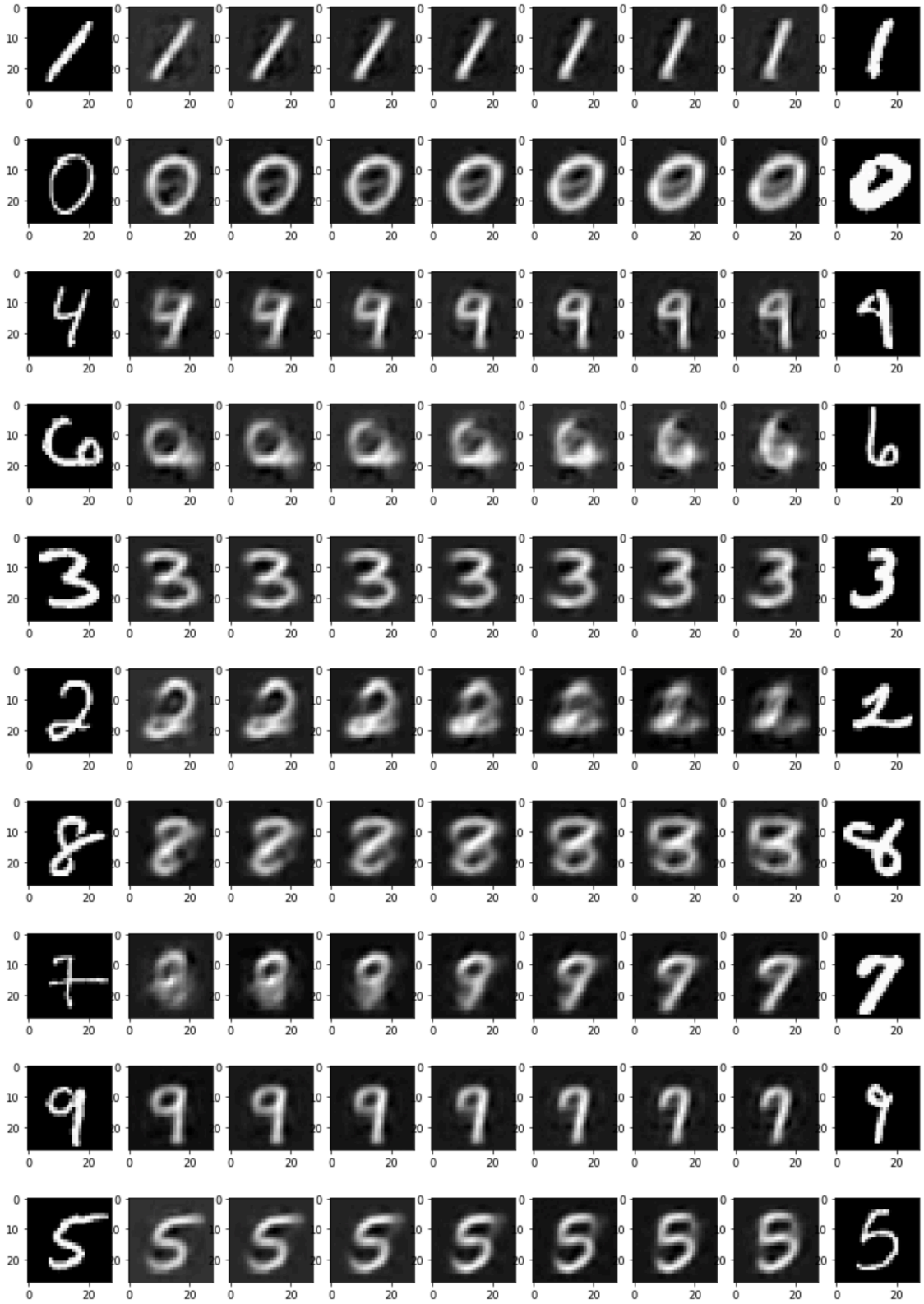
