
import pandas as pd
import sklearn
```

Torch handles data types a bit differently. Everything in torch is a *tensor*.

```
a = np.random.rand(2,3)
b = torch.from_numpy(a)

# Q4.1 Display the contents of a, b
# ...
# ...
print("Displaying contents of A")
print(a)

print("\nDisplaying contents of B")
print(b)
```

```
Displaying contents of A
[[0.86780681 0.45833768 0.55640672]
 [0.83049916 0.74459275 0.91870588]]

Displaying contents of B
tensor([[0.8678, 0.4583, 0.5564],
        [0.8305, 0.7446, 0.9187]], dtype=torch.float64)
```

The idea in Torch is that tensors allow for easy forward (function evaluations) and backward (gradient) passes.

```
A = torch.rand(2,2)
b = torch.rand(2,1)
x = torch.rand(2,1, requires_grad=True)
```

```
x = torch.rand(2,1, requires_grad=True)

y = torch.matmul(A,x) + b

print(y)
z = y.sum()
print(z)
z.backward()
print(x.grad)
print(x)
```

```
tensor([[1.3683],
        [1.3693]], grad_fn=<AddBackward0>)
tensor(2.7376, grad_fn=<SumBackward0>)
tensor([[1.4168],
        [1.1964]])
tensor([[0.8909],
        [0.5900]], requires_grad=True)
```

Notice how the backward pass computed the gradients using autograd. OK, enough background. Time to train some networks. Let us load the *Fashion MNIST* dataset, which is a database of grayscale images of clothing items.

```
trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=True,downloac
testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=False,download=Tı
```

Let us examine the size of the dataset.

```
# Q4.2 How many training and testing data points are there in the dataset?
# What is the number of features in each data point?
#
# ...
```

```
print(trainingdata)
print("\nTotal number of training data points = 60000\n")
print(testdata)
print("\nTotal number of testing data points is 10000")
```

```
Dataset FashionMNIST
    Number of datapoints: 60000
    Root location: ./FashionMNIST/
    Split: Train
    StandardTransform
Transform: ToTensor()

Total number of training data points = 60000
```

```
len(trainingdata)
```

```
60000
    Split: Test
```

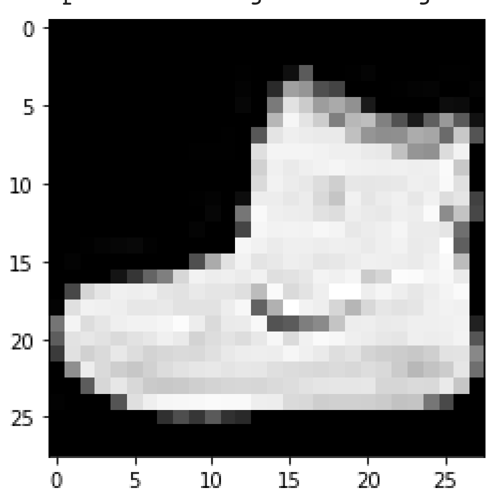
```
len(testdata)
```

```
10000
```

Let us try to visualize some of the images. Since each data point is a tensor (not an array) we need to postprocess to use matplotlib.

```
import matplotlib.pyplot as plt
%matplotlib inline

image, label = trainingdata[0]
# Q4.3 Assuming each sample is an image of size 28x28, show it in matplotlib.
# ...
# ...

plt.imshow(image[0].numpy().squeeze(), cmap = 'Greys_r')
```

```
<matplotlib.image.AxesImage at 0x7fee10464278>
```



Let's try plotting several images. This is conveniently achieved in PyTorch using a *data loader*, which loads data in batches.

