Problem1_VAE_print

April 4, 2023

1 Problem 1 - Variational Auto-Encoder (VAE)

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 (GANs, see Problem 2), 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. 6. Extend VAE's generative capabilities by conditioning it on the label you wish to generate.

Note: For faster training of the models in this assignment you can enable GPU support in this Colab. Navigate to "Runtime" –> "Change Runtime Type" and set the "Hardware Accelerator" to "GPU". However, you might hit compute limits of the colab free edition. Hence, you might want to debug locally (e.g. in a jupyter notebook) or in a CPU-only runtime on colab.

2 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
```

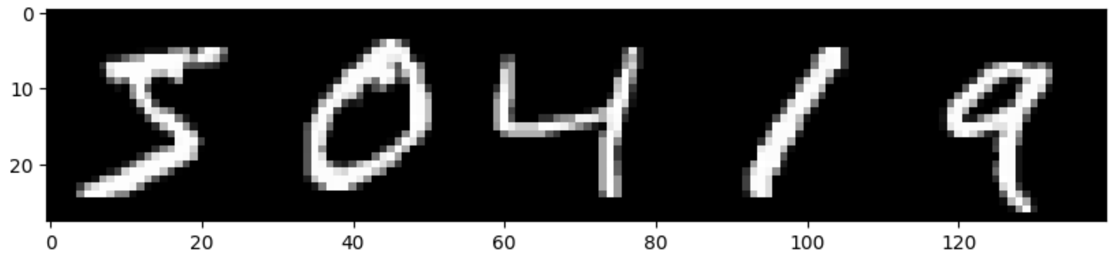
```
%load_ext autoreload
%autoreload 2
```

[ ]:
```
import torch
import torchvision
# device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')   #␣
 ↪use GPU if available
if torch.cuda.is_available():
    device = 'cuda:0'
elif torch.backends.mps.is_available():
    device = 'mps'
else:
    device = 'cpu'
print(f"Using device: {device}")

# 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)))
```

```
Using device: mps

 Download complete! Downloaded 60000 training examples!
```

[ ]:
```
from numpy.random.mtrand import sample
import matplotlib.pyplot as plt
import numpy as np

# Let's display some of the training samples.
sample_images = []
randomize = False # set to False for debugging
num_samples = 5 # simple data sampling for now, later we will use proper␣
 ↪DataLoader
if randomize:
  sample_idxs = np.random.randint(low=0,high=len(mnist_train), size=num_samples)
else:
  sample_idxs = list(range(num_samples))

for idx in sample_idxs:
  sample = mnist_train[idx]
  # print(f"Tensor w/ shape {sample[0][0].detach().cpu().numpy().shape} and␣
 ↪label {sample[1]}")
  sample_images.append(sample[0][0].data.cpu().numpy())
```

```
  # print(sample_images[0]) # Values are in [0, 1]

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



3 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 fully connected layers for the encoder. 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 decoder network that mirrors the architecture of the encoder. 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. nz 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.

3.1 Defining the Auto-Encoder Architecture [6pt]

[ ]:
```
import torch.nn as nn

# Prob1-1: Let's define encoder and decoder networks
class Encoder(nn.Module):
  def __init__(self, nz, input_size):
    super().__init__()
    self.input_size = input_size
    ################################# TODO␣
↪########################################
    # Create the network architecture using a nn.Sequential module wrapper.    ␣
↪   #
    # Encoder Architecture:                                                     ␣
↪   #
    # - input_size -> 256                                                       ␣
↪   #
    # - ReLU                                                                    ␣
↪   #
    # - 256 -> 64                                                               ␣
↪   #
    # - ReLU                                                                    ␣
↪   #
    # - 64 -> nz                                                                ␣
↪   #
    # HINT: Verify the shapes of intermediate layers by running partial␣
↪networks    #
    #         (with the next notebook cell) and visualizing the output shapes.  ␣
↪   #
   ␣
↪################################################################################
    # Create the network architecture using a nn.Sequential module wrapper
    self.net = nn.Sequential(
        nn.Linear(input_size, 256),
        nn.ReLU(),
        nn.Linear(256, 64),
        nn.ReLU(),
        nn.Linear(64, nz)
    )
    ################################# END TODO␣
↪########################################

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

```
class Decoder(nn.Module):
  def __init__(self, nz, output_size):
    super().__init__()
    self.output_size = output_size
    ################################# TODO␣
 ↪#########################################
    # Create the network architecture using a nn.Sequential module wrapper.    ␣
 ↪   #
    # Decoder Architecture (mirrors encoder architecture):                     ␣
 ↪   #
    # - nz -> 64                                                               ␣
 ↪   #
    # - ReLU                                                                   ␣
 ↪   #
    # - 64 -> 256                                                              ␣
 ↪   #
    # - ReLU                                                                   ␣
 ↪   #
    # - 256 -> output_size                                                     ␣
 ↪   #
   ␣
 ↪###############################################################################
    # Create the network architecture using a nn.Sequential module wrapper
    self.net = nn.Sequential(
        nn.Linear(nz, 64),
        nn.ReLU(),
        nn.Linear(64, 256),
        nn.ReLU(),
        nn.Linear(256, output_size),
        nn.Sigmoid()
    )

    ################################ END TODO␣
 ↪#######################################

  def forward(self, z):
    return self.net(z).reshape(-1, 1, self.output_size)
```

3.2 Testing the Auto-Encoder Forward Pass

[ ]:
```
# 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 = 2        # number of workers used for efficient data loading

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

```
# Create a PyTorch DataLoader object for efficiently generating training␣
↪batches.  #
# Make sure that the data loader automatically shuffles the training dataset.  ␣
↪   #
# Consider only *full* batches of data, to avoid torch errrors.              #
# 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. https://pytorch.org/docs/stable/data.html              ␣
↪   #
##############################################################################
mnist_data_loader = torch.utils.data.DataLoader(mnist_train,
                                                batch_size=batch_size,
                                                shuffle=True,
                                                num_workers=nworkers,
                                                drop_last=True)
##############################################################################

# now we can run a forward pass for encoder and decoder and check the produced␣
↪shapes
in_size = out_size = 28*28 # image size
nz = 32          # dimensionality of the learned embedding
encoder = Encoder(nz=nz, input_size=in_size)
decoder = Decoder(nz=nz, output_size=out_size)
for sample_img, sample_label in mnist_data_loader: # loads a batch of data
  input = sample_img.reshape([batch_size, in_size])
  print(f'{sample_img.shape=}, {type(sample_img)}, {input.shape=}')
  enc = encoder(input)
  print(f"Shape of encoding vector (should be [batch_size, nz]): {enc.shape}")
  dec = decoder(enc)
  print("Shape of decoded image (should be [batch_size, 1, out_size]): {}.".
↪format(dec.shape))
  break

del input, enc, dec, encoder, decoder, nworkers # remove to avoid confusion␣
↪later
```

```
sample_img.shape=torch.Size([64, 1, 28, 28]), <class 'torch.Tensor'>,
input.shape=torch.Size([64, 784])
Shape of encoding vector (should be [batch_size, nz]): torch.Size([64, 32])
Shape of decoded image (should be [batch_size, 1, out_size]): torch.Size([64, 1,
784]).
```

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=nz, input_size=in_size)
         self.decoder = Decoder(nz=nz, output_size=out_size)

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

       def reconstruct(self, x):
         """Only used later for visualization."""
         enc = self.encoder(x)
         flattened = self.decoder(enc)
         image = flattened.reshape(-1, 28, 28)
         return image
```

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

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

```
[ ]: import copy
     # Prob1-2
     epochs = 10
     learning_rate = 1e-3

     # build AE model
     print(f'Device available {device}')
     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␣
      ↪#######################################
     # Build the optimizer and loss classes. For the loss you can use a loss layer  ␣
      ↪  #
     # from the torch.nn package. We recommend binary cross entropy.                ␣
      ↪  #
     # HINT: We will use the Adam optimizer (learning rate given above, otherwise   ␣
      ↪  #
     #       default parameters).                                                    ␣
      ↪  #
     # NOTE: We could also use alternative losses like MSE and cross entropy,␣
      ↪depending #
     #       on the assumptions we are making about the output distribution.        ␣
      ↪  #
```

```
####################################################################################
optimizer = torch.optim.Adam(ae_model.parameters(), lr=learning_rate)
loss_fn = nn.BCELoss()
#################################### 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.    #
 ␣
 ↪####################################################################################
  for sample_img, sample_label in mnist_data_loader:
    input = sample_img.reshape([batch_size, in_size])
    input = input.to(device)
    recon = ae_model(input).reshape(-1,784)
    rec_loss = loss_fn(recon, input)
    optimizer.zero_grad()
    rec_loss.backward()
    optimizer.step()

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

print("Done!")
del epochs, learning_rate, sample_img, train_it, rec_loss #, opt
  #################################### END TODO␣
 ↪#####################################
```

```
Device available mps
Run Epoch 0
It 0: Reconstruction Loss: 0.6944908499717712
It 100: Reconstruction Loss: 0.2519420385360718
It 200: Reconstruction Loss: 0.23311863839626312
```

```
It 300: Reconstruction Loss: 0.19082558155059814
It 400: Reconstruction Loss: 0.18413308262825012
It 500: Reconstruction Loss: 0.16310472786426544
It 600: Reconstruction Loss: 0.1525726169347763
It 700: Reconstruction Loss: 0.14484992623329163
It 800: Reconstruction Loss: 0.14436911046504974
It 900: Reconstruction Loss: 0.1411139965057373
Run Epoch 1
It 1000: Reconstruction Loss: 0.12483111768960953
It 1100: Reconstruction Loss: 0.1331968754529953
It 1200: Reconstruction Loss: 0.1293070763349533
It 1300: Reconstruction Loss: 0.12582580745220184
It 1400: Reconstruction Loss: 0.12371337413787842
It 1500: Reconstruction Loss: 0.11845267564058304
It 1600: Reconstruction Loss: 0.12030372768640518
It 1700: Reconstruction Loss: 0.10330843925476074
It 1800: Reconstruction Loss: 0.1160634458065033
Run Epoch 2
It 1900: Reconstruction Loss: 0.1119118481874466
It 2000: Reconstruction Loss: 0.11211449652910233
It 2100: Reconstruction Loss: 0.10372737795114517
It 2200: Reconstruction Loss: 0.1118197813630104
It 2300: Reconstruction Loss: 0.10898920893669128
It 2400: Reconstruction Loss: 0.1107589453458786
It 2500: Reconstruction Loss: 0.10856975615024567
It 2600: Reconstruction Loss: 0.10179665684700012
It 2700: Reconstruction Loss: 0.10096292942762375
It 2800: Reconstruction Loss: 0.10604799538850784
Run Epoch 3
It 2900: Reconstruction Loss: 0.10868927836418152
It 3000: Reconstruction Loss: 0.10158700495958328
It 3100: Reconstruction Loss: 0.10117951780557632
It 3200: Reconstruction Loss: 0.1065385565161705
It 3300: Reconstruction Loss: 0.1074255108833313
It 3400: Reconstruction Loss: 0.10580065846443176
It 3500: Reconstruction Loss: 0.09795019030570984
It 3600: Reconstruction Loss: 0.10410849004983902
It 3700: Reconstruction Loss: 0.099971704185009
Run Epoch 4
It 3800: Reconstruction Loss: 0.10165152698755264
It 3900: Reconstruction Loss: 0.10274136811494827
It 4000: Reconstruction Loss: 0.10182314366102219
It 4100: Reconstruction Loss: 0.10511576384305954
It 4200: Reconstruction Loss: 0.09913355857133865
It 4300: Reconstruction Loss: 0.09909197688102722
It 4400: Reconstruction Loss: 0.10403913259506226
It 4500: Reconstruction Loss: 0.09910505264997482
It 4600: Reconstruction Loss: 0.0935821607708931
```

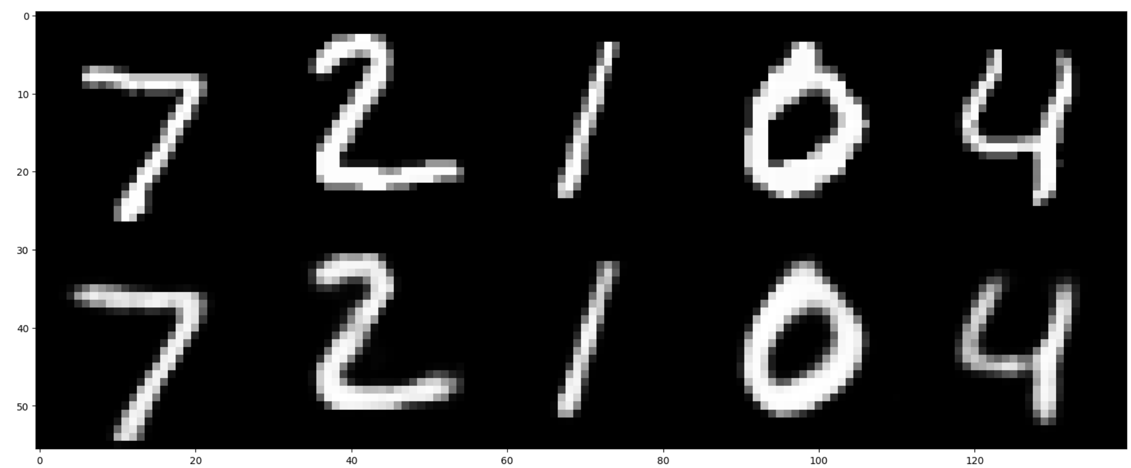
```
Run Epoch 5
It 4700: Reconstruction Loss: 0.0926743671298027
It 4800: Reconstruction Loss: 0.0933632031083107
It 4900: Reconstruction Loss: 0.08742174506187439
It 5000: Reconstruction Loss: 0.09385325014591217
It 5100: Reconstruction Loss: 0.09080643206834793
It 5200: Reconstruction Loss: 0.09340311586856842
It 5300: Reconstruction Loss: 0.09632482379674911
It 5400: Reconstruction Loss: 0.09385483711957932
It 5500: Reconstruction Loss: 0.08971249312162399
It 5600: Reconstruction Loss: 0.09635243564844131
Run Epoch 6
It 5700: Reconstruction Loss: 0.09132653474807739
It 5800: Reconstruction Loss: 0.09182647615671158
It 5900: Reconstruction Loss: 0.0957198292016983
It 6000: Reconstruction Loss: 0.0944468155503273
It 6100: Reconstruction Loss: 0.08400052040815353
It 6200: Reconstruction Loss: 0.09946037828922272
It 6300: Reconstruction Loss: 0.09410060197114944
It 6400: Reconstruction Loss: 0.09048306196928024
It 6500: Reconstruction Loss: 0.08898382633924484
Run Epoch 7
It 6600: Reconstruction Loss: 0.08681165426969528
It 6700: Reconstruction Loss: 0.0870070531964302
It 6800: Reconstruction Loss: 0.08675722032785416
It 6900: Reconstruction Loss: 0.0896509662270546
It 7000: Reconstruction Loss: 0.08388164639472961
It 7100: Reconstruction Loss: 0.08632978796958923
It 7200: Reconstruction Loss: 0.0847453698515892
It 7300: Reconstruction Loss: 0.09550989419221878
It 7400: Reconstruction Loss: 0.0886843129992485
Run Epoch 8
It 7500: Reconstruction Loss: 0.08556409925222397
It 7600: Reconstruction Loss: 0.09089229255914688
It 7700: Reconstruction Loss: 0.08732468634843826
It 7800: Reconstruction Loss: 0.08104748278856277
It 7900: Reconstruction Loss: 0.08924427628517151
It 8000: Reconstruction Loss: 0.08808325231075287
It 8100: Reconstruction Loss: 0.08264004439115524
It 8200: Reconstruction Loss: 0.0868585929274559
It 8300: Reconstruction Loss: 0.08666516840457916
It 8400: Reconstruction Loss: 0.08399076014757156
Run Epoch 9
It 8500: Reconstruction Loss: 0.0842433050274849
It 8600: Reconstruction Loss: 0.09101127833127975
It 8700: Reconstruction Loss: 0.08973096311092377
It 8800: Reconstruction Loss: 0.09299307316541672
It 8900: Reconstruction Loss: 0.08950456231832504
```

