HW3

December 1, 2021

1 2 Classification of handwritten digits

1.1 2.1 Downloading MNIST

[ ]:
```
from torch.utils.data import DataLoader, TensorDataset
import numpy as np
from torch import nn
from torch.nn.functional import softmax, relu, sigmoid, tanh
from torchvision import datasets, transforms
import os
from ps3_functions import train
import matplotlib.pyplot as plt
from torch.nn import functional as F

path_figures = os.getcwd().replace('code','figures')
if not os.path.exists(path_figures):
    os.makedirs(path_figures)

path_save = os.path.join(os.getcwd(), 'models')
if not os.path.exists(path_save):
    os.makedirs(path_save)

path_results = os.path.join(os.getcwd(), 'results')
if not os.path.exists(path_results):
    os.makedirs(path_results)

path_outputs = os.path.join(os.getcwd(), 'outputs')
if not os.path.exists(path_outputs):
    os.makedirs(path_outputs)

mnist_train = datasets.MNIST(root = 'data', train=True, download=True,␣
 ↪transform = transforms.ToTensor())
mnist_test = datasets.MNIST(root = 'data', train=False, download=True,␣
 ↪transform = transforms.ToTensor())
```

[ ]:
```
import torch

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[ ]:
```
print(device)
```

```
cuda
```

1.2 2.2 Feed-forward Neural Network

1.2.1 please see the code for section 1-6

1.2.2 Fill in the provided train function, completing the TODOs to build a training loop and an evaluation function. Then train the model for 100 epochs with a batch size of 128, and a learning rate of 0.5. This training might take a few minutes.

[ ]:
```
# training parameters
bs = 128
```

[ ]:
```
# create training set and testing set
train_dataloader = DataLoader(mnist_train, batch_size=bs, shuffle=True)
test_dataloader = DataLoader(mnist_test, batch_size=bs, shuffle=True)
```

[ ]:
```
lr = 0.5
num_epochs = 50
model_name = 'relu'

class FeedForwardNet(nn.Module):
    """ Simple feed forward network with one hidden layer."""
    def __init__(self): # initialize the model
        super(FeedForwardNet, self).__init__() # call for the parent class to␣
↪initialize
        self.W1 = nn.Parameter(nn.init.uniform_(torch.empty((784, 128)), a=-np.
↪sqrt(1/128), b=np.sqrt(1/128)))
        self.b1 = nn.Parameter(nn.init.uniform_(torch.empty((1, 128)), a=-np.
↪sqrt(1/128), b=np.sqrt(1/128)))

        self.W2 = nn.Parameter(nn.init.uniform_(torch.empty((128, 10)), a=-np.
↪sqrt(1/10), b=np.sqrt(1/10)))
        self.b2 = nn.Parameter(nn.init.uniform_(torch.empty((1, 10)), a=-np.
↪sqrt(1/10), b=np.sqrt(1/10)))

        # Make sure to add another weight and bias vector to represent the␣
↪hidden layer.

    def forward(self, x):
        # put the logic here.
        layer1_out = relu(torch.matmul(x, self.W1) + self.b1)
        layer2_out = softmax(relu(torch.matmul(layer1_out, self.W2) + self.b2))

        predictions = layer2_out
```

```
        return predictions

model = FeedForwardNet()
model.to(device)
ce_loss = nn.CrossEntropyLoss()
SGD = torch.optim.SGD(model.parameters(), lr = lr)
train_acc_list, test_acc_list = train(model, ce_loss, SGD, train_dataloader,␣
↪test_dataloader, num_epochs=num_epochs)

path_save = os.path.join(os.getcwd(), 'models')
if not os.path.exists(path_save):
    os.makedirs(path_save)

torch.save(model.state_dict(), os.path.join(path_save,␣
↪'model_{}_bs_{}_lr_{}_epoch_{}.pth'.format(str(model_name), str(bs),␣
↪str(lr), str(num_epochs))))
```

```
/tmp/ipykernel_20559/4262507057.py:20: UserWarning: Implicit dimension choice
for softmax has been deprecated. Change the call to include dim=X as an
argument.
  layer2_out = softmax(relu(torch.matmul(layer1_out, self.W2) + self.b2))
```

```
EPOCH 0. Progress: 0.0%.
Train accuracy: 0.6693333387374878. Test accuracy: 0.6650999784469604
EPOCH 10. Progress: 20.0%.
Train accuracy: 0.8807333707809448. Test accuracy: 0.8710999488830566
EPOCH 20. Progress: 40.0%.
Train accuracy: 0.8894000053405762. Test accuracy: 0.8791999816894531
EPOCH 30. Progress: 60.0%.
Train accuracy: 0.8918833136558533. Test accuracy: 0.8805999755859375
EPOCH 40. Progress: 80.0%.
Train accuracy: 0.9886333346366882. Test accuracy: 0.9739999771118164
```

1.2.3 Try training this without a non-linearity between the layers (linear activation), and then try adding a sigmoid non-linearity both before the hidden layer and after the hidden layer, recording your test accuracy results for each in a table.

[ ]:
```
lr = 0.5
num_epochs = 50
model_name = 'linear'

class FeedForwardNet(nn.Module):
    """ Simple feed forward network with one hidden layer."""
    def __init__(self): # initialize the model
        super(FeedForwardNet, self).__init__() # call for the parent class to␣
↪initialize
```

```
        self.W1 = nn.Parameter(nn.init.uniform_(torch.empty((784, 128)), a=-np.
↪sqrt(1/128), b=np.sqrt(1/128)))
        self.b1 = nn.Parameter(nn.init.uniform_(torch.empty((1, 128)), a=-np.
↪sqrt(1/128), b=np.sqrt(1/128)))

        self.W2 = nn.Parameter(nn.init.uniform_(torch.empty((128, 10)), a=-np.
↪sqrt(1/10), b=np.sqrt(1/10)))
        self.b2 = nn.Parameter(nn.init.uniform_(torch.empty((1, 10)), a=-np.
↪sqrt(1/10), b=np.sqrt(1/10)))

        # Make sure to add another weight and bias vector to represent the␣
↪hidden layer.

    def forward(self, x):
        # put the logic here.
        layer1_out = torch.matmul(x, self.W1) + self.b1
        layer2_out = softmax(torch.matmul(layer1_out, self.W2) + self.b2)

        predictions = layer2_out

        return predictions

from ps3_functions import train

model = FeedForwardNet()
model.to(device)
ce_loss = nn.CrossEntropyLoss()
SGD = torch.optim.SGD(model.parameters(), lr = lr)
train_acc_list, test_acc_list = train(model, ce_loss, SGD, train_dataloader,␣
↪test_dataloader, num_epochs=num_epochs)

path_save = os.path.join(os.getcwd(), 'models')
if not os.path.exists(path_save):
    os.makedirs(path_save)

torch.save(model, os.path.join(path_save, 'model_{}_bs_{}_lr_{}_epoch_{}.pth'.
↪format(str(model_name), str(bs), str(lr), str(num_epochs))))
```

```
/tmp/ipykernel_12537/251991051.py:20: UserWarning: Implicit dimension choice for
softmax has been deprecated. Change the call to include dim=X as an argument.
  layer2_out = softmax(torch.matmul(layer1_out, self.W2) + self.b2)
```

```
EPOCH 0. Progress: 0.0%.
Train accuracy: 0.9097499847412109. Test accuracy: 0.9124999642372131
EPOCH 10. Progress: 20.0%.
Train accuracy: 0.9343166947364807. Test accuracy: 0.9277999997138977
EPOCH 20. Progress: 40.0%.
```

```
Train accuracy: 0.9384166598320007. Test accuracy: 0.9284999966621399
EPOCH 30. Progress: 60.0%.
Train accuracy: 0.9427833557128906. Test accuracy: 0.9299999475479126
EPOCH 40. Progress: 80.0%.
Train accuracy: 0.9448000192642212. Test accuracy: 0.9304999709129333
```

[ ]:
```
lr = 0.5
num_epochs = 50
model_name = 'sigmoid'

class FeedForwardNet(nn.Module):
    """ Simple feed forward network with one hidden layer."""
    def __init__(self): # initialize the model
        super(FeedForwardNet, self).__init__() # call for the parent class to␣
 ↪initialize
        self.W1 = nn.Parameter(nn.init.uniform_(torch.empty((784, 128)), a=-np.
 ↪sqrt(1/128), b=np.sqrt(1/128)))
        self.b1 = nn.Parameter(nn.init.uniform_(torch.empty((1, 128)), a=-np.
 ↪sqrt(1/128), b=np.sqrt(1/128)))

        self.W2 = nn.Parameter(nn.init.uniform_(torch.empty((128, 10)), a=-np.
 ↪sqrt(1/10), b=np.sqrt(1/10)))
        self.b2 = nn.Parameter(nn.init.uniform_(torch.empty((1, 10)), a=-np.
 ↪sqrt(1/10), b=np.sqrt(1/10)))

        # Make sure to add another weight and bias vector to represent the␣
 ↪hidden layer.

    def forward(self, x):
        # put the logic here.
        layer1_out = sigmoid(torch.matmul(x, self.W1) + self.b1)
        layer2_out = softmax(sigmoid(torch.matmul(layer1_out, self.W2) + self.
 ↪b2))

        predictions = layer2_out

        return predictions

from ps3_functions import train

model = FeedForwardNet()
model.to(device)
ce_loss = nn.CrossEntropyLoss()
SGD = torch.optim.SGD(model.parameters(), lr = lr)
train_acc_list, test_acc_list = train(model, ce_loss, SGD, train_dataloader,␣
 ↪test_dataloader, num_epochs=num_epochs)
```

```
path_save = os.path.join(os.getcwd(), 'models')
if not os.path.exists(path_save):
    os.makedirs(path_save)

torch.save(model, os.path.join(path_save, 'model_{}_bs_{}_lr_{}_epoch_{}.pth'.
 ↪format(str(model_name), str(bs), str(lr), str(num_epochs))))
```

```
/home/xiaoranzhang/anaconda3/envs/torch_env/lib/python3.9/site-
packages/torch/nn/functional.py:1806: UserWarning: nn.functional.sigmoid is
deprecated. Use torch.sigmoid instead.
  warnings.warn("nn.functional.sigmoid is deprecated. Use torch.sigmoid
instead.")
/tmp/ipykernel_12537/1446017766.py:20: UserWarning: Implicit dimension choice
for softmax has been deprecated. Change the call to include dim=X as an
argument.
  layer2_out = softmax(sigmoid(torch.matmul(layer1_out, self.W2) + self.b2))
```

```
EPOCH 0. Progress: 0.0%.
Train accuracy: 0.7341499924659729. Test accuracy: 0.7436999678611755
EPOCH 10. Progress: 20.0%.
Train accuracy: 0.8754333257675171. Test accuracy: 0.882599949836731
EPOCH 20. Progress: 40.0%.
Train accuracy: 0.89410001039505. Test accuracy: 0.9005999565124512
EPOCH 30. Progress: 60.0%.
Train accuracy: 0.903083324432373. Test accuracy: 0.9096999764442444
EPOCH 40. Progress: 80.0%.
Train accuracy: 0.9084333181381226. Test accuracy: 0.9140999913215637
```

1.2.4 Try adjusting the learning rate (by making it smaller) if your model is not converging/improving in accuracy. You might also try increasing the number of epochs used.

[ ]:
```
lr = 1e-3
num_epochs = 50
model_name = 'learning_rate'

class FeedForwardNet(nn.Module):
    """ Simple feed forward network with one hidden layer."""
    def __init__(self): # initialize the model
        super(FeedForwardNet, self).__init__() # call for the parent class to␣
 ↪initialize
        self.W1 = nn.Parameter(nn.init.uniform_(torch.empty((784, 128)), a=-np.
 ↪sqrt(1/128), b=np.sqrt(1/128)))
        self.b1 = nn.Parameter(nn.init.uniform_(torch.empty((1, 128)), a=-np.
 ↪sqrt(1/128), b=np.sqrt(1/128)))
```

```
        self.W2 = nn.Parameter(nn.init.uniform_(torch.empty((128, 10)), a=-np.
↪sqrt(1/10), b=np.sqrt(1/10)))
        self.b2 = nn.Parameter(nn.init.uniform_(torch.empty((1, 10)), a=-np.
↪sqrt(1/10), b=np.sqrt(1/10)))

        # Make sure to add another weight and bias vector to represent the␣
↪hidden layer.

    def forward(self, x):
        # put the logic here.
        layer1_out = relu(torch.matmul(x, self.W1) + self.b1)
        layer2_out = softmax(relu(torch.matmul(layer1_out, self.W2) + self.b2))

        predictions = layer2_out

        return predictions

from ps3_functions import train

model = FeedForwardNet()
model.to(device)
ce_loss = nn.CrossEntropyLoss()
SGD = torch.optim.SGD(model.parameters(), lr = lr)
train_acc_list, test_acc_list = train(model, ce_loss, SGD, train_dataloader,␣
↪test_dataloader, num_epochs=num_epochs)

path_save = os.path.join(os.getcwd(), 'models')
if not os.path.exists(path_save):
    os.makedirs(path_save)

torch.save(model, os.path.join(path_save, 'model_{}_bs_{}_lr_{}_epoch_{}.pth'.
↪format(str(model_name), str(bs), str(lr), str(num_epochs))))
```

```
/tmp/ipykernel_12537/3423974620.py:20: UserWarning: Implicit dimension choice
for softmax has been deprecated. Change the call to include dim=X as an
argument.
  layer2_out = softmax(relu(torch.matmul(layer1_out, self.W2) + self.b2))
```