```
trainDataLoader = torch.utils.data.DataLoader(trainingdata, batch_size=64, shuffle=Tru
```

```
testDataLoader = torch.utils.data.DataLoader(testdata, batch_size=64, shuffle=False)
images, labels = iter(trainDataLoader).next()
print(images.size(), labels)
```

```
torch.Size([64, 1, 28, 28]) tensor([1, 3, 7, 4, 9, 8, 0, 8, 8, 3, 6, 1, 9, 9, 7,
        0, 5, 4, 4, 8, 0, 6, 6, 0, 4, 0, 4, 7, 4, 0, 7, 9, 2, 9, 4, 9, 3, 5, 7,
        7, 2, 7, 3, 0, 9, 5, 5, 1, 7, 8, 1, 7, 1, 5, 7])
```

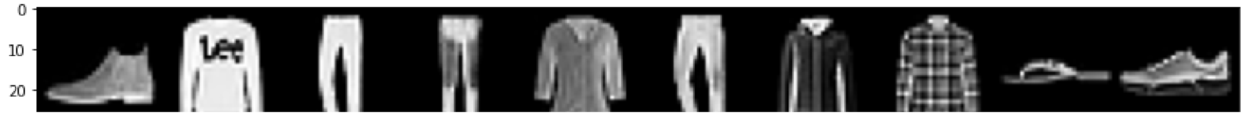
```
def output_label(label):
    output_mapping = {
                 0: "T-shirt/Top",
                 1: "Trouser",
                 2: "Pullover",
                 3: "Dress",
                 4: "Coat",
                 5: "Sandal",
                 6: "Shirt",
                 7: "Sneaker",
                 8: "Bag",
                 9: "Ankle Boot"
                 }
    input = (label.item() if type(label) == torch.Tensor else label)
    return output_mapping[input]
```

```
# Q4.4 Visualize the first 10 images of the first minibatch
# returned by testDataLoader.
# ...
# ...

printOutput = torch.utils.data.DataLoader(testdata, batch_size=10)
batch = next(iter(printOutput))
images, labels = batch
print(type(images), type(labels))
print(images.shape, labels.shape)
```

```
<class 'torch.Tensor'> <class 'torch.Tensor'>
torch.Size([10, 1, 28, 28]) torch.Size([10])
```

```
grid = torchvision.utils.make_grid(images, nrow=10)
plt.figure(figsize=(15, 20))
plt.imshow(np.transpose(grid, (1, 2, 0)))
print("labels: ", end=" ")
for i, label in enumerate(labels):
    print(output_label(label), end=", ")
```

```
labels:  Ankle Boot, Pullover, Trouser, Trouser, Shirt, Trouser, Coat, Shirt, Sa
```



Now we are ready to define our linear model. Here is some boilerplate PyTorch code that implements the forward model for a single layer network for logistic regression (similar to the one discussed in class notes).

```
class LinearReg(torch.nn.Module):
  def __init__(self, input_dim, output_dim):
    super(LinearReg, self).__init__()
    self.linear = torch.nn.Linear(input_dim, output_dim)

  def forward(self, x):
    x = x.view(-1,28*28)
    transformed_x = self.linear(x)
    return transformed_x

input_dim = 28*28
output_dim = 10

model = LinearReg(input_dim, output_dim)

Loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.03)
```

```
len(trainDataLoader)
```

```
938
```

```
print(model)
print(model.parameters())
print(len(list(model.parameters())))
```

```
LinearReg(
  (linear): Linear(in_features=784, out_features=10, bias=True)
)
<generator object Module.parameters at 0x7fee10034e08>
2
```

Cool! Now we have set everything up. Let's try to train the network.

