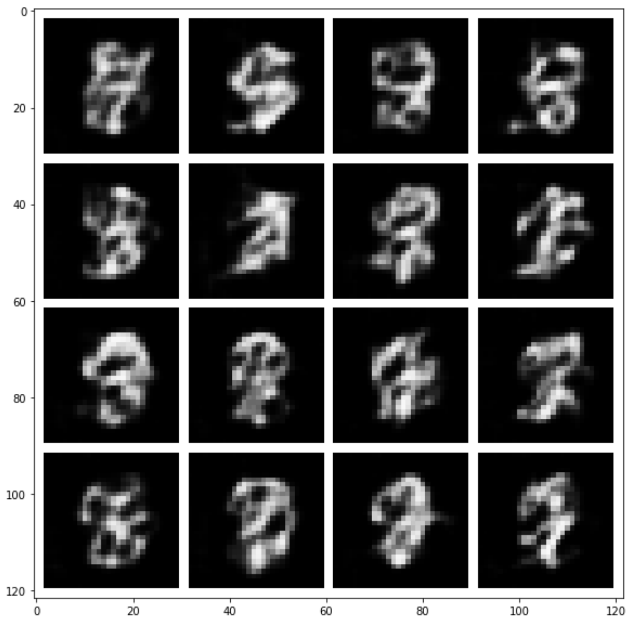
Question 1: AE-Sampling

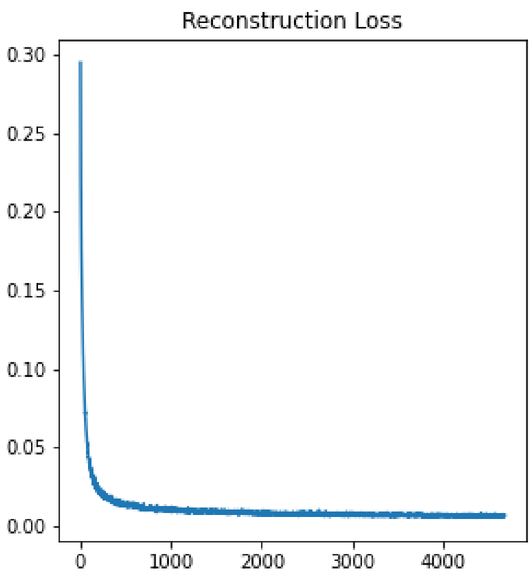


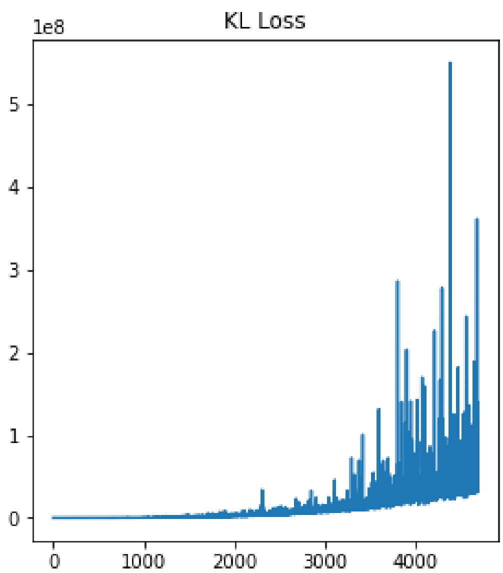
> **Inline Question: Describe your observations, why do you think they occur? [2pt]** \ (please limit your answer to <150 words) \ **Answer:**

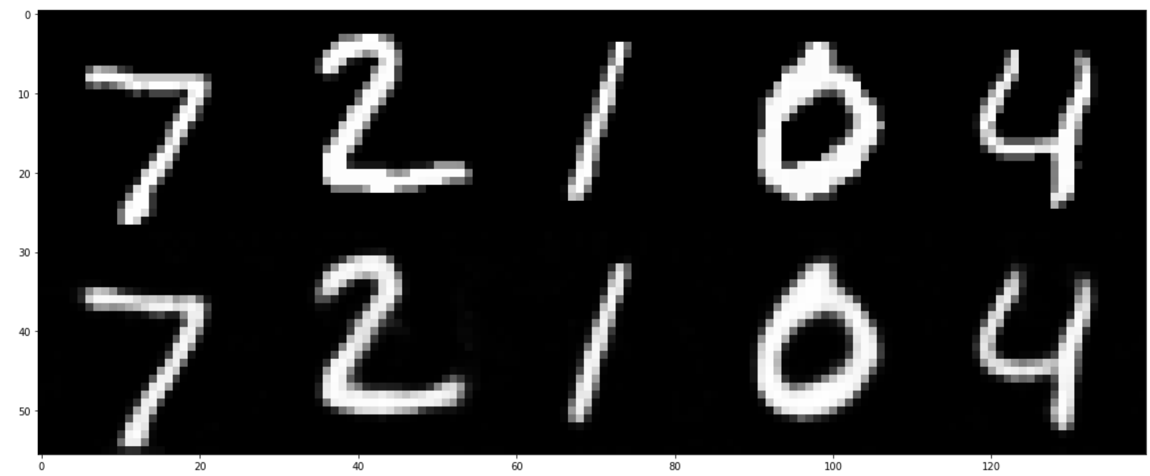
The decoding model has learned the reresentation of the MNIST dataset in such a way that it can reconstruct the original image from this reduced dimensional space with minimal error. Here it is trying to reconstruct the random values to a digit, and we can see that it resembles a few digits, like 8 and 3. These are the representations it has learned.

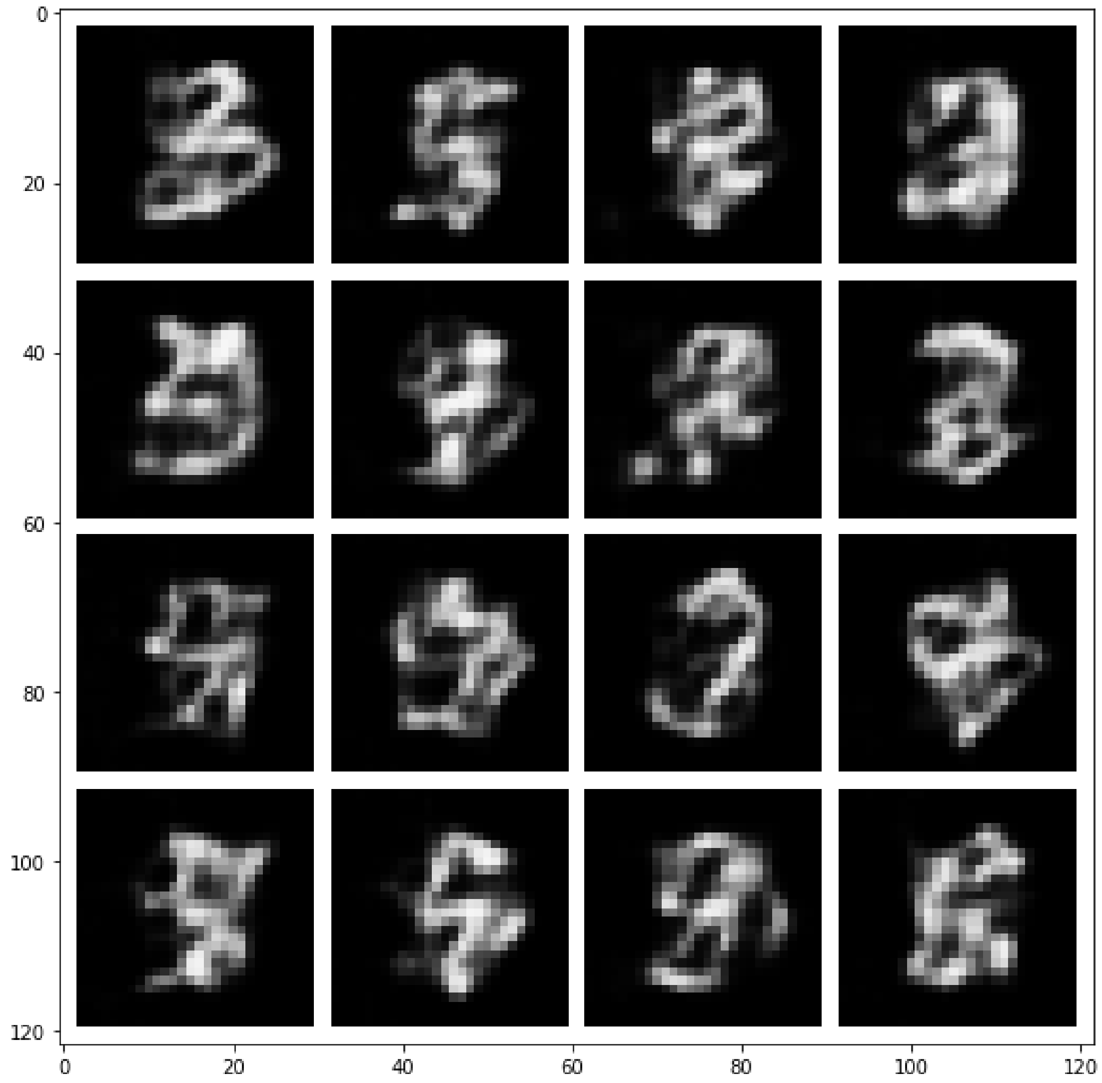
Question 2:

B = 0:

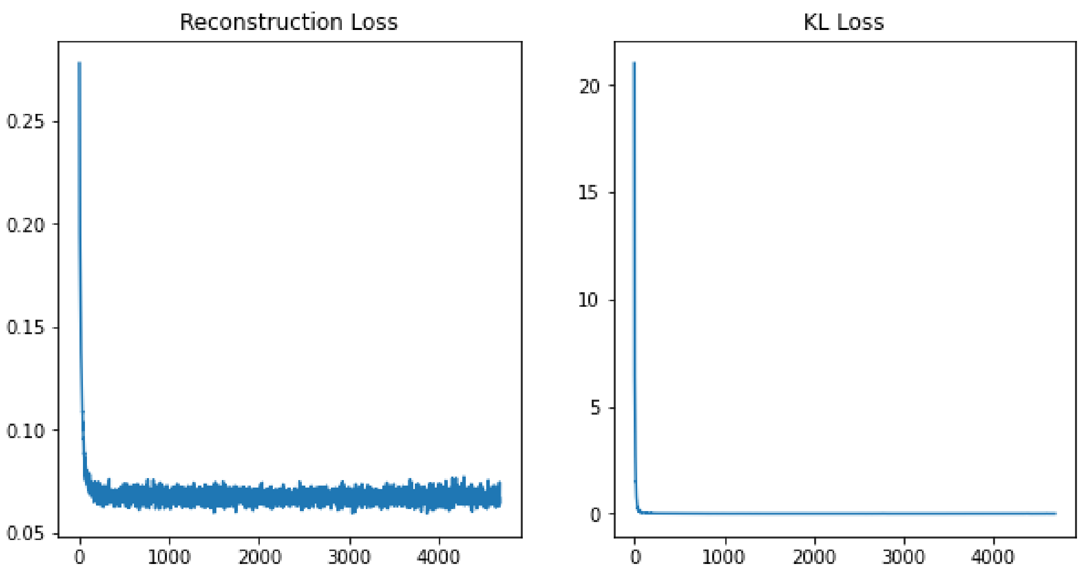




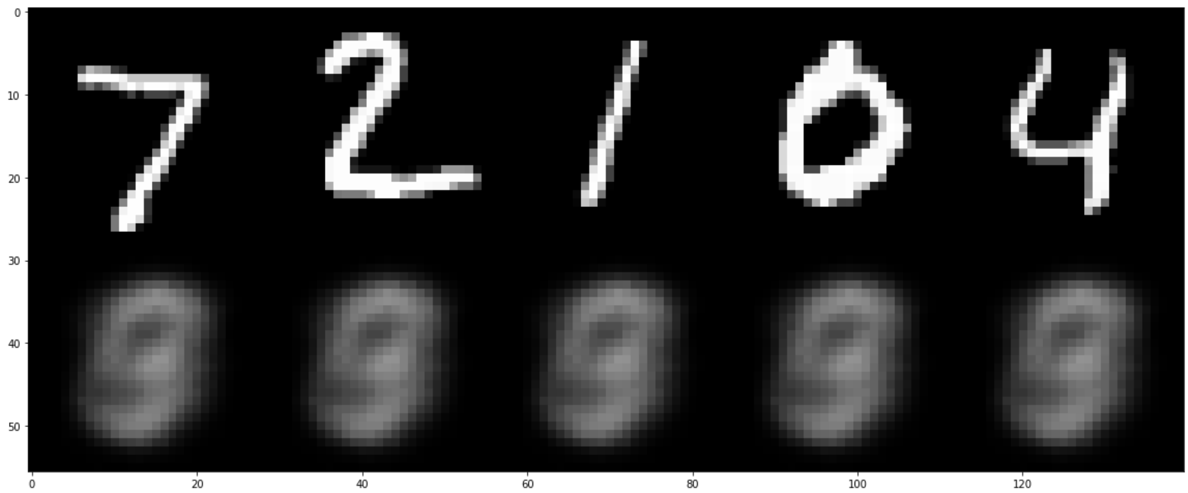




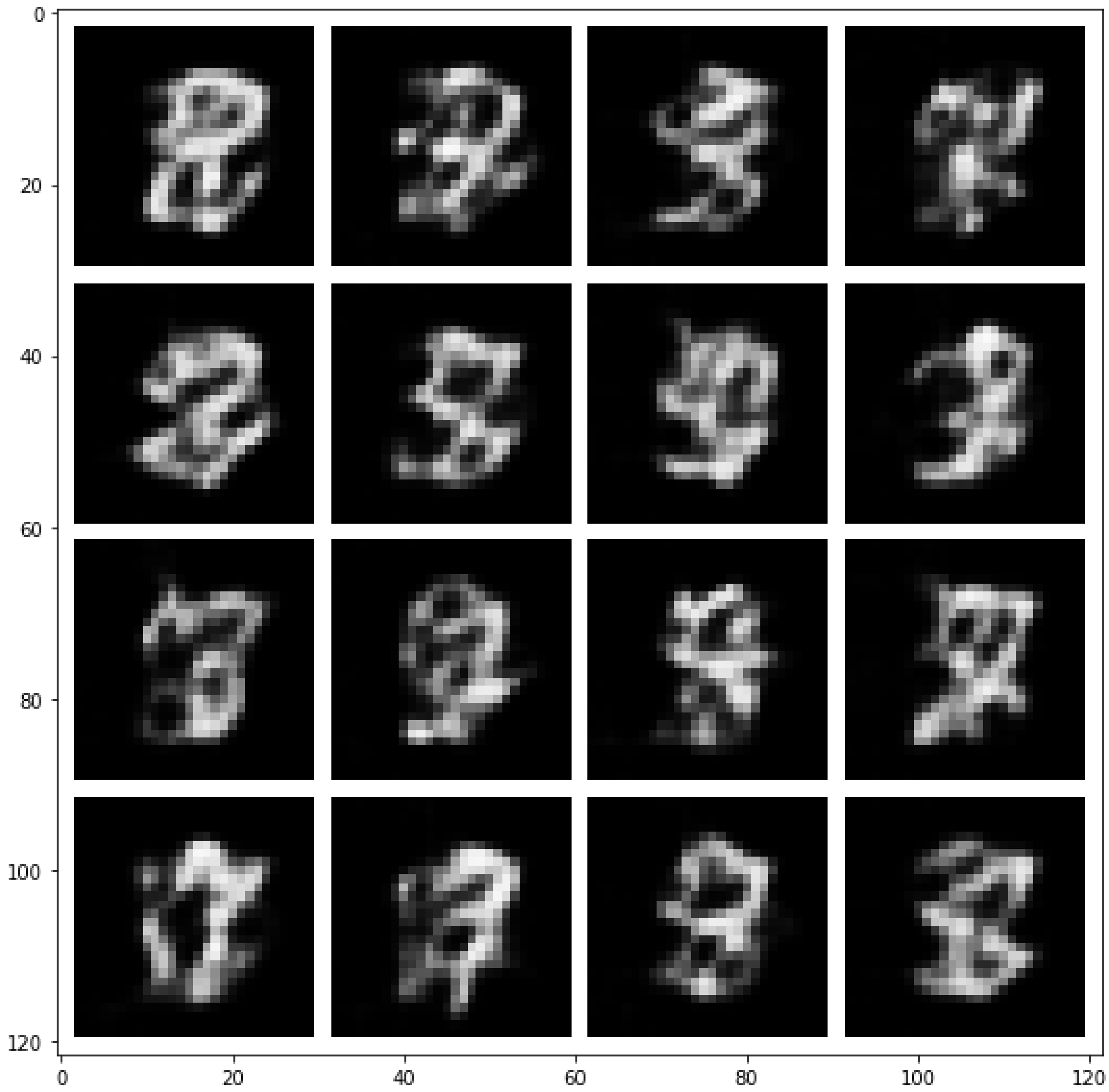
B = 10:



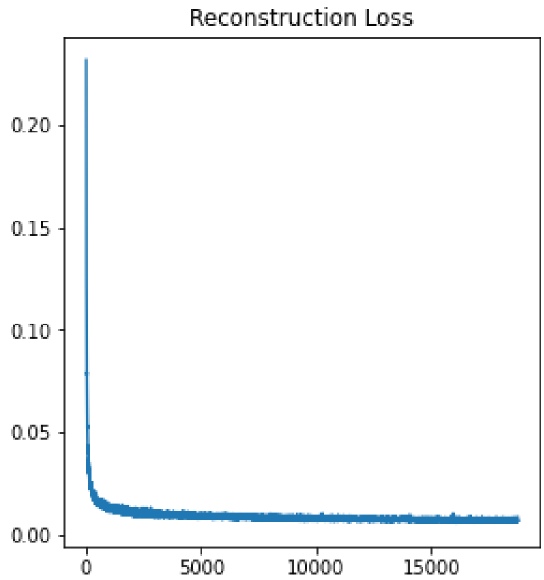
Reconstruction Loss
0.25
0.20
0.15
0.10
0.05
0
1000
2000
3000
4000
KL Loss
20
15
10
5
0
0
1000
2000
3000
4000

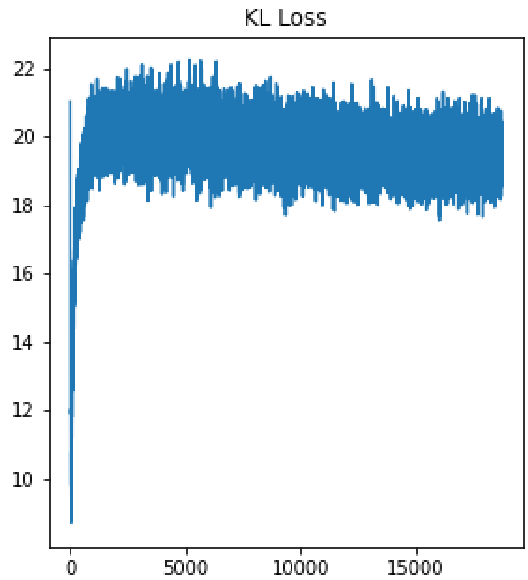


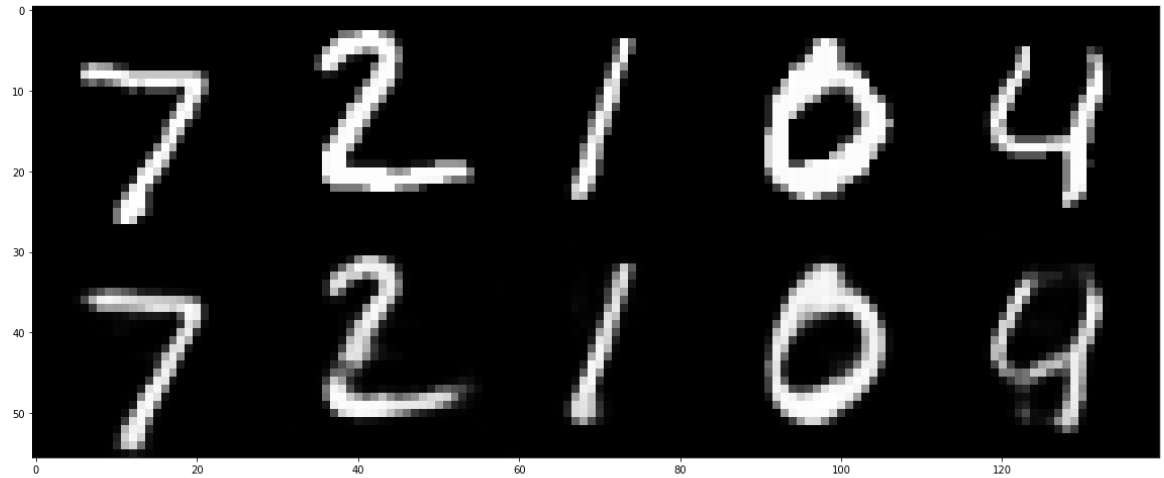
0
10
20
30
40
50
0
20
40
60
80
100
120

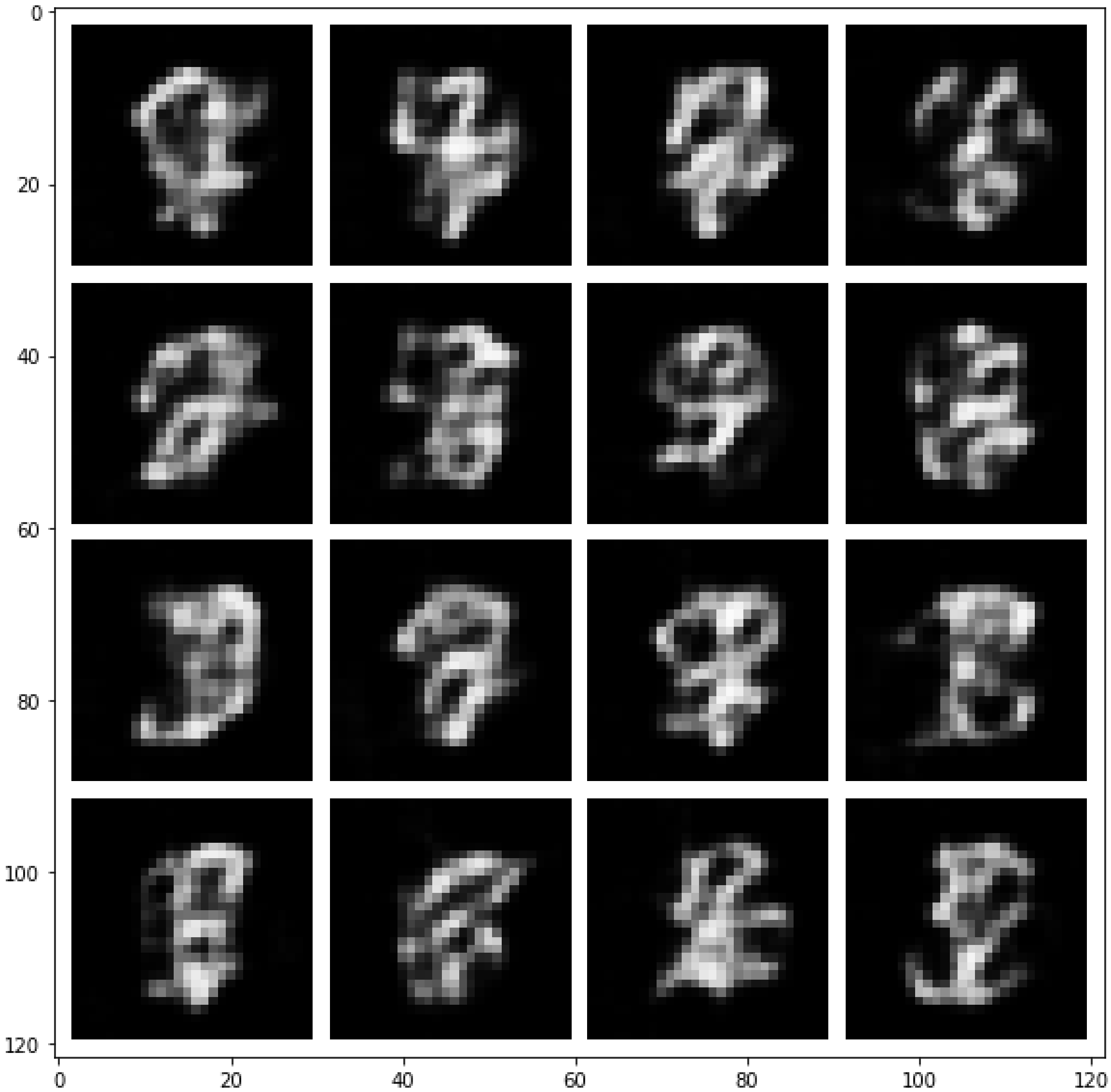


B=0.0001 (opt):









Question 3: VAE inline questions

> **Inline Question: What can you observe when setting $\beta = 0$? Explain your observations! [3pt]** \ (please limit your answer to <150 words) \ **Answer:**

+ Code + Markdown

When B=0, we use only logp(x|z) and ignore the prior divergence. We only use the reconstruction error. Hence we get the image shown above, without any disentanglement.

Let's repeat the same experiment for $\beta = 10$, a very high value for the coefficient. You can modify the β value in the cell above and rerun it (it is okay to overwrite the outputs of the previous experiment, but **make sure to copy the visualizations of training curves, reconstructions and samples for $\beta = 0$ into your solution PDF** before deleting them).

> **Inline Question: What can you observe when setting $\beta = 10$? Explain your observations! [3pt]** \ (please limit your answer to <200 words) \ **Answer**:

When B=10, we have a stronger contraint over the latent bottleneck. This greatly limits the representation capacity of z and results in further disentanglement, which in turn, results in the above image with very poor reconstruction. Optimal value exists between 0 and 10.

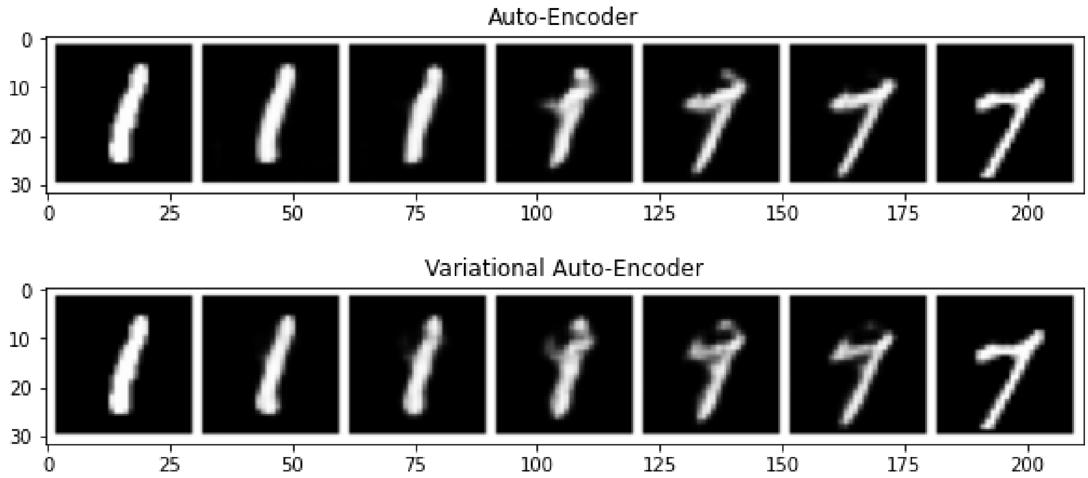
Now we can start tuning the beta value to achieve a good result. First describe what a "good result" would look like (focus what you would expect for reconstructions and sample quality).