```
EPOCH 0. Progress: 0.0%.
Train accuracy: 0.11543333530426025. Test accuracy: 0.11549999564886093
EPOCH 10. Progress: 20.0%.
Train accuracy: 0.4541333317756653. Test accuracy: 0.45819997787475586
EPOCH 20. Progress: 40.0%.
Train accuracy: 0.5333666801452637. Test accuracy: 0.5324000120162964
EPOCH 30. Progress: 60.0%.
Train accuracy: 0.5476999878883362. Test accuracy: 0.5475000143051147
EPOCH 40. Progress: 80.0%.
Train accuracy: 0.5537499785423279. Test accuracy: 0.5521000027656555
```

1.2.5 Lastly, experiment with the width of the hidden layer, keeping the activation function that performs best. Remember to add these results to your table.

[ ]:
```
lr = 0.5
num_epochs = 50
hidden_width = 64
model_name = 'width_hidden'

class FeedForwardNet(nn.Module):
    """ Simple feed forward network with one hidden layer."""
    def __init__(self): # initialize the model
        super(FeedForwardNet, self).__init__() # call for the parent class to␣
↪initialize
        self.W1 = nn.Parameter(nn.init.uniform_(torch.empty((784,␣
↪hidden_width)), a=-np.sqrt(1/hidden_width), b=np.sqrt(1/hidden_width)))
        self.b1 = nn.Parameter(nn.init.uniform_(torch.empty((1, hidden_width)),␣
↪a=-np.sqrt(1/hidden_width), b=np.sqrt(1/hidden_width)))

        self.W2 = nn.Parameter(nn.init.uniform_(torch.empty((hidden_width,␣
↪10)), a=-np.sqrt(1/10), b=np.sqrt(1/10)))
        self.b2 = nn.Parameter(nn.init.uniform_(torch.empty((1, 10)), a=-np.
↪sqrt(1/10), b=np.sqrt(1/10)))

        # Make sure to add another weight and bias vector to represent the␣
↪hidden layer.

    def forward(self, x):
        # put the logic here.
        layer1_out = relu(torch.matmul(x, self.W1) + self.b1)
        layer2_out = softmax(relu(torch.matmul(layer1_out, self.W2) + self.b2))

        predictions = layer2_out

        return predictions

from ps3_functions import train

model = FeedForwardNet()
model.to(device)
ce_loss = nn.CrossEntropyLoss()
SGD = torch.optim.SGD(model.parameters(), lr = lr)
train_acc_list, test_acc_list = train(model, ce_loss, SGD, train_dataloader,␣
↪test_dataloader, num_epochs=num_epochs)

torch.save(model, os.path.join(path_save, 'model_{}_{}_bs_{}_lr_{}_epoch_{}.
↪pth'.format(str(model_name), str(hidden_width), str(bs), str(lr),␣
↪str(num_epochs))))
```

```
/tmp/ipykernel_12537/2455853343.py:21: UserWarning: Implicit dimension choice
for softmax has been deprecated. Change the call to include dim=X as an
argument.
  layer2_out = softmax(relu(torch.matmul(layer1_out, self.W2) + self.b2))
```

```
EPOCH 0. Progress: 0.0%.
Train accuracy: 0.6707500219345093. Test accuracy: 0.6680999994277954
EPOCH 10. Progress: 20.0%.
Train accuracy: 0.8770833611488342. Test accuracy: 0.8714999556541443
EPOCH 20. Progress: 40.0%.
Train accuracy: 0.9767667055130005. Test accuracy: 0.967799961566925
EPOCH 30. Progress: 60.0%.
Train accuracy: 0.9853333234786987. Test accuracy: 0.972000002861023
EPOCH 40. Progress: 80.0%.
Train accuracy: 0.9880666732788086. Test accuracy: 0.9723999500274658
```

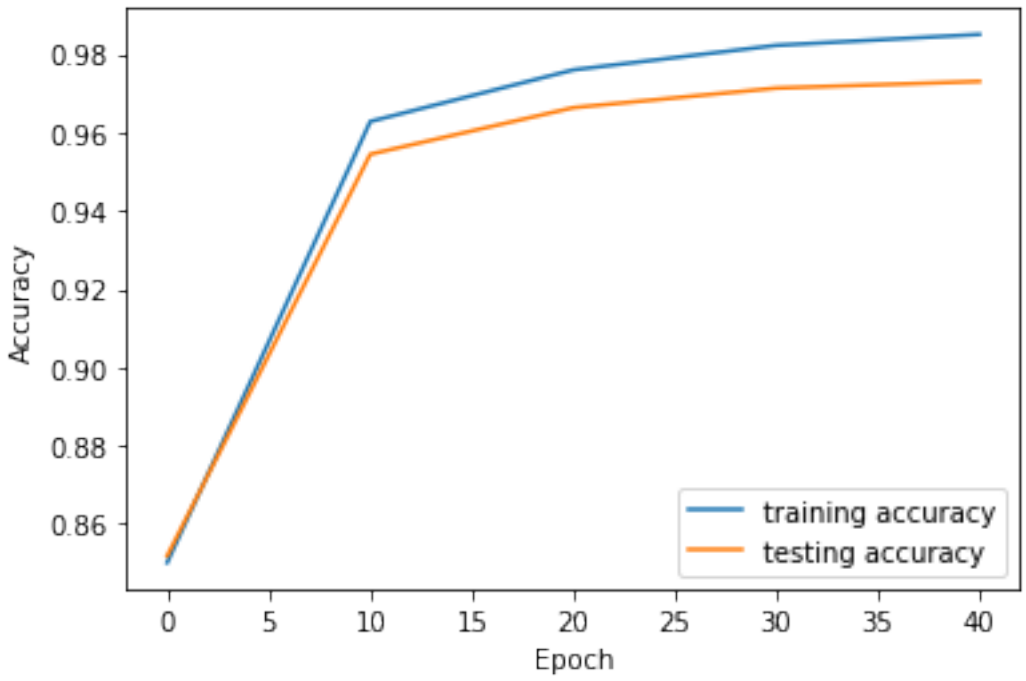
1.2.6 Question 2.2.1. What percentage classification accuracy does this network achieve?

Answer: The network is able to achieve about 0.96 testing accuracy.

1.2.7 Question 2.2.2. Create a plot of the training and test error vs the number of iterations. How many iterations are sufficient to reach good performance?

Answer: From the following figure, about 20 iterations is enough to reach good performance (using relu, lr=0.5 configuration).

[ ]:
```
import matplotlib.pyplot as plt

def func_plotAcc(train_acc_list, test_acc_list, plot_name):
    train_acc_list_py, test_acc_list_py = [], []

    for train_acc, test_acc in zip(train_acc_list, test_acc_list):
        train_acc.to(torch.device('cpu')), test_acc.to(torch.device('cpu'))
        train_acc_list_py.append(train_acc.cpu().numpy()), test_acc_list_py.
 ↪append(test_acc.cpu().numpy())
    epochs = np.arange(0, 50, 10)
    fig, ax = plt.subplots(1,1)
    ax.plot(epochs, train_acc_list_py, label='training accuracy')
    ax.plot(epochs, test_acc_list_py, label='testing accuracy')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Accuracy')
    ax.legend()
    plt.show()

    fig.savefig(os.path.join(path_figures, plot_name))

func_plotAcc(train_acc_list, test_acc_list, 'section2-1.pdf')
```



1.2.8 Question 2.2.3. Print the confusion matrix showing which digits were misclassified, and what they were misclassified as. What numbers are frequently confused with one another by your model?

Answer: The confusion matrix printed below indicates that generally the model correctly classifies the digits. The biggest confusion comes from digit 4 with digit 9.

[ ]:
```
from sklearn.metrics import confusion_matrix
output_list, label_list = [], []
with torch.no_grad():
    for data, targets in test_dataloader:
        data = data.to('cuda')
        targets = targets.to('cuda')
        test_input = data.view(-1, 784)
        test_output_onehot = model(test_input)
        test_output = torch.argmax(test_output_onehot, dim=1)
        output_list.extend(test_output.cpu().numpy()), label_list.
 ↪extend(targets.cpu().numpy())

confusion_matrix(label_list, output_list)
```

```
/tmp/ipykernel_12537/251991051.py:20: UserWarning: Implicit dimension choice for
softmax has been deprecated. Change the call to include dim=X as an argument.
  layer2_out = softmax(torch.matmul(layer1_out, self.W2) + self.b2)
```

[ ]:
```
array([[ 955,    0,    1,    1,    1,    6,   11,    2,    3,    0],
       [   0, 1113,    5,    3,    1,    4,    4,    2,    3,    0],
       [   9,    7,  926,   11,   16,    7,    8,   13,   29,    6],
       [   2,    0,   23,  917,    1,   28,    3,    9,   19,    8],
       [   1,    5,    4,    1,  930,    0,   10,    3,    3,   25],
       [   7,    2,    3,   25,   15,  791,   13,    3,   26,    7],
       [  10,    3,    6,    2,    7,   15,  912,    2,    1,    0],
       [   3,    7,   24,    4,   10,    1,    0,  950,    2,   27],
       [   5,    6,    3,   19,    9,   35,    9,   10,  873,    5],
       [  12,    7,    1,    8,   26,    8,    0,   16,    9,  922]])
```

1.2.9 Question 2.2.4. Experiment with the learning rate, optimizer and activation function of your network. Report the best accuracy and briefly describe the training scheme that reached this accuracy.

Answer: After experimenting with a series models shown above, the best testing accuracy (0.96) is achieved when lr = 0.5 with SGD and relu activation function.

2 3 Autoencoder

2.1 3.1 MNIST

2.1.1 section 1-3, please check the relevant functions

2.1.2 Adapt your training function for the autoencoder. Use the same batch size and number of epochs (128 and 100), but use the ADAM optimizer instead of Gradient Descent. Use Mean Squared Error for your reconstruction loss.

[ ]:
```
bs = 128
```

[ ]:
```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.enc_lin1 = nn.Linear(784, 1000)
        self.enc_lin2 = nn.Linear(1000, 500)
        self.enc_lin3 = nn.Linear(500, 250)
        self.enc_lin4 = nn.Linear(250, 2)

        self.dec_lin1 = nn.Linear(2, 250)
        self.dec_lin2 = nn.Linear(250, 500)
        self.dec_lin3 = nn.Linear(500, 1000)
        self.dec_lin4 = nn.Linear(1000, 784)
        # define additional layers here


    def encode(self, x):
        x = tanh(self.enc_lin1(x))
        x = tanh(self.enc_lin2(x))
```

```
        x = tanh(self.enc_lin3(x))
        x = sigmoid(self.enc_lin4(x))

        # ... additional layers, plus possible nonlinearities.
        return x

    def decode(self, z):
        # ditto, but in reverse
        z = tanh(self.dec_lin1(z))
        z = tanh(self.dec_lin2(z))
        z = tanh(self.dec_lin3(z))
        z = tanh(self.dec_lin4(z))

        return z

    def forward(self, x):
        z = self.encode(x)
        out = self.decode(z)
        return out

def train_ae(model, loss_fn, optimizer, train_loader, test_loader, num_epochs):
    # num_epochs = 100 # obviously, this is too many. I don't know what this␣
 ↪author was thinking.

    train_loss_list, test_loss_list = [], []

    for epoch in range(num_epochs):
        # loop through each data point in the training set
        for data, targets in train_loader:
            optimizer.zero_grad()

            # run the model on the data
            data = data.to('cuda')
            targets = targets.to('cuda')
            model_input = data.view(-1, 784)

            # TODO: Turn the 28 by 28 image tensors into a 784 dimensional␣
 ↪tensor.
            out = model(model_input)

            # Calculate the loss
            loss = loss_fn(out,model_input)

            # Find the gradients of our loss via backpropogation
            loss.backward()

            # Adjust accordingly with the optimizer
```

```
            optimizer.step()


        # Give status reports every 100 epochs
        if epoch % 10==0:
            train_loss, test_loss = evaluate_ae(model, train_loader, loss_fn),␣
 ↪evaluate_ae (model, test_loader, loss_fn)
            train_loss_list.append(train_loss), test_loss_list.append(test_loss)
            print(f" EPOCH {epoch}. Progress: {epoch/num_epochs*100}%. ")
            print(f" Train loss: {train_loss}. Test loss: {test_loss}") #TODO:␣
 ↪implement the evaluate function to provide performance statistics during␣
 ↪training.