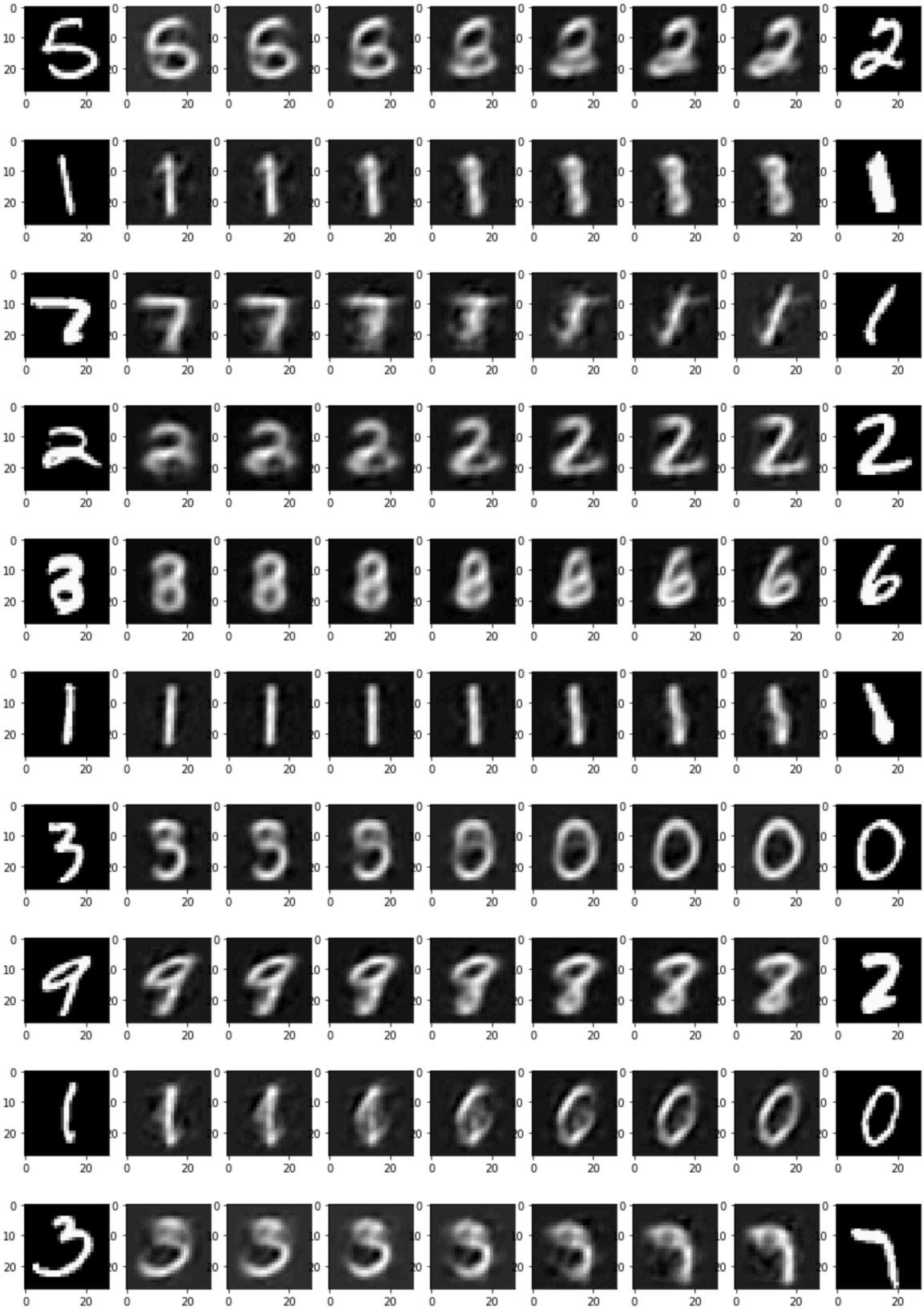
1 Same digit Interpolates