```
# Q4.5 Write down a for-loop that trains this network for 20 minibatch iterations,
# and print the train/test losses.
# Save them in the variables above. If done correctly, you should be able to
# execute the next code block.
# ...
```

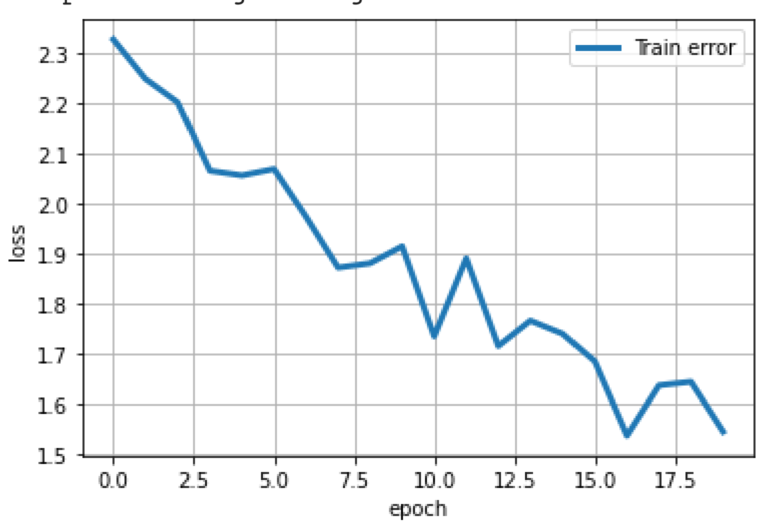
```
# ...
train_loss_history = []
test_loss_history = []
for i, (images, labels) in enumerate(trainDataLoader):

    images = images.view(-1, 28*28).requires_grad_()
    labels = labels

    # Clear gradients with respect to the parameters
    optimizer.zero_grad()

    # Forward pass
    outputs = model(images)

    # Calculate Loss
    calcLoss = Loss(outputs, labels)

    # Getting gradients with respect to the parameters
    calcLoss.backward()

    #Update the parameters
    optimizer.step()
    train_loss_history.append(calcLoss.item())
    print(calcLoss.item())

    correctPredictions = 0
    totalNumLabels = 0

    if(i >=19):
          break

    for i ,(images, labels) in enumerate(testDataLoader):

        images = images.view(-1, 28*28).requires_grad_()

        outputs = model(images)
        calcLoss = Loss(outputs, labels)

        if(i%500 == 0):
          test_loss_history.append(calcLoss.item())

        # Get predictions from the maximum value
        _, predicted = torch.max(outputs.data, 1)

        #Figuring the total number of labels
        totalNumLabels += labels.size(0)

        # Figuring the total number of correct predictions
        correctPredictions += (predicted == labels).sum()

        #Computing accuracy
        accuracy = 100 * correctPredictions.numpy() / totalNumLabels
```

```
    accuracy = 100 * correctPredictions.numpy() / totalNumLabels
```

```
2.325894355773926
2.247469663619995
2.20113205909729
2.0652406215667725
2.0560007095336914
2.0687193870544434
1.9734618663787842
1.8722034692764282
1.8808752298355103
1.9147193431854248
1.7352871894836426
1.8906794786453247
1.7159597873687744
1.7664744853973389
1.74052095413208
1.6854078769683838
1.5379551649093628
1.6379114389419556
1.6449737548828125
1.5461499691009521
```

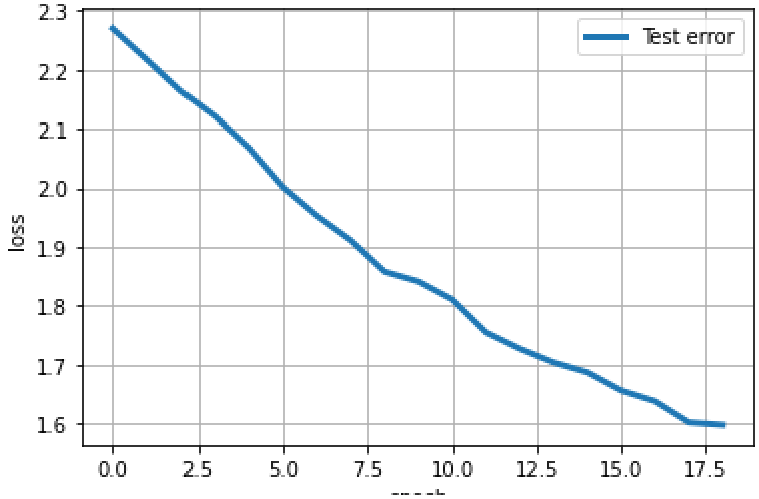
```
plt.plot(train_loss_history,'-',linewidth=3,label='Train error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fee101bc0b8>
```