    return train_loss_list, test_loss_list

def evaluate_ae(model, evaluation_set, loss_fn):
    with torch.no_grad(): # this disables backpropogation, which makes the␣
 ↪model run much more quickly.
        # TODO: Fill in the rest of the evaluation function.
        loss_sum = 0

        for data, targets in evaluation_set:
            data = data.to('cuda')
            # targets = targets.to('cuda')
            test_input = data.view(-1, 784)
            test_output = model(test_input)

            loss_sum += loss_fn(test_input, test_output)

        loss = loss_sum / len(evaluation_set.dataset)

    return loss
```

[ ]:
```
lr = 0.003
num_epochs = 10
model_name = 'autoencoder'

model_ae = Autoencoder()
model_ae.to(device)
mse_loss = nn.MSELoss()
optimizer_ae = torch.optim.Adam(model_ae.parameters(), lr=lr)
train_acc_list, test_acc_list = train_ae(model_ae, mse_loss, optimizer_ae,␣
 ↪train_dataloader, test_dataloader, num_epochs)

torch.save(model, os.path.join(path_save, 'model_{}_bs_{}_lr_{}_epoch_{}.pth'.
 ↪format(str(model_name), str(bs), str(lr), str(num_epochs))))
```

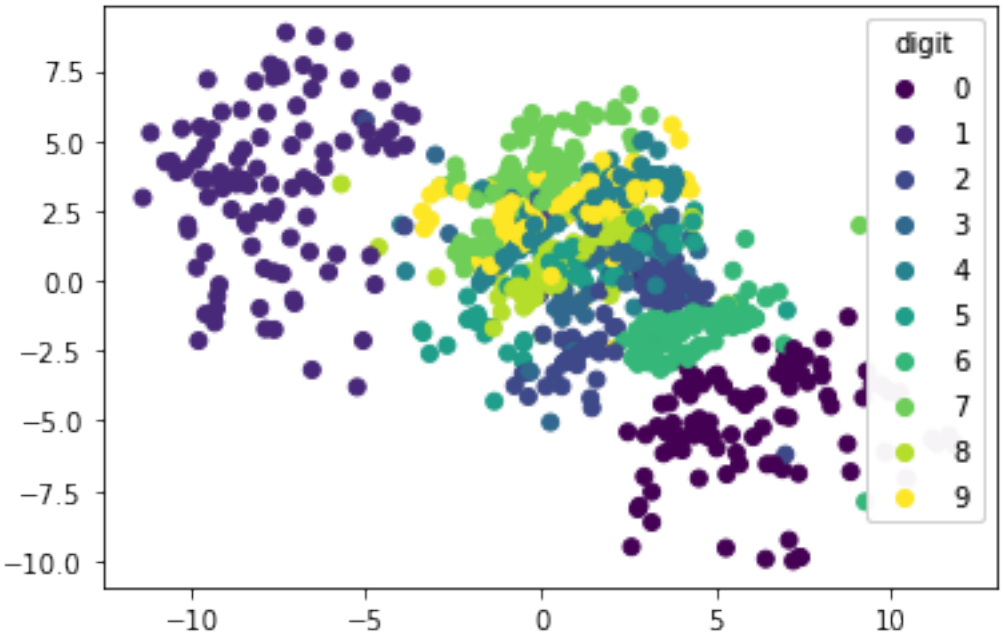
```
/home/xiaoranzhang/anaconda3/envs/torch_env/lib/python3.9/site-
packages/torch/nn/functional.py:1795: UserWarning: nn.functional.tanh is
deprecated. Use torch.tanh instead.
  warnings.warn("nn.functional.tanh is deprecated. Use torch.tanh instead.")
/home/xiaoranzhang/anaconda3/envs/torch_env/lib/python3.9/site-
packages/torch/nn/functional.py:1806: UserWarning: nn.functional.sigmoid is
deprecated. Use torch.sigmoid instead.
  warnings.warn("nn.functional.sigmoid is deprecated. Use torch.sigmoid
instead.")

 EPOCH 0. Progress: 0.0%.
 Train loss: 0.0003743564593605697. Test loss: 0.00037780642742291093
```

2.1.3 After training your model, plot the 2 dimensional embeddings of 1000 digits, colored by the image labels.

[ ]:
```
data_loader_1000_digits = DataLoader(mnist_train, batch_size=1000, shuffle=True)
for data, target in data_loader_1000_digits:
    with torch.no_grad():
        data = data.to(device)
        model_input = data.view(-1, 784)
        embedds = model_ae.encode(model_input).cpu().numpy()
        preds = model_ae(model_input).cpu().numpy()
        labels = target.cpu().numpy()
        inputs = model_input.cpu().numpy()
        break # just load one batch
```

```
/home/xiaoranzhang/anaconda3/envs/torch_env/lib/python3.9/site-
packages/torch/nn/functional.py:1795: UserWarning: nn.functional.tanh is
deprecated. Use torch.tanh instead.
  warnings.warn("nn.functional.tanh is deprecated. Use torch.tanh instead.")
/home/xiaoranzhang/anaconda3/envs/torch_env/lib/python3.9/site-
packages/torch/nn/functional.py:1806: UserWarning: nn.functional.sigmoid is
deprecated. Use torch.sigmoid instead.
  warnings.warn("nn.functional.sigmoid is deprecated. Use torch.sigmoid
instead.")
```

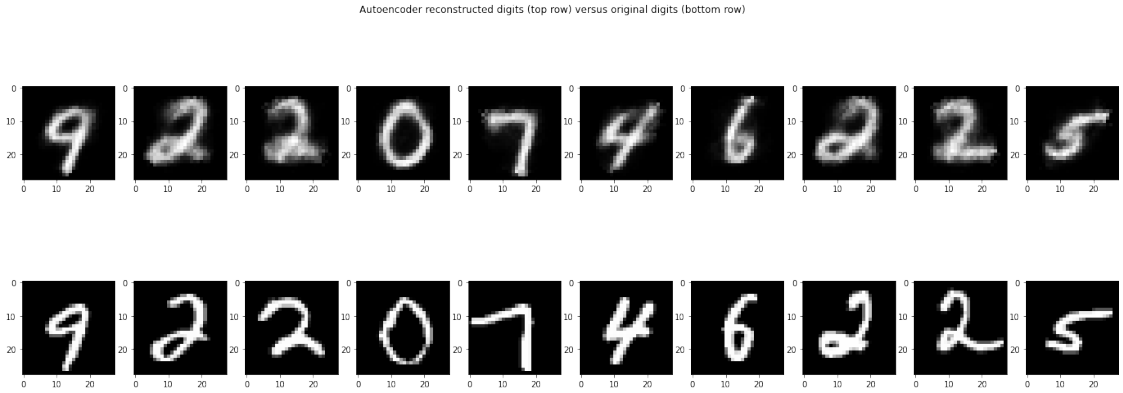
[ ]:
```
fig = plt.figure()
ax = fig.add_subplot()
scatter = ax.scatter(embedds[:,0], embedds[:,1], c=labels)
legend = ax.legend(*scatter.legend_elements(), title='digit')
ax.add_artist(legend)
fig.savefig(os.path.join(path_figures,'Section_3_1.pdf'))
plt.show()
```



2.1.4 Produce side-by-side plots of one original and reconstructed sample of each digit (0 - 9). You can use the save_image function from torchvision.utils.

[ ]:
```
# find a sample idx with each digit
sample_idx_list, digit_list = [], []
for idx in range(labels.shape[0]):
    for digit in range(10):
        if labels[idx] == digit:
            if digit not in digit_list:
                sample_idx_list.append(idx)
                digit_list.append(digit)
```

[ ]:
```
fig, axes = plt.subplots(2,10)
fig.set_size_inches(24,8)

for i in range(10):
    preds_digit = preds[sample_idx_list[i]]
    input_digit = inputs[sample_idx_list[i]]
    axes[0][i].imshow(np.reshape(preds_digit, (28,28)), cmap='gray')
    axes[1][i].imshow(np.reshape(input_digit, (28,28)), cmap='gray')
fig.suptitle('Autoencoder reconstructed digits (top row) versus original digits␣
 ↪(bottom row)')
fig.savefig(os.path.join(path_figures,'Section_3_2.pdf'))
plt.show()
```

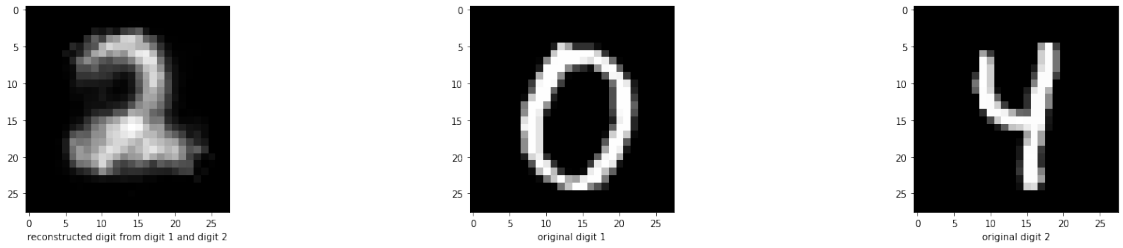


2.1.5 Now for something fun: locate the embeddings of two distinct images, and interpolate between them to produce some intermediate point in the latent space. Visualize this point in the 2D embedding. Then, run your decoder on this fabricated "embedding" to see if it the output looks anything like a handwritten digit. You might try interpolating between and within several different classes.

[ ]:
```
digit_idx_1 = 0
digit_idx_2 = 1

for data, target in data_loader_1000_digits:
    with torch.no_grad():
        data = data.to(device)
        model_input = data.view(-1, 784)
        embedds_1 = model_ae.encode(model_input[digit_list[digit_idx_1],:]).
↪cpu().numpy()
        embedds_2 = model_ae.encode(model_input[digit_list[digit_idx_2],:]).
↪cpu().numpy()
        embedds_frab = (embedds_1 + embedds_2)/2
        preds = model_ae.decode(torch.from_numpy(embedds_frab).to(device)).
↪cpu().numpy()
        break # just load one batch
```

```
/home/xiaoranzhang/anaconda3/envs/torch_env/lib/python3.9/site-
packages/torch/nn/functional.py:1795: UserWarning: nn.functional.tanh is
deprecated. Use torch.tanh instead.
  warnings.warn("nn.functional.tanh is deprecated. Use torch.tanh instead.")
/home/xiaoranzhang/anaconda3/envs/torch_env/lib/python3.9/site-
packages/torch/nn/functional.py:1806: UserWarning: nn.functional.sigmoid is
deprecated. Use torch.sigmoid instead.
  warnings.warn("nn.functional.sigmoid is deprecated. Use torch.sigmoid
instead.")
```

[ ]:
```
fig, axes = plt.subplots(1,3)
fig.set_size_inches(24,4)

preds_digit = preds
digit_1 = model_input[digit_list[digit_idx_1],:].cpu().numpy()
digit_2 = model_input[digit_list[digit_idx_2],:].cpu().numpy()

axes[0].imshow(np.reshape(preds_digit, (28,28)), cmap='gray')
axes[0].set_xlabel('reconstructed digit from digit 1 and digit 2')
axes[1].imshow(np.reshape(digit_1, (28,28)), cmap='gray')
axes[1].set_xlabel('original digit 1')
axes[2].imshow(np.reshape(digit_2, (28,28)), cmap='gray')
axes[2].set_xlabel('original digit 2')

fig.savefig(os.path.join(path_figures,'Section_3_3.pdf'))
plt.show()
```



2.1.6 Question 3.1.1. Do the colors easily separate, or are they all clumped together? Which numbers are frequently embedded close together, and what does this mean?

Answer: The colors are generally separable. The embeddings of 5 and 6 seem to clump together and this means that the handwritten digits of 5 and 6 are harder to separate.

2.1.7 Question 3.1.2. How realistic were the images you generated by interpolating between points in the latent space? Can you think of a better way to generate images with an autoencoder?

Answer: The image I interpolated looks quite realistic to digit 2. A better way to generate images is to sample the embeddings based on some continuous distribution such as Gaussian, which will be similar to the idea of variational autoencoder.

2.2 3.2 Biological Data: Retinal Bipolar Dataset

[ ]:
```
import pickle

path_RB_root = os.getcwd().replace('code','data')
```

```
with open(os.path.join(path_RB_root, 'retinal-bipolar-data.pickle'), 'rb') as f:
    RB_data = pickle.load(f)

with open(os.path.join(path_RB_root, 'retinal-bipolar-metadata.pickle'), 'rb')␣
 ↪as f:
    RB_meta_data = pickle.load(f)
```

2.2.1 Create a training and testing split of the data (choose an 80-20 split) and generate minibatches from your training data (shuffling the order of the points between epochs).

[ ]:
```
n_split = 0.8
n_feature = 784
bs = 128

totalNumOfSample = int(n_split*RB_data.shape[0])
RB_data_train = torch.Tensor(RB_data.values[:totalNumOfSample, :n_feature].
 ↪astype(np.float32))
RB_label_train = torch.Tensor(RB_meta_data['CLUSTER'].values[:totalNumOfSample].
 ↪astype(np.float32))
RB_data_test = torch.Tensor(RB_data.values[totalNumOfSample:, :n_feature].
 ↪astype(np.float32))
RB_label_test = torch.Tensor(RB_meta_data['CLUSTER'].values[totalNumOfSample:].
 ↪astype(np.float32))

RB_train_dataset = TensorDataset(RB_data_train, RB_label_train)
RB_test_dataset = TensorDataset(RB_data_test, RB_label_test)

RB_train_dataloader = DataLoader(RB_train_dataset, batch_size=bs, shuffle=True)
RB_test_dataloader = DataLoader(RB_test_dataset, batch_size=RB_data_test.cpu().
 ↪numpy().shape[0], shuffle=True)
```

[ ]:
```
print(RB_data_train.shape)
print(RB_label_train.shape)
print(RB_data_test.shape)
print(RB_label_test.shape)
```

```
torch.Size([17241, 784])
torch.Size([17241])
torch.Size([4311, 784])
torch.Size([4311])
```

2.2.2 Starting with the same autoencoder architecture as the last section, change the last layer's activation to a linear (instead of a sigmoid activation)

[ ]:
```
class Autoencoder_RB(Autoencoder):
    def encode(self, x):
        x = tanh(self.enc_lin1(x))
        x = tanh(self.enc_lin2(x))
        x = tanh(self.enc_lin3(x))
        x = self.enc_lin4(x)