2 Different digit Interpolates



3 Code

Homework 9: Variational Autoencoders

About

Due

Monday 4/22/19, 11:59 PM CST

Goal

This homework focuses on creating variational autoencoders applied to the MNIST dataset.

Dev Environment

Working on Google Colab

You may choose to work locally or on Google Colaboratory. You have access to free compute through this service.

1. Visit https://colab.research.google.com/drive
2. Navigate to the **`Upload`** tab, and upload your `HW10.ipynb`
3. Now on the top right corner, under the `Comment` and `Share` options, you should see a `Connect` option. Once you are connected, you will have access to a VM with 12GB RAM, 50 GB disk space and a single GPU. The dropdown menu will allow you to connect to a local runtime as well.

Notes:

- **If you do not have a working setup for Python 3, this is your best bet. It will also save you from heavy installations like `tensorflow` if you don't want to deal with those.**
- ***There is a downside*. You can only use this instance for a single 12-hour stretch, after which your data will be deleted, and you would have redownload all your datasets, any libraries not already on the VM, and regenerate your logs**.

Installing PyTorch and Dependencies

The instructions for installing and setting up PyTorch can be found at https://pytorch.org/get-started/locally/. Make sure you follow the instructions for your machine. For any of the remaining libraries used in this assignment:

- We have provided a `hw8_requirements.txt` file on the homework web page.
- Download this file, and in the same directory you can run `pip3 install -r hw8_requirements.txt`

Check that PyTorch installed correctly by running the following:

In [1]:

```
import torch
```

```
torch.rand(5, 3)
```

Out[1]:

```
tensor([[0.3107, 0.3403, 0.9030],
        [0.3666, 0.2765, 0.4050],
        [0.9817, 0.2669, 0.7267],
        [0.6591, 0.7676, 0.4878],
        [0.6755, 0.2200, 0.9744]])
```

The output should look something like

```
tensor([[0.3380, 0.3845, 0.3217],

        [0.8337, 0.9050, 0.2650],

        [0.2979, 0.7141, 0.9069],

        [0.1449, 0.1132, 0.1375],

        [0.4675, 0.3947, 0.1426]])
```

Let's get started with the assignment.

Instructions

Part 1 - Datasets and Dataloaders

This part of the assignment is similar to HW 8.

Create a directory named hw9_data with the following command.

In [2]:

```
!mkdir hw9_data
```

Now use `torch.datasets.MNIST` to load the Train and Test data into `hw9_data`.

- **Use the directory you created above as the `root` directory for your datasets**
- **Populate the `transformations` variable with any transformations you would like to perform on your data.** (Hint: You will need to do at least one)
- **Pass your `transformations` variable to `torch.datasets.MNIST`. This allows you to perform arbitrary transformations to your data at loading time.**

In [3]:

```
from torchvision import datasets, transforms

## YOUR CODE HERE ##
transformations = transforms.Compose([transforms.ToTensor()])
mnist_train = datasets.MNIST(root='./hw9_data',train=True,download=True,transfo
rm=transformations)
mnist_test = datasets.MNIST(root='./hw9_data',train=False,download=True,transfo
rm=transformations)
0it [00:00, ?it/s]
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./hw
9_data/MNIST/raw/train-images-idx3-ubyte.gz
 98%|[redacted]| 9732096/9912422 [00:11<00:00, 2623780.04it/s]
Extracting ./hw9_data/MNIST/raw/train-images-idx3-ubyte.gz
0it [00:00, ?it/s]
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./hw
9_data/MNIST/raw/train-labels-idx1-ubyte.gz
  0%|          | 0/28881 [00:00<?, ?it/s]
 57%|[redacted]     | 16384/28881 [00:00<00:00, 53968.44it/s]
32768it [00:00, 37085.20it/s]
0it [00:00, ?it/s]
Extracting ./hw9_data/MNIST/raw/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./hw9
_data/MNIST/raw/t10k-images-idx3-ubyte.gz
  0%|          | 0/1648877 [00:00<?, ?it/s]
  1%|          | 16384/1648877 [00:00<00:30, 52681.17it/s]
  2%|[redacted]         | 40960/1648877 [00:01<00:25, 62743.01it/s]
  6%|[redacted]          | 98304/1648877 [00:01<00:19, 81341.85it/s]
  9%|[redacted]          | 155648/1648877 [00:01<00:14, 102891.30it/s]
 13%|[redacted]         | 212992/1648877 [00:01<00:11, 126321.62it/s]
 16%|[redacted]         | 270336/1648877 [00:01<00:09, 150066.30it/s]
 20%|[redacted]         | 335872/1648877 [00:02<00:07, 175063.70it/s]
 24%|[redacted]        | 401408/1648877 [00:02<00:06, 201463.07it/s]
 28%|[redacted]        | 466944/1648877 [00:02<00:05, 225119.55it/s]
 32%|[redacted]       | 532480/1648877 [00:02<00:04, 261786.70it/s]
 37%|[redacted]       | 606208/1648877 [00:02<00:03, 324277.39it/s]
 40%|[redacted]       | 655360/1648877 [00:03<00:03, 298799.55it/s]
 42%|[redacted]      | 696320/1648877 [00:03<00:03, 263902.56it/s]
 47%|[redacted]      | 770048/1648877 [00:03<00:02, 307857.65it/s]
 52%|[redacted]     | 851968/1648877 [00:03<00:02, 377266.53it/s]
 55%|[redacted]     | 909312/1648877 [00:03<00:02, 340015.53it/s]
 58%|[redacted]     | 958464/1648877 [00:03<00:02, 321866.85it/s]
 64%|[redacted]    | 1048576/1648877 [00:04<00:01, 370289.64it/s]
 67%|[redacted]    | 1097728/1648877 [00:04<00:01, 393349.78it/s]
 70%|[redacted]    | 1146880/1648877 [00:04<00:01, 409950.15it/s]
 73%|[redacted]   | 1196032/1648877 [00:04<00:01, 423171.11it/s]
 76%|[redacted]   | 1253376/1648877 [00:04<00:00, 451525.98it/s]
 79%|[redacted]   | 1302528/1648877 [00:04<00:00, 459325.64it/s]
 82%|[redacted] | 1359872/1648877 [00:04<00:00, 488121.76it/s]
 86%|[redacted]  | 1417216/1648877 [00:04<00:00, 496073.30it/s]
 90%|[redacted] | 1482752/1648877 [00:04<00:00, 527349.97it/s]
 93%|[redacted]| 1540096/1648877 [00:05<00:00, 529585.06it/s]
 97%|[redacted]| 1605632/1648877 [00:05<00:00, 560881.52it/s]
1654784it [00:05, 316396.19it/s]
0it [00:00, ?it/s]
Extracting ./hw9_data/MNIST/raw/t10k-images-idx3-ubyte.gz
```

```
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./hw9
_data/MNIST/raw/t10k-labels-idx1-ubyte.gz
  0%|          | 0/4542 [00:00<?, ?it/s]
8192it [00:00, 17965.86it/s]
Extracting ./hw9_data/MNIST/raw/t10k-labels-idx1-ubyte.gz
Processing...
Done!
```