> **Inline Question: Characterize what properties you would expect for reconstructions (1pt) and samples (2pt) of a well-tuned VAE! [3pt]** \ (please limit your answer to <200 words) \ **Answer**:

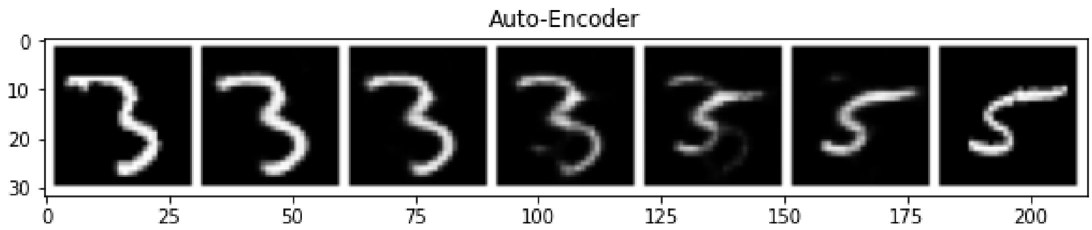
A well tuned B-VAE will learn the disentangled representations while still having less reconstruction errors. They will however have lesser interpretability of the latent space.

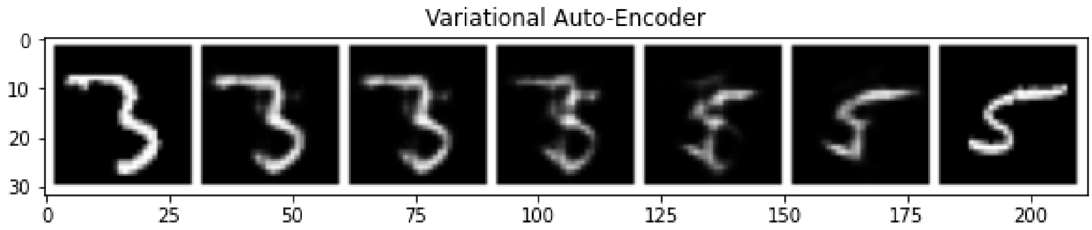
Question 4: Interpolation images

1-7:

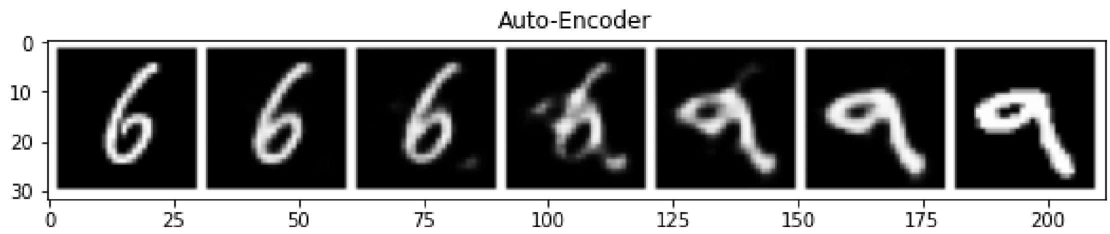


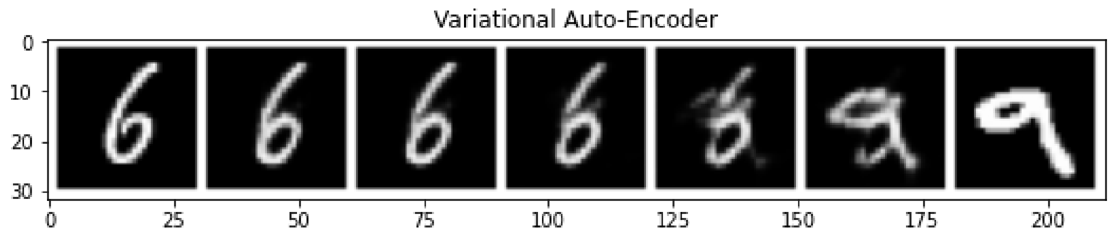
3-5:





6-9:





Question 5: Interpolation inline

Repeat the experiment for different start / end labels and different samples. Describe your observations.

> **Inline Question: Repeat the interpolation experiment with different start / end labels and multiple samples. Describe your observations! Focus on**: \

1. **How do AE and VAE embedding space interpolations differ?** \
2. **How do you expect these differences to affect the usefulness of the learned representation for downstream learning?** \ (please limit your answer to <300 words)

> **Answer**:

1. The latent space where the encoder encodes the input in an autoencoder may not be continuous and hence dont allow easy interpolation. On the other hand, the latent spaces of VAE are continous and allow random sampling.

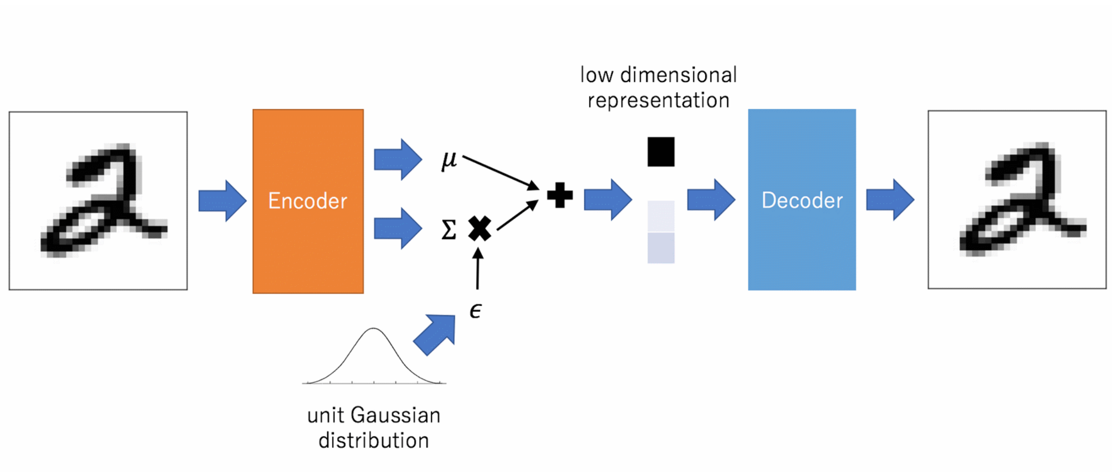
2. VAE learns both the mean and variance of the inputs in the latent space and hence can be used to generate new data from feeding random values to the decoder part. The autoencoder will try its best to generate an image which looks close to a digit, while the VAE can generate an image closer to a digit because it learns a distribution of possible latent inputs that can lead to the specific digit.

Hence, we can observe that the transition between the 2 classes is smoother in the case of VAE and very abrupt (you can clearly see 2 different digits in the case of 3&5) in the case of AE. This is due to the nature of the latent space.

Problem 1 - Variational Auto-Encoder (VAE)

Open in Colab

Variational Auto-Encoders (VAEs) are a widely used class of generative models. They are simple to implement and, in contrast to other generative model classes like Generative Adversarial Networks, they optimize an explicit maximum likelihood objective to train the model. Finally, their architecture makes them well-suited for unsupervised representation learning, i.e. learning low-dimensional representations of high-dimenionsal inputs, like images, with only self-supervised objectives (data reconstruction in the case of VAEs).



(image source: https://mlexplained.com/2017/12/28/an-intuitive-explanation-of-variational-autoencoders-vaes-part-1)

By working on this problem you will learn and practice the following steps:

1. Set up a data loading pipeline in PyTorch.
2. Implement, train and visualize an auto-encoder architecture.
3. Extend your implementation to a variational auto-encoder.
4. Learn how to tune the critical beta parameter of your VAE.
5. Inspect the learned representation of your VAE.

Note: For faster training of the models in this assignment you can use Colab with enabled GPU support. In Colab, navigate to "Runtime" --> "Change Runtime Type" and set the "Hardware Accelerator" to "GPU".

1. MNIST Dataset

We will perform all experiments for this problem using the MNIST dataset, a standard dataset of handwritten digits. The main benefits of this dataset are that it is small and relatively easy to model. It therefore allows for quick experimentation and serves as initial test bed in many papers.

Another benefit is that it is so widely used that PyTorch even provides functionality to automatically download it.

Let's start by downloading the data and visualizing some samples.

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

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-
modules-in-ipython
%load_ext autoreload
%autoreload 2

import torch
import torchvision

# this will automatically download the MNIST training set
mnist_train = torchvision.datasets.MNIST(root='./data',
                                         train=True,
                                         download=True,

transform=torchvision.transforms.ToTensor())
print("\n Download complete! Downloaded {} training
examples!".format(len(mnist_train)))
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-
ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-
ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
```

{"version_major":2,"version_minor":0,"model_id":"f9a8cedde07e4b39a6f1a
81d5c26a87d"}

```
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to
./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-
ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-
ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
```

{"version_major":2,"version_minor":0,"model_id":"b6cf64adc5e0467ca658b
6985b299629"}

```
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to
./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
```

{"version_major":2,"version_minor":0,"model_id":"6c39f62de6fb457dbf34272df7cefd2b"}

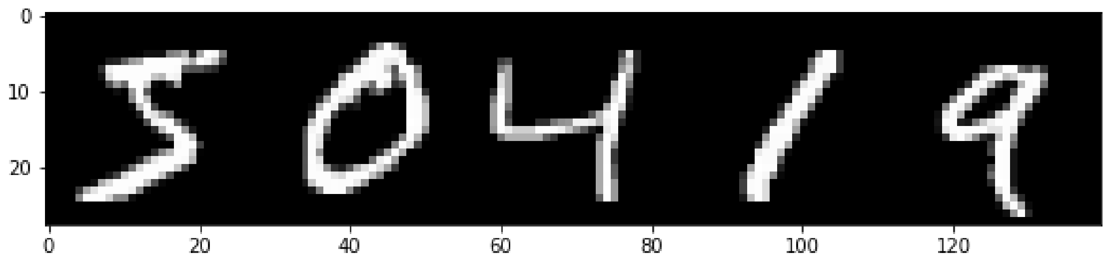
```
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to
./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
```

{"version_major":2,"version_minor":0,"model_id":"f8e33690c8174c998aac65af0d34a287"}

```
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to
./data/MNIST/raw


 Download complete! Downloaded 60000 training examples!
```

```
/opt/conda/lib/python3.7/site-packages/torchvision/datasets/
mnist.py:498: UserWarning: The given NumPy array is not writeable, and
PyTorch does not support non-writeable tensors. This means you can
write to the underlying (supposedly non-writeable) NumPy array using
the tensor. You may want to copy the array to protect its data or make
it writeable before converting it to a tensor. This type of warning
will be suppressed for the rest of this program. (Triggered internally
at  /usr/local/src/pytorch/torch/csrc/utils/tensor_numpy.cpp:174.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

```
import matplotlib.pyplot as plt
import numpy as np

# Let's display some of the training samples.
sample_images = []
mnist_it = iter(mnist_train)  # create simple iterator, later we will
use proper DataLoader
for _ in range(5):
  sample = next(mnist_it)     # samples a tuple (image, label)
  sample_images.append(sample[0][0].data.cpu().numpy())

fig = plt.figure(figsize = (10, 50))
ax1 = plt.subplot(111)
ax1.imshow(np.concatenate(sample_images, axis=1), cmap='gray')
plt.show()
```



2. Auto-Encoder

Before implementing the full VAE, we will first implement an **auto-encoder architecture**. Auto-encoders feature the same encoder-decoder architecture as VAEs and therefore also learn a low-dimensional representation of the input data without supervision. In contrast to VAEs they are **fully deterministic** models and do not employ variational inference for optimization.

The **architecture** is very simple: we will encode the input image into a low-dimensional representation using a convolutional network with strided convolutions that reduce the image resolution in every layer. This results in a low-dimensional representation of the input image. This representation will get decoded back into the dimensionality of the input image using a convolutional decoder network that mirrors the architecture of the encoder. It employs transposed convolutions to increase the resolution of its input in every layer. The whole model is trained by **minimizing a reconstruction loss** between the input and the decoded image.

Intuitively, the **auto-encoder needs to compress the information contained in the input image** into a much lower dimensional representation (e.g. 28x28=784px vs. 64 embedding dimensions for our MNIST model). This is possible since the information captured in the pixels is *highly redundant*. E.g. encoding an MNIST image requires <4 bits to encode which of the 10 possible digits is displayed and a few additional bits to capture information about shape and orientation. This is much less than the $255^{28 \cdot 28}$ bits of information that could be theoretically captured in the input image.

Learning such a **compressed representation can make downstream task learning easier**. For example, learning to add two numbers based on the inferred digits is much easier than performing the task based on two piles of pixel values that depict the digits.

In the following, we will first define the architecture of encoder and decoder and then train the auto-encoder model.