        # ... additional layers, plus possible nonlinearities.
        return x
```

[ ]:
```
lr = 0.003
num_epochs = 100
model_name = 'autoencoder_RB'

model_ae = Autoencoder_RB()
model_ae.to(device)
mse_loss = nn.MSELoss()
optimizer_ae = torch.optim.Adam(model_ae.parameters(), lr=lr)
train_acc_list, test_acc_list = train_ae(model_ae, mse_loss, optimizer_ae,␣
↪RB_train_dataloader, RB_test_dataloader, num_epochs)

torch.save(model_ae, os.path.join(path_save, 'model_{}_bs_{}_lr_{}_epoch_{}.
↪pth'.format(str(model_name), str(bs), str(lr), str(num_epochs))))
```

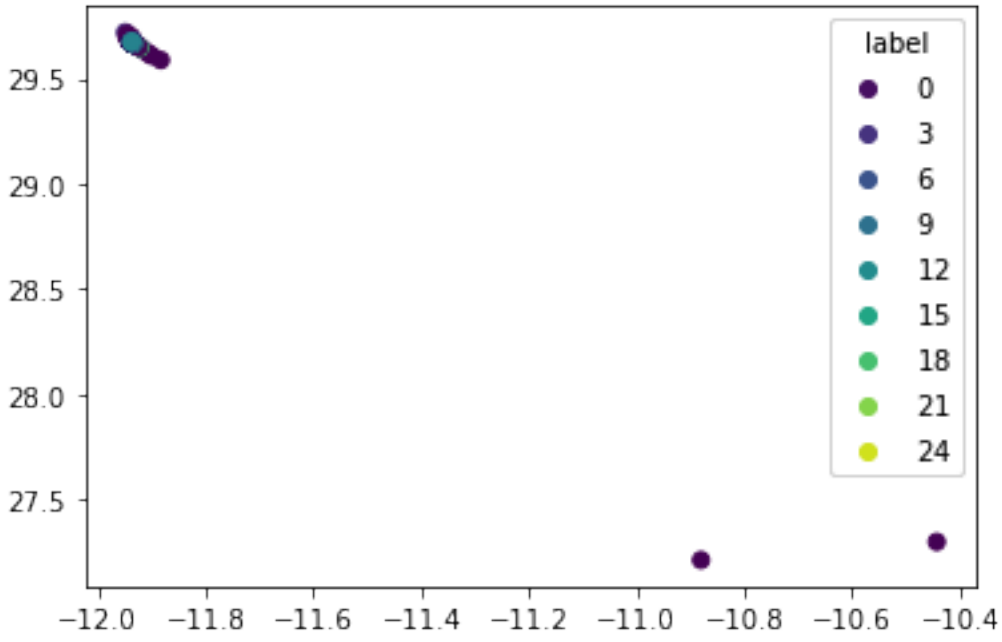
```
/home/xiaoranzhang/anaconda3/envs/torch_env/lib/python3.9/site-
packages/torch/nn/functional.py:1795: UserWarning: nn.functional.tanh is
deprecated. Use torch.tanh instead.
  warnings.warn("nn.functional.tanh is deprecated. Use torch.tanh instead.")
```

```
EPOCH 0. Progress: 0.0%.
Train loss: 0.0017081410624086857. Test loss: 4.702854130300693e-05
EPOCH 10. Progress: 10.0%.
Train loss: 0.0017036006320267916. Test loss: 4.686633837991394e-05
EPOCH 20. Progress: 20.0%.
Train loss: 0.0017036466160789132. Test loss: 4.673840157920495e-05
EPOCH 30. Progress: 30.0%.
Train loss: 0.009501755237579346. Test loss: 0.00027749643777497113
EPOCH 40. Progress: 40.0%.
Train loss: 0.009502754546701908. Test loss: 0.00027753328322432935
EPOCH 50. Progress: 50.0%.
Train loss: 0.009498313069343567. Test loss: 0.00027740810764953494
EPOCH 60. Progress: 60.0%.
Train loss: 0.009491397067904472. Test loss: 0.0002772490552160889
EPOCH 70. Progress: 70.0%.
Train loss: 0.009491091594099998. Test loss: 0.0002772490552160889
```

```
EPOCH 80. Progress: 80.0%.
Train loss: 0.00949031487107277. Test loss: 0.0002772211446426809
EPOCH 90. Progress: 90.0%.
Train loss: 0.009490543976426125. Test loss: 0.0002772211446426809
```

2.2.3 After training your model, plot the 2-dimensional embedding of the test set. Color this with the ground truth cluster labels.

[ ]:
```
for data, target in RB_test_dataloader:
    with torch.no_grad():
        data = data.to(device)
        model_input = data.view(-1, 784)
        embedds = model_ae.encode(model_input).cpu().numpy()
        preds = model_ae(model_input).cpu().numpy()
        labels = target.cpu().numpy()
        inputs = model_input.cpu().numpy()
        break # just load one batch

fig = plt.figure()
ax = fig.add_subplot()
scatter = ax.scatter(embedds[:,0], embedds[:,1], c=labels)
legend = ax.legend(*scatter.legend_elements(), title='label')
ax.add_artist(legend)
fig.savefig(os.path.join(path_figures,'Section_3_4.pdf'))
plt.show()
```



2.2.4 Question 3.2.1. How many clusters are visible in the embedding? Do they correspond to the cluster labels?

Answer: It seems that only three clusters are forming in the embedding. The variance in y is small and numbers found in the sample value of x ends up in one of the three bins.

3 4 Generative models

3.1 4.1 The Variational Autoencoder

3.1.1 Section 1 please check vae.py

3.1.2 Train your VAE on MNIST. How well does it perform on the test set relative to your vanilla autoen- coder? (You can run the VAE by executing python vae.py --arguments here).

[ ]:
```
path_results = os.path.join(os.getcwd(),'results')
if not os.path.exists(path_results):
    os.makedirs(path_results)
!python vae.py
```

```
Train Epoch: 1 [0/60000 (0%)]   Loss: 550.513916
Train Epoch: 1 [1280/60000 (2%)]        Loss: 310.610535
Train Epoch: 1 [2560/60000 (4%)]        Loss: 240.650162
Train Epoch: 1 [3840/60000 (6%)]        Loss: 219.223282
Train Epoch: 1 [5120/60000 (9%)]        Loss: 215.092834
Train Epoch: 1 [6400/60000 (11%)]       Loss: 208.450150
Train Epoch: 1 [7680/60000 (13%)]       Loss: 203.374268
Train Epoch: 1 [8960/60000 (15%)]       Loss: 193.761398
Train Epoch: 1 [10240/60000 (17%)]      Loss: 194.701691
Train Epoch: 1 [11520/60000 (19%)]      Loss: 192.943787
Train Epoch: 1 [12800/60000 (21%)]      Loss: 179.935974
Train Epoch: 1 [14080/60000 (23%)]      Loss: 174.059174
Train Epoch: 1 [15360/60000 (26%)]      Loss: 182.615799
Train Epoch: 1 [16640/60000 (28%)]      Loss: 168.166245
Train Epoch: 1 [17920/60000 (30%)]      Loss: 166.474197
Train Epoch: 1 [19200/60000 (32%)]      Loss: 161.467590
Train Epoch: 1 [20480/60000 (34%)]      Loss: 162.560577
Train Epoch: 1 [21760/60000 (36%)]      Loss: 151.491119
Train Epoch: 1 [23040/60000 (38%)]      Loss: 157.329208
Train Epoch: 1 [24320/60000 (41%)]      Loss: 154.508270
Train Epoch: 1 [25600/60000 (43%)]      Loss: 155.947128
Train Epoch: 1 [26880/60000 (45%)]      Loss: 150.704407
Train Epoch: 1 [28160/60000 (47%)]      Loss: 153.140961
Train Epoch: 1 [29440/60000 (49%)]      Loss: 146.401535
Train Epoch: 1 [30720/60000 (51%)]      Loss: 141.688751
Train Epoch: 1 [32000/60000 (53%)]      Loss: 144.736603
Train Epoch: 1 [33280/60000 (55%)]      Loss: 144.681290
Train Epoch: 1 [34560/60000 (58%)]      Loss: 145.765518
```

```
Train Epoch: 1 [35840/60000 (60%)]      Loss: 137.349380
Train Epoch: 1 [37120/60000 (62%)]      Loss: 142.360214
Train Epoch: 1 [38400/60000 (64%)]      Loss: 139.197067
Train Epoch: 1 [39680/60000 (66%)]      Loss: 135.043091
Train Epoch: 1 [40960/60000 (68%)]      Loss: 141.403214
Train Epoch: 1 [42240/60000 (70%)]      Loss: 141.775909
Train Epoch: 1 [43520/60000 (72%)]      Loss: 140.087769
Train Epoch: 1 [44800/60000 (75%)]      Loss: 135.145599
Train Epoch: 1 [46080/60000 (77%)]      Loss: 140.357910
Train Epoch: 1 [47360/60000 (79%)]      Loss: 132.954529
Train Epoch: 1 [48640/60000 (81%)]      Loss: 131.553711
Train Epoch: 1 [49920/60000 (83%)]      Loss: 130.639999
Train Epoch: 1 [51200/60000 (85%)]      Loss: 131.357391
Train Epoch: 1 [52480/60000 (87%)]      Loss: 131.015793
Train Epoch: 1 [53760/60000 (90%)]      Loss: 132.620331
Train Epoch: 1 [55040/60000 (92%)]      Loss: 126.406624
Train Epoch: 1 [56320/60000 (94%)]      Loss: 137.493317
Train Epoch: 1 [57600/60000 (96%)]      Loss: 129.700119
Train Epoch: 1 [58880/60000 (98%)]      Loss: 132.308182
====> Epoch: 1 Average loss: 164.3932
====> Test set loss: 127.1262
Train Epoch: 2 [0/60000 (0%)]   Loss: 125.535873
Train Epoch: 2 [1280/60000 (2%)]        Loss: 131.477020
Train Epoch: 2 [2560/60000 (4%)]        Loss: 128.273682
Train Epoch: 2 [3840/60000 (6%)]        Loss: 129.233337
Train Epoch: 2 [5120/60000 (9%)]        Loss: 127.049393
Train Epoch: 2 [6400/60000 (11%)]       Loss: 124.260353
Train Epoch: 2 [7680/60000 (13%)]       Loss: 125.568207
Train Epoch: 2 [8960/60000 (15%)]       Loss: 131.587036
Train Epoch: 2 [10240/60000 (17%)]      Loss: 124.160423
Train Epoch: 2 [11520/60000 (19%)]      Loss: 129.391907
Train Epoch: 2 [12800/60000 (21%)]      Loss: 124.939484
Train Epoch: 2 [14080/60000 (23%)]      Loss: 123.899483
Train Epoch: 2 [15360/60000 (26%)]      Loss: 129.144165
Train Epoch: 2 [16640/60000 (28%)]      Loss: 117.747879
Train Epoch: 2 [17920/60000 (30%)]      Loss: 128.577728
Train Epoch: 2 [19200/60000 (32%)]      Loss: 126.573441
Train Epoch: 2 [20480/60000 (34%)]      Loss: 121.923729
Train Epoch: 2 [21760/60000 (36%)]      Loss: 121.998512
Train Epoch: 2 [23040/60000 (38%)]      Loss: 123.425346
Train Epoch: 2 [24320/60000 (41%)]      Loss: 126.384003
Train Epoch: 2 [25600/60000 (43%)]      Loss: 123.315178
Train Epoch: 2 [26880/60000 (45%)]      Loss: 122.613266
Train Epoch: 2 [28160/60000 (47%)]      Loss: 120.674988
Train Epoch: 2 [29440/60000 (49%)]      Loss: 121.835434
Train Epoch: 2 [30720/60000 (51%)]      Loss: 121.354088
Train Epoch: 2 [32000/60000 (53%)]      Loss: 117.470314
Train Epoch: 2 [33280/60000 (55%)]      Loss: 114.733414
```

```
Train Epoch: 2 [34560/60000 (58%)]      Loss: 128.560532
Train Epoch: 2 [35840/60000 (60%)]      Loss: 118.608986
Train Epoch: 2 [37120/60000 (62%)]      Loss: 119.012131
Train Epoch: 2 [38400/60000 (64%)]      Loss: 118.315994
Train Epoch: 2 [39680/60000 (66%)]      Loss: 118.106857
Train Epoch: 2 [40960/60000 (68%)]      Loss: 118.166054
Train Epoch: 2 [42240/60000 (70%)]      Loss: 118.680847
Train Epoch: 2 [43520/60000 (72%)]      Loss: 116.305542
Train Epoch: 2 [44800/60000 (75%)]      Loss: 114.640320
Train Epoch: 2 [46080/60000 (77%)]      Loss: 117.191109
Train Epoch: 2 [47360/60000 (79%)]      Loss: 118.408707
Train Epoch: 2 [48640/60000 (81%)]      Loss: 119.088081
Train Epoch: 2 [49920/60000 (83%)]      Loss: 123.146927
Train Epoch: 2 [51200/60000 (85%)]      Loss: 117.070839
Train Epoch: 2 [52480/60000 (87%)]      Loss: 115.501419
Train Epoch: 2 [53760/60000 (90%)]      Loss: 119.065201
Train Epoch: 2 [55040/60000 (92%)]      Loss: 116.253250
Train Epoch: 2 [56320/60000 (94%)]      Loss: 116.849663
Train Epoch: 2 [57600/60000 (96%)]      Loss: 119.891922
Train Epoch: 2 [58880/60000 (98%)]      Loss: 116.468521
====> Epoch: 2 Average loss: 121.4586
====> Test set loss: 116.0417
Train Epoch: 3 [0/60000 (0%)]   Loss: 120.296066
Train Epoch: 3 [1280/60000 (2%)]        Loss: 120.106651
Train Epoch: 3 [2560/60000 (4%)]        Loss: 114.243118
Train Epoch: 3 [3840/60000 (6%)]        Loss: 114.746696
Train Epoch: 3 [5120/60000 (9%)]        Loss: 116.854485
Train Epoch: 3 [6400/60000 (11%)]       Loss: 110.232544
Train Epoch: 3 [7680/60000 (13%)]       Loss: 116.535187
Train Epoch: 3 [8960/60000 (15%)]       Loss: 115.031769
Train Epoch: 3 [10240/60000 (17%)]      Loss: 116.219406
Train Epoch: 3 [11520/60000 (19%)]      Loss: 113.384880
Train Epoch: 3 [12800/60000 (21%)]      Loss: 113.316544
Train Epoch: 3 [14080/60000 (23%)]      Loss: 116.036942
Train Epoch: 3 [15360/60000 (26%)]      Loss: 112.938385
Train Epoch: 3 [16640/60000 (28%)]      Loss: 116.452530
Train Epoch: 3 [17920/60000 (30%)]      Loss: 112.208824
Train Epoch: 3 [19200/60000 (32%)]      Loss: 115.711273
Train Epoch: 3 [20480/60000 (34%)]      Loss: 118.686096
Train Epoch: 3 [21760/60000 (36%)]      Loss: 117.976364
Train Epoch: 3 [23040/60000 (38%)]      Loss: 113.902908
Train Epoch: 3 [24320/60000 (41%)]      Loss: 120.138832
Train Epoch: 3 [25600/60000 (43%)]      Loss: 114.611214
Train Epoch: 3 [26880/60000 (45%)]      Loss: 115.597183
Train Epoch: 3 [28160/60000 (47%)]      Loss: 114.485115
Train Epoch: 3 [29440/60000 (49%)]      Loss: 115.054443
Train Epoch: 3 [30720/60000 (51%)]      Loss: 118.494202
Train Epoch: 3 [32000/60000 (53%)]      Loss: 114.693443
```