```
It 9000: Reconstruction Loss: 0.08407291024923325
It 9100: Reconstruction Loss: 0.08939047157764435
It 9200: Reconstruction Loss: 0.08108043670654297
It 9300: Reconstruction Loss: 0.07942094653844833
Done!
```

3.4 Verifying reconstructions

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, randomize=False):
  # download MNIST test set + build Dataset object
  mnist_test = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          download=True,
                                          transform=torchvision.transforms.
 ↪ToTensor())
  model.eval()      # set model in evalidation mode (eg freeze batchnorm params)
  num_samples = 5
  if randomize:
    sample_idxs = np.random.randint(low=0,high=len(mnist_test),␣
 ↪size=num_samples)
  else:
    sample_idxs = list(range(num_samples))

  input_imgs, test_reconstructions = [], []
  for idx in sample_idxs:
    sample = mnist_test[idx]
    input_img = np.asarray(sample[0])
    input_flat = input_img.reshape(784)
    reconstruction = model.reconstruct(torch.tensor(input_flat, device=device))

    input_imgs.append(input_img[0])
    test_reconstructions.append(reconstruction[0].data.cpu().numpy())
    # print(f'{input_img[0].shape=}\t{reconstruction.shape}')

  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, randomize=False) # set randomize to False for␣
 ↪debugging
```

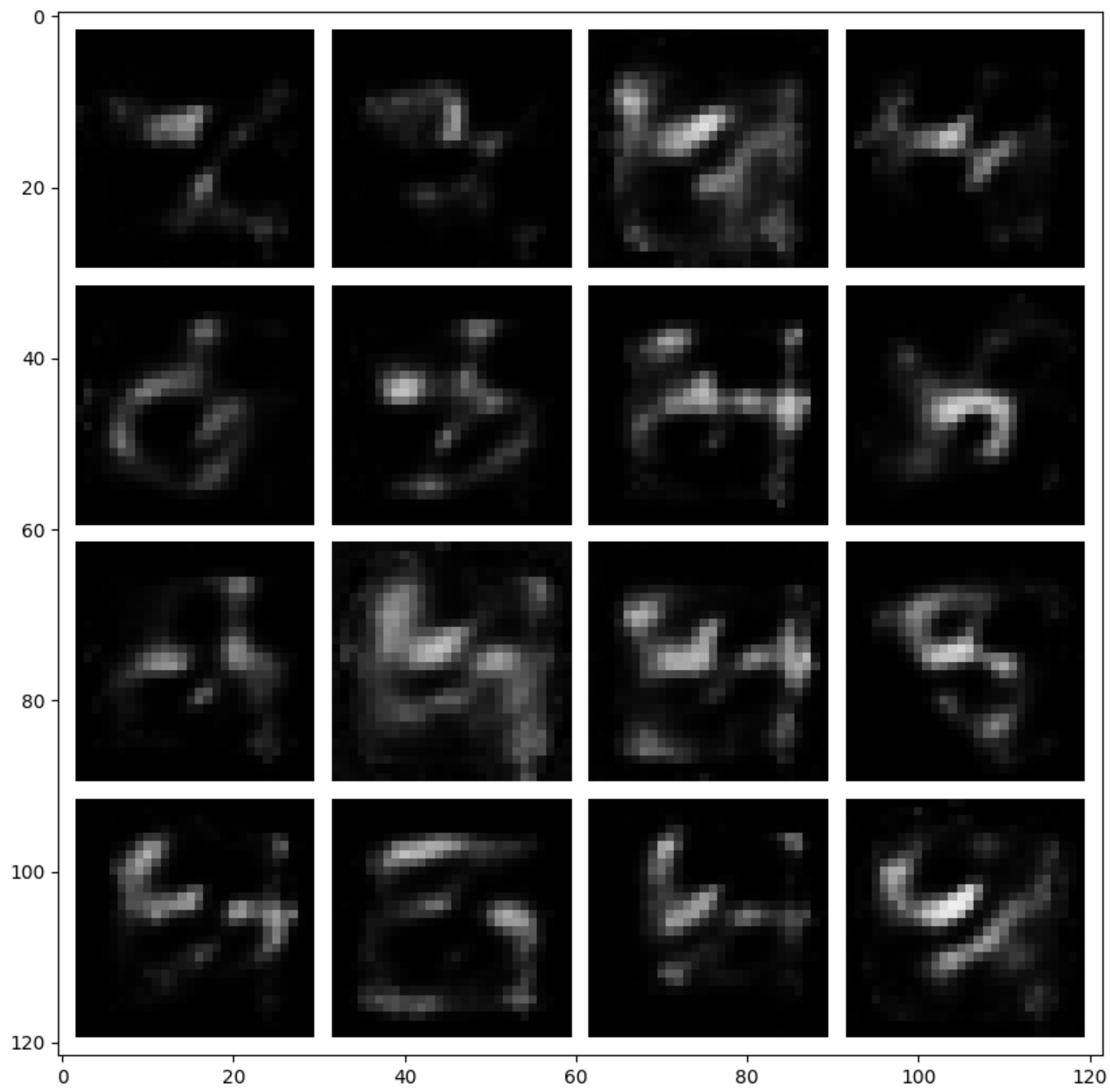


3.5 Sampling from the Auto-Encoder [2pt]

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.

[ ]:
```python
# we will sample N embeddings, then decode and visualize them
def vis_samples(model):
    ######################################## TODO ########################################
    # Prob1-3 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.                        #
    ########################################################################################
```

```
    z = torch.randn(batch_size, nz).to(device)

    with torch.no_grad():
        decoded_samples = model.decoder(z)
    decoded_samples = decoded_samples.clamp(0, 1)
    decoded_samples = decoded_samples.reshape((-1, 1, 28, 28))

    #################################### 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)
```



Prob1-3 continued: Inline Question: Describe your observations, why do you think they occur? [2pt] (max 150 words)

Answer:

Although we can use AutoEncoders as a generative model, they are not specifically designed for this purpose. The primary objective of an autoencoder is to learn a compressed representation of the input data, with the goal of reconstructing the original input as accurately as possible.

The compressed representation learned by an AutoEncoder may not able to capture the full structure of the original data distribution. As a result, the generated samples may appear random or not belong to the original distribution, as they are essentially being generated by sampling from an incomplete or inaccurate representation of the data.

To address this, techniques such as Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs) have been developed, which explicitly model the structure of the data distribution and can generate more realistic samples.

4 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):

$$p(x) > \underbrace{\mathbb{E}_{z\sim q(z|x)} p(x|z)}_{\text{reconstruction}} - \underbrace{D_{\text{KL}}(q(z|x), p(z))}_{\text{prior divergence}}$$

Here, $D_{\text{KL}}(q, p)$ denotes the Kullback-Leibler (KL) divergence between the posterior distribution $q(z|x)$, 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 choose a unit Gaussian prior again. The first term is the reconstruction term we already know from training the auto-encoder. When assuming a Gaussian output distribution for both encoder $q(z|x)$ and decoder $p(x|z)$ the objective reduces to:

$$\mathcal{L}_{\text{VAE}} = \sum_{x\sim\mathcal{D}} \mathcal{L}_{\text{rec}}(x, \hat{x}) - \beta \cdot D_{\text{KL}}(\mathcal{N}(\mu_q, \sigma_q), \mathcal{N}(0, I))$$

Here, $\hat{x}$ is the reconstruction output of the decoder. In comparison to the auto-encoder objective, 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

4.0.1 Reparametrization Trick

The sampling procedure inside the VAE's forward pass for obtaining a sample z from the posterior distribution $q(z|x)$, when implemented naively, is non-differentiable. However, since $q(z|x)$ 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 $z \sim \mathcal{N}(\mu_q, \sigma_q)$ we can "separate" the network's predictions and the random sampling by computing the sample as:

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

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.

4.1 Defining the VAE Model [7pt]

[ ]:
```
import torch.nn.functional as F

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


# Prob1-4
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 needs to    ␣
↪       #
    #       double. The decoder works with an embedding sampled from this␣
↪output.  #
  ␣
↪#####################################################################################
    self.encoder = Encoder(nz=nz*2, input_size=in_size)
    self.decoder = Decoder(nz=nz, output_size=out_size)
    self.nz = nz
    self.loss_fn = nn.BCELoss(reduction='mean')
    ###################################### 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                          ␣
↪       #
        ␣
↪#########################################################################################
        if x.dim() > 2:
                x = x.view(-1, 28*28)

        q = self.encoder(x)
        mu, log_sigma = torch.chunk(q, 2, dim=-1)

        # sample latent variable z with reparametrization
        eps = torch.normal(mean=torch.zeros_like(mu), std=torch.
↪ones_like(log_sigma))
        # eps = torch.randn_like(mu) # Alternatively use this
        z = mu + eps * torch.exp(log_sigma)

        # compute reconstruction
        reconstruction = self.decoder(z)

        #################################### 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 loss␣
↪above      #
        #          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)                        ␣
↪       #
  ␣
↪##################################################################################
    rec_loss = self.loss_fn(outputs['rec'].reshape(-1,784), x)
    mu, log_sigma = torch.chunk(outputs['q'], 2, dim=-1)
    kl_loss = kl_divergence(mu, log_sigma, torch.zeros_like(mu), torch.
↪zeros_like(log_sigma)).mean()
    #################################### END TODO␣
↪######################################

    # return weighted 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.                                              ␣
↪       #
  ␣
↪##################################################################################
    enc = self.encoder(x)
    mu, log_sigma = torch.chunk(enc, 2, dim=-1)
    flattened = self.decoder(mu)
    image = flattened.reshape(-1, 28, 28)

    #################################### END TODO␣
↪######################################
    return image
```