Defining the Auto-Encoder Architecture [6pt]

```
import torch.nn as nn

# Let's define encoder and decoder networks
######################################################################
# Encoder Architecture:                                              #
```

```
#   - Conv2d, hidden units: 32, output resolution: 14x14, kernel: 4 #
#   - LeakyReLU                                                     #
#   - Conv2d, hidden units: 64, output resolution: 7x7, kernel: 4   #
#   - BatchNorm2d                                                   #
#   - LeakyReLU                                                     #
#   - Conv2d, hidden units: 128, output resolution: 3x3, kernel: 3  #
#   - BatchNorm2d                                                   #
#   - LeakyReLU                                                     #
#   - Conv2d, hidden units: 256, output resolution: 1x1, kernel: 3  #
#   - BatchNorm2d                                                   #
#   - LeakyReLU                                                     #
#   - Flatten                                                       #
#   - Linear, output units: nz (= representation dimensionality)    #
####################################################################

class Encoder(nn.Module):
  def __init__(self, nz):
    super().__init__()
    ################################# TODO
#########################################
    # Create the network architecture using a nn.Sequential module
wrapper.        #
    # All convolutional layers should also learn a bias.
#
    # HINT: use the given information to compute stride and padding
#
    #       for each convolutional layer. Verify the shapes of
intermediate layers #
    #       by running partial networks (with the next cell) and
visualizing the   #
    #       output shapes.
#

######################################################################
##########
    self.net = nn.Sequential(
        nn.Conv2d(1, 32, 4, stride=2, padding=1, bias=True),
        nn.LeakyReLU(),
        nn.Conv2d(32, 64, 4, stride=2, padding=1, bias=True),
        nn.BatchNorm2d(64),
        nn.LeakyReLU(),
        nn.Conv2d(64, 128, 3, stride=5, padding=1, bias=True),
        nn.BatchNorm2d(128),
        nn.LeakyReLU(),
        nn.Conv2d(128, 256, 3, stride=5, padding=1, bias=True),
        nn.BatchNorm2d(256),
        nn.LeakyReLU(),
        nn.Flatten(),
        nn.Linear(256, nz)
        # add your network layers here
```

```
        # ...
    )
    ################################ END TODO
#######################################

  def forward(self, x):
    return self.net(x)


#####################################################################
# Decoder Architecture (mirrors encoder architecture):              #
#   - Linear, output units: 256                                     #
#   - Reshape, output shape: (256, 1, 1)                            #
#   - BatchNorm2d                                                   #
#   - LeakyReLU                                                     #
#   - ConvT2d, hidden units: 128, output resolution: 3x3, kernel: 3 #
#   - BatchNorm2d                                                   #
#   - LeakyReLU                                                     #
#   - ConvT2d, hidden units: 64, output resolution: 7x7, kernel: 3  #
#   - ...                                                           #
#   - ...                                                           #
#   - ConvT2d, output units: 1, output resolution: 28x28, kernel: 4 #
#   - Sigmoid (to limit output in range [0...1])                    #
#####################################################################

class Decoder(nn.Module):
  def __init__(self, nz):
    super().__init__()
    ################################# TODO
#########################################
    # Create the network architecture using a nn.Sequential module
wrapper.        #
    # Again, all (transposed) convolutional layers should also learn a
bias.       #
    # We need to separate the intial linear layer into a separate
variable since   #
    # nn.Sequential does not support reshaping. Instead the "Reshape"
is performed #
    # in the forward() function below and does not need to be added to
self.net    #
    # HINT: use the class nn.ConvTranspose2d for the transposed
convolutions.      #
    #       Verify the shapes of intermediate layers by running
partial networks   #
    #       (using the next cell) and visualizing the output shapes.
#

######################################################################
##########
    self.map = nn.Linear(64, 256)   # for initial Linear layer
```

```
    self.net = nn.Sequential(
        nn.BatchNorm2d(256),
        nn.LeakyReLU(),
        nn.ConvTranspose2d(256, 128, 3, bias=True),
        nn.BatchNorm2d(128),
        nn.LeakyReLU(),
        nn.ConvTranspose2d(128, 64, 3, stride=2, padding=0,
bias=True),
        nn.BatchNorm2d(64),
        nn.LeakyReLU(),
        nn.ConvTranspose2d(64, 1, 4, stride=4, padding=0, bias=True),
        nn.Sigmoid()
        # add your network layers here
        # ...
    )
    ################################ END TODO
#######################################

  def forward(self, x):
    return self.net(self.map(x).reshape(-1, 256, 1, 1))
```

Testing the Auto-Encoder Forward Pass [1pt]

```
# To test your encoder/decoder, let's encode/decode some sample images
# first, make a PyTorch DataLoader object to sample data batches
batch_size = 64
nworkers = 4        # number of wrokers used for efficient data
loading

####################################### TODO
#######################################
# Create a PyTorch DataLoader object for efficiently generating
training batches.  #
# Make sure that the data loader automatically shuffles the training
dataset.      #
# HINT: The DataLoader wraps the MNIST dataset class we created
earlier.           #
#       Use the given batch_size and number of data loading workers
when creating  #
#       the DataLoader.
#
######################################################################
##############
mnist_data_loader = torch.utils.data.DataLoader(mnist_train,
batch_size=batch_size, shuffle=True, num_workers=nworkers)
#################################### END TODO
#######################################

# now we can run a forward pass for encoder and decoder and check the
produced shapes
nz = 64          # dimensionality of the learned embedding
```

```
encoder = Encoder(nz)
decoder = Decoder(nz)
for sample_img, sample_label in mnist_data_loader:
  print(sample_img.shape)
  enc = encoder(sample_img)
  print("Shape of encoding vector (should be [batch_size, nz]):
{}".format(enc.shape))
  dec = decoder(enc)
  print("Shape of decoded image (should be [batch_size, 1, 28, 28]):
{}".format(dec.shape))
  break
```

```
/opt/conda/lib/python3.7/site-packages/torch/utils/data/
dataloader.py:481: UserWarning: This DataLoader will create 4 worker
processes in total. Our suggested max number of worker in current
system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get
DataLoader running slow or even freeze, lower the worker number to
avoid potential slowness/freeze if necessary.
  cpuset_checked))
```

```
torch.Size([64, 1, 28, 28])
Shape of encoding vector (should be [batch_size, nz]): torch.Size([64,
64])
Shape of decoded image (should be [batch_size, 1, 28, 28]):
torch.Size([64, 1, 28, 28])
```

Now that we defined encoder and decoder network our architecture is nearly complete. However, before we start training, we can wrap encoder and decoder into an auto-encoder class for easier handling.

```
class AutoEncoder(nn.Module):
  def __init__(self, nz):
    super().__init__()
    self.encoder = Encoder(nz)
    self.decoder = Decoder(nz)

  def forward(self, x):
    return self.decoder(self.encoder(x))

  def reconstruct(self, x):
    """Only used later for visualization."""
    return self.forward(x)
```

Setting up the Auto-Encoder Training Loop [6pt]

After implementing the network architecture, we can now set up the training loop and run training.

```
epochs = 10
learning_rate = 1e-3
```

```
# build AE model
device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')   # use GPU if available
ae_model = AutoEncoder(nz).to(device)    # transfer model to GPU if
available
ae_model = ae_model.train()   # set model in train mode (eg batchnorm
params get updated)

# build optimizer and loss function
####################################### TODO
#######################################
# Create the optimizer and loss classes. For the loss you can use a
loss layer     #
# from the torch.nn package.
#
# HINT: We will use the Adam optimizer (learning rate given above,
otherwise       #
#       default parameters) and MSE loss for the criterion / loss.
#
# NOTE: We could also use alternative loss functions like cross
entropy, depending #
#       on the assumptions we are making about the output
distribution. Here we    #
#       will use MSE loss as it is the most common choice, assuming a
Gaussian     #
#       output distribution.
#
######################################################################
##############
opt = torch.optim.Adam(ae_model.parameters(), lr=learning_rate)
# create optimizer instance
criterion = nn.MSELoss()    # create loss layer instance
#################################### END TODO
#######################################

train_it = 0
for ep in range(epochs):
  print("Run Epoch {}".format(ep))
  ####################################### TODO
#######################################
  # Implement the main training loop for the auto-encoder model.
#
  # HINT: Your training loop should sample batches from the data
loader, run the     #
  #       forward pass of the AE, compute the loss, perform the
backward pass and    #
  #       perform one gradient step with the optimizer.
#
  # HINT: Don't forget to erase old gradients before performing the
```

```
backward pass.   #

####################################################################
##############
    # add training loop commands here
    # ...
  for (x, _) in mnist_data_loader:
    x = x.to(device)
    pred = ae_model.forward(x)
    rec_loss = criterion(pred, x)
    ae_model.zero_grad()
    opt.zero_grad()
    rec_loss.backward()
    opt.step()
    ######################################## END TODO
#####################################

    if train_it % 100 == 0:
      print("It {}: Reconstruction Loss: {}".format(train_it,
rec_loss))
    train_it += 1

print("Done!")
```

```
Run Epoch 0
It 0: Reconstruction Loss: 0.26972419023513794
It 100: Reconstruction Loss: 0.03894268348813057
It 200: Reconstruction Loss: 0.019504092633724213
It 300: Reconstruction Loss: 0.01590818539261818
It 400: Reconstruction Loss: 0.013952402397990227
It 500: Reconstruction Loss: 0.013186920434236526
It 600: Reconstruction Loss: 0.011852686293423176
It 700: Reconstruction Loss: 0.01009414717555046
It 800: Reconstruction Loss: 0.011076966300606728
It 900: Reconstruction Loss: 0.010262689553201199
Run Epoch 1
It 1000: Reconstruction Loss: 0.009367209859192371
It 1100: Reconstruction Loss: 0.00891298707574606
It 1200: Reconstruction Loss: 0.008198852650821209
It 1300: Reconstruction Loss: 0.00843334011733532
It 1400: Reconstruction Loss: 0.007510135415941477
It 1500: Reconstruction Loss: 0.008696877397596836
It 1600: Reconstruction Loss: 0.008465096354484558
It 1700: Reconstruction Loss: 0.0077793169766664505
It 1800: Reconstruction Loss: 0.008721661753952503
Run Epoch 2
It 1900: Reconstruction Loss: 0.007744807284325361
It 2000: Reconstruction Loss: 0.007225108332931995
It 2100: Reconstruction Loss: 0.007337048649787903
It 2200: Reconstruction Loss: 0.007528262212872505
```

```
It 2300: Reconstruction Loss: 0.006834432482719421
It 2400: Reconstruction Loss: 0.007883540354669094
It 2500: Reconstruction Loss: 0.00792559701949358
It 2600: Reconstruction Loss: 0.0069623845629394054
It 2700: Reconstruction Loss: 0.006955763790756464
It 2800: Reconstruction Loss: 0.0073212794959545135
Run Epoch 3
It 2900: Reconstruction Loss: 0.00676879333332181
It 3000: Reconstruction Loss: 0.006619803607463837
It 3100: Reconstruction Loss: 0.006619736552238464
It 3200: Reconstruction Loss: 0.006932450458407402
It 3300: Reconstruction Loss: 0.006519296672195196
It 3400: Reconstruction Loss: 0.006259030196815729
It 3500: Reconstruction Loss: 0.0069648632779717445
It 3600: Reconstruction Loss: 0.006809859536588192
It 3700: Reconstruction Loss: 0.006310960277915001
Run Epoch 4
It 3800: Reconstruction Loss: 0.00664236955344677
It 3900: Reconstruction Loss: 0.006767441052943468
It 4000: Reconstruction Loss: 0.005658551584929228
It 4100: Reconstruction Loss: 0.006615108344703913
It 4200: Reconstruction Loss: 0.005319413263350725
It 4300: Reconstruction Loss: 0.006962804589420557
It 4400: Reconstruction Loss: 0.006364195607602596
It 4500: Reconstruction Loss: 0.006309201940894127
It 4600: Reconstruction Loss: 0.005965700838714838
Run Epoch 5
It 4700: Reconstruction Loss: 0.006194955203682184
It 4800: Reconstruction Loss: 0.005932510830461979
It 4900: Reconstruction Loss: 0.00668210769072175
It 5000: Reconstruction Loss: 0.005948406644165516
It 5100: Reconstruction Loss: 0.00530467601493001
It 5200: Reconstruction Loss: 0.006765497848391533
It 5300: Reconstruction Loss: 0.006568537559360266
It 5400: Reconstruction Loss: 0.005505607929080725
It 5500: Reconstruction Loss: 0.006162336561828852
It 5600: Reconstruction Loss: 0.005985570140182972
Run Epoch 6
It 5700: Reconstruction Loss: 0.005671333055943251
It 5800: Reconstruction Loss: 0.00556244095787406
It 5900: Reconstruction Loss: 0.006094952579587698
It 6000: Reconstruction Loss: 0.0058354646898806095
It 6100: Reconstruction Loss: 0.005263315048068762
It 6200: Reconstruction Loss: 0.006222279742360115
It 6300: Reconstruction Loss: 0.005979042965918779
It 6400: Reconstruction Loss: 0.005646682344377041
It 6500: Reconstruction Loss: 0.0062297373078763485
Run Epoch 7
It 6600: Reconstruction Loss: 0.005251958500593901
It 6700: Reconstruction Loss: 0.005530822090804577
```

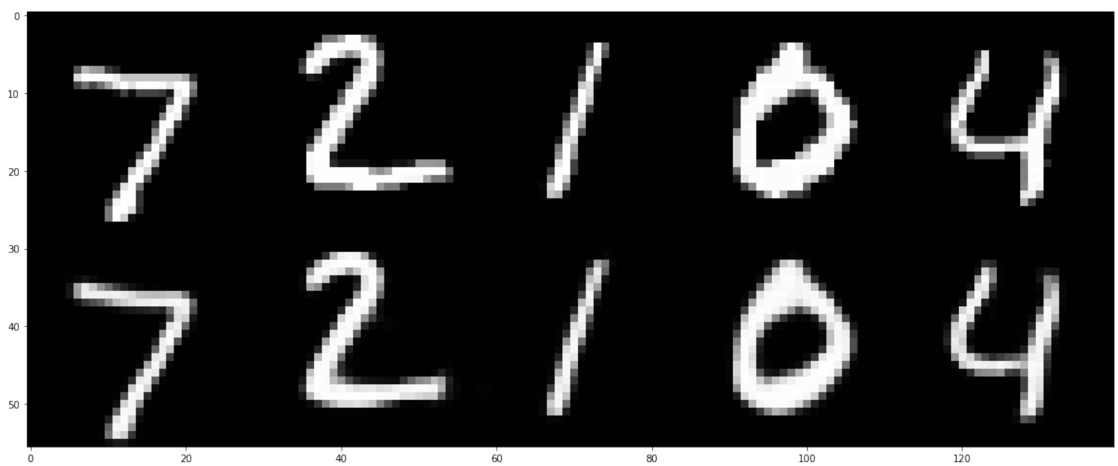
```
It 6800: Reconstruction Loss: 0.00569359865039587
It 6900: Reconstruction Loss: 0.005079316440969706
It 7000: Reconstruction Loss: 0.0056202104315161705
It 7100: Reconstruction Loss: 0.005669177509844303
It 7200: Reconstruction Loss: 0.0064501347951591015
It 7300: Reconstruction Loss: 0.005197315476834774
It 7400: Reconstruction Loss: 0.005425662267953157
It 7500: Reconstruction Loss: 0.005457249004393816
Run Epoch 8
It 7600: Reconstruction Loss: 0.0051828231662511826
It 7700: Reconstruction Loss: 0.005223572254180908
It 7800: Reconstruction Loss: 0.005383913405239582
It 7900: Reconstruction Loss: 0.004581862594932318
It 8000: Reconstruction Loss: 0.005069257691502571
It 8100: Reconstruction Loss: 0.005365584511309862
It 8200: Reconstruction Loss: 0.005165536887943745
It 8300: Reconstruction Loss: 0.00555442413315177
It 8400: Reconstruction Loss: 0.005510886199772358
Run Epoch 9
It 8500: Reconstruction Loss: 0.005067567806690931
It 8600: Reconstruction Loss: 0.005851340480148792
It 8700: Reconstruction Loss: 0.004378402605652809
It 8800: Reconstruction Loss: 0.004706318955868483
It 8900: Reconstruction Loss: 0.005418729968369007
It 9000: Reconstruction Loss: 0.004921534564346075
It 9100: Reconstruction Loss: 0.005481590982526541
It 9200: Reconstruction Loss: 0.004407491069287062
It 9300: Reconstruction Loss: 0.0056149763986468315
Done!
```