```
Train Epoch: 3 [33280/60000 (55%)]      Loss: 115.068649
Train Epoch: 3 [34560/60000 (58%)]      Loss: 115.588120
Train Epoch: 3 [35840/60000 (60%)]      Loss: 116.201035
Train Epoch: 3 [37120/60000 (62%)]      Loss: 116.104370
Train Epoch: 3 [38400/60000 (64%)]      Loss: 115.169296
Train Epoch: 3 [39680/60000 (66%)]      Loss: 115.497551
Train Epoch: 3 [40960/60000 (68%)]      Loss: 116.682724
Train Epoch: 3 [42240/60000 (70%)]      Loss: 110.929100
Train Epoch: 3 [43520/60000 (72%)]      Loss: 111.843788
Train Epoch: 3 [44800/60000 (75%)]      Loss: 111.168198
Train Epoch: 3 [46080/60000 (77%)]      Loss: 112.718018
Train Epoch: 3 [47360/60000 (79%)]      Loss: 117.361702
Train Epoch: 3 [48640/60000 (81%)]      Loss: 114.392502
Train Epoch: 3 [49920/60000 (83%)]      Loss: 114.049614
Train Epoch: 3 [51200/60000 (85%)]      Loss: 113.015976
Train Epoch: 3 [52480/60000 (87%)]      Loss: 113.492493
Train Epoch: 3 [53760/60000 (90%)]      Loss: 110.972893
Train Epoch: 3 [55040/60000 (92%)]      Loss: 115.558853
Train Epoch: 3 [56320/60000 (94%)]      Loss: 112.501053
Train Epoch: 3 [57600/60000 (96%)]      Loss: 114.945267
Train Epoch: 3 [58880/60000 (98%)]      Loss: 109.844193
====> Epoch: 3 Average loss: 114.4005
====> Test set loss: 111.5367
Train Epoch: 4 [0/60000 (0%)]   Loss: 113.615578
Train Epoch: 4 [1280/60000 (2%)]        Loss: 117.178650
Train Epoch: 4 [2560/60000 (4%)]        Loss: 114.397842
Train Epoch: 4 [3840/60000 (6%)]        Loss: 112.539162
Train Epoch: 4 [5120/60000 (9%)]        Loss: 114.348732
Train Epoch: 4 [6400/60000 (11%)]       Loss: 113.261871
Train Epoch: 4 [7680/60000 (13%)]       Loss: 113.401688
Train Epoch: 4 [8960/60000 (15%)]       Loss: 112.113800
Train Epoch: 4 [10240/60000 (17%)]      Loss: 111.566582
Train Epoch: 4 [11520/60000 (19%)]      Loss: 108.375977
Train Epoch: 4 [12800/60000 (21%)]      Loss: 112.694687
Train Epoch: 4 [14080/60000 (23%)]      Loss: 111.022934
Train Epoch: 4 [15360/60000 (26%)]      Loss: 112.899078
Train Epoch: 4 [16640/60000 (28%)]      Loss: 109.182274
Train Epoch: 4 [17920/60000 (30%)]      Loss: 112.508850
Train Epoch: 4 [19200/60000 (32%)]      Loss: 107.165749
Train Epoch: 4 [20480/60000 (34%)]      Loss: 106.866333
Train Epoch: 4 [21760/60000 (36%)]      Loss: 112.705818
Train Epoch: 4 [23040/60000 (38%)]      Loss: 110.920410
Train Epoch: 4 [24320/60000 (41%)]      Loss: 110.755798
Train Epoch: 4 [25600/60000 (43%)]      Loss: 114.568665
Train Epoch: 4 [26880/60000 (45%)]      Loss: 115.242157
Train Epoch: 4 [28160/60000 (47%)]      Loss: 111.938110
Train Epoch: 4 [29440/60000 (49%)]      Loss: 110.974945
Train Epoch: 4 [30720/60000 (51%)]      Loss: 108.887085
```

```
Train Epoch: 4 [32000/60000 (53%)]      Loss: 110.181458
Train Epoch: 4 [33280/60000 (55%)]      Loss: 108.104034
Train Epoch: 4 [34560/60000 (58%)]      Loss: 109.684921
Train Epoch: 4 [35840/60000 (60%)]      Loss: 106.182327
Train Epoch: 4 [37120/60000 (62%)]      Loss: 110.109459
Train Epoch: 4 [38400/60000 (64%)]      Loss: 111.840340
Train Epoch: 4 [39680/60000 (66%)]      Loss: 114.688583
Train Epoch: 4 [40960/60000 (68%)]      Loss: 112.778084
Train Epoch: 4 [42240/60000 (70%)]      Loss: 108.759384
Train Epoch: 4 [43520/60000 (72%)]      Loss: 113.430641
Train Epoch: 4 [44800/60000 (75%)]      Loss: 106.302727
Train Epoch: 4 [46080/60000 (77%)]      Loss: 110.591820
Train Epoch: 4 [47360/60000 (79%)]      Loss: 113.711258
Train Epoch: 4 [48640/60000 (81%)]      Loss: 110.368332
Train Epoch: 4 [49920/60000 (83%)]      Loss: 114.458008
Train Epoch: 4 [51200/60000 (85%)]      Loss: 112.680672
Train Epoch: 4 [52480/60000 (87%)]      Loss: 112.330994
Train Epoch: 4 [53760/60000 (90%)]      Loss: 109.988037
Train Epoch: 4 [55040/60000 (92%)]      Loss: 109.544281
Train Epoch: 4 [56320/60000 (94%)]      Loss: 111.551651
Train Epoch: 4 [57600/60000 (96%)]      Loss: 113.451370
Train Epoch: 4 [58880/60000 (98%)]      Loss: 111.917374
====> Epoch: 4 Average loss: 111.4813
====> Test set loss: 109.8046
Train Epoch: 5 [0/60000 (0%)]   Loss: 112.991783
Train Epoch: 5 [1280/60000 (2%)]        Loss: 108.989746
Train Epoch: 5 [2560/60000 (4%)]        Loss: 108.004852
Train Epoch: 5 [3840/60000 (6%)]        Loss: 108.123047
Train Epoch: 5 [5120/60000 (9%)]        Loss: 109.295609
Train Epoch: 5 [6400/60000 (11%)]       Loss: 111.604500
Train Epoch: 5 [7680/60000 (13%)]       Loss: 112.504807
Train Epoch: 5 [8960/60000 (15%)]       Loss: 111.840721
Train Epoch: 5 [10240/60000 (17%)]      Loss: 106.736221
Train Epoch: 5 [11520/60000 (19%)]      Loss: 109.404976
Train Epoch: 5 [12800/60000 (21%)]      Loss: 111.003838
Train Epoch: 5 [14080/60000 (23%)]      Loss: 106.550278
Train Epoch: 5 [15360/60000 (26%)]      Loss: 108.456451
Train Epoch: 5 [16640/60000 (28%)]      Loss: 108.284164
Train Epoch: 5 [17920/60000 (30%)]      Loss: 105.083572
Train Epoch: 5 [19200/60000 (32%)]      Loss: 108.913528
Train Epoch: 5 [20480/60000 (34%)]      Loss: 108.706261
Train Epoch: 5 [21760/60000 (36%)]      Loss: 111.635483
Train Epoch: 5 [23040/60000 (38%)]      Loss: 113.879669
Train Epoch: 5 [24320/60000 (41%)]      Loss: 108.390800
Train Epoch: 5 [25600/60000 (43%)]      Loss: 110.565384
Train Epoch: 5 [26880/60000 (45%)]      Loss: 109.312050
Train Epoch: 5 [28160/60000 (47%)]      Loss: 104.949638
Train Epoch: 5 [29440/60000 (49%)]      Loss: 111.242813
```

```
Train Epoch: 5 [30720/60000 (51%)]      Loss: 110.453484
Train Epoch: 5 [32000/60000 (53%)]      Loss: 109.355637
Train Epoch: 5 [33280/60000 (55%)]      Loss: 112.656090
Train Epoch: 5 [34560/60000 (58%)]      Loss: 109.362747
Train Epoch: 5 [35840/60000 (60%)]      Loss: 106.413055
Train Epoch: 5 [37120/60000 (62%)]      Loss: 109.362267
Train Epoch: 5 [38400/60000 (64%)]      Loss: 110.037140
Train Epoch: 5 [39680/60000 (66%)]      Loss: 105.842995
Train Epoch: 5 [40960/60000 (68%)]      Loss: 110.943802
Train Epoch: 5 [42240/60000 (70%)]      Loss: 105.749878
Train Epoch: 5 [43520/60000 (72%)]      Loss: 113.949707
Train Epoch: 5 [44800/60000 (75%)]      Loss: 107.526169
Train Epoch: 5 [46080/60000 (77%)]      Loss: 108.988037
Train Epoch: 5 [47360/60000 (79%)]      Loss: 108.074226
Train Epoch: 5 [48640/60000 (81%)]      Loss: 111.953331
Train Epoch: 5 [49920/60000 (83%)]      Loss: 107.496529
Train Epoch: 5 [51200/60000 (85%)]      Loss: 108.496704
Train Epoch: 5 [52480/60000 (87%)]      Loss: 109.846619
Train Epoch: 5 [53760/60000 (90%)]      Loss: 109.132050
Train Epoch: 5 [55040/60000 (92%)]      Loss: 110.426025
Train Epoch: 5 [56320/60000 (94%)]      Loss: 109.541641
Train Epoch: 5 [57600/60000 (96%)]      Loss: 110.152527
Train Epoch: 5 [58880/60000 (98%)]      Loss: 105.980377
====> Epoch: 5 Average loss: 109.7551
====> Test set loss: 108.4793
Train Epoch: 6 [0/60000 (0%)]   Loss: 108.505859
Train Epoch: 6 [1280/60000 (2%)]        Loss: 109.834488
Train Epoch: 6 [2560/60000 (4%)]        Loss: 106.853470
Train Epoch: 6 [3840/60000 (6%)]        Loss: 109.281166
Train Epoch: 6 [5120/60000 (9%)]        Loss: 104.007095
Train Epoch: 6 [6400/60000 (11%)]       Loss: 108.719223
Train Epoch: 6 [7680/60000 (13%)]       Loss: 108.563705
Train Epoch: 6 [8960/60000 (15%)]       Loss: 107.902664
Train Epoch: 6 [10240/60000 (17%)]      Loss: 108.561569
Train Epoch: 6 [11520/60000 (19%)]      Loss: 109.297264
Train Epoch: 6 [12800/60000 (21%)]      Loss: 109.594994
Train Epoch: 6 [14080/60000 (23%)]      Loss: 103.630516
Train Epoch: 6 [15360/60000 (26%)]      Loss: 110.680672
Train Epoch: 6 [16640/60000 (28%)]      Loss: 108.052086
Train Epoch: 6 [17920/60000 (30%)]      Loss: 105.734749
Train Epoch: 6 [19200/60000 (32%)]      Loss: 107.807793
Train Epoch: 6 [20480/60000 (34%)]      Loss: 107.628380
Train Epoch: 6 [21760/60000 (36%)]      Loss: 108.349213
Train Epoch: 6 [23040/60000 (38%)]      Loss: 106.557243
Train Epoch: 6 [24320/60000 (41%)]      Loss: 111.445297
Train Epoch: 6 [25600/60000 (43%)]      Loss: 109.805031
Train Epoch: 6 [26880/60000 (45%)]      Loss: 106.308975
Train Epoch: 6 [28160/60000 (47%)]      Loss: 110.326881
```