4.2 Setting up the VAE Training Loop [4pt]

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

[ ]:
```
# Prob1-5 VAE training loop
learning_rate = 1e-3
nz = 32
beta = 0

####################################### TODO␣
 ↪#######################################
epochs = 10      # recommended 5-20 epochs
#################################### 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.                  ␣
 ↪   #
###################################################################################
# same as AE
optimizer = torch.optim.Adam(vae_model.parameters(), lr=learning_rate)
#################################### END TODO␣
 ↪######################################

train_it = 0
rec_loss, kl_loss = [], []
print(f"Running {epochs} epochs with {beta=}")
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 sample_images, sample_labels in mnist_data_loader:
      # reshape images to (batch_size, 784)
      images = sample_images.reshape([batch_size, in_size])
      images = images.to(device)

      # reset gradients
      optimizer.zero_grad()

      # forward pass
      outputs = vae_model(images)

      # compute loss
      loss_dict = vae_model.loss(images, outputs)
      total_loss = loss_dict[0]
      losses = loss_dict[1]
      # print(total_loss)

      # backward pass
      total_loss.backward()

      # perform one optimization step
      optimizer.step()

      # save losses for plotting
      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
    ################################### END TODO␣
  ↪###################################

print("Done!")
```

```python
rec_loss_plotdata = [foo.detach().cpu() for foo in rec_loss]
kl_loss_plotdata = [foo.detach().cpu() for foo in kl_loss]

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

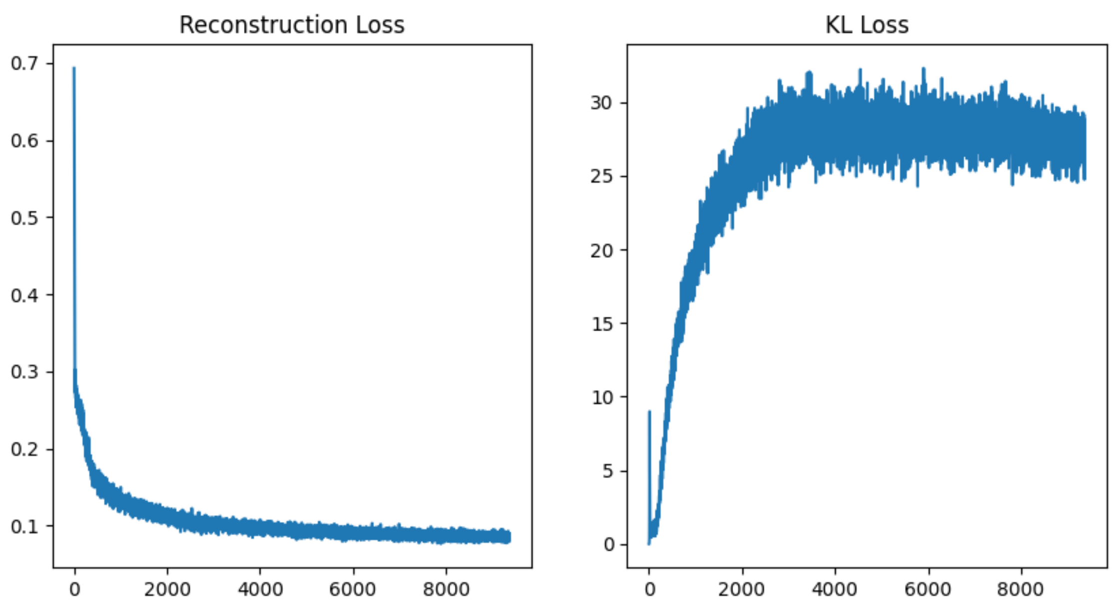
```
Running 10 epochs with beta=0
Run Epoch 0
It 0: Total Loss: 0.693139374256134,     Rec Loss: 0.693139374256134,    KL
Loss: 0.007166516967117786
It 100: Total Loss: 0.25726062059402466,         Rec Loss: 0.25726062059402466,
KL Loss: 0.7997620105743408
It 200: Total Loss: 0.2321595847606659,          Rec Loss: 0.2321595847606659,
KL Loss: 1.933419942855835
It 300: Total Loss: 0.1853952407836914,          Rec Loss: 0.1853952407836914,
KL Loss: 5.878175735473633
It 400: Total Loss: 0.18141287565231323,         Rec Loss: 0.18141287565231323,
KL Loss: 9.587563514709473
It 500: Total Loss: 0.1593417078256607,          Rec Loss: 0.1593417078256607,
KL Loss: 11.438508033752441
It 600: Total Loss: 0.15346018970012665,         Rec Loss: 0.15346018970012665,
KL Loss: 13.681218147277832
It 700: Total Loss: 0.16121311485767365,         Rec Loss: 0.16121311485767365,
KL Loss: 13.72098445892334
It 800: Total Loss: 0.13179124891757965,         Rec Loss: 0.13179124891757965,
KL Loss: 18.52638816833496
It 900: Total Loss: 0.13383376598358154,         Rec Loss: 0.13383376598358154,
KL Loss: 18.259340286254883
Run Epoch 1
It 1000: Total Loss: 0.13233406841754913,        Rec Loss: 0.13233406841754913,
KL Loss: 19.455059051513672
It 1100: Total Loss: 0.12734082341194153,        Rec Loss: 0.12734082341194153,
KL Loss: 20.336734771728516
It 1200: Total Loss: 0.1261293590068817,         Rec Loss: 0.1261293590068817,
KL Loss: 20.51131820678711
It 1300: Total Loss: 0.12314817309379578,        Rec Loss: 0.12314817309379578,
KL Loss: 23.14322853088379
It 1400: Total Loss: 0.12378254532814026,        Rec Loss: 0.12378254532814026,
KL Loss: 24.080059051513672
It 1500: Total Loss: 0.13319449126720428,        Rec Loss: 0.13319449126720428,
```

```
KL Loss: 22.664691925048828
It 1600: Total Loss: 0.1110578328371048,        Rec Loss: 0.1110578328371048,
KL Loss: 23.385356903076172
It 1700: Total Loss: 0.11692888289690018,       Rec Loss: 0.11692888289690018,
KL Loss: 25.336336135864258
It 1800: Total Loss: 0.11153991520404816,       Rec Loss: 0.11153991520404816,
KL Loss: 22.57352066040039
Run Epoch 2
It 1900: Total Loss: 0.11998095363378525,       Rec Loss: 0.11998095363378525,
KL Loss: 25.09699249267578
It 2000: Total Loss: 0.10968552529811859,       Rec Loss: 0.10968552529811859,
KL Loss: 24.46449089050293
It 2100: Total Loss: 0.1115984097123146,        Rec Loss: 0.1115984097123146,
KL Loss: 26.62747573852539
It 2200: Total Loss: 0.10767600685358047,       Rec Loss: 0.10767600685358047,
KL Loss: 25.76189422607422
It 2300: Total Loss: 0.11042513698339462,       Rec Loss: 0.11042513698339462,
KL Loss: 25.85028076171875
It 2400: Total Loss: 0.10536207258701324,       Rec Loss: 0.10536207258701324,
KL Loss: 26.7633056640625
It 2500: Total Loss: 0.10463270545005798,       Rec Loss: 0.10463270545005798,
KL Loss: 25.36116600036621
It 2600: Total Loss: 0.10580159723758698,       Rec Loss: 0.10580159723758698,
KL Loss: 27.369918823242188
It 2700: Total Loss: 0.10498428344726562,       Rec Loss: 0.10498428344726562,
KL Loss: 27.5137882232666
It 2800: Total Loss: 0.10050584375858307,       Rec Loss: 0.10050584375858307,
KL Loss: 28.111988067626953
Run Epoch 3
It 2900: Total Loss: 0.10421314835548401,       Rec Loss: 0.10421314835548401,
KL Loss: 27.645832061767578
It 3000: Total Loss: 0.09871480613946915,       Rec Loss: 0.09871480613946915,
KL Loss: 28.14415168762207
It 3100: Total Loss: 0.1006021648645401,        Rec Loss: 0.1006021648645401,
KL Loss: 28.360763549804688
It 3200: Total Loss: 0.09511252492666245,       Rec Loss: 0.09511252492666245,
KL Loss: 26.122705459594727
It 3300: Total Loss: 0.10536639392375946,       Rec Loss: 0.10536639392375946,
KL Loss: 28.16770362854004
It 3400: Total Loss: 0.10239254683256149,       Rec Loss: 0.10239254683256149,
KL Loss: 28.043190002441406
It 3500: Total Loss: 0.09968145191669464,       Rec Loss: 0.09968145191669464,
KL Loss: 28.412986755371094
It 3600: Total Loss: 0.09947700798511505,       Rec Loss: 0.09947700798511505,
KL Loss: 27.731338500976562
It 3700: Total Loss: 0.10559655725955963,       Rec Loss: 0.10559655725955963,
KL Loss: 28.734169006347656
Run Epoch 4
```

```
It 3800: Total Loss: 0.10000119358301163,        Rec Loss: 0.10000119358301163,
KL Loss: 29.41818618774414
It 3900: Total Loss: 0.09355659782886505,        Rec Loss: 0.09355659782886505,
KL Loss: 27.31559181213379
It 4000: Total Loss: 0.09513528645038605,        Rec Loss: 0.09513528645038605,
KL Loss: 30.087352752685547
It 4100: Total Loss: 0.10039179772138596,        Rec Loss: 0.10039179772138596,
KL Loss: 27.11927604675293
It 4200: Total Loss: 0.09850852936506271,        Rec Loss: 0.09850852936506271,
KL Loss: 28.463106155395508
It 4300: Total Loss: 0.09391020238399506,        Rec Loss: 0.09391020238399506,
KL Loss: 26.624359130859375
It 4400: Total Loss: 0.09612996131181717,        Rec Loss: 0.09612996131181717,
KL Loss: 28.24752426147461
It 4500: Total Loss: 0.09539452195167542,        Rec Loss: 0.09539452195167542,
KL Loss: 29.633399963378906
It 4600: Total Loss: 0.1000458374619484,         Rec Loss: 0.1000458374619484,
KL Loss: 29.553466796875
Run Epoch 5
It 4700: Total Loss: 0.09218546003103256,        Rec Loss: 0.09218546003103256,
KL Loss: 29.564186096191406
It 4800: Total Loss: 0.09185317903757095,        Rec Loss: 0.09185317903757095,
KL Loss: 28.5970458984375
It 4900: Total Loss: 0.08902092278003693,        Rec Loss: 0.08902092278003693,
KL Loss: 26.972000122070312
It 5000: Total Loss: 0.1009882465004921,         Rec Loss: 0.1009882465004921,
KL Loss: 28.596431732177734
It 5100: Total Loss: 0.08787818998098373,        Rec Loss: 0.08787818998098373,
KL Loss: 26.96091079711914
It 5200: Total Loss: 0.09619930386543274,        Rec Loss: 0.09619930386543274,
KL Loss: 26.59955596923828
It 5300: Total Loss: 0.09046325087547302,        Rec Loss: 0.09046325087547302,
KL Loss: 26.898035049438477
It 5400: Total Loss: 0.09099748730659485,        Rec Loss: 0.09099748730659485,
KL Loss: 27.338809967041016
It 5500: Total Loss: 0.08790378272533417,        Rec Loss: 0.08790378272533417,
KL Loss: 24.809370040893555
It 5600: Total Loss: 0.09502513706684113,        Rec Loss: 0.09502513706684113,
KL Loss: 28.927932739257812
Run Epoch 6
It 5700: Total Loss: 0.08754625171422958,        Rec Loss: 0.08754625171422958,
KL Loss: 27.347936630249023
It 5800: Total Loss: 0.096025250852108,          Rec Loss: 0.096025250852108,
KL Loss: 29.578842163085938
It 5900: Total Loss: 0.08966736495494843,        Rec Loss: 0.08966736495494843,
KL Loss: 29.024734497070312
It 6000: Total Loss: 0.09179459512233734,        Rec Loss: 0.09179459512233734,
KL Loss: 27.7093448638916
```

It 6100: Total Loss: 0.08653895556926727, Rec Loss: 0.08653895556926727,
KL Loss: 27.525041580200195
It 6200: Total Loss: 0.08655117452144623, Rec Loss: 0.08655117452144623,
KL Loss: 28.31140899658203
It 6300: Total Loss: 0.09244192391633987, Rec Loss: 0.09244192391633987,
KL Loss: 28.949811935424805
It 6400: Total Loss: 0.08696801960468292, Rec Loss: 0.08696801960468292,
KL Loss: 27.495460510253906
It 6500: Total Loss: 0.08717242628335953, Rec Loss: 0.08717242628335953,
KL Loss: 26.773033142089844
Run Epoch 7
It 6600: Total Loss: 0.09222833067178726, Rec Loss: 0.09222833067178726,
KL Loss: 27.747562408447266
It 6700: Total Loss: 0.09318392723798752, Rec Loss: 0.09318392723798752,
KL Loss: 28.844112396240234
It 6800: Total Loss: 0.08574678003787994, Rec Loss: 0.08574678003787994,
KL Loss: 28.68630599975586
It 6900: Total Loss: 0.0885276272892952, Rec Loss: 0.0885276272892952,
KL Loss: 28.993236541748047
It 7000: Total Loss: 0.08123018592596054, Rec Loss: 0.08123018592596054,
KL Loss: 27.073055267333984
It 7100: Total Loss: 0.08998767286539078, Rec Loss: 0.08998767286539078,
KL Loss: 29.50719451904297
It 7200: Total Loss: 0.08967763185501099, Rec Loss: 0.08967763185501099,
KL Loss: 27.491901397705078
It 7300: Total Loss: 0.08911441266536713, Rec Loss: 0.08911441266536713,
KL Loss: 28.332143783569336
It 7400: Total Loss: 0.08155931532382965, Rec Loss: 0.08155931532382965,
KL Loss: 26.974740982055664
Run Epoch 8
It 7500: Total Loss: 0.0900396779179573, Rec Loss: 0.0900396779179573,
KL Loss: 27.175785064697266
It 7600: Total Loss: 0.0886014774441719, Rec Loss: 0.0886014774441719,
KL Loss: 28.093124389648438
It 7700: Total Loss: 0.08395019918680191, Rec Loss: 0.08395019918680191,
KL Loss: 27.360580444335938
It 7800: Total Loss: 0.08890915662050247, Rec Loss: 0.08890915662050247,
KL Loss: 28.563270568847656
It 7900: Total Loss: 0.08736532181501389, Rec Loss: 0.08736532181501389,
KL Loss: 29.589824676513672
It 8000: Total Loss: 0.0899948924779892, Rec Loss: 0.0899948924779892,
KL Loss: 27.658733367919922
It 8100: Total Loss: 0.08425834774971008, Rec Loss: 0.08425834774971008,
KL Loss: 28.400962829589844
It 8200: Total Loss: 0.0844934806227684, Rec Loss: 0.0844934806227684,
KL Loss: 27.537267684936523
It 8300: Total Loss: 0.08953127264976501, Rec Loss: 0.08953127264976501,
KL Loss: 27.549537658691406