Verifying reconstructions [0pt]

Now that we trained the auto-encoder we can visualize some of the reconstructions on the test set to verify that it is converged and did not overfit. **Before continuing, make sure that your auto-encoder is able to reconstruct these samples near-perfectly.**

```
# visualize test data reconstructions
def vis_reconstruction(model):
  # download MNIST test set + build Dataset object
  mnist_test = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          download=True,

transform=torchvision.transforms.ToTensor())
  mnist_test_iter = iter(mnist_test)
  model.eval()      # set model in evalidation mode (eg freeze
batchnorm params)
  input_imgs, test_reconstructions = [], []
  for _ in range(5):
    input_img = np.asarray(next(mnist_test_iter)[0])
```

```
    reconstruction = model.reconstruct(torch.tensor(input_img[None],
device=device))
    input_imgs.append(input_img[0])
    test_reconstructions.append(reconstruction[0,
0].data.cpu().numpy())

  fig = plt.figure(figsize = (20, 50))
  ax1 = plt.subplot(111)
  ax1.imshow(np.concatenate([np.concatenate(input_imgs, axis=1),
                             np.concatenate(test_reconstructions,
axis=1)], axis=0), cmap='gray')
  plt.show()

vis_reconstruction(ae_model)
```

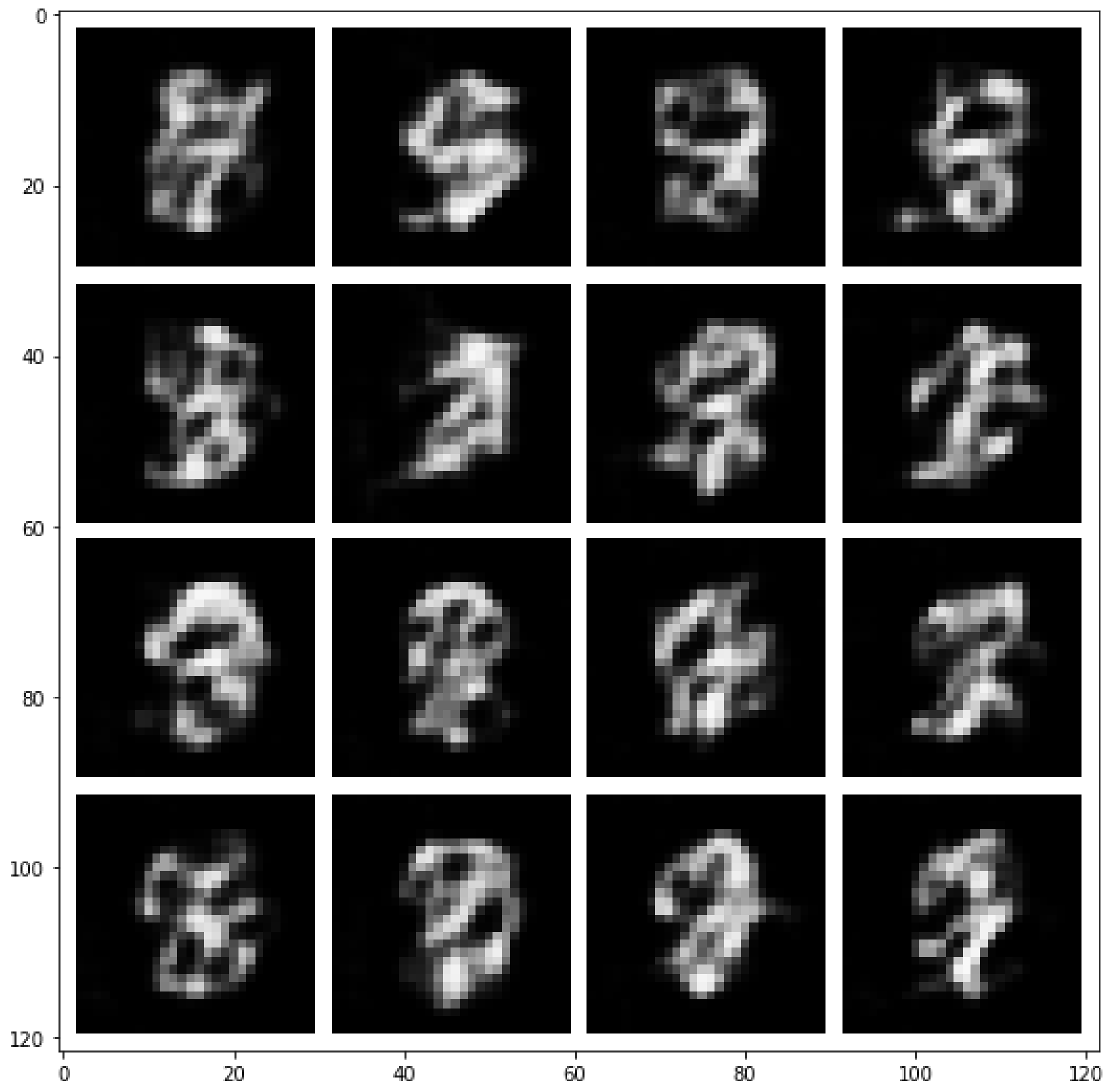


Sampling from the Auto-Encoder [2pt]

To test whether the auto-encoder is useful as a generative model, we can use it like any other generative model: draw embedding samples from a prior distribution and decode them through the decoder network. We will choose a unit Gaussian prior to allow for easy comparison to the VAE later.

```
# we will sample N embeddings, then decode and visualize them
def vis_samples(model):
  ####################################### TODO
#######################################
  # Sample embeddings from a diagonal unit Gaussian distribution and
decode them     #
  # using the model.
#
  # HINT: The sampled embeddings should have shape [batch_size, nz].
Diagonal unit   #
  #       Gaussians have mean 0 and a covariance matrix with ones on
the diagonal    #
  #       and zeros everywhere else.
#
```

```
  # HINT: If you are unsure whether you sampled the correct
distribution, you can    #
  #       sample a large batch and compute the empirical mean and
variance using the #
  #       .mean() and .var() functions.
#
  # HINT: You can directly use model.decoder() to decode the samples.
#

######################################################################
##############
  sampled_embeddings = torch.randn(64, 64).to(device)  # sample batch
of embedding from prior
  decoded_samples = ae_model.decoder(sampled_embeddings)       #
decoder output images for sampled embeddings
  ###################################### END TODO
######################################

  fig = plt.figure(figsize = (10, 10))
  ax1 = plt.subplot(111)
  ax1.imshow(torchvision.utils.make_grid(decoded_samples[:16], nrow=4,
pad_value=1.)\
                .data.cpu().numpy().transpose(1, 2, 0), cmap='gray')
  plt.show()

vis_samples(ae_model)
```



Inline Question: Describe your observations, why do you think they occur? [2pt] \ (please limit your answer to <150 words) \ **Answer:**

The decoding model has learned the reresentation of the MNIST dataset in such a way that it can reconstruct the original image from this reduced dimensional space with minimal error. Here it is trying to reconstruct the random values to a digit, and we can see that it resembles a few digits, like 8 and 3. These are the representations it has learned.

3. Variational Auto-Encoder (VAE)

Variational auto-encoders use a very similar architecture to deterministic auto-encoders, but are inherently storchastic models, i.e. we perform a stochastic sampling operation during the forward pass, leading to different different outputs every time we run the network for the same input. This sampling is required to optimize the VAE objective also known as the evidence lower bound (ELBO):

$$p(x) > E_{z \sim q(z \vee x)} p(x \vee z) - D_{\mathrm{KL}}(q(z \vee x), p(z))$$

Here, $D_{\mathrm{KL}}(q, p)$ denotes the Kullback-Leibler (KL) divergence between the posterior distribution $q(z \vee x)$, i.e. the output of our encoder, and $p(z)$, the prior over the embedding variable z, which we can choose freely.

For simplicity, we will again choose a unit Gaussian prior. The left term is the reconstruction term we already know from training the auto-encoder. When assuming a Gaussian output distribution for both encoder $q(z \vee x)$ and decoder $p(x \vee z)$ the objective reduces to:

$$L_{\mathrm{VAE}} = \sum_{x \sim D} (x - \hat{x})^2 - \beta \cdot D_{\mathrm{KL}}(N(\mu_q, \sigma_q), N(0, I))$$

Here, $\hat{x}$ is the reconstruction output of the decoder. In comparison to the auto-encoder objetive, the VAE adds a regularizing term between the output of the encoder and a chosen prior distribution, effectively forcing the encoder output to not stray too far from the prior during training. As a result the decoder gets trained with samples that look pretty similar to samples from the prior, which will hopefully allow us to generate better images when using the VAE as a generative model and actually feeding it samples from the prior (as we have done for the AE before).

The coefficient β is a scalar weighting factor that trades off between reconstruction and regularization objective. We will investigate the influence of this factor in out experiments below.

If you need a refresher on VAEs you can check out this tutorial paper: https://arxiv.org/abs/1606.05908

Reparametrization Trick

The sampling procedure inside the VAE's forward pass for obtaining a sample z from the posterior distribution $q(z \vee x)$, when implemented naively, is non-differentiable. However, since $q(z \vee x)$ is parametrized with a Gaussian function, there is a simple trick to obtain a differentiable sampling operator, known as the *reparametrization trick*.

Instead of directly sampling $z \sim N(\mu_q, \sigma_q)$ we can "separate" the network's predictions and the random sampling by computing the sample as:

$$z = \mu_q + \sigma_q * \epsilon, \epsilon \sim N(0, I)$$

Note that in this equation, the sample z is computed as a deterministic function of the network's predictions μ_q and σ_q and therefore allows to propagate gradients through the sampling procedure.

Note: While in the equations above the encoder network parametrizes the standard deviation σ_q of the Gaussian posterior distribution, in practice we usually parametrize the **logarithm of the standard deviation** $\log \sigma_q$ for numerical stability. Before sampling z we will then exponentiate the network's output to obtain σ_q.

Defining the VAE Model [7pt]

```
def kl_divergence(mu1, log_sigma1, mu2, log_sigma2):
  """Computes KL[p||q] between two Gaussians defined by [mu,
log_sigma]."""
  return (log_sigma2 - log_sigma1) + (torch.exp(log_sigma1) ** 2 +
(mu1 - mu2) ** 2) \
               / (2 * torch.exp(log_sigma2) ** 2) - 0.5


class VAE(nn.Module):
  def __init__(self, nz, beta=1.0):
    super().__init__()
    self.beta = beta          # factor trading off between two loss
components
    ######################################## TODO
########################################
    # Instantiate Encoder and Decoder.
#
    # HINT: Remember that the encoder is now parametrizing a Gaussian
distribution's   #
    #       mean and log_sigma, so the dimensionality of the output
embedding needs to #
    #       double.
#

######################################################################
##############
    self.encoder = Encoder(2 * nz)
    self.decoder = Decoder(nz)
    ##################################### END TODO
######################################

  def forward(self, x):
    ######################################## TODO
########################################
    # Implement the forward pass of the VAE.
#
    # HINT: Your code should implement the following steps:
#
    #          1. encode input x, split encoding into mean and
log_sigma of Gaussian   #
    #          2. sample z from inferred posterior distribution using
#
    #             reparametrization trick
#
    #          3. decode the sampled z to obtain the reconstructed
image               #

######################################################################
```

```
##############
    # encode input into posterior distribution q(z | x)
    q = self.encoder(x)        # output of encoder (concatenated mean
and log_sigma)

    # sample latent variable z with reparametrization
    z = q[:, :nz] + torch.rand_like(torch.log(q[:,
nz:]))*torch.exp(q[:, nz:])     # batch of sampled embeddings
    # compute reconstruction
    reconstruction = self.decoder(z)   # decoder reconstruction from
embedding
    #################################### END TODO
######################################

    return {'q': q,
            'rec': reconstruction}

  def loss(self, x, outputs):
    ######################################## TODO
#######################################
    # Implement the loss computation of the VAE.
#
    # HINT: Your code should implement the following steps:
#
    #          1. compute the image reconstruction loss, similar to AE
we use MSE loss #
    #          2. compute the KL divergence loss between the inferred
posterior        #
    #             distribution and a unit Gaussian prior; you can use
the provided     #
    #             function above for computing the KL divergence
between two Gaussians #
    #             parametrized by mean and log_sigma
#
    # HINT: Make sure to compute the KL divergence in the correct
order since it is    #
    #       not symmetric, ie. KL(p, q) != KL(q, p)!
#

######################################################################
##############
    # compute reconstruction loss
    rec = outputs['rec']
    q = outputs['q']
    rec_loss = nn.functional.mse_loss(x, rec)
    # compute KL divergence loss
#    kl_loss = torch.mean(torch.sum(kl_divergence(q[:, :nz],
torch.log(q[:, nz:]), torch.zeros(nz).to(device),
torch.zeros(nz).to(device)).nan_to_num(), axis=1))# make sure that
this is a scalar, not a vector / array
```

```
    kl_loss = 
torch.mean(torch.sum(kl_divergence(torch.zeros(nz).to(device), 
torch.zeros(nz).to(device), q[:, :nz], q[:, nz:]).nan_to_num(), 
axis=1))# make sure that this is a scalar, not a vector / array
    #################################### END TODO 
#####################################

    # return weihgted objective
    return rec_loss + self.beta * kl_loss, \
           {'rec_loss': rec_loss, 'kl_loss': kl_loss}

  def reconstruct(self, x):
    """Use mean of posterior estimate for visualization 
reconstruction."""
    ####################################### TODO 
#######################################
    # This function is used for visualizing reconstructions of our VAE 
model. To       #
    # obtain the maximum likelihood estimate we bypass the sampling 
procedure of the   #
    # inferred latent and instead directly use the mean of the 
inferred posterior.     #
    # HINT: encode the input image and then decode the mean of the 
posterior to obtain #
    #       the reconstruction.                                                        
#
    
######################################################################
##############
    q = self.encoder(x)
    reconstruction = self.decoder(q[:, :nz])
    #################################### END TODO 
#####################################
    return reconstruction
```