```
Train Epoch: 6 [29440/60000 (49%)]      Loss: 106.622849
Train Epoch: 6 [30720/60000 (51%)]      Loss: 109.760956
Train Epoch: 6 [32000/60000 (53%)]      Loss: 112.280128
Train Epoch: 6 [33280/60000 (55%)]      Loss: 110.038147
Train Epoch: 6 [34560/60000 (58%)]      Loss: 108.453644
Train Epoch: 6 [35840/60000 (60%)]      Loss: 106.972229
Train Epoch: 6 [37120/60000 (62%)]      Loss: 104.929672
Train Epoch: 6 [38400/60000 (64%)]      Loss: 108.170044
Train Epoch: 6 [39680/60000 (66%)]      Loss: 108.734467
Train Epoch: 6 [40960/60000 (68%)]      Loss: 108.966034
Train Epoch: 6 [42240/60000 (70%)]      Loss: 111.175690
Train Epoch: 6 [43520/60000 (72%)]      Loss: 108.248665
Train Epoch: 6 [44800/60000 (75%)]      Loss: 110.408234
Train Epoch: 6 [46080/60000 (77%)]      Loss: 108.973221
Train Epoch: 6 [47360/60000 (79%)]      Loss: 110.595215
Train Epoch: 6 [48640/60000 (81%)]      Loss: 111.061417
Train Epoch: 6 [49920/60000 (83%)]      Loss: 107.301987
Train Epoch: 6 [51200/60000 (85%)]      Loss: 109.759499
Train Epoch: 6 [52480/60000 (87%)]      Loss: 107.855042
Train Epoch: 6 [53760/60000 (90%)]      Loss: 106.868843
Train Epoch: 6 [55040/60000 (92%)]      Loss: 109.547920
Train Epoch: 6 [56320/60000 (94%)]      Loss: 108.891655
Train Epoch: 6 [57600/60000 (96%)]      Loss: 107.840752
Train Epoch: 6 [58880/60000 (98%)]      Loss: 106.073227
====> Epoch: 6 Average loss: 108.6459
====> Test set loss: 107.2975
Train Epoch: 7 [0/60000 (0%)]   Loss: 105.730301
Train Epoch: 7 [1280/60000 (2%)]        Loss: 109.658615
Train Epoch: 7 [2560/60000 (4%)]        Loss: 106.546844
Train Epoch: 7 [3840/60000 (6%)]        Loss: 106.082489
Train Epoch: 7 [5120/60000 (9%)]        Loss: 109.986687
Train Epoch: 7 [6400/60000 (11%)]       Loss: 103.700600
Train Epoch: 7 [7680/60000 (13%)]       Loss: 111.700012
Train Epoch: 7 [8960/60000 (15%)]       Loss: 111.790642
Train Epoch: 7 [10240/60000 (17%)]      Loss: 108.633942
Train Epoch: 7 [11520/60000 (19%)]      Loss: 105.828362
Train Epoch: 7 [12800/60000 (21%)]      Loss: 108.727997
Train Epoch: 7 [14080/60000 (23%)]      Loss: 110.896446
Train Epoch: 7 [15360/60000 (26%)]      Loss: 105.045837
Train Epoch: 7 [16640/60000 (28%)]      Loss: 113.660316
Train Epoch: 7 [17920/60000 (30%)]      Loss: 104.312210
Train Epoch: 7 [19200/60000 (32%)]      Loss: 108.212662
Train Epoch: 7 [20480/60000 (34%)]      Loss: 109.566238
Train Epoch: 7 [21760/60000 (36%)]      Loss: 109.173126
Train Epoch: 7 [23040/60000 (38%)]      Loss: 107.205643
Train Epoch: 7 [24320/60000 (41%)]      Loss: 108.830742
Train Epoch: 7 [25600/60000 (43%)]      Loss: 105.068192
Train Epoch: 7 [26880/60000 (45%)]      Loss: 107.802322
```

```
Train Epoch: 7 [28160/60000 (47%)]      Loss: 107.595398
Train Epoch: 7 [29440/60000 (49%)]      Loss: 104.832344
Train Epoch: 7 [30720/60000 (51%)]      Loss: 106.664177
Train Epoch: 7 [32000/60000 (53%)]      Loss: 106.527817
Train Epoch: 7 [33280/60000 (55%)]      Loss: 109.081787
Train Epoch: 7 [34560/60000 (58%)]      Loss: 103.392342
Train Epoch: 7 [35840/60000 (60%)]      Loss: 107.233299
Train Epoch: 7 [37120/60000 (62%)]      Loss: 108.840073
Train Epoch: 7 [38400/60000 (64%)]      Loss: 105.958107
Train Epoch: 7 [39680/60000 (66%)]      Loss: 105.913879
Train Epoch: 7 [40960/60000 (68%)]      Loss: 108.513351
Train Epoch: 7 [42240/60000 (70%)]      Loss: 104.383499
Train Epoch: 7 [43520/60000 (72%)]      Loss: 108.467857
Train Epoch: 7 [44800/60000 (75%)]      Loss: 106.140060
Train Epoch: 7 [46080/60000 (77%)]      Loss: 109.998230
Train Epoch: 7 [47360/60000 (79%)]      Loss: 106.991348
Train Epoch: 7 [48640/60000 (81%)]      Loss: 104.442917
Train Epoch: 7 [49920/60000 (83%)]      Loss: 108.629639
Train Epoch: 7 [51200/60000 (85%)]      Loss: 106.790794
Train Epoch: 7 [52480/60000 (87%)]      Loss: 106.558090
Train Epoch: 7 [53760/60000 (90%)]      Loss: 106.096481
Train Epoch: 7 [55040/60000 (92%)]      Loss: 107.494843
Train Epoch: 7 [56320/60000 (94%)]      Loss: 109.315727
Train Epoch: 7 [57600/60000 (96%)]      Loss: 106.098938
Train Epoch: 7 [58880/60000 (98%)]      Loss: 109.080879
====> Epoch: 7 Average loss: 107.7092
====> Test set loss: 106.8546
Train Epoch: 8 [0/60000 (0%)]   Loss: 105.041023
Train Epoch: 8 [1280/60000 (2%)]        Loss: 112.572037
Train Epoch: 8 [2560/60000 (4%)]        Loss: 108.288094
Train Epoch: 8 [3840/60000 (6%)]        Loss: 105.973122
Train Epoch: 8 [5120/60000 (9%)]        Loss: 107.764000
Train Epoch: 8 [6400/60000 (11%)]       Loss: 107.273743
Train Epoch: 8 [7680/60000 (13%)]       Loss: 107.516846
Train Epoch: 8 [8960/60000 (15%)]       Loss: 109.439896
Train Epoch: 8 [10240/60000 (17%)]      Loss: 108.526367
Train Epoch: 8 [11520/60000 (19%)]      Loss: 106.048325
Train Epoch: 8 [12800/60000 (21%)]      Loss: 111.425194
Train Epoch: 8 [14080/60000 (23%)]      Loss: 106.403908
Train Epoch: 8 [15360/60000 (26%)]      Loss: 108.006668
Train Epoch: 8 [16640/60000 (28%)]      Loss: 104.422546
Train Epoch: 8 [17920/60000 (30%)]      Loss: 109.322662
Train Epoch: 8 [19200/60000 (32%)]      Loss: 109.860077
Train Epoch: 8 [20480/60000 (34%)]      Loss: 108.677666
Train Epoch: 8 [21760/60000 (36%)]      Loss: 103.639267
Train Epoch: 8 [23040/60000 (38%)]      Loss: 109.182808
Train Epoch: 8 [24320/60000 (41%)]      Loss: 105.691177
Train Epoch: 8 [25600/60000 (43%)]      Loss: 105.480774
```

```
Train Epoch: 8 [26880/60000 (45%)]      Loss: 103.903641
Train Epoch: 8 [28160/60000 (47%)]      Loss: 109.933189
Train Epoch: 8 [29440/60000 (49%)]      Loss: 106.123283
Train Epoch: 8 [30720/60000 (51%)]      Loss: 109.700577
Train Epoch: 8 [32000/60000 (53%)]      Loss: 110.203171
Train Epoch: 8 [33280/60000 (55%)]      Loss: 105.755280
Train Epoch: 8 [34560/60000 (58%)]      Loss: 111.894440
Train Epoch: 8 [35840/60000 (60%)]      Loss: 106.857391
Train Epoch: 8 [37120/60000 (62%)]      Loss: 104.462814
Train Epoch: 8 [38400/60000 (64%)]      Loss: 106.989044
Train Epoch: 8 [39680/60000 (66%)]      Loss: 105.153961
Train Epoch: 8 [40960/60000 (68%)]      Loss: 107.828552
Train Epoch: 8 [42240/60000 (70%)]      Loss: 103.774956
Train Epoch: 8 [43520/60000 (72%)]      Loss: 107.153854
Train Epoch: 8 [44800/60000 (75%)]      Loss: 106.761154
Train Epoch: 8 [46080/60000 (77%)]      Loss: 102.698029
Train Epoch: 8 [47360/60000 (79%)]      Loss: 105.693497
Train Epoch: 8 [48640/60000 (81%)]      Loss: 105.985497
Train Epoch: 8 [49920/60000 (83%)]      Loss: 104.574074
Train Epoch: 8 [51200/60000 (85%)]      Loss: 106.048454
Train Epoch: 8 [52480/60000 (87%)]      Loss: 107.059875
Train Epoch: 8 [53760/60000 (90%)]      Loss: 109.193024
Train Epoch: 8 [55040/60000 (92%)]      Loss: 105.050751
Train Epoch: 8 [56320/60000 (94%)]      Loss: 108.374588
Train Epoch: 8 [57600/60000 (96%)]      Loss: 107.632698
Train Epoch: 8 [58880/60000 (98%)]      Loss: 105.859360
====> Epoch: 8 Average loss: 107.1235
====> Test set loss: 106.4418
Train Epoch: 9 [0/60000 (0%)]   Loss: 108.966171
Train Epoch: 9 [1280/60000 (2%)]        Loss: 110.048325
Train Epoch: 9 [2560/60000 (4%)]        Loss: 105.516266
Train Epoch: 9 [3840/60000 (6%)]        Loss: 111.503983
Train Epoch: 9 [5120/60000 (9%)]        Loss: 108.238014
Train Epoch: 9 [6400/60000 (11%)]       Loss: 105.653687
Train Epoch: 9 [7680/60000 (13%)]       Loss: 109.413826
Train Epoch: 9 [8960/60000 (15%)]       Loss: 108.930466
Train Epoch: 9 [10240/60000 (17%)]      Loss: 108.397537
Train Epoch: 9 [11520/60000 (19%)]      Loss: 109.781113
Train Epoch: 9 [12800/60000 (21%)]      Loss: 104.643471
Train Epoch: 9 [14080/60000 (23%)]      Loss: 107.087463
Train Epoch: 9 [15360/60000 (26%)]      Loss: 109.663437
Train Epoch: 9 [16640/60000 (28%)]      Loss: 106.789886
Train Epoch: 9 [17920/60000 (30%)]      Loss: 107.948532
Train Epoch: 9 [19200/60000 (32%)]      Loss: 102.518036
Train Epoch: 9 [20480/60000 (34%)]      Loss: 104.895012
Train Epoch: 9 [21760/60000 (36%)]      Loss: 111.627441
Train Epoch: 9 [23040/60000 (38%)]      Loss: 106.619759
Train Epoch: 9 [24320/60000 (41%)]      Loss: 101.878372
```

```
Train Epoch: 9 [25600/60000 (43%)]      Loss: 109.109238
Train Epoch: 9 [26880/60000 (45%)]      Loss: 108.014893
Train Epoch: 9 [28160/60000 (47%)]      Loss: 106.875748
Train Epoch: 9 [29440/60000 (49%)]      Loss: 104.649162
Train Epoch: 9 [30720/60000 (51%)]      Loss: 109.301636
Train Epoch: 9 [32000/60000 (53%)]      Loss: 111.715187
Train Epoch: 9 [33280/60000 (55%)]      Loss: 107.886047
Train Epoch: 9 [34560/60000 (58%)]      Loss: 106.527412
Train Epoch: 9 [35840/60000 (60%)]      Loss: 106.261078
Train Epoch: 9 [37120/60000 (62%)]      Loss: 105.550201
Train Epoch: 9 [38400/60000 (64%)]      Loss: 103.308533
Train Epoch: 9 [39680/60000 (66%)]      Loss: 105.401375
Train Epoch: 9 [40960/60000 (68%)]      Loss: 109.298508
Train Epoch: 9 [42240/60000 (70%)]      Loss: 106.445663
Train Epoch: 9 [43520/60000 (72%)]      Loss: 102.687225
Train Epoch: 9 [44800/60000 (75%)]      Loss: 103.630402
Train Epoch: 9 [46080/60000 (77%)]      Loss: 110.898590
Train Epoch: 9 [47360/60000 (79%)]      Loss: 106.520500
Train Epoch: 9 [48640/60000 (81%)]      Loss: 108.130737
Train Epoch: 9 [49920/60000 (83%)]      Loss: 107.943069
Train Epoch: 9 [51200/60000 (85%)]      Loss: 106.273987
Train Epoch: 9 [52480/60000 (87%)]      Loss: 103.861237
Train Epoch: 9 [53760/60000 (90%)]      Loss: 101.042374
Train Epoch: 9 [55040/60000 (92%)]      Loss: 102.077835
Train Epoch: 9 [56320/60000 (94%)]      Loss: 104.360863
Train Epoch: 9 [57600/60000 (96%)]      Loss: 106.404030
Train Epoch: 9 [58880/60000 (98%)]      Loss: 101.811279
====> Epoch: 9 Average loss: 106.6244
====> Test set loss: 105.7810
Train Epoch: 10 [0/60000 (0%)]  Loss: 105.610443
Train Epoch: 10 [1280/60000 (2%)]       Loss: 103.960693
Train Epoch: 10 [2560/60000 (4%)]       Loss: 110.520874
Train Epoch: 10 [3840/60000 (6%)]       Loss: 108.841141
Train Epoch: 10 [5120/60000 (9%)]       Loss: 107.548325
Train Epoch: 10 [6400/60000 (11%)]      Loss: 111.096054
Train Epoch: 10 [7680/60000 (13%)]      Loss: 107.063446
Train Epoch: 10 [8960/60000 (15%)]      Loss: 104.129494
Train Epoch: 10 [10240/60000 (17%)]     Loss: 107.383888
Train Epoch: 10 [11520/60000 (19%)]     Loss: 105.091736
Train Epoch: 10 [12800/60000 (21%)]     Loss: 106.034531
Train Epoch: 10 [14080/60000 (23%)]     Loss: 103.662704
Train Epoch: 10 [15360/60000 (26%)]     Loss: 110.323196
Train Epoch: 10 [16640/60000 (28%)]     Loss: 106.924675
Train Epoch: 10 [17920/60000 (30%)]     Loss: 104.277954
Train Epoch: 10 [19200/60000 (32%)]     Loss: 105.485260
Train Epoch: 10 [20480/60000 (34%)]     Loss: 105.521912
Train Epoch: 10 [21760/60000 (36%)]     Loss: 104.254440
Train Epoch: 10 [23040/60000 (38%)]     Loss: 108.495293
```