```
It 8400: Total Loss: 0.09083836525678635,        Rec Loss: 0.09083836525678635,
KL Loss: 29.148975372314453
Run Epoch 9
It 8500: Total Loss: 0.07932011038064957,        Rec Loss: 0.07932011038064957,
KL Loss: 26.48763084411621
It 8600: Total Loss: 0.08147520571947098,        Rec Loss: 0.08147520571947098,
KL Loss: 28.215667724609375
It 8700: Total Loss: 0.08931191265583038,        Rec Loss: 0.08931191265583038,
KL Loss: 27.98303985595703
It 8800: Total Loss: 0.08093380182981491,        Rec Loss: 0.08093380182981491,
KL Loss: 27.026912689208984
It 8900: Total Loss: 0.08588673919439316,        Rec Loss: 0.08588673919439316,
KL Loss: 26.11998176574707
It 9000: Total Loss: 0.08817160874605179,        Rec Loss: 0.08817160874605179,
KL Loss: 25.74666976928711
It 9100: Total Loss: 0.08734923601150513,        Rec Loss: 0.08734923601150513,
KL Loss: 27.335716247558594
It 9200: Total Loss: 0.09080950170755386,        Rec Loss: 0.09080950170755386,
KL Loss: 28.744606018066406
It 9300: Total Loss: 0.07991605997085571,        Rec Loss: 0.07991605997085571,
KL Loss: 26.440622329711914
Done!
```

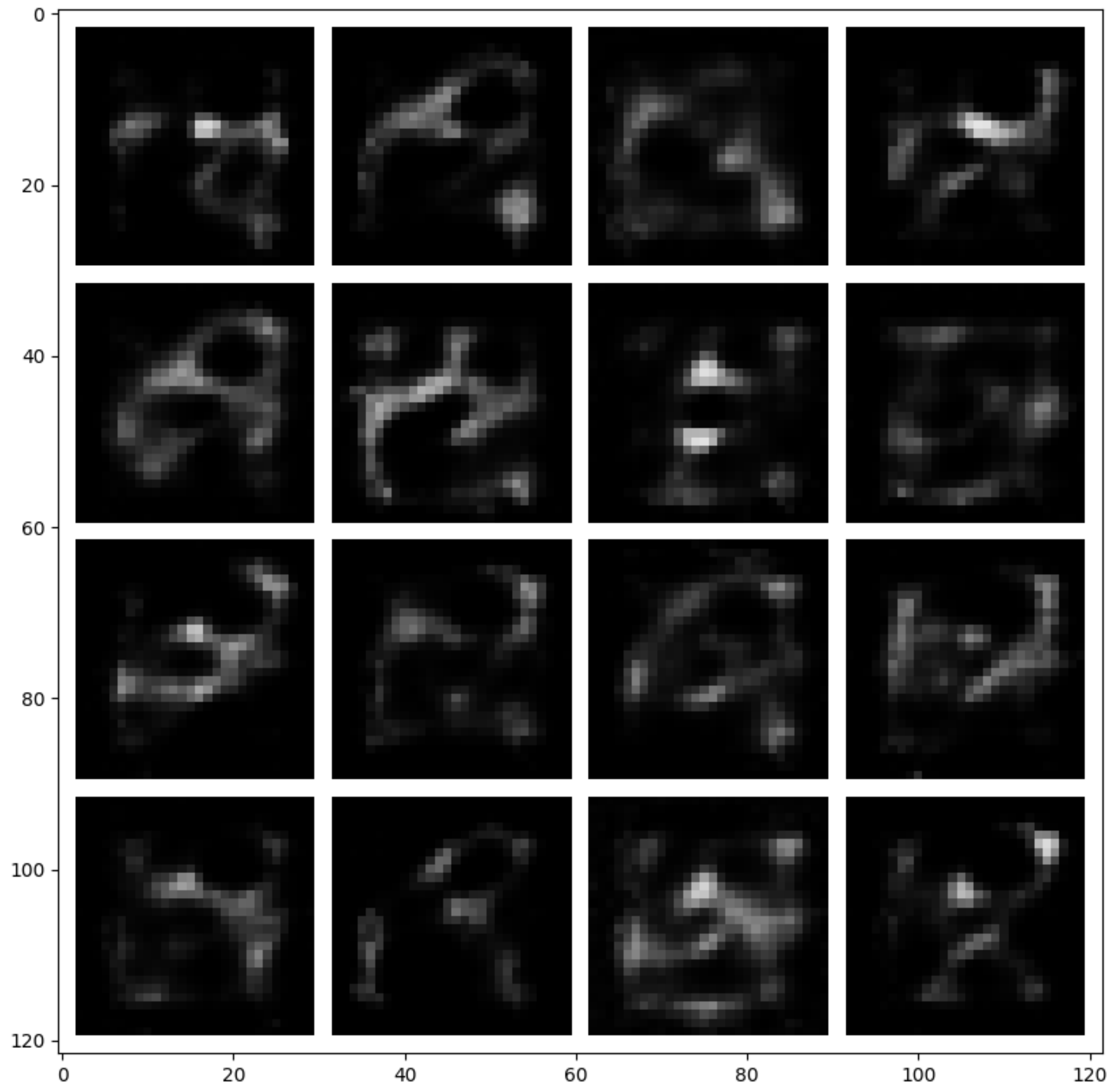


Let's look at some reconstructions and decoded embedding samples!

[ ]:
```python
# visualize VAE reconstructions and samples from the generative model
print("beta = ", beta)
vis_reconstruction(vae_model, randomize=True)
```

```
vis_samples(vae_model)
```

```
beta =  0
```



4.3 Tweaking the loss function β [2pt]

Prob1-6: Let's repeat the same experiment for $\beta = 10$, a very high value for the coefficient.

[ ]:
```
# VAE training loop
learning_rate = 1e-3
nz = 32
beta = 10

######################################## TODO␣
 ↪########################################
epochs = 10        # recommended 5-20 epochs
```

```
##################################### 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.                    ␣
↪   #
#####################################################################################
# same as AE
optimizer = torch.optim.Adam(vae_model.parameters(), lr=learning_rate)
##################################### END TODO␣
↪#######################################

train_it = 0
rec_loss, kl_loss = [], []
print(f"Running {epochs} epochs with {beta=}")
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 sample_images, sample_labels in mnist_data_loader:
    # reshape images to (batch_size, 784)
    images = sample_images.reshape([batch_size, in_size])
    images = images.to(device)

    # reset gradients
    optimizer.zero_grad()

    # forward pass
    outputs = vae_model(images)

    # compute loss
    loss_dict = vae_model.loss(images, outputs)
    total_loss = loss_dict[0]
    losses = loss_dict[1]
    # print(total_loss)

    # backward pass
    total_loss.backward()

    # perform one optimization step
    optimizer.step()

    # save losses for plotting
    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
  ###################################### END TODO␣
 ↪#####################################

print("Done!")

rec_loss_plotdata = [foo.detach().cpu() for foo in rec_loss]
kl_loss_plotdata = [foo.detach().cpu() for foo in kl_loss]

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

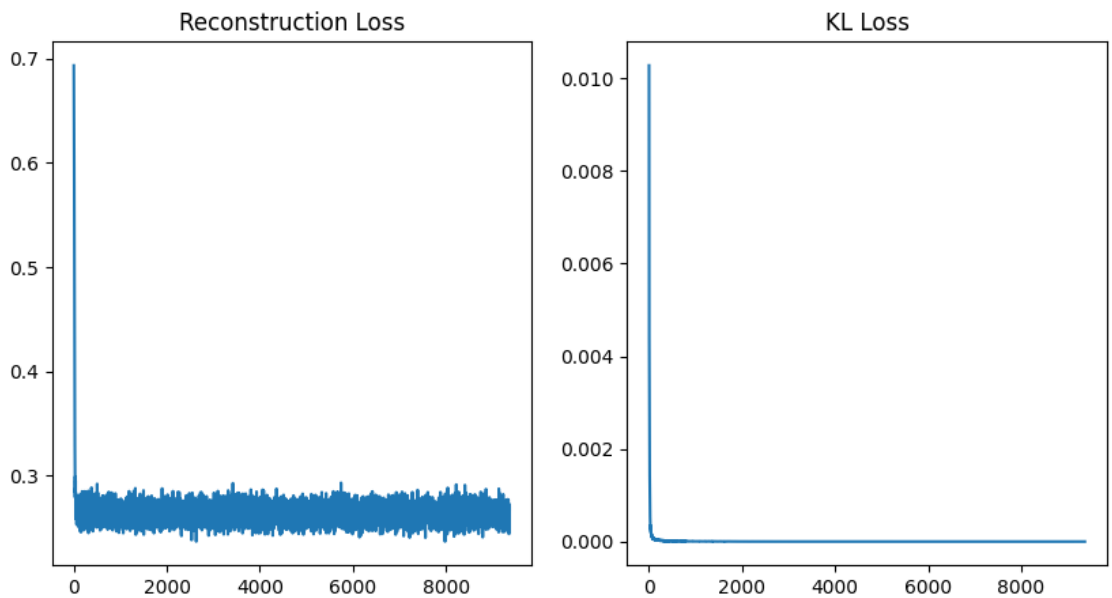
```
Running 10 epochs with beta=10
Run Epoch 0
It 0: Total Loss: 0.7963533401489258,    Rec Loss: 0.6936022639274597,   KL
Loss: 0.010275107808411121
It 100: Total Loss: 0.25947821140289307,         Rec Loss: 0.2588992416858673,
KL Loss: 5.789594433736056e-05
It 200: Total Loss: 0.25660958886146545,         Rec Loss: 0.2563475966453552,
KL Loss: 2.619785664137453e-05
It 300: Total Loss: 0.2566053569316864,          Rec Loss: 0.25639572739601135,
KL Loss: 2.096427488140762e-05
It 400: Total Loss: 0.27249875664711,    Rec Loss: 0.272353857755661,    KL
Loss: 1.4489094610325992e-05
It 500: Total Loss: 0.26842647790908813,         Rec Loss: 0.2683154046535492,
KL Loss: 1.1106327292509377e-05
It 600: Total Loss: 0.2634994685649872,          Rec Loss: 0.2634139657020569,
KL Loss: 8.550225174985826e-06
It 700: Total Loss: 0.2615213096141815,          Rec Loss: 0.2614284157752991,
KL Loss: 9.288400178775191e-06
It 800: Total Loss: 0.26968181133270264,         Rec Loss: 0.2696245014667511,
KL Loss: 5.729816621169448e-06
It 900: Total Loss: 0.27115723490715027,         Rec Loss: 0.27109894156455994,
KL Loss: 5.829439032822847e-06
Run Epoch 1
It 1000: Total Loss: 0.2629452645778656,         Rec Loss: 0.26290157437324524,
KL Loss: 4.37026028521359e-06
It 1100: Total Loss: 0.2637653350830078,         Rec Loss: 0.2637219727039337,
KL Loss: 4.336150595918298e-06
It 1200: Total Loss: 0.26680466532707214,        Rec Loss: 0.266767293214798,
KL Loss: 3.738256054930389e-06
It 1300: Total Loss: 0.26636070013046265,        Rec Loss: 0.2663077116012573,
KL Loss: 5.298614269122481e-06
It 1400: Total Loss: 0.2657104730606079,         Rec Loss: 0.26568371057510376,
KL Loss: 2.675544237717986e-06
It 1500: Total Loss: 0.2635616958141327,         Rec Loss: 0.2635292410850525,
KL Loss: 3.245004336349666e-06
It 1600: Total Loss: 0.2799365520477295,         Rec Loss: 0.2799019515514374,
KL Loss: 3.4589029382914305e-06
It 1700: Total Loss: 0.2598848044872284,         Rec Loss: 0.25986117124557495,
KL Loss: 2.3647735361009836e-06
It 1800: Total Loss: 0.25960513949394226,        Rec Loss: 0.2595805525779724,
KL Loss: 2.458080416545272e-06
Run Epoch 2
It 1900: Total Loss: 0.2637275755405426,         Rec Loss: 0.263708233833313,
KL Loss: 1.9342260202392936e-06
It 2000: Total Loss: 0.266984760761261,          Rec Loss: 0.2669712007045746,
KL Loss: 1.3558019418269396e-06
It 2100: Total Loss: 0.26951026916503906,        Rec Loss: 0.26949355006217957,
KL Loss: 1.671127392910421e-06
```

```
It 2200: Total Loss: 0.2641616761684418,        Rec Loss: 0.2641494870185852,
KL Loss: 1.2177915778011084e-06
It 2300: Total Loss: 0.2603447437286377,        Rec Loss: 0.26033255457878113,
KL Loss: 1.219086698256433e-06
It 2400: Total Loss: 0.26416951417922974,       Rec Loss: 0.26415854692459106,
KL Loss: 1.0963121894747019e-06
It 2500: Total Loss: 0.26564595103263855,       Rec Loss: 0.2656334340572357,
KL Loss: 1.251770299859345e-06
It 2600: Total Loss: 0.2627035677433014,        Rec Loss: 0.26269668340682983,
KL Loss: 6.893533281981945e-07
It 2700: Total Loss: 0.2720264196395874,        Rec Loss: 0.27201807498931885,
KL Loss: 8.344068191945553e-07
It 2800: Total Loss: 0.26738885045051575,       Rec Loss: 0.26737770438194275,
KL Loss: 1.114996848627925e-06
Run Epoch 3
It 2900: Total Loss: 0.27197501063346863,       Rec Loss: 0.27196574211120605,
KL Loss: 9.25589120015502e-07
It 3000: Total Loss: 0.26365235447883606,       Rec Loss: 0.2636394798755646,
KL Loss: 1.2877280823886395e-06
It 3100: Total Loss: 0.26127931475639343,       Rec Loss: 0.26127177476882935,
KL Loss: 7.550843292847276e-07
It 3200: Total Loss: 0.2582951486110687,        Rec Loss: 0.25828829407691956,
KL Loss: 6.85802660882473e-07
It 3300: Total Loss: 0.26722797751426697,       Rec Loss: 0.2672210931777954,
KL Loss: 6.898917490616441e-07
It 3400: Total Loss: 0.263291597366333,         Rec Loss: 0.2632862329483032,
KL Loss: 5.367182893678546e-07
It 3500: Total Loss: 0.2673724591732025,        Rec Loss: 0.2673671543598175,
KL Loss: 5.309702828526497e-07
It 3600: Total Loss: 0.26910069584846497,       Rec Loss: 0.2690948247909546,
KL Loss: 5.861802492290735e-07
It 3700: Total Loss: 0.27974390983581543,       Rec Loss: 0.2797384262084961,
KL Loss: 5.492620402947068e-07
Run Epoch 4
It 3800: Total Loss: 0.26390475034713745,       Rec Loss: 0.26390257477760315,
KL Loss: 2.1737650968134403e-07
It 3900: Total Loss: 0.2580288350582123,        Rec Loss: 0.2580243945121765,
KL Loss: 4.455359885469079e-07
It 4000: Total Loss: 0.2617935240268707,        Rec Loss: 0.26179179549217224,
KL Loss: 1.7430284060537815e-07
It 4100: Total Loss: 0.2529808580875397,        Rec Loss: 0.2529771029949188,
KL Loss: 3.7462450563907623e-07
It 4200: Total Loss: 0.2544715106487274,        Rec Loss: 0.25446903705596924,
KL Loss: 2.4602923076599836e-07
It 4300: Total Loss: 0.26757344603538513,       Rec Loss: 0.2675706744194031,
KL Loss: 2.771848812699318e-07
It 4400: Total Loss: 0.2784442901611328,        Rec Loss: 0.27844110131263733,
KL Loss: 3.181776264682412e-07
```

```
It 4500: Total Loss: 0.27326521277427673,        Rec Loss: 0.2732624411582947,
KL Loss: 2.7776695787906647e-07
It 4600: Total Loss: 0.2546224296092987,         Rec Loss: 0.2546194791793823,
KL Loss: 2.9652437660843134e-07
Run Epoch 5
It 4700: Total Loss: 0.25699248909950256,        Rec Loss: 0.2569904327392578,
KL Loss: 2.048327587544918e-07
It 4800: Total Loss: 0.2685321569442749,         Rec Loss: 0.2685300409793854,
KL Loss: 2.1011510398238897e-07
It 4900: Total Loss: 0.26363807916641235,        Rec Loss: 0.26363590359687805,
KL Loss: 2.1797313820570707e-07
It 5000: Total Loss: 0.25046202540397644,        Rec Loss: 0.250460684299469,
KL Loss: 1.3342651072889566e-07
It 5100: Total Loss: 0.2744469940662384,         Rec Loss: 0.27444544434547424,
KL Loss: 1.5583646018058062e-07
It 5200: Total Loss: 0.27468639612197876,        Rec Loss: 0.27468428015708923,
KL Loss: 2.117303665727377e-07
It 5300: Total Loss: 0.25915753841400146,        Rec Loss: 0.25915589928627014,
KL Loss: 1.643202267587185e-07
It 5400: Total Loss: 0.2608167231082916,         Rec Loss: 0.2608147859573364,
KL Loss: 1.9384606275707483e-07
It 5500: Total Loss: 0.26321643590927124,        Rec Loss: 0.2632143497467041,
KL Loss: 2.1008600015193224e-07
It 5600: Total Loss: 0.26686009764671326,        Rec Loss: 0.26685723662376404,
KL Loss: 2.872257027775049e-07
Run Epoch 6
It 5700: Total Loss: 0.27423155307769775,        Rec Loss: 0.274230420589447,
KL Loss: 1.1369411367923021e-07
It 5800: Total Loss: 0.2557062804698944,         Rec Loss: 0.25570493936538696,
KL Loss: 1.3489625416696072e-07
It 5900: Total Loss: 0.25306937098503113,        Rec Loss: 0.25306838750839233,
KL Loss: 9.922950994223356e-08
It 6000: Total Loss: 0.26050496101379395,        Rec Loss: 0.26050353050231934,
KL Loss: 1.4278339222073555e-07
It 6100: Total Loss: 0.27564287185668945,        Rec Loss: 0.27564239501953125,
KL Loss: 4.882167559117079e-08
It 6200: Total Loss: 0.2707749605178833,         Rec Loss: 0.27077344059944153,
KL Loss: 1.5081604942679405e-07
It 6300: Total Loss: 0.2695075273513794,         Rec Loss: 0.26950469613075256,
KL Loss: 2.8418435249477625e-07
It 6400: Total Loss: 0.26897817850112915,        Rec Loss: 0.26897576451301575,
KL Loss: 2.4212931748479605e-07
It 6500: Total Loss: 0.26411059498786926,        Rec Loss: 0.2641102373600006,
KL Loss: 3.470631781965494e-08
Run Epoch 7
It 6600: Total Loss: 0.2532949447631836,         Rec Loss: 0.2532942295074463,
KL Loss: 7.101334631443024e-08
It 6700: Total Loss: 0.2645516097545624,         Rec Loss: 0.26455092430114746,
```