Any file in our dataset will now be read at runtime, and the specified transformations we need on it will be applied when we need it..

We could iterate through these directly using a loop, but this is not idiomatic. PyTorch provides us with this abstraction in the form of `DataLoaders`. The module of interest is `torch.utils.data.DataLoader`.

`DataLoader` allows us to do lots of useful things

- Group our data into batches
- Shuffle our data
- Load the data in parallel using `multiprocessing` workers

Use `DataLoader` to create a loader for the training set and one for the testing set

- **Use a `batch_size` of 32 to start, you may change it if you wish.**
- **Set the `shuffle` parameter to `True`.**

Check that the data was loaded successfully before proceeding to the next sections.

In [4]:

```
from torch.utils.data import DataLoader

## YOUR CODE HERE ##
train_loader = DataLoader(mnist_train, batch_size=100, shuffle=True, num_worker
s=1)
test_loader = DataLoader(mnist_test, batch_size=100, shuffle=True, num_workers=
1)
```

Part 2 - Encoder and Decoders (0 points)

In this section we will be creating the encoder and decoder for our variational autoencoder (VAE).

You can take a look at the following to understand how VAE's work.

- https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf
- http://kvfrans.com/variational-autoencoders-explained/
- https://jmetzen.github.io/2015-11-27/vae.html

VAEs work around a latent space who's dimension can be chosen by us. We will leave this as a parameter for the Encoder and Decoder classes that you will have to populate.

Feel free to use any network architecture that you wish. Try simpler network structures like a few linear layers before trying anything more complicated.

For the Encoder:

- **Finish the init() function.**
- **Finish the forward() function.**
- **Assume that input to forward, x, is of shape (batch_size, 28,28)**
- **forward() should return two tensors of size latent_dim like a standard encoder of a VAE**
- **One of the tensors should correspond to the mean of the encoding and the other tensor should correspond to the variance. In practice, it is easier to model the output as the log of the variance (logvar) and we will too**

For the Decoder:

- **Finish the init() function.**
- **Finish the forward() function.**
- **Assume that input to forward, x, is of shape (batch_size, latent_dim)**
- **forward() should return a tensor of shape (batch_size, 28,28)**
- **Make sure that the output lies in the same range as the input to the encoder (Hint: Sigmoid?)**