Setting up the VAE Training Loop [4pt]

Let's start training the VAE model! We will first verify our implementation by setting $\beta=0$.

```
learning_rate = 1e-3
nz = 64

####################################### TODO 
#######################################
# Tune the beta parameter to obtain good VAE training results. 
However, for the    #
# initial experiments leave beta = 0 in order to verify our 
implementation.        #
######################################################################
##############
```

```
epochs = 20          # using 5 epochs is sufficient for the first two
experiments
                   # for the experiment where you tune beta, 20 epochs
are appropriate
beta = 0.0001
#################################### END TODO
######################################

# build VAE model
vae_model = VAE(nz, beta).to(device)    # transfer model to GPU if
available
vae_model = vae_model.train()   # set model in train mode (eg
batchnorm params get updated)

# build optimizer and loss function
####################################### TODO
#######################################
# Build the optimizer for the vae_model. We will again use the Adam
optimizer with #
# the given learning rate and otherwise default parameters.
#
#######################################################################
##############
opt = torch.optim.Adam(vae_model.parameters(), lr=learning_rate)
#################################### END TODO
######################################

train_it = 0
rec_loss, kl_loss = [], []
for ep in range(epochs):
  print("Run Epoch {}".format(ep))
  ####################################### TODO
#######################################
  # Implement the main training loop for the VAE model.
#
  # HINT: Your training loop should sample batches from the data
loader, run the     #
  #       forward pass of the VAE, compute the loss, perform the
backward pass and   #
  #       perform one gradient step with the optimizer.
#
  # HINT: Don't forget to erase old gradients before performing the
backward pass.   #
  # HINT: This time we will use the loss() function of our model for
computing the   #
  #       training loss. It outputs the total training loss and a dict
containing    #
  #       the breakdown of reconstruction and KL loss.
#
```

```
######################################################################
##############
  for (x, _) in mnist_data_loader:
    x = x.to(device)
    forward_outs = vae_model.forward(x)
    total_loss, losses = vae_model.loss(x, forward_outs)
    total_loss = total_loss.cpu()
    vae_model.zero_grad()
    opt.zero_grad()
    total_loss.backward()
    losses['rec_loss'] = losses['rec_loss'].detach().cpu()
    losses['kl_loss'] = losses['kl_loss'].detach().cpu()
    opt.step()
    # add VAE training loop commands here
    # ...
    #################################### END TODO
####################################

    rec_loss.append(losses['rec_loss']);
kl_loss.append(losses['kl_loss'])
    if train_it % 100 == 0:
      print("It {}: Total Loss: {}, \t Rec Loss: {},\t KL Loss: {}"\
            .format(train_it, total_loss, losses['rec_loss'],
losses['kl_loss']))
    train_it += 1

print("Done!")

# log the loss training curves
fig = plt.figure(figsize = (10, 5))
ax1 = plt.subplot(121)
ax1.plot(rec_loss)
ax1.title.set_text("Reconstruction Loss")
ax2 = plt.subplot(122)
ax2.plot(kl_loss)
ax2.title.set_text("KL Loss")
plt.show()

Run Epoch 0
It 0: Total Loss: 0.2337312400341034,  Rec Loss: 0.23162946105003357,
      KL Loss: 21.01778221130371
It 100: Total Loss: 0.036748774349689484,    Rec Loss:
0.03539176657795906,   KL Loss: 13.570079803466797
It 200: Total Loss: 0.024348070845007896,    Rec Loss:
0.02283547632396221,   KL Loss: 15.125946998596191
It 300: Total Loss: 0.02180352248251438,     Rec Loss:
0.01993614435195923,   KL Loss: 18.673782348632812
It 400: Total Loss: 0.018534326925873756,    Rec Loss:
0.016756225377321243,  KL Loss: 17.781015396118164
It 500: Total Loss: 0.01699286326766014,     Rec Loss:
```

```
0.015122673474252224,  KL Loss: 18.701904296875
It 600: Total Loss: 0.01705174706876278,    Rec Loss:
0.015139229595661163,  KL Loss: 19.125171661376953
It 700: Total Loss: 0.01530380453914404,    Rec Loss:
0.013450547121465206,  KL Loss: 18.53257179260254
It 800: Total Loss: 0.016046034172177315,   Rec Loss:
0.014114723540842533,  KL Loss: 19.3131046295166
It 900: Total Loss: 0.013427069410681725,   Rec Loss:
0.011467210948467255,  KL Loss: 19.59857940673828
Run Epoch 1
It 1000: Total Loss: 0.015417631715536118,  Rec Loss:
0.013443171977996826,  KL Loss: 19.74459457397461
It 1100: Total Loss: 0.014303203672170639,  Rec Loss:
0.012274106964468956,  KL Loss: 20.290969848632812
It 1200: Total Loss: 0.013512368313968182,  Rec Loss:
0.011571221053600311,  KL Loss: 19.411470413208008
It 1300: Total Loss: 0.014181634411215782,  Rec Loss:
0.012233473360538483,  KL Loss: 19.481609344482422
It 1400: Total Loss: 0.01398109644651413,   Rec Loss:
0.011965928599238396,  KL Loss: 20.151676177978516
It 1500: Total Loss: 0.0135706327855587,    Rec Loss:
0.011567069217562675,  KL Loss: 20.0356388092041
It 1600: Total Loss: 0.012504805810749 53,  Rec Loss:
0.010474496521055698,  KL Loss: 20.303096771240234
It 1700: Total Loss: 0.012361840344965458,  Rec Loss:
0.010396509431302547,  KL Loss: 19.65330696105957
It 1800: Total Loss: 0.013560134917497635,  Rec Loss:
0.011530271731317043,  KL Loss: 20.29863739013672
Run Epoch 2
It 1900: Total Loss: 0.012573613785207272,  Rec Loss:
0.010611371137201786,  KL Loss: 19.622426986694336
It 2000: Total Loss: 0.012237310409545898,  Rec Loss:
0.01025467086583376,   KL Loss: 19.826400756835938
It 2100: Total Loss: 0.011951389722526073,  Rec Loss:
0.010073976591229439,  KL Loss: 18.774131774902344
It 2200: Total Loss: 0.013662638142704964,  Rec Loss:
0.011595742776989937,  KL Loss: 20.668954849243164
It 2300: Total Loss: 0.011914394795894623,  Rec Loss:
0.009933087974786758,  KL Loss: 19.813068389892578
It 2400: Total Loss: 0.01227217074483633,   Rec Loss:
0.010211063548922539,  KL Loss: 20.611074447631836
It 2500: Total Loss: 0.011185338720679283,  Rec Loss:
0.009228352457284927,  KL Loss: 19.569866180419922
It 2600: Total Loss: 0.013756802305579185,  Rec Loss:
0.011653892695903778,  KL Loss: 21.029094696044922
It 2700: Total Loss: 0.01224315632134676,   Rec Loss:
0.010243695229291916,  KL Loss: 19.994609832763672
It 2800: Total Loss: 0.011746814474463463,  Rec Loss:
0.009737318381667137,  KL Loss: 20.0949649810791
Run Epoch 3
```

```
It 2900: Total Loss: 0.01197094563394785,   Rec Loss:
0.009961074218153954,  KL Loss: 20.098712921142578
It 3000: Total Loss: 0.011041415855288506,  Rec Loss:
0.009078983217477798,  KL Loss: 19.62432861328125
It 3100: Total Loss: 0.012746147811412811,  Rec Loss:
0.010597671382129192,  KL Loss: 21.484764099121094
It 3200: Total Loss: 0.010808959603309631,  Rec Loss:
0.008815997280180454,  KL Loss: 19.929628372192383
It 3300: Total Loss: 0.012135517783463001,  Rec Loss:
0.01000713836401701,   KL Loss: 21.283798217773438
It 3400: Total Loss: 0.012651904486119747,  Rec Loss:
0.01066006813198328,   KL Loss: 19.91836166381836
It 3500: Total Loss: 0.011761651374399662,  Rec Loss:
0.00978136621415615,   KL Loss: 19.802852630615234
It 3600: Total Loss: 0.013106169179081917,  Rec Loss:
0.011067613959312439,  KL Loss: 20.385547637939453
It 3700: Total Loss: 0.010718959383666515,  Rec Loss:
0.008789443410933018,  KL Loss: 19.295164108276367
Run Epoch 4
It 3800: Total Loss: 0.011049985885620117,  Rec Loss:
0.008932968601584435,  KL Loss: 21.170177459716797
It 3900: Total Loss: 0.011663142591714859,  Rec Loss:
0.009615457616746426,  KL Loss: 20.476852416992188
It 4000: Total Loss: 0.011229131370782852,  Rec Loss:
0.009225407615303993,  KL Loss: 20.037242889404297
It 4100: Total Loss: 0.012598467990756035,  Rec Loss:
0.01062050648033619,   KL Loss: 19.779621124267578
It 4200: Total Loss: 0.0114980423822999,    Rec Loss:
0.00949908047914505,   KL Loss: 19.98961639404297
It 4300: Total Loss: 0.01127099059522152,   Rec Loss:
0.009196130558848381,  KL Loss: 20.748598098754883
It 4400: Total Loss: 0.011609885841608047,  Rec Loss:
0.009655353613197803,  KL Loss: 19.54532814025879
It 4500: Total Loss: 0.010655476711690426,  Rec Loss:
0.00867842324078083,   KL Loss: 19.770538330078125
It 4600: Total Loss: 0.011383214965462685,  Rec Loss:
0.009324452839791775,  KL Loss: 20.587627410888672
Run Epoch 5
It 4700: Total Loss: 0.010912141762673855,  Rec Loss:
0.008901283144950867,  KL Loss: 20.10858726501465
It 4800: Total Loss: 0.01162390224635601,   Rec Loss:
0.00957073736935854,   KL Loss: 20.531654357910156
It 4900: Total Loss: 0.010170238092541695,  Rec Loss:
0.008329479955136776,  KL Loss: 18.40758514404297
It 5000: Total Loss: 0.011235630139708519,  Rec Loss:
0.009235206991434097,  KL Loss: 20.00423240661621
It 5100: Total Loss: 0.011403016746044159,  Rec Loss:
0.009313545189797878,  KL Loss: 20.894710540771484
It 5200: Total Loss: 0.010562351904809475,  Rec Loss:
0.008551323786377907,  KL Loss: 20.110279083251953
```

```
It 5300: Total Loss: 0.011084072291851044,  Rec Loss:
0.009131333790719509, KL Loss: 19.527389526367188
It 5400: Total Loss: 0.010685810819268227,  Rec Loss:
0.008706529624760151, KL Loss: 19.79280662536621
It 5500: Total Loss: 0.011725720949470997,  Rec Loss:
0.009676782414317131, KL Loss: 20.48938751220703
It 5600: Total Loss: 0.010762940160930157,  Rec Loss:
0.00870999600738287,  KL Loss: 20.52943992614746
Run Epoch 6
It 5700: Total Loss: 0.010728634893894196,  Rec Loss:
0.00882148090749979,  KL Loss: 19.071537017822266
It 5800: Total Loss: 0.011182880029082298,  Rec Loss:
0.009180882014334202, KL Loss: 20.019981384277344
It 5900: Total Loss: 0.011242968030273914,  Rec Loss:
0.009294232353568077, KL Loss: 19.487354278564453
It 6000: Total Loss: 0.011706531047821045,  Rec Loss:
0.00965104065835476,  KL Loss: 20.554901123046875
It 6100: Total Loss: 0.010770891793072224,  Rec Loss:
0.008772864006459713, KL Loss: 19.98027992248535
It 6200: Total Loss: 0.010298667475581169,  Rec Loss:
0.008425189182162285, KL Loss: 18.734783172607422
It 6300: Total Loss: 0.010512295179069042,  Rec Loss:
0.008560956455767155, KL Loss: 19.513385772705078
It 6400: Total Loss: 0.010647623799741268,  Rec Loss:
0.008569744415581226, KL Loss: 20.778797149658203
It 6500: Total Loss: 0.009606734849512577,  Rec Loss:
0.007599283009767532, KL Loss: 20.07451629638672
Run Epoch 7
It 6600: Total Loss: 0.009882181882858276,  Rec Loss:
0.007866312749683857, KL Loss: 20.158687591552734
It 6700: Total Loss: 0.011033924296498299,  Rec Loss:
0.009002123028039932, KL Loss: 20.318016052246094
It 6800: Total Loss: 0.011210756376385689,  Rec Loss:
0.009148713201284409, KL Loss: 20.620433807373047
It 6900: Total Loss: 0.010074814781546593,  Rec Loss:
0.008128092624247074, KL Loss: 19.46722412109375
It 7000: Total Loss: 0.009631154127418995,  Rec Loss:
0.00773963239043951,  KL Loss: 18.91521453857422
It 7100: Total Loss: 0.00967472791671753,   Rec Loss:
0.007734966930001974, KL Loss: 19.397615432739258
It 7200: Total Loss: 0.01078040711581707,   Rec Loss:
0.008755271323025227, KL Loss: 20.251361846923828
It 7300: Total Loss: 0.010695084929466248,  Rec Loss:
0.008751566521823406, KL Loss: 19.4351806640625
It 7400: Total Loss: 0.010861076414585114,  Rec Loss:
0.00889766775071621,  KL Loss: 19.634092330932617
It 7500: Total Loss: 0.010362161323428154,  Rec Loss:
0.008475693874061108, KL Loss: 18.864673614501953
Run Epoch 8
It 7600: Total Loss: 0.011189166456460953,  Rec Loss:
```