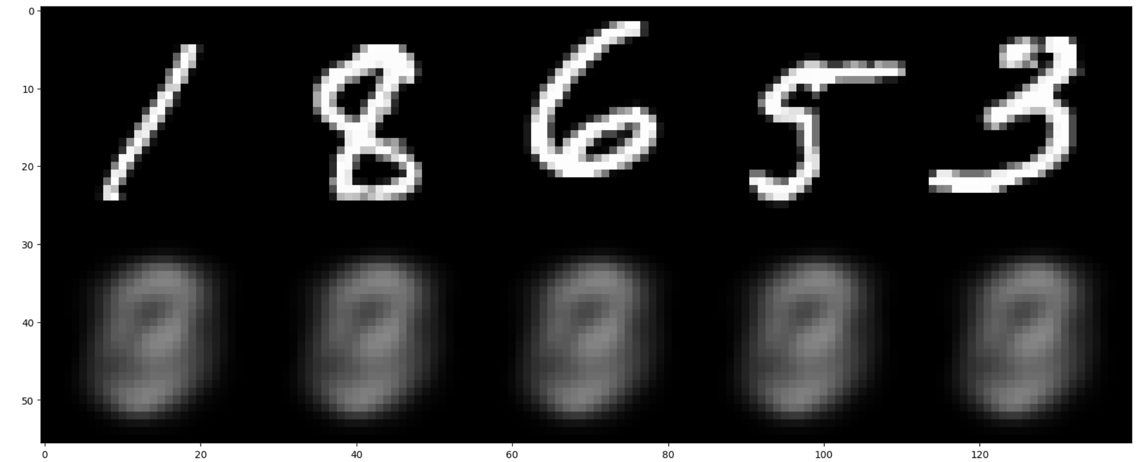
```
KL Loss: 6.727350410073996e-08
It 6800: Total Loss: 0.25699394941329956,        Rec Loss: 0.2569906711578369,
KL Loss: 3.2730167731642723e-07
It 6900: Total Loss: 0.2614345848560333,         Rec Loss: 0.261432409286499,
KL Loss: 2.1628511603921652e-07
It 7000: Total Loss: 0.2658095359802246,         Rec Loss: 0.2658092677593231,
KL Loss: 2.7997884899377823e-08
It 7100: Total Loss: 0.2738460600376129,         Rec Loss: 0.2738453149795532,
KL Loss: 7.549533620476723e-08
It 7200: Total Loss: 0.25123685598373413,        Rec Loss: 0.2512364387512207,
KL Loss: 4.253524821251631e-08
It 7300: Total Loss: 0.2738327980041504,         Rec Loss: 0.27383166551589966,
KL Loss: 1.1351949069648981e-07
It 7400: Total Loss: 0.2770816683769226,         Rec Loss: 0.2770787477493286,
KL Loss: 2.9089278541505337e-07
Run Epoch 8
It 7500: Total Loss: 0.2571987509727478,         Rec Loss: 0.2571980655193329,
KL Loss: 6.948539521545172e-08
It 7600: Total Loss: 0.2597567141056061,         Rec Loss: 0.2597564458847046,
KL Loss: 2.759043127298355e-08
It 7700: Total Loss: 0.2606600224971771,         Rec Loss: 0.26065945625305176,
KL Loss: 5.758192855864763e-08
It 7800: Total Loss: 0.2611430287361145,         Rec Loss: 0.2611417770385742,
KL Loss: 1.2645614333450794e-07
It 7900: Total Loss: 0.2610930800437927,         Rec Loss: 0.2610926628112793,
KL Loss: 4.1167368181049824e-08
It 8000: Total Loss: 0.252152681350708,          Rec Loss: 0.2521519064903259,
KL Loss: 7.711059879511595e-08
It 8100: Total Loss: 0.26121285557746887,        Rec Loss: 0.26121264696121216,
KL Loss: 2.0605511963367462e-08
It 8200: Total Loss: 0.25202319025993347,        Rec Loss: 0.25202271342277527,
KL Loss: 4.876346793025732e-08
It 8300: Total Loss: 0.25537511706352234,        Rec Loss: 0.2553749680519104,
KL Loss: 1.5235855244100094e-08
It 8400: Total Loss: 0.27664124965667725,        Rec Loss: 0.27664127945899963,
KL Loss: -3.5943230614066124e-09
Run Epoch 9
It 8500: Total Loss: 0.26559528708457947,        Rec Loss: 0.2655944526195526,
KL Loss: 8.339702617377043e-08
It 8600: Total Loss: 0.26581066846847534,        Rec Loss: 0.26581013202667236,
KL Loss: 5.25469658896327e-08
It 8700: Total Loss: 0.2503766119480133,         Rec Loss: 0.25037574768066406,
KL Loss: 8.64820322021842e-08
It 8800: Total Loss: 0.26834410429000854,        Rec Loss: 0.2683439254760742,
KL Loss: 1.9150320440530777e-08
It 8900: Total Loss: 0.26416900753974915,        Rec Loss: 0.26416879892349243,
KL Loss: 2.0547304302453995e-08
It 9000: Total Loss: 0.27294719219207764,        Rec Loss: 0.27294692397117615,
```

```
KL Loss: 2.6499037630856037e-08
It 9100: Total Loss: 0.2601778507232666,        Rec Loss: 0.2601776421070099,
KL Loss: 2.062006387859583e-08
It 9200: Total Loss: 0.26124072074890137,       Rec Loss: 0.2612399756908417,
KL Loss: 7.37927621230483e-08
It 9300: Total Loss: 0.26350051164627075,       Rec Loss: 0.2634981572628021,
KL Loss: 2.3414031602442265e-07
Done!
```

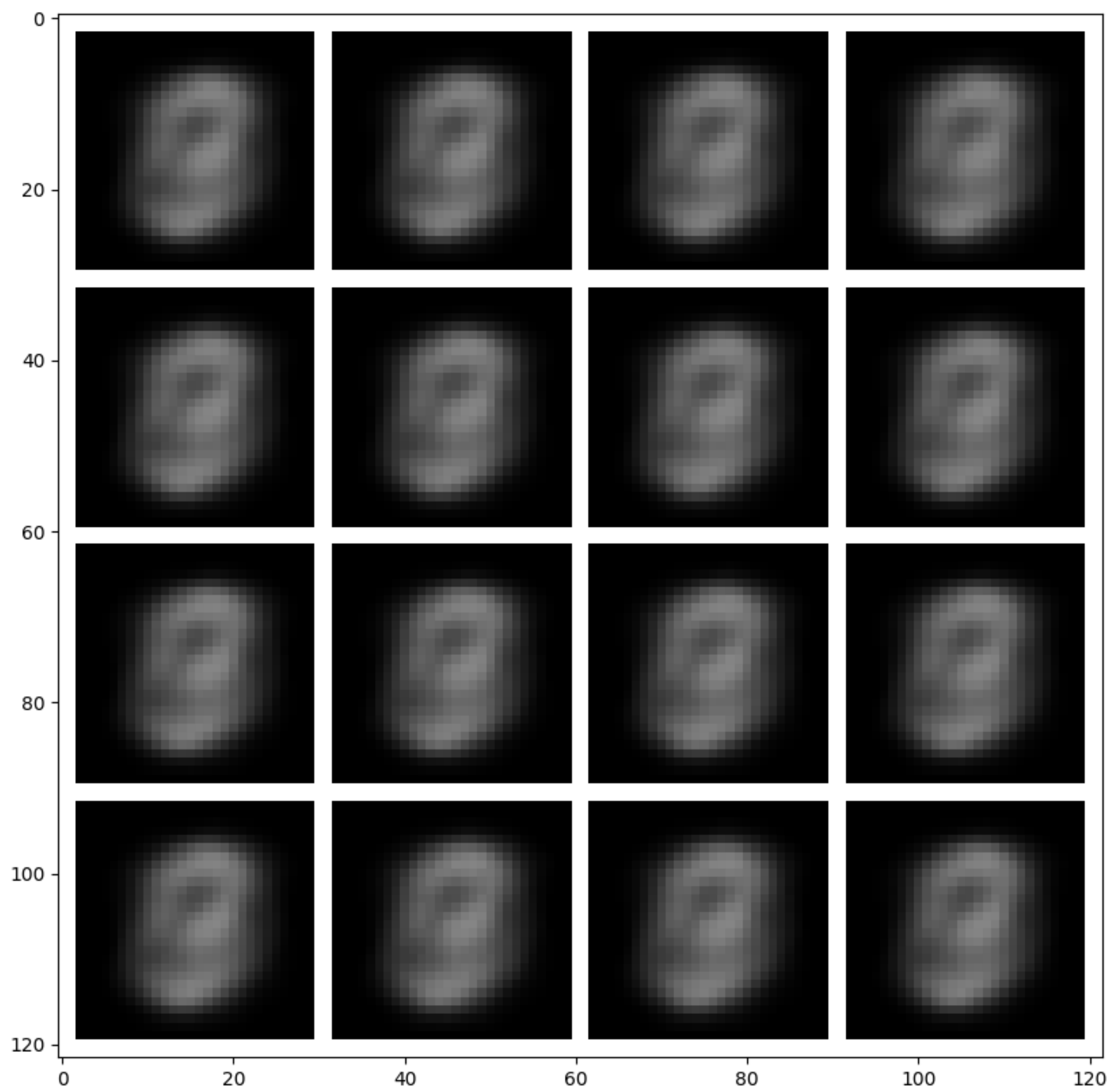


[ ]:
```python
# visualize VAE reconstructions and samples from the generative model
print("beta = ", beta)
vis_reconstruction(vae_model, randomize=True)
vis_samples(vae_model)
```

```
beta =  10
```