```
plt.plot(test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fee10152748>
```



Neat! Now let's evaluate our model accuracy on the entire dataset. The predicted class label for a given input image can computed by looking at the output of the neural network model and computing the index corresponding to the maximum activation. Something like

predicted_output = net(images) _, predicted_labels = torch.max(predicted_output,1)

```
predicted_output = model(images)
print(torch.max(predicted_output, 1))
fit = Loss(predicted_output, labels)
print(labels)
```

```
torch.return_types.max(
values=tensor([1.0211, 1.1970, 0.6002, 1.0526, 1.4884, 0.8140, 1.0334, 0.9941, 1
        1.3968, 2.0169, 1.2345, 1.7619, 0.5816, 1.1849, 1.8226, 0.7461, 1.2836,
        1.6389, 1.8291, 1.8179, 1.2951, 1.2433, 1.5005, 0.9930, 0.4443, 1.2734,
        1.8999, 0.7364, 1.6322, 1.9442, 1.4469, 1.5511, 0.7837, 1.7236, 0.7170,
        0.9840, 0.8445, 1.0878, 1.7428, 1.3926, 1.1502, 1.7046, 0.9351, 1.1246,
        1.8468, 1.0007, 1.4352, 1.4131, 0.9919, 1.3044, 1.0443, 1.6108, 2.0653,
        1.1146, 1.8927, 1.3643, 1.6146, 1.7828, 0.6247, 0.9762, 1.4032, 0.6954,
        0.4679], grad_fn=<MaxBackward0>),
indices=tensor([7, 4, 4, 7, 0, 3, 7, 7, 4, 7, 9, 0, 1, 4, 4, 2, 3, 4, 9, 9, 4, 0
        9, 8, 3, 4, 4, 2, 1, 1, 1, 4, 4, 4, 8, 4, 7, 4, 3, 0, 7, 7, 7, 4, 4, 7,
        4, 4, 7, 2, 7, 9, 8, 4, 1, 4, 4, 4, 0, 4, 7, 4]))
tensor([8, 4, 8, 5, 6, 3, 5, 7, 6, 7, 9, 0, 1, 2, 6, 2, 3, 6, 9, 9, 4, 0, 4, 4,
        5, 8, 0, 4, 6, 2, 1, 1, 1, 6, 2, 6, 2, 6, 7, 4, 3, 0, 7, 7, 7, 4, 6, 7,
        6, 4, 7, 2, 9, 9, 8, 4, 1, 2, 2, 8, 0, 4, 5, 3])
```

```
def evaluate(dataloader):
# Q4.6 Implement a function here that evaluates training and testing accuracy.
# Here, accuracy is measured by probability of successful classification.
  correctPredictions = 0
  totalNumLabels = 0
  c = 0
  for images, labels in dataloader:

          images = images.view(-1, 28*28).requires_grad_()
```

```
            # Forward pass
            outputs = model(images)
            # Get predictions from the maximum value
            _, predicted = torch.max(outputs.data, 1)
            # Computing the total number of labels
            totalNumLabels += labels.size(0)
            # Computing the total number of correct predictions
            correctPredictions += (predicted == labels).sum()
    correctPredictions = correctPredictions.numpy()
    FinalAccuracy = correctPredictions*100/totalNumLabels
    return FinalAccuracy

print('Train acc = %0.2f, test acc = %0.2f' % (evaluate(trainDataLoader), evaluate(tes
```

```
Train acc = 60.20, test acc = 59.27
```