In [5]:

```
from torch import nn
class Encoder(nn.Module):
    def __init__(self, latent_dim):
        super(Encoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(784, 400),
            nn.ReLU()
            )
        self.em = nn.Linear(400,latent_dim)   # mu layer
        self.ev = nn.Linear(400,latent_dim)   # logvariance layer

    def forward(self, x):
        out = self.encoder(x)
        mu = self.em(out)
        logvar = self.ev(out)
        return mu,logvar

class Decoder(nn.Module):
    def __init__(self, latent_dim):
        super(Decoder, self).__init__()
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 400),
            nn.ReLU(),
            nn.Linear(400, 784)
```

```
            )
    def forward(self,x):
        x_hat = self.decoder(x)
        return x_hat
```

Part 3: Training and loss functions (0 points)

Recall that the encoder outputs the mean (mu) and the log of the variance (logvar). This implies that the latent vector of the input image follows a gaussian distribution with mean (mu) and standard deviation (e^[0.5*logvar]). To decode this information, the decoder needs a sample from this distribution.

Complete the sample function to generate these samples

In [6]:

```
def sample(mu, logvar):
    std = logvar.mul(0.5).exp_()
    eps = torch.FloatTensor(std.size()).normal_()
    eps = torch.autograd.Variable(eps)
    return eps.mul(std).add_(mu)
```

We also need to create the loss function. Assume that x are your input images and x_hat are your reconstructions of these input images, complete the following loss for a VAE. (Hint: You will need to use mu and logvar as well)

In [7]:

```
def vae_loss(x, x_hat, mu, logvar):
    reconstruction_function = nn.MSELoss(size_average=False)
    BCE = reconstruction_function(x_hat, x)
    KLD = mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).add_(logvar)
    KLD = torch.sum(KLD).mul_(-0.5)
    return BCE + KLD
```

In the following we will instantiate an Encoder and Decoder with a latent dimension of 32.

We also define a single optimizer that optimizes the parameters of both the Encoder and the Decoder together. Feel free to use any optimizer of your choice.

In [8]:

```
from torch import optim

## YOUR CODE HERE ##
encoder = Encoder(20)
decoder = Decoder(20)
params = list(encoder.parameters())+list(decoder.parameters())
optimizer = optim.Adam(params, lr=1e-3)
```

Complete the train function that takes input encoder, decoder, train_loader, optimizer, and number of epochs you wish to train your model for.

Training will involve:

1. **One epoch is defined as a full pass of your dataset through your model. We choose the number of epochs we wish to train our model for.**
2. **For each batch, use the encoder to generate the mu and logvar.**

3. **Sample a latent vector for each image in the batch and feed this to the decoder to generate the decoded images.**
4. **Calculate the loss function for this batch.**
5. **Now calculate the gradients for each parameter you are optimizing over. (Hint: Your loss function object can do this for you)**
6. **Update your model parameters (Hint: The optimizer comes in here)**
7. **Set the gradients in your model to zero for the next batch.**

In [9]:

```
def train(encoder, decoder, train_loader, optimizer, num_epochs = 10):
    for epoch in range(num_epochs):
        train_loss = 0
        for batch_idx, data in enumerate(train_loader):
            img, _ = data #Extract image
            img = img.view(img.size(0), -1) #Re-Size the Image
            img = torch.autograd.Variable(img) #Variable the Image
            optimizer.zero_grad() #Making Zero for all the Gradient

            mu, logvar = encoder(img)
            z = sample(mu, logvar)
            x_hat=decoder(z)

            loss = vae_loss(x_hat, img, mu, logvar)
            loss.backward()
            train_loss += loss.data
            optimizer.step()
            if batch_idx % 100 == 0:
                print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                    epoch,
                    batch_idx * len(img),
                    len(train_loader.dataset), 100. * batch_idx / len(train_loa
der),
                    loss.data / len(img)))
        print('====> Epoch: {} Average loss: {:.4f}'.format(
            epoch, train_loss / len(train_loader.dataset)))
```

Finally call train with the relevant parameters.

Note : This function may take a while to complete if you're training for many epochs on a cpu. This is where it comes in handy to be running on Google Colab, or just have a GPU on hand.