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



0
20
40
60
80
100
120
0
20
40
60
80
100
120

Inline Question: What can you observe when setting $\beta = 0$ and $\beta = 10$? Explain your observations! [2pt] (max 200 words)

Answer: When we train the VAE with beta = 0, the KL divergence term in loss is not taken into account. The VAE is trained to minimize the reconstruction loss only. As a result, the VAE is not forced to learn a compressed representation that belongs to the original distribution. The VAE will learn to reconstruct the input data decently (much like AE), but the learned representation will not be able to capture the full structure of the original data distribution. As a result, the generated samples may appear random or not belong to the original distribution, as they are essentially being generated by sampling from an incomplete or inaccurate representation of the data.

In contrast, when we train the VAE with beta = 10, the KL divergence loss weight in total_loss is very high. The VAE is trained to minimize the KL divergence term heavily. As a result, the VAE is forced to learn a compressed representation that is close to the prior distribution. So, it does exactly that. It learns to generate shapes (with pixel intensity) that would be close to average–i.e. the blurry 8 like shape. Hence, VAE will reconstruct the input data poorly.

4.4 Obtaining the best β-factor [5pt]

Prob 1-6 continued: 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 and samples of a well-tuned VAE! [3pt] (max 200 words)

Answer:

When we choose a beta value that is not too high or too low, there is a balance between reconstruction loss and KL divergence loss. The reconstruction loss will coverge properly (unlike beta = 10) as well as the KL loss will warm up but plateau at a lower value (unlike beta = 0). Hence, the VAE will learn to reconstruct the input data decently and generate samples that belong to the original distribution.

Moreover, when we draw embedding samples from a unit Gaussian prior and decode them through the decoder network, the generated samples might appear to be random, but they will be random in a way that is consistent with the original data distribution. This is what we want to observe in the below experiment.

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. Training reconstructions should be high quality, test samples should be diverse, distinguishable numbers, most samples recognizable as numbers.

Answer: Tuned beta value = 0.15 [2pt]

[ ]:

```
# Tuning for best beta
learning_rate = 1e-3
nz = 32

######################################## TODO␣
↪########################################
epochs = 20     # recommended 5-20 epochs
beta = 0.15 # Tune this for best results
##################################### 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.                    ␣
↪   #
#####################################################################################
# same as AE
optimizer = torch.optim.Adam(vae_model.parameters(), lr=learning_rate)
##################################### END TODO␣
↪########################################

train_it = 0
rec_loss, kl_loss = [], []
print(f"Running {epochs} epochs with {beta=}")
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.    #
```

```python
  # 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 sample_images, sample_labels in mnist_data_loader:
    # reshape images to (batch_size, 784)
    images = sample_images.reshape([batch_size, in_size])
    images = images.to(device)

    # reset gradients
    optimizer.zero_grad()

    # forward pass
    outputs = vae_model(images)

    # compute loss
    loss_dict = vae_model.loss(images, outputs)
    total_loss = loss_dict[0]
    losses = loss_dict[1]
    # print(total_loss)

    # backward pass
    total_loss.backward()

    # perform one optimization step
    optimizer.step()

    # save losses for plotting
    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
  ################################### END TODO␣
 ↪####################################

print("Done!")

rec_loss_plotdata = [foo.detach().cpu() for foo in rec_loss]
kl_loss_plotdata = [foo.detach().cpu() for foo in kl_loss]

# log the loss training curves
fig = plt.figure(figsize = (10, 5))
```

```
ax1 = plt.subplot(121)
ax1.plot(rec_loss_plotdata)
ax1.title.set_text("Reconstruction Loss")
ax2 = plt.subplot(122)
ax2.plot(kl_loss_plotdata)
ax2.title.set_text("KL Loss")
plt.show()
```

```
Running 20 epochs with beta=0.15
Run Epoch 0
It 0: Total Loss: 0.6945223212242126,    Rec Loss: 0.6931150555610657,   KL
Loss: 0.009381677955389023
It 100: Total Loss: 0.2559598982334137,          Rec Loss: 0.25218522548675537,
KL Loss: 0.025164486840367317
It 200: Total Loss: 0.2509142756462097,          Rec Loss: 0.24587085843086243,
KL Loss: 0.033622775226831436
It 300: Total Loss: 0.24623756110668182,         Rec Loss: 0.24025869369506836,
KL Loss: 0.0398590974509716
It 400: Total Loss: 0.256846159696579,   Rec Loss: 0.2512294352054596,   KL
Loss: 0.03744490072131157
It 500: Total Loss: 0.23882855474948883,         Rec Loss: 0.22909285128116608,
KL Loss: 0.06490468978881836
It 600: Total Loss: 0.2416294366121292,          Rec Loss: 0.23177403211593628,
KL Loss: 0.0657026544213295
It 700: Total Loss: 0.23339667916297913,         Rec Loss: 0.22189535200595856,
KL Loss: 0.07667550444602966
It 800: Total Loss: 0.23902738094329834,         Rec Loss: 0.2252204567193985,
KL Loss: 0.09204614162445068
It 900: Total Loss: 0.24071602523326874,         Rec Loss: 0.22479180991649628,
KL Loss: 0.1061614602804184
Run Epoch 1
It 1000: Total Loss: 0.23546724021434784,        Rec Loss: 0.21789732575416565,
KL Loss: 0.117132768034935
It 1100: Total Loss: 0.23365187644958496,        Rec Loss: 0.21351440250873566,
KL Loss: 0.13424980640411377
It 1200: Total Loss: 0.2297917604446411,         Rec Loss: 0.21029847860336304,
KL Loss: 0.12995515763759613
It 1300: Total Loss: 0.2203046679496765,         Rec Loss: 0.19705171883106232,
KL Loss: 0.15501965582370758
It 1400: Total Loss: 0.2157001793384552,         Rec Loss: 0.19169865548610687,
KL Loss: 0.16001009941101074
It 1500: Total Loss: 0.21636177599430084,        Rec Loss: 0.19165408611297607,
KL Loss: 0.1647179126739502
It 1600: Total Loss: 0.22026576101779938,        Rec Loss: 0.1935461014509201,
KL Loss: 0.17813104391098022
It 1700: Total Loss: 0.20787960290908813,        Rec Loss: 0.1821402758359909,
KL Loss: 0.1715955287218094
```

```
It 1800: Total Loss: 0.2060813158750534,        Rec Loss: 0.17817461490631104,
KL Loss: 0.1860446333885193
Run Epoch 2
It 1900: Total Loss: 0.20363818109035492,       Rec Loss: 0.17514556646347046,
KL Loss: 0.18995076417922974
It 2000: Total Loss: 0.19725383818149567,       Rec Loss: 0.1696663796901703,
KL Loss: 0.18391641974449158
It 2100: Total Loss: 0.2024865299463272,        Rec Loss: 0.17373478412628174,
KL Loss: 0.19167830049991608
It 2200: Total Loss: 0.1994306445121765,        Rec Loss: 0.16895653307437897,
KL Loss: 0.20316070318222046
It 2300: Total Loss: 0.2129776030778885,        Rec Loss: 0.1827264428138733,
KL Loss: 0.20167440176010132
It 2400: Total Loss: 0.20997145771980286,       Rec Loss: 0.17873847484588623,
KL Loss: 0.20821993052959442
It 2500: Total Loss: 0.20269350707530975,       Rec Loss: 0.17177386581897736,
KL Loss: 0.2061309516429901
It 2600: Total Loss: 0.19502492249011993,       Rec Loss: 0.16543275117874146,
KL Loss: 0.1972810924053192
It 2700: Total Loss: 0.20229169726371765,       Rec Loss: 0.17124876379966736,
KL Loss: 0.20695286989212036
It 2800: Total Loss: 0.20481246709823608,       Rec Loss: 0.173354372382164,
KL Loss: 0.20972062647342682
Run Epoch 3
It 2900: Total Loss: 0.18945154547691345,       Rec Loss: 0.15795938670635223,
KL Loss: 0.20994767546653748
It 3000: Total Loss: 0.20964694023132324,       Rec Loss: 0.17867150902748108,
KL Loss: 0.20650288462638855
It 3100: Total Loss: 0.19145219027996063,       Rec Loss: 0.1595744788646698,
KL Loss: 0.21251808106899261
It 3200: Total Loss: 0.20241358876228333,       Rec Loss: 0.1668911725282669,
KL Loss: 0.23681607842445374
It 3300: Total Loss: 0.20141595602035522,       Rec Loss: 0.16761048138141632,
KL Loss: 0.22536984086036682
It 3400: Total Loss: 0.18627199530601501,       Rec Loss: 0.15279115736484528,
KL Loss: 0.22320556640625
It 3500: Total Loss: 0.17638933658599854,       Rec Loss: 0.14364702999591827,
KL Loss: 0.21828202903270721
It 3600: Total Loss: 0.19410309195518494,       Rec Loss: 0.16094918549060822,
KL Loss: 0.2210259884595871
It 3700: Total Loss: 0.2025146186351776,        Rec Loss: 0.16882537305355072,
KL Loss: 0.22459501028060913
Run Epoch 4
It 3800: Total Loss: 0.20147547125816345,       Rec Loss: 0.1657753884792328,
KL Loss: 0.23800048232078552
It 3900: Total Loss: 0.1976301074028015,        Rec Loss: 0.16207696497440338,
KL Loss: 0.23702093958854675
It 4000: Total Loss: 0.19302994012832642,       Rec Loss: 0.15887859463691711,
```

```
KL Loss: 0.2276756316423416
It 4100: Total Loss: 0.18461830914020538,        Rec Loss: 0.14982128143310547,
KL Loss: 0.23198020458221436
It 4200: Total Loss: 0.1903066635131836,         Rec Loss: 0.15776997804641724,
KL Loss: 0.21691125631332397
It 4300: Total Loss: 0.189383864402771,          Rec Loss: 0.1553182750940323,
KL Loss: 0.22710391879081726
It 4400: Total Loss: 0.1899825781583786,         Rec Loss: 0.15531226992607117,
KL Loss: 0.23113541305065155
It 4500: Total Loss: 0.19837383925914764,        Rec Loss: 0.16212154924869537,
KL Loss: 0.24168188869953156
It 4600: Total Loss: 0.18669266998767853,        Rec Loss: 0.15114209055900574,
KL Loss: 0.23700383305549622
Run Epoch 5
It 4700: Total Loss: 0.19952918589115143,        Rec Loss: 0.16497109830379486,
KL Loss: 0.23038722574710846
It 4800: Total Loss: 0.18362458050251007,        Rec Loss: 0.14905929565429688,
KL Loss: 0.23043525218963623
It 4900: Total Loss: 0.1960720270872116,         Rec Loss: 0.16242477297782898,
KL Loss: 0.22431501746177673
It 5000: Total Loss: 0.1927088499069214,         Rec Loss: 0.15822991728782654,
KL Loss: 0.22985953092575073
It 5100: Total Loss: 0.18288008868694305,        Rec Loss: 0.14722946286201477,
KL Loss: 0.23767083883285522
It 5200: Total Loss: 0.1841963529586792,         Rec Loss: 0.1479508876800537,
KL Loss: 0.2416364550590515
It 5300: Total Loss: 0.1948273628950119,         Rec Loss: 0.1589684784412384,
KL Loss: 0.2390591949224472
It 5400: Total Loss: 0.18671831488609314,        Rec Loss: 0.15214522182941437,
KL Loss: 0.23048733174800873
It 5500: Total Loss: 0.19182497262954712,        Rec Loss: 0.15666206181049347,
KL Loss: 0.23441937565803528
It 5600: Total Loss: 0.1824888288974762,         Rec Loss: 0.14625662565231323,
KL Loss: 0.24154803156852722
Run Epoch 6
It 5700: Total Loss: 0.18715901672840118,        Rec Loss: 0.15158407390117645,
KL Loss: 0.23716627061367035
It 5800: Total Loss: 0.19622774422168732,        Rec Loss: 0.1593206375837326,
KL Loss: 0.24604733288288116
It 5900: Total Loss: 0.1872219294309616,         Rec Loss: 0.14956553280353546,
KL Loss: 0.25104260444641113
It 6000: Total Loss: 0.19246305525302887,        Rec Loss: 0.15472042560577393,
KL Loss: 0.25161752104759216
It 6100: Total Loss: 0.19548547267913818,        Rec Loss: 0.15782327950000763,
KL Loss: 0.25108128786087036
It 6200: Total Loss: 0.18571837246418,   Rec Loss: 0.1497429460287094,   KL
Loss: 0.23983615636825562
It 6300: Total Loss: 0.1955876499414444,         Rec Loss: 0.1595742106437683,
```

```
KL Loss: 0.24008959531784058
It 6400: Total Loss: 0.19410189986228943,        Rec Loss: 0.1565549671649933,
KL Loss: 0.2503129243850708
It 6500: Total Loss: 0.1685449182987213,         Rec Loss: 0.13335931301116943,
KL Loss: 0.2345707267522812
Run Epoch 7
It 6600: Total Loss: 0.18503661453723907,        Rec Loss: 0.14781348407268524,
KL Loss: 0.24815420806407928
It 6700: Total Loss: 0.17800773680210114,        Rec Loss: 0.14022260904312134,
KL Loss: 0.251900851726532
It 6800: Total Loss: 0.19797620177268982,        Rec Loss: 0.15985411405563354,
KL Loss: 0.2541472613811493
It 6900: Total Loss: 0.1986517608165741,         Rec Loss: 0.163430854678154,
KL Loss: 0.23480603098869324
It 7000: Total Loss: 0.18024766445159912,        Rec Loss: 0.14534342288970947,
KL Loss: 0.23269498348236084
It 7100: Total Loss: 0.19838769733905792,        Rec Loss: 0.1623523086309433,
KL Loss: 0.24023593962192535
It 7200: Total Loss: 0.18276619911193848,        Rec Loss: 0.1472986787557602,
KL Loss: 0.2364501655101776
It 7300: Total Loss: 0.18226219713687897,        Rec Loss: 0.1462600976228714,
KL Loss: 0.240013986825943
It 7400: Total Loss: 0.185263991355896,          Rec Loss: 0.14833416044712067,
KL Loss: 0.24619892239570618
Run Epoch 8
It 7500: Total Loss: 0.20500069856643677,        Rec Loss: 0.1675364077091217,
KL Loss: 0.24976196885108948
It 7600: Total Loss: 0.1925702691078186,         Rec Loss: 0.15588806569576263,
KL Loss: 0.24454806745052338
It 7700: Total Loss: 0.18677589297294617,        Rec Loss: 0.1501273810863495,
KL Loss: 0.24432335793972015
It 7800: Total Loss: 0.1876429319381714,         Rec Loss: 0.14985975623130798,
KL Loss: 0.25188785791397095
It 7900: Total Loss: 0.18305620551109314,        Rec Loss: 0.14798255264759064,
KL Loss: 0.23382437229156494
It 8000: Total Loss: 0.1935311257839203,         Rec Loss: 0.15568523108959198,
KL Loss: 0.2523060142993927
It 8100: Total Loss: 0.18878336250782013,        Rec Loss: 0.1520400047302246,
KL Loss: 0.24495573341846466
It 8200: Total Loss: 0.18375319242477417,        Rec Loss: 0.14667503535747528,
KL Loss: 0.24718770384788513
It 8300: Total Loss: 0.18561501801013947,        Rec Loss: 0.14919447898864746,
KL Loss: 0.24280358850955963
It 8400: Total Loss: 0.18679489195346832,        Rec Loss: 0.14839573204517365,
KL Loss: 0.25599437952041626
Run Epoch 9
It 8500: Total Loss: 0.19359934329986572,        Rec Loss: 0.15545713901519775,
KL Loss: 0.2542813718318939
```