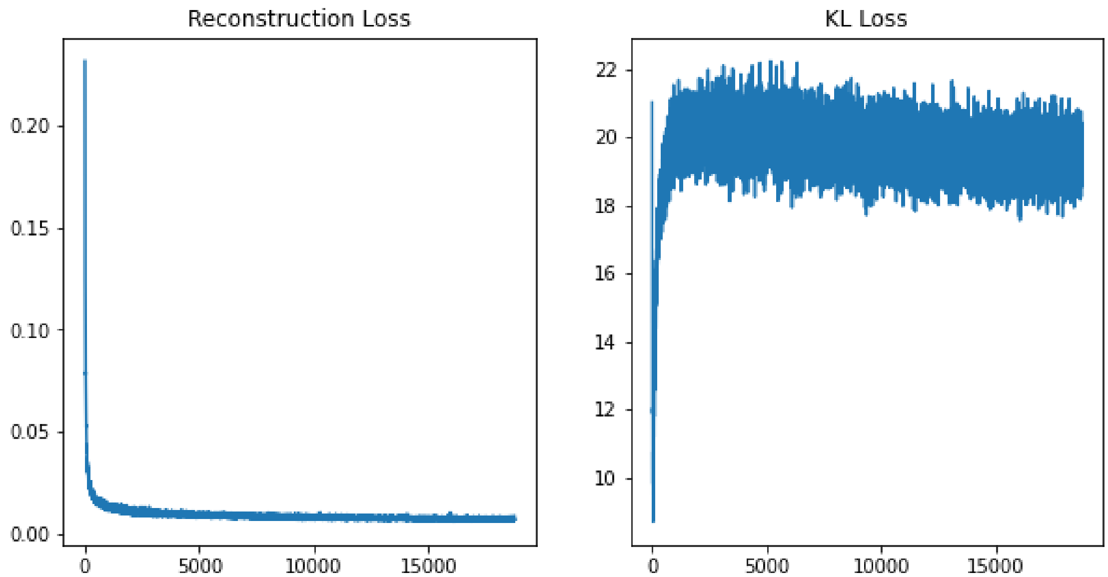
```
0.009079241193830967,  KL Loss: 21.09925079345703
It 7700: Total Loss: 0.010027027688920498,   Rec Loss:
0.008037402294576168,  KL Loss: 19.89625358581543
It 7800: Total Loss: 0.009904044680297375,   Rec Loss:
0.007940786890685558,  KL Loss: 19.632579803466797
It 7900: Total Loss: 0.009269644506275654,   Rec Loss:
0.00732524786144495,   KL Loss: 19.443967819213867
It 8000: Total Loss: 0.011116919107735157,   Rec Loss:
0.009120163507759571,  KL Loss: 19.96755599975586
It 8100: Total Loss: 0.009996791370213032,   Rec Loss:
0.007930874824523926,  KL Loss: 20.659164428710938
It 8200: Total Loss: 0.009633170440793037,   Rec Loss:
0.00773466844111681,   KL Loss: 18.985021591186523
It 8300: Total Loss: 0.009711318649351597,   Rec Loss:
0.007771733216941357,  KL Loss: 19.395851135253906
It 8400: Total Loss: 0.00980121549218893,    Rec Loss:
0.007797365076839924,  KL Loss: 20.038501739501953
Run Epoch 9
It 8500: Total Loss: 0.010241026990115643,   Rec Loss:
0.008257930167019367,  KL Loss: 19.830970764160156
It 8600: Total Loss: 0.01074901595711708,    Rec Loss:
0.008725578896701336,  KL Loss: 20.234376907348633
It 8700: Total Loss: 0.011062851175665855,   Rec Loss:
0.009063187055289745,  KL Loss: 19.996646881103516
It 8800: Total Loss: 0.010644344612956047,   Rec Loss:
0.008598100394010544,  KL Loss: 20.46244239807129
It 8900: Total Loss: 0.009751505218446255,   Rec Loss:
0.0077975899912416935,      KL Loss: 19.539155960083008
It 9000: Total Loss: 0.010029444471001625,   Rec Loss:
0.008111625909805298,  KL Loss: 19.178184509277344
It 9100: Total Loss: 0.010985367931425571,   Rec Loss:
0.008956646546721458,  KL Loss: 20.287212371826172
It 9200: Total Loss: 0.009876340627670288,   Rec Loss:
0.007843519560992718,  KL Loss: 20.328210830688477
It 9300: Total Loss: 0.009982575662434101,   Rec Loss:
0.008055259473621845,  KL Loss: 19.273160934448242
Run Epoch 10
It 9400: Total Loss: 0.010141980834305286,   Rec Loss:
0.008086148649454117,  KL Loss: 20.558320999145508
It 9500: Total Loss: 0.010368820279836655,   Rec Loss:
0.00838407315313816,   KL Loss: 19.847469329833984
It 9600: Total Loss: 0.010149510577321053,   Rec Loss:
0.00813536997884512,   KL Loss: 20.141407012939453
It 9700: Total Loss: 0.009718629531562328,   Rec Loss:
0.007757882121950388,  KL Loss: 19.60747718811035
It 9800: Total Loss: 0.008868706412613392,   Rec Loss:
0.007033166475594044,  KL Loss: 18.355396270751953
It 9900: Total Loss: 0.009782112203538418,   Rec Loss:
0.00785546749830246,   KL Loss: 19.26644515991211
It 10000: Total Loss: 0.009920748881995678,       Rec Loss:
```

```
0.008012413047254086,  KL Loss: 19.083362579345703
It 10100: Total Loss: 0.011020917445421219,      Rec Loss:
0.008964565582573414,  KL Loss: 20.563518524169922
It 10200: Total Loss: 0.009082206524908543,      Rec Loss:
0.007148674223572016,  KL Loss: 19.33531951904297
It 10300: Total Loss: 0.010194560512900352,      Rec Loss:
0.008293364197015762,  KL Loss: 19.011959075927734
Run Epoch 11
It 10400: Total Loss: 0.00940373633056879,   Rec Loss:
0.007542531006038189,  KL Loss: 18.612051010131836
It 10500: Total Loss: 0.009233261458575726,      Rec Loss:
0.007266642991453409,  KL Loss: 19.666183471679688
It 10600: Total Loss: 0.00848186295479536,   Rec Loss:
0.006556359119713306,  KL Loss: 19.255041122436523
It 10700: Total Loss: 0.009250662289559841,      Rec Loss:
0.007337313145399094,  KL Loss: 19.13349151611328
It 10800: Total Loss: 0.00948668085038662,   Rec Loss:
0.007473149802535772,  KL Loss: 20.135305404663086
It 10900: Total Loss: 0.010043583810329437,      Rec Loss:
0.008041223511099815,  KL Loss: 20.023609161376953
It 11000: Total Loss: 0.010067366994917393,      Rec Loss:
0.008146263659000397,  KL Loss: 19.211030960083008
It 11100: Total Loss: 0.010544667951762676,      Rec Loss:
0.00850841123610735,   KL Loss: 20.362565994262695
It 11200: Total Loss: 0.009751818142831326,      Rec Loss:
0.007756262086331844,  KL Loss: 19.9555606842041
Run Epoch 12
It 11300: Total Loss: 0.009799700230360031,      Rec Loss:
0.007853757590055466,  KL Loss: 19.459426879882812
It 11400: Total Loss: 0.010336529463529587,      Rec Loss:
0.0083177974447608,    KL Loss: 20.187318801879883
It 11500: Total Loss: 0.00938414130359888,   Rec Loss:
0.00731789181008935,   KL Loss: 20.662498474121094
It 11600: Total Loss: 0.010069755837321281,      Rec Loss:
0.008138229139149189,  KL Loss: 19.315269470214844
It 11700: Total Loss: 0.009680209681391716,      Rec Loss:
0.007745346054434776,  KL Loss: 19.34864044189453
It 11800: Total Loss: 0.009413301944732666,      Rec Loss:
0.007457858417183161,  KL Loss: 19.554431915283203
It 11900: Total Loss: 0.009493906982243061,      Rec Loss:
0.007567024789750576,  KL Loss: 19.26882553100586
It 12000: Total Loss: 0.009318679571151733,      Rec Loss:
0.007405176293104887,  KL Loss: 19.135028839111328
It 12100: Total Loss: 0.00895090214908123,   Rec Loss:
0.007010401226580143,  KL Loss: 19.405010223388672
Run Epoch 13
It 12200: Total Loss: 0.008735612034797668,      Rec Loss:
0.006883055437356234,  KL Loss: 18.52556610107422
It 12300: Total Loss: 0.009696024470031261,      Rec Loss:
0.007652694825083017,  KL Loss: 20.433298110961914
```

```
It 12400: Total Loss: 0.009836899116635323,      Rec Loss:
0.007860262878239155,  KL Loss: 19.76636505126953
It 12500: Total Loss: 0.010244989767670631,      Rec Loss:
0.008260565809905529,  KL Loss: 19.84423828125
It 12600: Total Loss: 0.010686383582651615,      Rec Loss:
0.008695952594280243,  KL Loss: 19.904312133789062
It 12700: Total Loss: 0.010180667974054813,      Rec Loss:
0.008162700571119785,  KL Loss: 20.17967414855957
It 12800: Total Loss: 0.009238140657544136,      Rec Loss:
0.007156890816986561,  KL Loss: 20.812503814697266
It 12900: Total Loss: 0.009176095947623253,      Rec Loss:
0.007221668027341366,  KL Loss: 19.544281005859375
It 13000: Total Loss: 0.009474145248532295,      Rec Loss:
0.007469289004802704,  KL Loss: 20.048564910888672
It 13100: Total Loss: 0.009604974649846554,      Rec Loss:
0.00766359269618988,   KL Loss: 19.413820266723633
Run Epoch 14
It 13200: Total Loss: 0.009036051109433174,      Rec Loss:
0.007228773087263107,  KL Loss: 18.072784423828125
It 13300: Total Loss: 0.009338753297924995,      Rec Loss:
0.007435858249664307,  KL Loss: 19.028945922851562
It 13400: Total Loss: 0.010226594284176826,      Rec Loss:
0.008227042853832245,  KL Loss: 19.99551010131836
It 13500: Total Loss: 0.009303375147283077,      Rec Loss:
0.007377778645604849,  KL Loss: 19.255966186523438
It 13600: Total Loss: 0.009831802919507027,      Rec Loss:
0.007921365089714527,  KL Loss: 19.10437774658203
It 13700: Total Loss: 0.010256019420921803,      Rec Loss:
0.00825662724673748,   KL Loss: 19.993919372558594
It 13800: Total Loss: 0.009492616169154644,      Rec Loss:
0.007542094215750694,  KL Loss: 19.505220413208008
It 13900: Total Loss: 0.009448716416954994,      Rec Loss:
0.007441073656082153,  KL Loss: 20.076427459716797
It 14000: Total Loss: 0.009207211434841156,      Rec Loss:
0.007303313352167606,  KL Loss: 19.038978576660156
Run Epoch 15
It 14100: Total Loss: 0.009012767113745213,      Rec Loss:
0.007089658640325069,  KL Loss: 19.2310848236084
It 14200: Total Loss: 0.009808267466723919,      Rec Loss:
0.007831469178199768,  KL Loss: 19.767982482910156
It 14300: Total Loss: 0.009388512931764126,      Rec Loss:
0.007406635209918022,  KL Loss: 19.818775177001953
It 14400: Total Loss: 0.00828483421355486,   Rec Loss:
0.006444678641855717,  KL Loss: 18.401559829711914
It 14500: Total Loss: 0.008984452113509178,      Rec Loss:
0.007109673693776131,  KL Loss: 18.747779846191406
It 14600: Total Loss: 0.009112023748457432,      Rec Loss:
0.007162486203014851,  KL Loss: 19.495376586914062
It 14700: Total Loss: 0.009432138875126839,      Rec Loss:
0.0075379470363259315,      KL Loss: 18.941917419433594
```