In [10]:

```
train(encoder, decoder, train_loader, optimizer, num_epochs = 10)
/Users/sushanta/anaconda3/lib/python3.6/site-packages/torch/nn/_reduction.py:49
: UserWarning: size_average and reduce args will be deprecated, please use redu
ction='sum' instead.
  warnings.warn(warning.format(ret))
Train Epoch: 0 [0/60000 (0%)]  Loss: 132.178009
Train Epoch: 0 [10000/60000 (17%)]    Loss: 43.739262
Train Epoch: 0 [20000/60000 (33%)]    Loss: 40.663799
Train Epoch: 0 [30000/60000 (50%)]    Loss: 38.585915
```

```
Train Epoch: 0 [40000/60000 (67%)]    Loss: 40.661022
Train Epoch: 0 [50000/60000 (83%)]    Loss: 36.237770
====> Epoch: 0 Average loss: 41.4419
Train Epoch: 1 [0/60000 (0%)] Loss: 36.198639
Train Epoch: 1 [10000/60000 (17%)]    Loss: 35.159542
Train Epoch: 1 [20000/60000 (33%)]    Loss: 36.147301
Train Epoch: 1 [30000/60000 (50%)]    Loss: 35.252235
Train Epoch: 1 [40000/60000 (67%)]    Loss: 33.608196
Train Epoch: 1 [50000/60000 (83%)]    Loss: 35.908756
====> Epoch: 1 Average loss: 35.0167
Train Epoch: 2 [0/60000 (0%)] Loss: 33.541672
9920512it [00:30, 2623780.04it/s]
Train Epoch: 2 [10000/60000 (17%)]    Loss: 32.694824
Train Epoch: 2 [20000/60000 (33%)]    Loss: 34.187637
Train Epoch: 2 [30000/60000 (50%)]    Loss: 32.953907
Train Epoch: 2 [40000/60000 (67%)]    Loss: 34.721653
Train Epoch: 2 [50000/60000 (83%)]    Loss: 34.795624
====> Epoch: 2 Average loss: 33.9908
Train Epoch: 3 [0/60000 (0%)] Loss: 32.882137
Train Epoch: 3 [10000/60000 (17%)]    Loss: 32.848618
Train Epoch: 3 [20000/60000 (33%)]    Loss: 33.019115
Train Epoch: 3 [30000/60000 (50%)]    Loss: 34.161644
Train Epoch: 3 [40000/60000 (67%)]    Loss: 34.738590
Train Epoch: 3 [50000/60000 (83%)]    Loss: 33.969429
====> Epoch: 3 Average loss: 33.3914
Train Epoch: 4 [0/60000 (0%)] Loss: 33.041035
Train Epoch: 4 [10000/60000 (17%)]    Loss: 34.164875
Train Epoch: 4 [20000/60000 (33%)]    Loss: 34.213131
Train Epoch: 4 [30000/60000 (50%)]    Loss: 33.009594
Train Epoch: 4 [40000/60000 (67%)]    Loss: 32.651241
Train Epoch: 4 [50000/60000 (83%)]    Loss: 31.368668
====> Epoch: 4 Average loss: 32.9518
Train Epoch: 5 [0/60000 (0%)] Loss: 33.879459
Train Epoch: 5 [10000/60000 (17%)]    Loss: 32.795914
Train Epoch: 5 [20000/60000 (33%)]    Loss: 31.573969
Train Epoch: 5 [30000/60000 (50%)]    Loss: 32.813580
Train Epoch: 5 [40000/60000 (67%)]    Loss: 32.360184
Train Epoch: 5 [50000/60000 (83%)]    Loss: 33.120499
====> Epoch: 5 Average loss: 32.6001
Train Epoch: 6 [0/60000 (0%)] Loss: 33.366531
Train Epoch: 6 [10000/60000 (17%)]    Loss: 32.090954
Train Epoch: 6 [20000/60000 (33%)]    Loss: 31.806473
Train Epoch: 6 [30000/60000 (50%)]    Loss: 32.521469
Train Epoch: 6 [40000/60000 (67%)]    Loss: 32.154099
Train Epoch: 6 [50000/60000 (83%)]    Loss: 30.545992
====> Epoch: 6 Average loss: 32.3410
Train Epoch: 7 [0/60000 (0%)] Loss: 31.471373
```

```
Train Epoch: 7 [10000/60000 (17%)]    Loss: 32.185341
Train Epoch: 7 [20000/60000 (33%)]    Loss: 31.739243
Train Epoch: 7 [30000/60000 (50%)]    Loss: 32.678883
Train Epoch: 7 [40000/60000 (67%)]    Loss: 32.802925
Train Epoch: 7 [50000/60000 (83%)]    Loss: 32.281803
====> Epoch: 7 Average loss: 32.1217
Train Epoch: 8 [0/60000 (0%)]  Loss: 31.157930
Train Epoch: 8 [10000/60000 (17%)]    Loss: 32.228493
Train Epoch: 8 [20000/60000 (33%)]    Loss: 31.751133
Train Epoch: 8 [30000/60000 (50%)]    Loss: 31.654814
Train Epoch: 8 [40000/60000 (67%)]    Loss: 31.871538
Train Epoch: 8 [50000/60000 (83%)]    Loss: 33.077606
====> Epoch: 8 Average loss: 31.9796
Train Epoch: 9 [0/60000 (0%)]  Loss: 31.155523
Train Epoch: 9 [10000/60000 (17%)]    Loss: 32.256847
Train Epoch: 9 [20000/60000 (33%)]    Loss: 31.256075
Train Epoch: 9 [30000/60000 (50%)]    Loss: 32.240906
Train Epoch: 9 [40000/60000 (67%)]    Loss: 31.891096
Train Epoch: 9 [50000/60000 (83%)]    Loss: 32.404610
====> Epoch: 9 Average loss: 31.8042
```