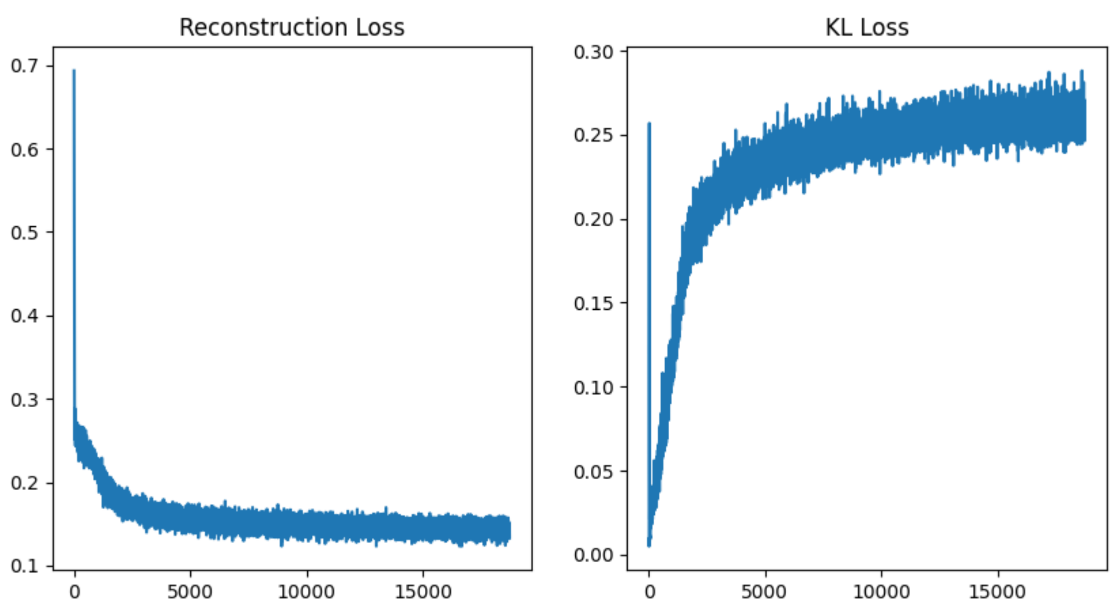
```
It 8600: Total Loss: 0.18286937475204468,       Rec Loss: 0.14648078382015228,
KL Loss: 0.24259065091609955
It 8700: Total Loss: 0.18086101114749908,       Rec Loss: 0.14431267976760864,
KL Loss: 0.24365556240081787
It 8800: Total Loss: 0.18275877833366394,       Rec Loss: 0.14557327330112457,
KL Loss: 0.24790331721305847
It 8900: Total Loss: 0.1835402399301529,        Rec Loss: 0.14711478352546692,
KL Loss: 0.2428363859653473
It 9000: Total Loss: 0.1933164894580841,        Rec Loss: 0.15622444450855255,
KL Loss: 0.2472803294658661
It 9100: Total Loss: 0.17420198023319244,       Rec Loss: 0.1376597136259079,
KL Loss: 0.24361509084701538
It 9200: Total Loss: 0.17834638059139252,       Rec Loss: 0.14007365703582764,
KL Loss: 0.2551514506340027
It 9300: Total Loss: 0.1772734522819519,        Rec Loss: 0.1396905928850174,
KL Loss: 0.2505524456501007
Run Epoch 10
It 9400: Total Loss: 0.1887914538383484,        Rec Loss: 0.15088962018489838,
KL Loss: 0.25267893075942993
It 9500: Total Loss: 0.18143099546432495,       Rec Loss: 0.1423765867948532,
KL Loss: 0.26036277413368225
It 9600: Total Loss: 0.18276375532150269,       Rec Loss: 0.14465764164924622,
KL Loss: 0.25404080748558044
It 9700: Total Loss: 0.18795371055603027,       Rec Loss: 0.1503460854291916,
KL Loss: 0.25071749091148376
It 9800: Total Loss: 0.19144810736179352,       Rec Loss: 0.15264354646205902,
KL Loss: 0.2586970329284668
It 9900: Total Loss: 0.19202271103858948,       Rec Loss: 0.15396066009998322,
KL Loss: 0.25374698638916016
It 10000: Total Loss: 0.16916580498218536,      Rec Loss: 0.1336364895105362,
KL Loss: 0.23686212301254272
It 10100: Total Loss: 0.18236826360225677,      Rec Loss: 0.14443975687026978,
KL Loss: 0.2528567314147949
It 10200: Total Loss: 0.17945052683353424,      Rec Loss: 0.142220601439476,
KL Loss: 0.2481994926929474
It 10300: Total Loss: 0.17789019644260406,      Rec Loss: 0.1425066888332367,
KL Loss: 0.23589007556438446
Run Epoch 11
It 10400: Total Loss: 0.18686243891716003,      Rec Loss: 0.14609336853027344,
KL Loss: 0.27179384231567383
It 10500: Total Loss: 0.1765826791524887,       Rec Loss: 0.1392141580581665,
KL Loss: 0.2491234540939331
It 10600: Total Loss: 0.18677091598510742,      Rec Loss: 0.1492961198091507,
KL Loss: 0.2498319447040558
It 10700: Total Loss: 0.19632844626903534,      Rec Loss: 0.15736299753189087,
KL Loss: 0.25976961851119995
It 10800: Total Loss: 0.1958196759223938,       Rec Loss: 0.15731342136859894,
KL Loss: 0.2567083537578583
```

It 10900: Total Loss: 0.19510504603385925, Rec Loss: 0.15757820010185242,
KL Loss: 0.2501789331436157
It 11000: Total Loss: 0.1736837923526764, Rec Loss: 0.13690228760242462,
KL Loss: 0.24520999193191528
It 11100: Total Loss: 0.1941429078578949, Rec Loss: 0.15667183697223663,
KL Loss: 0.2498071789741516
It 11200: Total Loss: 0.17362403869628906, Rec Loss: 0.13705430924892426,
KL Loss: 0.2437981367111206
Run Epoch 12
It 11300: Total Loss: 0.1797618865966797, Rec Loss: 0.14082224667072296,
KL Loss: 0.2595975995063782
It 11400: Total Loss: 0.16930341720581055, Rec Loss: 0.13152047991752625,
KL Loss: 0.2518862187862396
It 11500: Total Loss: 0.1834414303302765, Rec Loss: 0.14573608338832855,
KL Loss: 0.2513689696788788
It 11600: Total Loss: 0.1832035332918167, Rec Loss: 0.14482450485229492,
KL Loss: 0.25586017966270447
It 11700: Total Loss: 0.18686312437057495, Rec Loss: 0.14835090935230255,
KL Loss: 0.2567480802536011
It 11800: Total Loss: 0.18962186574935913, Rec Loss: 0.15208189189434052,
KL Loss: 0.2502664625644684
It 11900: Total Loss: 0.18505288660526276, Rec Loss: 0.14777298271656036,
KL Loss: 0.24853265285491943
It 12000: Total Loss: 0.1911475658416748, Rec Loss: 0.15199492871761322,
KL Loss: 0.2610176205635071
It 12100: Total Loss: 0.17983531951904297, Rec Loss: 0.1408105343580246,
KL Loss: 0.2601652145385742
Run Epoch 13
It 12200: Total Loss: 0.18427078425884247, Rec Loss: 0.14707624912261963,
KL Loss: 0.2479635775089264
It 12300: Total Loss: 0.1814236342906952, Rec Loss: 0.14273163676261902,
KL Loss: 0.2579466998577118
It 12400: Total Loss: 0.19016429781913757, Rec Loss: 0.15153543651103973,
KL Loss: 0.257525771856308
It 12500: Total Loss: 0.179831400513649, Rec Loss: 0.1422404944896698,
KL Loss: 0.2506060302257538
It 12600: Total Loss: 0.18243028223514557, Rec Loss: 0.14493651688098907,
KL Loss: 0.24995845556259155
It 12700: Total Loss: 0.18942886590957642, Rec Loss: 0.15257999300956726,
KL Loss: 0.24565915763378143
It 12800: Total Loss: 0.18411344289779663, Rec Loss: 0.14552420377731323,
KL Loss: 0.25726163387298584
It 12900: Total Loss: 0.18717333674430847, Rec Loss: 0.15022628009319305,
KL Loss: 0.24631372094154358
It 13000: Total Loss: 0.17755982279777527, Rec Loss: 0.14041124284267426,
KL Loss: 0.24765720963478088
It 13100: Total Loss: 0.18731114268302917, Rec Loss: 0.1506614089012146,
KL Loss: 0.24433156847953796