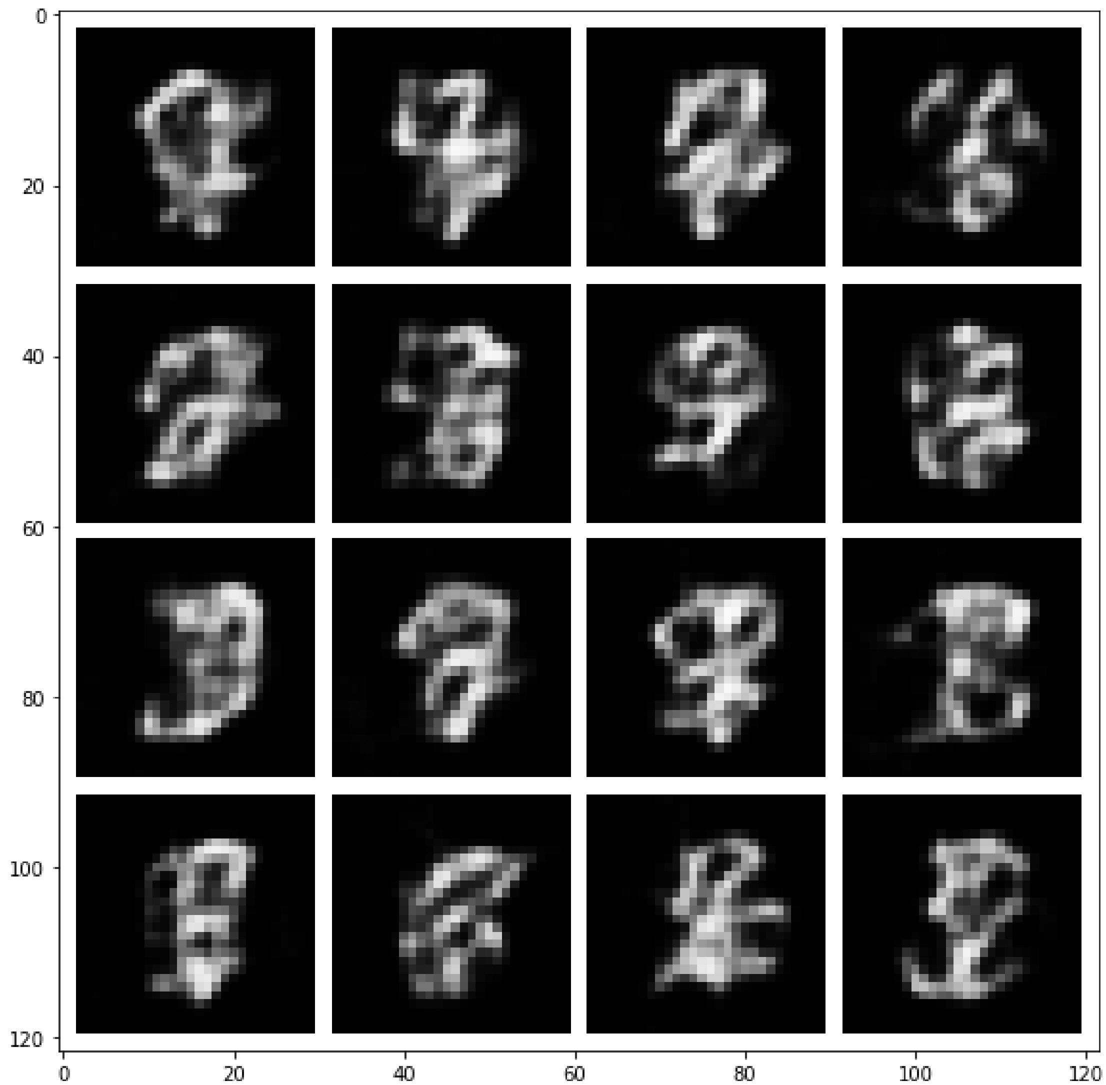
It 14800: Total Loss: 0.009679015725851059, Rec Loss: 0.007672532461583614, KL Loss: 20.064836502075195
It 14900: Total Loss: 0.008375903591513634, Rec Loss: 0.006535575725138187, KL Loss: 18.403274536132812
It 15000: Total Loss: 0.009418453089892864, Rec Loss: 0.007482761051505804, KL Loss: 19.356922149658203
Run Epoch 16
It 15100: Total Loss: 0.00890457071363926, Rec Loss: 0.0069479020312428474, KL Loss: 19.56668472290039
It 15200: Total Loss: 0.009318983182311058, Rec Loss: 0.007291019894182682, KL Loss: 20.279630661010742
It 15300: Total Loss: 0.008803917095065117, Rec Loss: 0.00691474974155426, KL Loss: 18.89167022705078
It 15400: Total Loss: 0.008747617714107037, Rec Loss: 0.006930071394890547, KL Loss: 18.175464630126953
It 15500: Total Loss: 0.008583329617977142, Rec Loss: 0.006689179688692093, KL Loss: 18.941497802734375
It 15600: Total Loss: 0.008900785818696022, Rec Loss: 0.0069357422180473804, KL Loss: 19.650434494018555
It 15700: Total Loss: 0.009168310090899467, Rec Loss: 0.007292360533028841, KL Loss: 18.759498596191406
It 15800: Total Loss: 0.009695341810584068, Rec Loss: 0.007654087617993355, KL Loss: 20.41254425048828
It 15900: Total Loss: 0.009341772645711899, Rec Loss: 0.007373183034360409, KL Loss: 19.685894012451172
Run Epoch 17
It 16000: Total Loss: 0.009631404653191566, Rec Loss: 0.007655958645045757, KL Loss: 19.75446128845215
It 16100: Total Loss: 0.00925498828291893, Rec Loss: 0.007257900666445494, KL Loss: 19.970874786376953
It 16200: Total Loss: 0.010051802732050419, Rec Loss: 0.008001520298421383, KL Loss: 20.502826690673828
It 16300: Total Loss: 0.009166950359940529, Rec Loss: 0.007285760249942541, KL Loss: 18.81190299987793
It 16400: Total Loss: 0.008989590220153332, Rec Loss: 0.00714452937245369, KL Loss: 18.450605392456055
It 16500: Total Loss: 0.008583547547459602, Rec Loss: 0.006651151925325394, KL Loss: 19.32396125793457
It 16600: Total Loss: 0.008448736742138863, Rec Loss: 0.006481637712568045, KL Loss: 19.670995712280273
It 16700: Total Loss: 0.009410754777491093, Rec Loss: 0.007448025979101658, KL Loss: 19.627286911010742
It 16800: Total Loss: 0.008248954080045223, Rec Loss: 0.006393996998667717, KL Loss: 18.54957389831543
Run Epoch 18
It 16900: Total Loss: 0.00913525465875864, Rec Loss: 0.0072027710266411304, KL Loss: 19.32483673095703
It 17000: Total Loss: 0.009384341537952423, Rec Loss: 0.007358510512858629, KL Loss: 20.258312225341797
It 17100: Total Loss: 0.008171929977834225, Rec Loss:

```
0.006286827847361565,  KL Loss: 18.85102081298828
It 17200: Total Loss: 0.009632471948862076,       Rec Loss:
0.007718869484961033,  KL Loss: 19.13602066040039
It 17300: Total Loss: 0.008548183366656303,       Rec Loss:
0.006607479881495237,  KL Loss: 19.407033920288086
It 17400: Total Loss: 0.009624561294913292,       Rec Loss:
0.0076821427792310715,      KL Loss: 19.42418670654297
It 17500: Total Loss: 0.009428214281797409,       Rec Loss:
0.007521343417465687,  KL Loss: 19.06870460510254
It 17600: Total Loss: 0.009941276162862778,       Rec Loss:
0.007968501187860966,  KL Loss: 19.727754592895508
It 17700: Total Loss: 0.008030988276004791,       Rec Loss:
0.006184965837746859,  KL Loss: 18.460229873657227
It 17800: Total Loss: 0.008651576936244965,       Rec Loss:
0.006743073463439941,  KL Loss: 19.085037231445312
Run Epoch 19
It 17900: Total Loss: 0.00953714083880186,   Rec Loss:
0.007608341984450817,  KL Loss: 19.287992477416992
It 18000: Total Loss: 0.009084980934858322,       Rec Loss:
0.007228220347315073,  KL Loss: 18.567604064941406
It 18100: Total Loss: 0.008538378402590752,       Rec Loss:
0.00660879397764802,   KL Loss: 19.295848846435547
It 18200: Total Loss: 0.008852812461555004,       Rec Loss:
0.006932985968887806,  KL Loss: 19.19826889038086
It 18300: Total Loss: 0.008395504206418991,       Rec Loss:
0.00651401374489069,   KL Loss: 18.814905166625977
It 18400: Total Loss: 0.009026801213622093,       Rec Loss:
0.007132979109883308,  KL Loss: 18.93822479248047
It 18500: Total Loss: 0.00881512276828289,   Rec Loss:
0.006911963690072298,  KL Loss: 19.03158950805664
It 18600: Total Loss: 0.008559603244066238,       Rec Loss:
0.006624417379498482,  KL Loss: 19.351856231689453
It 18700: Total Loss: 0.009245181456208229,       Rec Loss:
0.007343700621277094,  KL Loss: 19.014808654785156
Done!
```



Let's look at some reconstructions and decoded embedding samples!

```
# visualize VAE reconstructions and samples from the generative model
vis_reconstruction(vae_model)
vis_samples(vae_model)
```



> **Inline Question: What can you observe when setting $\beta = 0$? Explain your observations! [3pt]** \ (please limit your answer to <150 words) \ **Answer:**

When B=0, we use only logp(x|z) and ignore the prior divergence. We only use the reconstruction error. Hence we get the image shown above, without any disentanglement.

Let's repeat the same experiment for $\beta = 10$, a very high value for the coefficient. You can modify the β value in the cell above and rerun it (it is okay to overwrite the outputs of the previous experiment, but **make sure to copy the visualizations of training curves, reconstructions and samples for $\beta = 0$ into your solution PDF** before deleting them).

> **Inline Question: What can you observe when setting $\beta = 10$? Explain your observations! [3pt]** \ (please limit your answer to <200 words) \ **Answer**:

When B=10, we have a stronger contraint over the latent bottleneck. This greatly limits the representation capacity of z and results in further disentanglement, which in turn, results in the above image with very poor reconstruction. Optimal value exists between 0 and 10.

Now we can start tuning the beta value to achieve a good result. First describe what a "good result" would look like (focus what you would expect for reconstructions and sample quality).

> **Inline Question: Characterize what properties you would expect for reconstructions (1pt) and samples (2pt) of a well-tuned VAE! [3pt]** \ (please limit your answer to <200 words) \ **Answer**:

A well tuned B-VAE will learn the disentangled representations while still having less reconstruction errors. They will however have lesser interpretability of the latent space.

Tuning the β-factor [5pt]

Now that you know what outcome we would like to obtain, try to tune β to achieve this result.

(logarithmic search in steps of 10x will be helpful, good results can be achieved after ~20 epochs of training). It is again okay to overwrite the results of the previous $\beta=10$ experiment after copying them to the solution PDF.

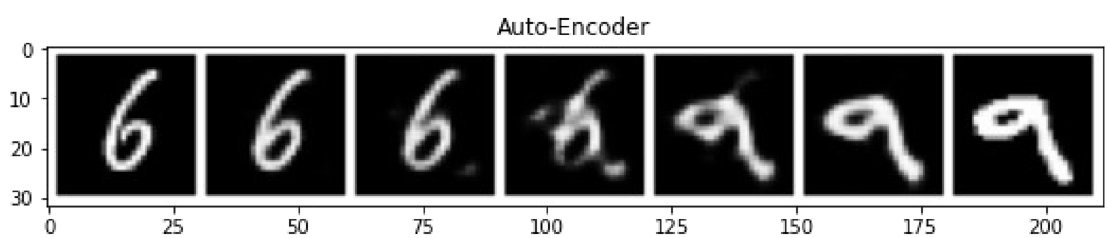
Your final notebook should include the visualizations of your best-tuned VAE.

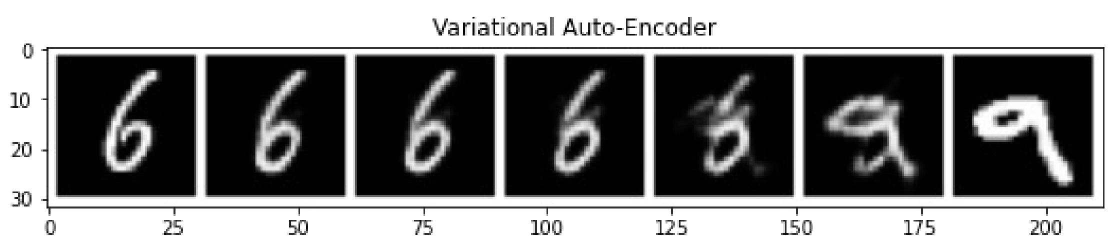
4. Embedding Space Interpolation [3pt]

As mentioned in the introduction, AEs and VAEs cannot only be used to generate images, but also to learn low-dimensional representations of their inputs. In this final section we will investigate the representations we learned with both models by **interpolating in embedding space** between different images. We will encode two images into their low-dimensional embedding representations, then interpolate these embeddings and reconstruct the result.

```
START_LABEL = 6
END_LABEL = 9
nz=64

def get_image_with_label(target_label):
  """Returns a random image from the training set with the requested digit."""
  for img_batch, label_batch in mnist_data_loader:
    for img, label in zip(img_batch, label_batch):
      if label == target_label:
        return img.to(device)

def interpolate_and_visualize(model, tag, start_img, end_img):
  """Encodes images and performs interpolation. Displays decodings."""
  model.eval()    # put model in eval mode to avoid updating batchnorm

  # encode both images into embeddings (use posterior mean for
```

```
interpolation)
  z_start = model.encoder(start_img[None])[..., :nz]
  z_end = model.encoder(end_img[None])[..., :nz]

  # compute interpolated latents
  N_INTER_STEPS = 5
  z_inter = [z_start + i/N_INTER_STEPS * (z_end - z_start) for i in
range(N_INTER_STEPS)]

  # decode interpolated embeddings (as a single batch)
  img_inter = model.decoder(torch.cat(z_inter))

  # reshape result and display interpolation
  vis_imgs = torch.cat([start_img[None], img_inter, end_img[None]])
  fig = plt.figure(figsize = (10, 10))
  ax1 = plt.subplot(111)
  ax1.imshow(torchvision.utils.make_grid(vis_imgs,
nrow=N_INTER_STEPS+2, pad_value=1.)\
                    .data.cpu().numpy().transpose(1, 2, 0), cmap='gray')
  plt.title(tag)
  plt.show()


# sample two training images with given labels
start_img = get_image_with_label(START_LABEL)
end_img = get_image_with_label(END_LABEL)

# visualize interpolations for AE and VAE models
interpolate_and_visualize(ae_model, "Auto-Encoder", start_img,
end_img)
interpolate_and_visualize(vae_model, "Variational Auto-Encoder",
start_img, end_img)
```





Repeat the experiment for different start / end labels and different samples. Describe your observations.

> **Inline Question: Repeat the interpolation experiment with different start / end labels and multiple samples. Describe your observations! Focus on**: \

1. **How do AE and VAE embedding space interpolations differ?** \
2. **How do you expect these differences to affect the usefulness of the learned representation for downstream learning?** \ (please limit your answer to <300 words)

> **Answer**:

1. The latent space where the encoder encodes the input in an autoencoder may not be continuous and hence dont allow easy interpolation. On the other hand, the latent spaces of VAE are continous and allow random sampling.

2. VAE learns both the mean and variance of the inputs in the latent space and hence can be used to generate new data from feeding random values to the decoder part. The autoencoder will try its best to generate an image which looks close to a digit, while the VAE can generate an image closer to a digit because it learns a distribution of possible latent inputs that can lead to the specific digit.

Hence, we can observe that the transition between the 2 classes is smoother in the case of VAE and very abrupt (you can clearly see 2 different digits in the case of 3&5) in the case of AE. This is due to the nature of the latent space.

Submission PDF

As in assignment 1, please prepare a separate submission PDF for each problem. **Do not simply render your notebook as a pdf**. For this problem, please include the following plots & answers in a PDF called `problem_1_solution.pdf`:

1. Auto-encoder samples and AE sampling inline question answer.
2. VAE training curves, reconstructions and samples for:

- $\beta=0$
- $\beta=10$
- your tuned β (also listing the tuned value for β)

1. Answers to all inline questions in VAE section (ie 4 inline questions).
2. Three representative interpolation comparisons that show AE and VAE embedding interpolation between the same images.
3. Answer to interpolation inline question.

Note that you still need to submit the jupyter notebook with all generated solutions. We will randomly pick submissions and check that the plots in the PDF and in the notebook are equivalent (except for those $\beta=0/10$ plots that we allowed to overwrite).