Part 4: Visualizing the VAE output (90 points)

We will look at how well the codes produced by the VAE can be interpolated. **For this section we will only use the MNIST test set.**

To create an interpolation between two images A and B, we encode both these images and generate a sample code for each of them. We now consider 7 equally spaced points in between these two sample codes giving us a total of 9 points including the samples. We then decode these images to get interpolated images in between A and B.

Complete the interpolation function below that takes a pair of images A and B and returns 9 images. (You are free to use any data structure you want to return these images)

In [11]:

```
import matplotlib.pyplot as plt
from torchvision import utils
%matplotlib inline
import numpy as np

def create_interpolates(A, B, encoder, decoder):
    IMG_DECODER = np.zeros((784,9))

    IMG_DECODER[:,0]=A
    IMG_DECODER[:,8]=B

    mu, logvar = encoder(A)
    z_A = sample(mu, logvar)
    mu, logvar = encoder(B)
```

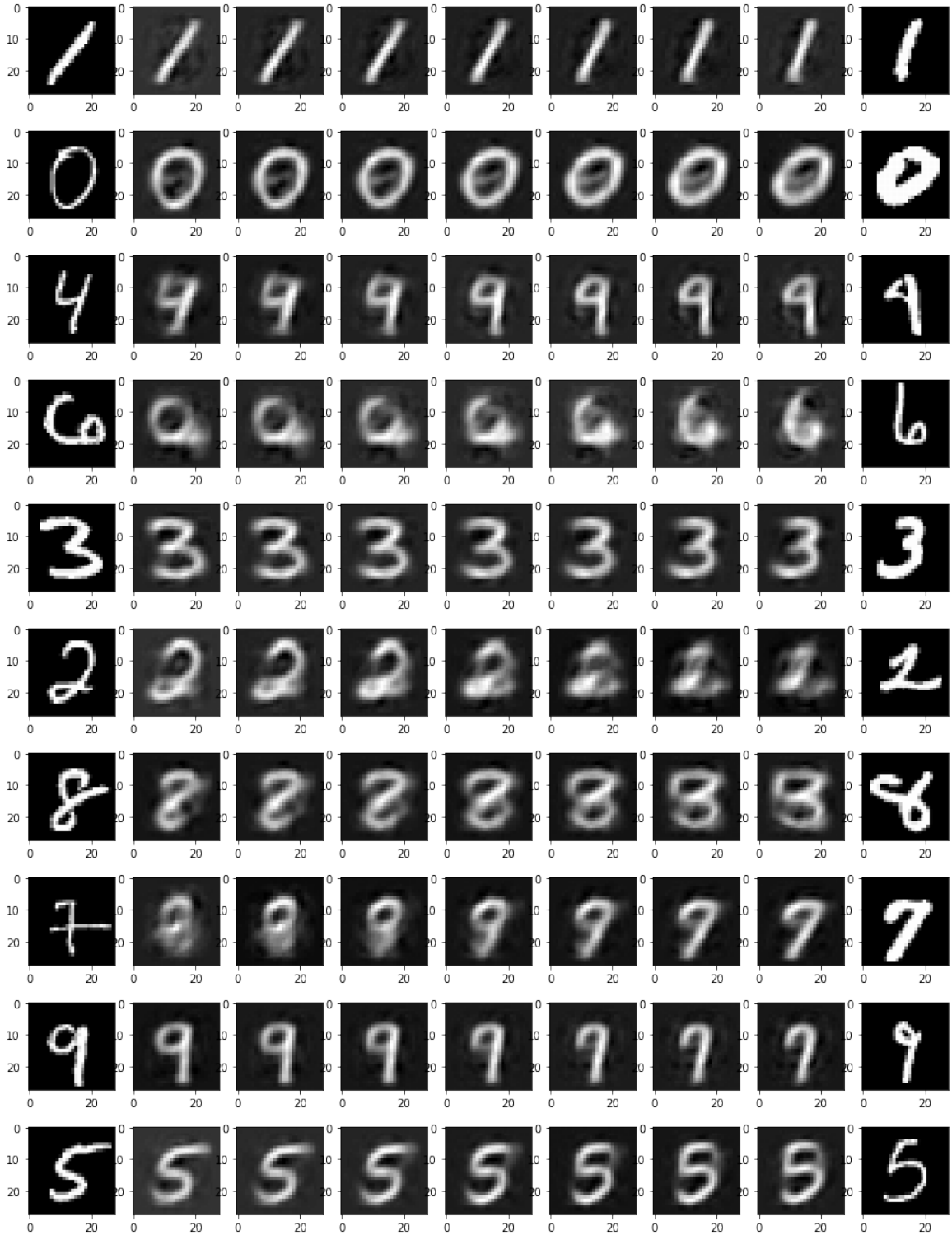
```
    z_B = sample(mu, logvar)
    Z_I = torch.zeros((20,7))

    for i in range(7):
        I = z_A + ((z_B - z_A)*((i+1)/8))
        Z_I[:,i]=I

    for i in range(7):
        img = decoder(Z_I[:,i])
        IMG_DECODER[:,i+1] = img.detach().numpy()

    fig, ax  = plt.subplots(1, 9,squeeze=False)
    fig.set_size_inches(15, 2)
    for i in range(1):
        for j in range(9):
            ax[i][j].imshow(IMG_DECODER[:,j].reshape(28,28), cmap='gray')

    return IMG_DECODER
```