```
Train Epoch: 10 [24320/60000 (41%)]     Loss: 104.598412
Train Epoch: 10 [25600/60000 (43%)]     Loss: 102.214706
Train Epoch: 10 [26880/60000 (45%)]     Loss: 105.787109
Train Epoch: 10 [28160/60000 (47%)]     Loss: 106.983864
Train Epoch: 10 [29440/60000 (49%)]     Loss: 105.721764
Train Epoch: 10 [30720/60000 (51%)]     Loss: 107.219543
Train Epoch: 10 [32000/60000 (53%)]     Loss: 109.746094
Train Epoch: 10 [33280/60000 (55%)]     Loss: 103.272308
Train Epoch: 10 [34560/60000 (58%)]     Loss: 107.094254
Train Epoch: 10 [35840/60000 (60%)]     Loss: 108.087059
Train Epoch: 10 [37120/60000 (62%)]     Loss: 106.604698
Train Epoch: 10 [38400/60000 (64%)]     Loss: 109.573204
Train Epoch: 10 [39680/60000 (66%)]     Loss: 104.669708
Train Epoch: 10 [40960/60000 (68%)]     Loss: 105.169456
Train Epoch: 10 [42240/60000 (70%)]     Loss: 104.462662
Train Epoch: 10 [43520/60000 (72%)]     Loss: 103.219894
Train Epoch: 10 [44800/60000 (75%)]     Loss: 108.372055
Train Epoch: 10 [46080/60000 (77%)]     Loss: 105.556030
Train Epoch: 10 [47360/60000 (79%)]     Loss: 103.479088
Train Epoch: 10 [48640/60000 (81%)]     Loss: 107.296158
Train Epoch: 10 [49920/60000 (83%)]     Loss: 107.129837
Train Epoch: 10 [51200/60000 (85%)]     Loss: 101.788994
Train Epoch: 10 [52480/60000 (87%)]     Loss: 105.631973
Train Epoch: 10 [53760/60000 (90%)]     Loss: 106.162102
Train Epoch: 10 [55040/60000 (92%)]     Loss: 102.664093
Train Epoch: 10 [56320/60000 (94%)]     Loss: 103.518196
Train Epoch: 10 [57600/60000 (96%)]     Loss: 106.986435
Train Epoch: 10 [58880/60000 (98%)]     Loss: 105.761192
====> Epoch: 10 Average loss: 106.1560
====> Test set loss: 105.4753
```

3.1.3 Visualize the latent space as a 2D plot, coloring each point by its label. Since our VAE is using a 20 dimensional latent space, you can try some of our dimensionality reduction tricks from the previous pset (PCA, PHATE, tSNE) to get a coherent 2 dimensional representation.

[ ]:
```python
import phate
import scprep
# from vae import VAE

class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
```

```
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)

    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5*logvar)
        eps = torch.randn_like(std)
        return mu + eps*std

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar


model_vae = VAE().to(device)
model_vae.load_state_dict(torch.load(os.path.join(path_save, 'model_vae.pth')))
```

[ ]:
```
<All keys matched successfully>
```

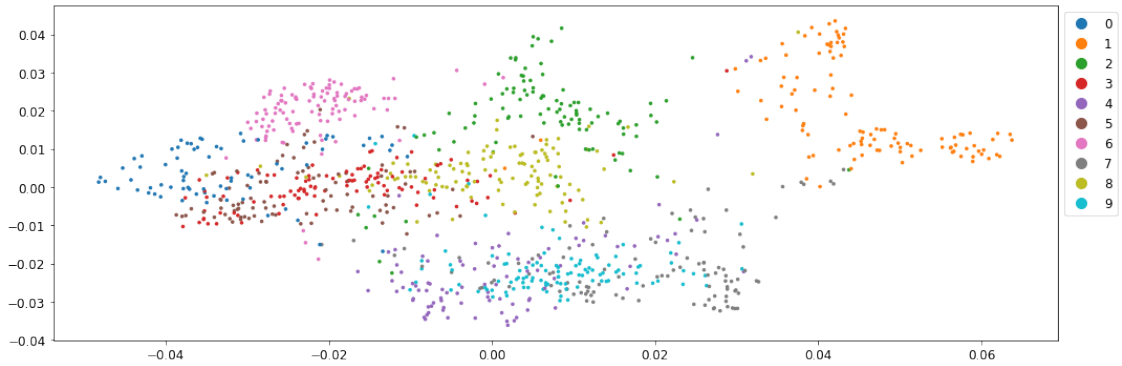
[ ]:
```
data_loader_1000_digits = DataLoader(mnist_train, batch_size=1000, shuffle=True)
for data, target in data_loader_1000_digits:
    with torch.no_grad():
        data = data.to(device)
        model_input = data.view(-1, 784)
        mu, logvar = model_vae.encode(model_input)
        embedds = model_vae.reparameterize(mu, logvar).cpu().numpy()
        # preds, mu, logvar = model_vae(model_input).cpu().numpy()
        labels = target.cpu().numpy()
        # inputs = model_input.cpu().numpy()

subsample_data_pc, subsample_meta = scprep.select.subsample(embedds, labels, n␣
 ↪= 1000)
data_phate = phate.PHATE().fit_transform(subsample_data_pc)

scprep.plot.scatter2d(data_phate, c=list(subsample_meta), figsize=(15,5),␣
 ↪legend_anchor=(1,1))
```

```
Calculating PHATE...
  Running PHATE on 1000 observations and 20 variables.
```

```
  Calculating graph and diffusion operator…
    Calculating KNN search…
    Calculated KNN search in 0.04 seconds.
    Calculating affinities…
    Calculated affinities in 0.01 seconds.
  Calculated graph and diffusion operator in 0.07 seconds.
  Calculating optimal t…
    Automatically selected t = 30
  Calculated optimal t in 0.39 seconds.
  Calculating diffusion potential…
  Calculated diffusion potential in 0.11 seconds.
  Calculating metric MDS…
  Calculated metric MDS in 1.12 seconds.
Calculated PHATE in 1.71 seconds.
```

```
/home/xiaoranzhang/anaconda3/envs/torch_env/lib/python3.9/site-
packages/scprep/plot/utils.py:104: UserWarning: Matplotlib is currently using
module://matplotlib_inline.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  fig.show()
```

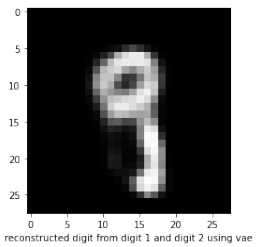
[ ]: <AxesSubplot:>

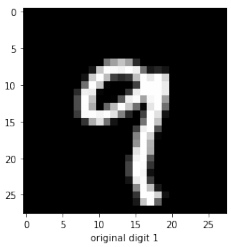


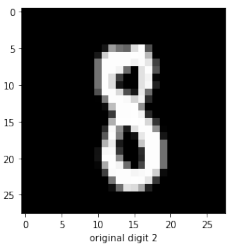
3.1.4 As before, try interpolating between two different images in the latent space. Run the fabricated embedding through the decoder to generate a never-before seen digit. You may wish to try interpolating between digits of the same class in addition to digits of different classes.

[ ]:
```
sample_idx_list, digit_list = [], []
for idx in range(labels.shape[0]):
    for digit in range(10):
        if labels[idx] == digit:
            if digit not in digit_list:
                sample_idx_list.append(idx)
```

```
                digit_list.append(digit)

digit_idx_1 = 0
digit_idx_2 = 1

for data, target in data_loader_1000_digits:
    with torch.no_grad():
        data = data.to(device)
        model_input = data.view(-1, 784)
        mu_1, logvar_1 = model_vae.encode(model_input[digit_list[digit_idx_1],:
 ↪])
        embedds_1 = model_vae.reparameterize(mu_1, logvar_1).cpu().numpy()

        mu_2, logvar_2 = model_vae.encode(model_input[digit_list[digit_idx_2],:
 ↪])
        embedds_2 = model_vae.reparameterize(mu_2, logvar_2).cpu().numpy()

        embedds_frab = (embedds_1 + embedds_2)/2
        preds = model_vae.decode(torch.from_numpy(embedds_frab).to(device)).
 ↪cpu().numpy()
        break

preds_digit = preds
digit_1 = model_input[digit_list[digit_idx_1],:].cpu().numpy()
digit_2 = model_input[digit_list[digit_idx_2],:].cpu().numpy()

fig, axes = plt.subplots(1,3)
fig.set_size_inches(24,4)

axes[0].imshow(np.reshape(preds_digit, (28,28)), cmap='gray')
axes[0].set_xlabel('reconstructed digit from digit 1 and digit 2 using vae')
axes[1].imshow(np.reshape(digit_1, (28,28)), cmap='gray')
axes[1].set_xlabel('original digit 1')
axes[2].imshow(np.reshape(digit_2, (28,28)), cmap='gray')
axes[2].set_xlabel('original digit 2')

fig.savefig(os.path.join(path_figures,'Section_4_1.pdf'))
plt.show()
```







3.1.5 Question 4.1.1. How does the VAE's latent space compare to the latent space of your previous autoencoder? Do the generated images have more clarity? Is this most noticeable between or within classes?

Answer: The latent space of VAE seems to provide better visualizations compared to the latent space of the previous autoencoder. The generated images have more clarity. It is noticeable between classes.

3.1.6 Question 4.1.2. In what situations would a VAE be more useful than a vanilla autoencoder, and when would you prefer a vanilla autoencoder to a VAE?

Answer: VAE is more suitable to employ on larger models. However, when the dataset is small and simple, AE should be tried first as it is a simpler model.

3.1.7 Question 4.1.3. The distance between embeddings in your first autoencoder provided some measure of the similarity between digits. To what extent is this preserved, or improved, by the VAE?

Answer: The distance is preserved in a more continuous fashion between classes as it is designed by the reparameterization trick by VAE.

4 4.2 GANs

4.0.1 Section 1-4, please check the code in GAN.py

[ ]:
```
!python GAN.py
```

```
EPOCH 0. Progress: 0.0%.
Train gen loss: 0.33152133226394653. Train dis loss: 0.6595855355262756
EPOCH 10. Progress: 10.0%.
Train gen loss: 0.733722984790802. Train dis loss: 0.4779012203216553
EPOCH 20. Progress: 20.0%.
Train gen loss: 0.5379916429519653. Train dis loss: 0.5625685453414917
EPOCH 30. Progress: 30.0%.
Train gen loss: 0.373367041349411. Train dis loss: 0.7363168001174927
EPOCH 40. Progress: 40.0%.
Train gen loss: 0.3073600232601166. Train dis loss: 0.6076046824455261
EPOCH 50. Progress: 50.0%.
Train gen loss: 1.5791900157928467. Train dis loss: 1.3175315856933594
EPOCH 60. Progress: 60.0%.
Train gen loss: 0.7346891760826111. Train dis loss: 0.6737794280052185
EPOCH 70. Progress: 70.0%.
Train gen loss: 0.787720263004303. Train dis loss: 0.5112329721450806
EPOCH 80. Progress: 80.0%.
Train gen loss: 0.7522504925727844. Train dis loss: 0.6918277740478516
EPOCH 90. Progress: 90.0%.
Train gen loss: 0.6981362700462341. Train dis loss: 0.7891863584518433
```

4.0.2 Using your best performing classifier from Part 2, classify these samples.

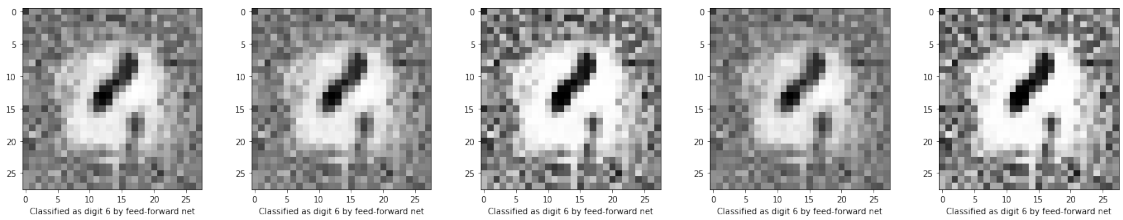
[ ]:
```
class Generator(nn.Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz # the dimension of the random noise used to seed the␣
↪Generator
        self.main = nn.Sequential( # nn.sequential is a handy way of combining␣
↪multiple layers.
            nn.Linear(self.nz, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, 784),
            nn.Tanh(),
        )
    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)

nz = 25
numOfSample = 1000

model_gen = Generator(nz).to(device)
model_gen.load_state_dict(torch.load(os.path.join(path_save, 'model_gen.pth')))
noise = torch.randn((numOfSample, nz)).to('cuda')
noise = torch.clamp(noise, 1e-8, 1)
generated_img = model_gen(noise).detach()
```

[ ]:
```
class FeedForwardNet(nn.Module):
    """ Simple feed forward network with one hidden layer."""
    def __init__(self): # initialize the model
        super(FeedForwardNet, self).__init__() # call for the parent class to␣
↪initialize
        self.W1 = nn.Parameter(nn.init.uniform_(torch.empty((784, 128)), a=-np.
↪sqrt(1/128), b=np.sqrt(1/128)))
        self.b1 = nn.Parameter(nn.init.uniform_(torch.empty((1, 128)), a=-np.
↪sqrt(1/128), b=np.sqrt(1/128)))

        self.W2 = nn.Parameter(nn.init.uniform_(torch.empty((128, 10)), a=-np.
↪sqrt(1/10), b=np.sqrt(1/10)))
        self.b2 = nn.Parameter(nn.init.uniform_(torch.empty((1, 10)), a=-np.
↪sqrt(1/10), b=np.sqrt(1/10)))