For 10 pairs of MNIST test images of the same digit (1 pair for "0", 1 pair for "1", etc.), selected at random, compute the code for each image of the pair. Now compute 7 evenly spaced linear interpolates between these codes, and decode the result into images. Prepare a figure showing this interpolate. Lay out the figure so each interpolate is a row. On the left of the row is the first test image; then the interpolate closest to it; etc; to the last test image. You should have a 10 rows (1 row per digit) and 9 columns (7 interpolates + 2 selected test images) of images. (45 points)

In [12]:

```
similar_pairs = {}
for _, (x, y) in enumerate(test_loader):
  for i in range(len(y)):
    if y[i].item() not in similar_pairs:
      similar_pairs[y[i].item()] = []
    if len(similar_pairs[y[i].item()])<2:
      similar_pairs[y[i].item()].append(x[i])

  done = True
  for i in range(10):
    if i not in similar_pairs or len(similar_pairs[i])<2:
      done = False

  if done:
    break

# similar_pairs[i] contains two images indexed at 0 and 1 that have images of t
he digit i

for k in similar_pairs.keys():
    A = similar_pairs[k][0].view(-1,)
    B = similar_pairs[k][1].view(-1,)
```

```
create_interpolates(A, B, encoder, decoder)
```



For 10 pairs of MNIST test images, selected at random, compute the code for each image of the pair. Now compute 7 evenly spaced linear interpolates between these codes, and decode the result into images. Prepare a figure showing this interpolate. Lay out the figure so each interpolate is a row. On the left of the row is the first test image; then the interpolate closest to it; etc; to the last test image. You should have a 10 rows and 9 columns of images. (45 points)

In [13]:

```
random_pairs = {}
for _, (x, y) in enumerate(test_loader):
```

```
  # Make sure the batch size is greater than 20
  for i in range(10):
    random_pairs[i] = []
    random_pairs[i].append(x[2*i])
    random_pairs[i].append(x[2*i+1])
  break

# random_pairs[i] contains two images indexed at 0 and 1 that are chosen at ran
dom

for k in random_pairs.keys():
    A = random_pairs[k][0].view(-1,)
    B = random_pairs[k][1].view(-1,)
    create_interpolates(A, B, encoder, decoder)
```