        # Make sure to add another weight and bias vector to represent the␣
↪hidden layer.
```

```
    def forward(self, x):
        # put the logic here.
        layer1_out = relu(torch.matmul(x, self.W1) + self.b1)
        layer2_out = softmax(relu(torch.matmul(layer1_out, self.W2) + self.b2))

        predictions = layer2_out

        return predictions

model_FCN = FeedForwardNet().to(device)
model_FCN.load_state_dict(torch.load(os.path.join(path_save,␣
 ↪'model_relu_bs_128_lr_0.5_epoch_50.pth')))

prediction_onehot = model_FCN(generated_img.to(device).view(-1, 784))
prediction = torch.argmax(prediction_onehot, dim=1)
```

```
/tmp/ipykernel_20559/969223363.py:16: UserWarning: Implicit dimension choice for
softmax has been deprecated. Change the call to include dim=X as an argument.
  layer2_out = softmax(relu(torch.matmul(layer1_out, self.W2) + self.b2))
```

[ ]:
```
fig, axes = plt.subplots(1,5)
fig.set_size_inches(24,4)

for i in range(5):
    axes[i].imshow(np.reshape(generated_img[i].cpu().numpy(), (28, 28)),␣
 ↪cmap='gray')
    axes[i].set_xlabel('Classified as digit {} by feed-forward net'.
 ↪format(str(prediction.cpu().numpy()[i])))

fig.savefig(os.path.join(path_figures,'Section_4_2.pdf'))
plt.show()
```



4.0.3 Question 4.2.1. Which generates more realistic images: your GAN, or your VAE? Why do you think this is?

Answer: VAE generates more realistic images compared to GAN. This might due to that VAE learns from handwritten digit but the GAN learns from the noise.

4.0.4 Does your GAN appear to generate all digits in equal number, or has it specialized in a smaller number of digits? If so, why might this be?

Answer: The GAN does not appear to generate all digits in equal number. It seems to be specialized in a smaller number of digits. It seems to generate digits such as 0 as a lot of numbers (9, 6, 8) has curves.

5 5 Information Theory

5.1 5.1 Simple Distribution

5.1.1 Compute the Kullback-Leibler (KL) Divergence of the two distributions.

[ ]:
```
mean = [0, 0, 0]
cov = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
numOfSample = 1000
X_normal = np.random.multivariate_normal(mean, cov, numOfSample)
X_uniform = np.random.uniform(0, 1, (numOfSample,3))
```

[ ]:
```
from scipy.stats import entropy
from sklearn.neighbors import KernelDensity
from time import perf_counter

kde_normal = KernelDensity(kernel='gaussian').fit(X_normal)
kde_uniform = KernelDensity(kernel='gaussian').fit(X_uniform)

prob_normal = np.exp(kde_normal.score_samples(X_normal))
prob_uniform = np.exp(kde_uniform.score_samples(X_uniform))
```

[ ]:
```
def func_KL_div(prob_distr1, prob_distr2):

    start_time = perf_counter()
    KL_div = entropy(prob_distr1, prob_distr2)
    end_time = perf_counter()

    return KL_div, end_time-start_time

print('KL normal to uniform: {}. Time elapsed: {}'.
 ↪format(func_KL_div(prob_normal, prob_uniform)[0], func_KL_div(prob_normal,␣
 ↪prob_uniform)[1]))
print('KL uniform to normal: {}. Time elapsed: {}'.
 ↪format(func_KL_div(prob_uniform, prob_normal)[0], func_KL_div(prob_uniform,␣
 ↪prob_normal)[1]))
```

```
KL normal to uniform: 0.1044634109528535. Time elapsed: 0.0014169570058584213
KL uniform to normal: 0.1365064541750548. Time elapsed: 0.0003813920193351805
```

5.1.2 Compute the Earth Mover's Distance (EMD) between the two distributions.

[ ]:
```
from scipy.spatial import distance_matrix
from pyemd import emd

def func_EMD(prob_distr1, prob_distr2):
    if prob_distr1.ndim == 1 and prob_distr2.ndim == 1:
        dist_mat = distance_matrix(np.expand_dims(prob_distr1, 1), np.
 ↪expand_dims(prob_distr2, 1))
    else:
        dist_mat = distance_matrix(prob_distr1, prob_distr2)

    start_time = perf_counter()
    emd_dist = emd(prob_distr1, prob_distr2, dist_mat)
    end_time = perf_counter()

    return emd_dist, end_time-start_time

print('EMD distance: {}. Time elapsed: {}'.format(func_EMD(prob_normal,␣
 ↪prob_uniform)[0], func_EMD(prob_normal, prob_uniform)[1]))
```

```
EMD distance: 2.1257755820697755. Time elapsed: 0.08436427998822182
```

5.1.3 Compute the Maximum Mean Discrepancy (MMD) between the two distributions.

[ ]:
```
def mmd(X,Y, kernel_fn):
    """
    Implementation of Maximum Mean Discrepancy.
    :param X: An n x 1 numpy vector containing the samples from distribution 1.
    :param Y: An n x 1 numpy vector containing the samples from distribution 2.
    :param kernel_fn: supply the kernel function to use.
    :return: the maximum mean discrepancy:
    MMD(X,Y) = Expected value of k(X,X) + Expected value of k(Y,Y) - Expected␣
 ↪value of k(X,Y)
    where k is a kernel function
    """
    mmd = np.mean(kernel_fn(X, X)) + np.mean(kernel_fn(Y, Y)) - np.
 ↪mean(kernel_fn(X, Y))
    return mmd


def kernel(A, B):
    """
    A gaussian kernel on two arrays.
    :param A: An n x d numpy matrix containing the samples from distribution 1
    :param B: An n x d numpy matrix containing the samples from distribution 2.
```

```
    :return K:  An n x n numpy matrix k, in which k_{i,j} = e^{-||A_i - B_j||^2/
↪(2*sigma^2)}
    """
    sigma = 1
    K = np.exp(-np.abs(A-B)**2 / 2*sigma**2 )
    return K
```

[ ]:
```
def func_MMD(X_distr1, X_distr2):
    start_time = perf_counter()
    mmd_dist = mmd(X_distr1, X_distr2, kernel)
    end_time = perf_counter()

    return mmd_dist, end_time-start_time

print('MMD distance: {}. Time elapsed: {}'.format(func_MMD(X_normal,␣
↪X_uniform)[0], func_MMD(X_normal, X_uniform)[1]))
```

```
MMD distance: 1.3476897524249987. Time elapsed: 0.0004335150006227195
```

5.1.4 Question 5.1.1. Based on the above measures alone, which divergence seems most accurate?

Answer: I think MMD distance seems most accurate and KL divergence is small and EMD seems to be way too large.

5.2 5.2 MNIST Sample Distributions

5.2.1 Compute the Kullback-Leibler (KL) Divergence of the two distributions.

[ ]:
```
from sklearn.decomposition import PCA

indices = torch.randperm(len(mnist_test))[:2000]

subset1 = torch.utils.data.Subset(mnist_test, indices[:1000])
subset2 = torch.utils.data.Subset(mnist_test, indices[1000:])

subset1_img_list, subset2_img_list = [], []
for img, lab in subset1:
    subset1_img_list.append(img.view(-1, 784).cpu().numpy())

for img, lab in subset2:
    subset2_img_list.append(img.view(-1, 784).cpu().numpy())

subset1_img_arr = np.concatenate(subset1_img_list, axis=0)
subset2_img_arr = np.concatenate(subset2_img_list, axis=0)

PCA_subset1 = PCA(n_components=8)
PCA_subset2 = PCA(n_components=8)
```

```
subset1_img_arr_PCA = PCA_subset1.fit_transform(subset1_img_arr)
subset2_img_arr_PCA = PCA_subset2.fit_transform(subset2_img_arr)

kde_subset1 = KernelDensity(kernel='gaussian').fit(subset1_img_arr_PCA)
kde_subset2 = KernelDensity(kernel='gaussian').fit(subset2_img_arr_PCA)

prob_subset1 = np.exp(kde_subset1.score_samples(subset1_img_arr_PCA))
prob_subset2 = np.exp(kde_subset2.score_samples(subset2_img_arr_PCA))

print('KL subset 1 to subset 2: {}. Time elapsed: {}'.
 ↪format(func_KL_div(prob_subset1, prob_subset2)[0], func_KL_div(prob_subset1,␣
 ↪prob_subset2)[1]))
print('KL subset 2 to subset 1: {}. Time elapsed: {}'.
 ↪format(func_KL_div(prob_subset2, prob_subset1)[0], func_KL_div(prob_subset2,␣
 ↪prob_subset1)[1]))
```

```
KL subset 1 to subset 2: 0.6715729293338801. Time elapsed: 9.958405280485749e-05
KL subset 2 to subset 1: 0.6040136198016678. Time elapsed: 0.001682029978837818
```

5.2.2 Compute the Earth Mover's Distance (EMD) between the two distributions.

[ ]:
```
print('EMD distance: {}. Time elapsed: {}'.format(func_EMD(prob_subset1,␣
 ↪prob_subset2)[0], func_EMD(prob_subset1, prob_subset2)[1]))
```

```
EMD distance: 4.286459269002608e-09. Time elapsed: 102.3937361909775
```

5.2.3 Compute the Maximum Mean Discrepancy (MMD) between the two distributions.

[ ]:
```
print('MMD distance: {}. Time elapsed: {}'.format(func_MMD(subset1_img_arr_PCA,␣
 ↪subset2_img_arr_PCA)[0], func_MMD(subset1_img_arr_PCA,␣
 ↪subset2_img_arr_PCA)[1]))
```

```
MMD distance: 1.6058905124664307. Time elapsed: 0.0007556750206276774
```

5.3 5.3 The GAN Distribution

5.3.1 Compute the Kullback-Leibler (KL) Divergence of the two distributions.

[ ]:
```
GAN_subset = generated_img.to(device).view(-1, 784).cpu().numpy()

PCA_GAN_subset = PCA(n_components=8)

GAN_subset_PCA = PCA_GAN_subset.fit_transform(GAN_subset)

kde_GAN_subset = KernelDensity(kernel='gaussian').fit(GAN_subset_PCA)
```

```
prob_GAN_subset = np.exp(kde_GAN_subset.score_samples(GAN_subset_PCA))

print('KL GAN to subset 2: {}. Time elapsed: {}'.
 ↪format(func_KL_div(prob_GAN_subset, prob_subset2)[0],␣
 ↪func_KL_div(prob_GAN_subset, prob_subset2)[1]))
print('KL subset 2 to GAN: {}. Time elapsed: {}'.
 ↪format(func_KL_div(prob_subset2, prob_GAN_subset)[0],␣
 ↪func_KL_div(prob_subset2, prob_GAN_subset)[1]))
```

```
KL GAN to subset 2: 0.32365281083383224. Time elapsed: 8.877599611878395e-05
KL subset 2 to GAN: 0.3972692551239079. Time elapsed: 0.0003860080032609403
```

5.3.2 Compute the Earth Mover's Distance (EMD) between the two distributions.

[ ]:
```
print('EMD distance: {}. Time elapsed: {}'.format(func_EMD(prob_GAN_subset,␣
 ↪prob_subset2)[0], func_EMD(prob_GAN_subset, prob_subset2)[1]))
```

```
EMD distance: 5.2296565691887656e-05. Time elapsed: 0.08263151004211977
```

5.3.3 Compute the Maximum Mean Discrepancy (MMD) between the two distributions.

[ ]:
```
print('MMD distance: {}. Time elapsed: {}'.format(func_MMD(GAN_subset_PCA,␣
 ↪subset2_img_arr_PCA)[0], func_MMD(GAN_subset_PCA, subset2_img_arr_PCA)[1]))
```

```
MMD distance: 1.503627061843872. Time elapsed: 0.003175096004270017
```

5.3.4 Question 5.3.1. Which divergence or distance showed the greatest discrepancy between the comparison between real MNIST data and the comparison with the GAN?

Answer: MMD shows the greatest discrepancy between the real MNISt samples and the generated fake samples from GAN.

5.3.5 Question 5.3.2. Which of these information measures would you recommend for judging a GAN's output? Why?

Answer: I recommend using EMD to judge a GAN's output. EMD is the only metric that shows larger discrepancy compared to the MNIST distribution example. As the GAN's fake output is visually worse than the real data, EMD seems to capture such charateristics.

5.3.6 Question 5.3.3. How do the runtimes of these measures compare?

Answer: It seems that the EMD takes the longest time as a distance matrix needs to be computed. The EMD computation of two distinct MNIST data takes very long. KL divergence seems to be the fastest metric and its runtime is comparable to MMD.