Run Epoch 14
It 13200: Total Loss: 0.1940775066614151, Rec Loss: 0.15495163202285767, KL Loss: 0.2608391344547272
It 13300: Total Loss: 0.18168707191944122, Rec Loss: 0.14237023890018463, KL Loss: 0.26211223006248474
It 13400: Total Loss: 0.17820130288600922, Rec Loss: 0.1419081687927246, KL Loss: 0.2419542521238327
It 13500: Total Loss: 0.17717429995536804, Rec Loss: 0.13910983502864838, KL Loss: 0.2537631094455719
It 13600: Total Loss: 0.1997235119342804, Rec Loss: 0.16054826974868774, KL Loss: 0.2611682116985321
It 13700: Total Loss: 0.1887679398059845, Rec Loss: 0.15087199211120605, KL Loss: 0.25263962149620056
It 13800: Total Loss: 0.19064456224441528, Rec Loss: 0.1513727605342865, KL Loss: 0.2618120014667511
It 13900: Total Loss: 0.17374810576438904, Rec Loss: 0.13690108060836792, KL Loss: 0.24564680457115173
It 14000: Total Loss: 0.183475524187088, Rec Loss: 0.1447165459394455, KL Loss: 0.25839316844940186
Run Epoch 15
It 14100: Total Loss: 0.1858111023902893, Rec Loss: 0.14598213136196136, KL Loss: 0.26552650332450867
It 14200: Total Loss: 0.18445253372192383, Rec Loss: 0.14575918018817902, KL Loss: 0.2579556405544281
It 14300: Total Loss: 0.18991675972938538, Rec Loss: 0.1507902294397354, KL Loss: 0.26084351539611816
It 14400: Total Loss: 0.18787294626235962, Rec Loss: 0.14759276807308197, KL Loss: 0.26853451132774353
It 14500: Total Loss: 0.18347637355327606, Rec Loss: 0.14504902064800262, KL Loss: 0.25618231296539307
It 14600: Total Loss: 0.17881277203559875, Rec Loss: 0.1399983912706375, KL Loss: 0.25876253843307495
It 14700: Total Loss: 0.19243967533111572, Rec Loss: 0.1527901291847229, KL Loss: 0.26433035731315613
It 14800: Total Loss: 0.18522027134895325, Rec Loss: 0.14784058928489685, KL Loss: 0.24919793009757996
It 14900: Total Loss: 0.18357713520526886, Rec Loss: 0.14482493698596954, KL Loss: 0.2583479583263397
Run Epoch 16
It 15000: Total Loss: 0.1744711995124817, Rec Loss: 0.1356046050786972, KL Loss: 0.25911056995391846
It 15100: Total Loss: 0.18742598593235016, Rec Loss: 0.14882560074329376, KL Loss: 0.2573358714580536
It 15200: Total Loss: 0.18614870309829712, Rec Loss: 0.14723999798297882, KL Loss: 0.2593913674354553
It 15300: Total Loss: 0.18230897188186646, Rec Loss: 0.1451542228460312, KL Loss: 0.2476983368396759
It 15400: Total Loss: 0.18996894359588623, Rec Loss: 0.15032599866390228,

```
KL Loss: 0.2642862796783447
It 15500: Total Loss: 0.18909913301467896,      Rec Loss: 0.1510647088289261,
KL Loss: 0.2535628378391266
It 15600: Total Loss: 0.17700082063674927,      Rec Loss: 0.13831393420696259,
KL Loss: 0.2579125165939331
It 15700: Total Loss: 0.17992082238197327,      Rec Loss: 0.1404876708984375,
KL Loss: 0.2628876864910126
It 15800: Total Loss: 0.19012130796909332,      Rec Loss: 0.15131938457489014,
KL Loss: 0.25867950916290283
It 15900: Total Loss: 0.18532560765743256,      Rec Loss: 0.14526675641536713,
KL Loss: 0.26705896854400635
Run Epoch 17
It 16000: Total Loss: 0.18586623668670654,      Rec Loss: 0.14862322807312012,
KL Loss: 0.24828673899173737
It 16100: Total Loss: 0.18684646487236023,      Rec Loss: 0.14655250310897827,
KL Loss: 0.2686263918876648
It 16200: Total Loss: 0.17894604802131653,      Rec Loss: 0.14072172343730927,
KL Loss: 0.25482887029647827
It 16300: Total Loss: 0.18335804343223572,      Rec Loss: 0.14323660731315613,
KL Loss: 0.2674762010574341
It 16400: Total Loss: 0.18545877933502197,      Rec Loss: 0.14589422941207886,
KL Loss: 0.2637636959552765
It 16500: Total Loss: 0.1818978190422058,       Rec Loss: 0.1432458907365799,
KL Loss: 0.2576795220375061
It 16600: Total Loss: 0.18489982187747955,      Rec Loss: 0.14615659415721893,
KL Loss: 0.2582882046699524
It 16700: Total Loss: 0.18043574690818787,      Rec Loss: 0.14075195789337158,
KL Loss: 0.2645586133003235
It 16800: Total Loss: 0.18430426716804504,      Rec Loss: 0.1469610333442688,
KL Loss: 0.24895493686199188
Run Epoch 18
It 16900: Total Loss: 0.16657856106758118,      Rec Loss: 0.13074159622192383,
KL Loss: 0.23891305923461914
It 17000: Total Loss: 0.17387013137340546,      Rec Loss: 0.13370883464813232,
KL Loss: 0.26774194836616516
It 17100: Total Loss: 0.17404237389564514,      Rec Loss: 0.13450254499912262,
KL Loss: 0.26359885931015015
It 17200: Total Loss: 0.1861250251531601,       Rec Loss: 0.14875733852386475,
KL Loss: 0.249117910861969
It 17300: Total Loss: 0.18225115537643433,      Rec Loss: 0.14450392127037048,
KL Loss: 0.2516481876373291
It 17400: Total Loss: 0.17603039741516113,      Rec Loss: 0.1363660842180252,
KL Loss: 0.2644287049770355
It 17500: Total Loss: 0.18026381731033325,      Rec Loss: 0.14001981914043427,
KL Loss: 0.26829326152801514
It 17600: Total Loss: 0.19129809737205505,      Rec Loss: 0.15074391663074493,
KL Loss: 0.2703612446784973
It 17700: Total Loss: 0.18959413468837738,      Rec Loss: 0.15051968395709991,
```

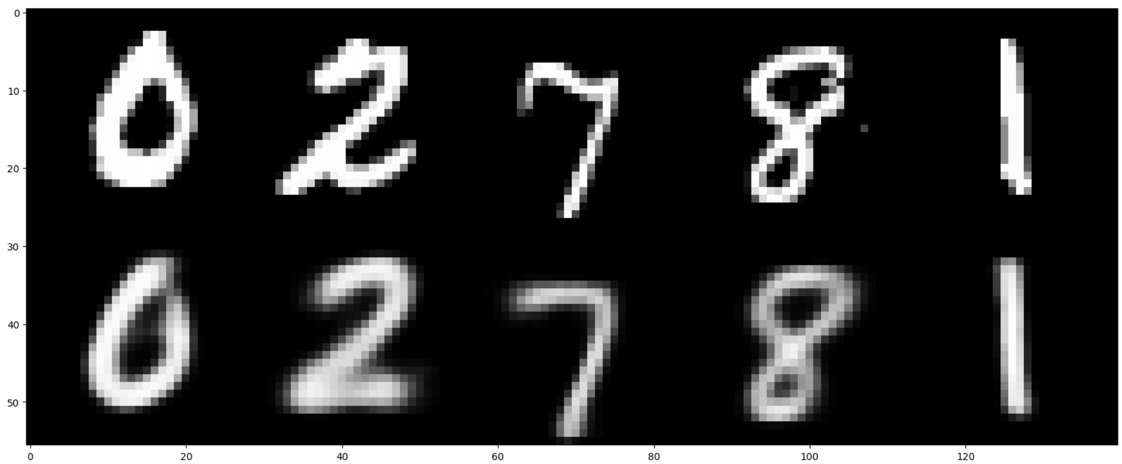
```
KL Loss: 0.2604963183403015
It 17800: Total Loss: 0.17974938452243805,       Rec Loss: 0.14082439243793488,
KL Loss: 0.2594999074935913
Run Epoch 19
It 17900: Total Loss: 0.17428100109100342,       Rec Loss: 0.13530534505844116,
KL Loss: 0.25983765721321106
It 18000: Total Loss: 0.17895013093948364,       Rec Loss: 0.13926251232624054,
KL Loss: 0.26458412408828735
It 18100: Total Loss: 0.1761186271905899,        Rec Loss: 0.13569538295269012,
KL Loss: 0.26948827505111694
It 18200: Total Loss: 0.1641996055841446,        Rec Loss: 0.12396464496850967,
KL Loss: 0.2682330906391144
It 18300: Total Loss: 0.17059367895126343,       Rec Loss: 0.13086889684200287,
KL Loss: 0.2648318409919739
It 18400: Total Loss: 0.17890691757202148,       Rec Loss: 0.14082174003124237,
KL Loss: 0.2539011240005493
It 18500: Total Loss: 0.18027442693710327,       Rec Loss: 0.14031733572483063,
KL Loss: 0.26638063788414
It 18600: Total Loss: 0.1774519383907318,        Rec Loss: 0.13770398497581482,
KL Loss: 0.2649863064289093
It 18700: Total Loss: 0.18536825478076935,       Rec Loss: 0.14445343613624573,
KL Loss: 0.27276545763015747
Done!
```

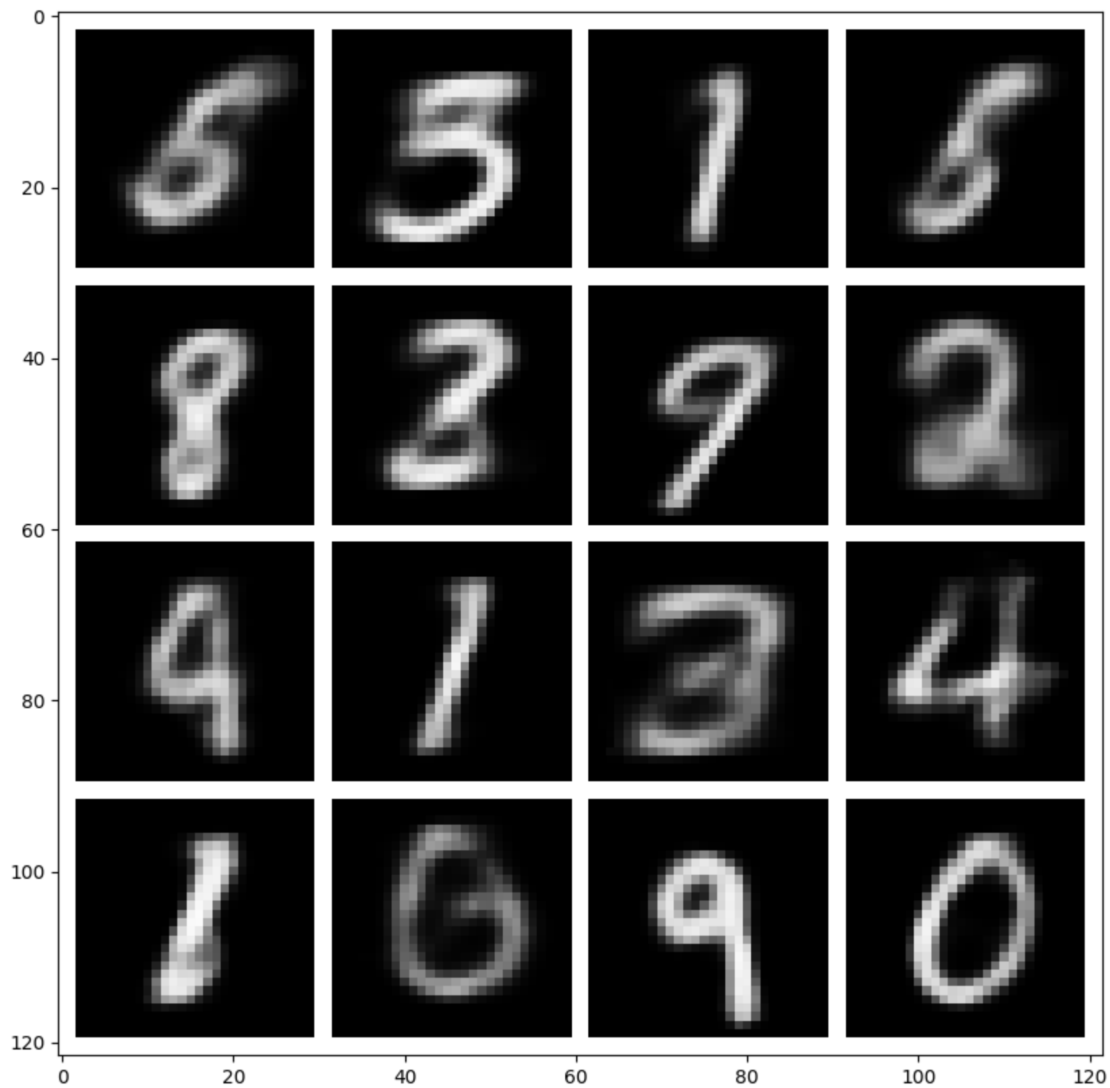


Let’s look at some reconstructions and decoded embedding samples for this beta!

[ ]:
```python
# [OPTIONAL] visualize VAE reconstructions and samples from the generative model
print("BEST beta = ", beta)
vis_reconstruction(vae_model, randomize=True)
vis_samples(vae_model)
```

```
BEST beta =  0.15
```





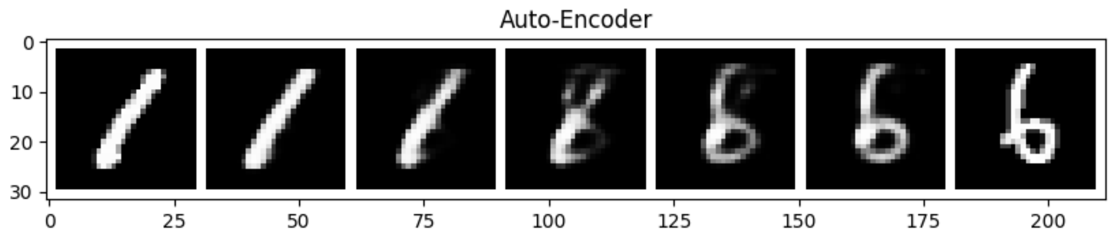
5 4. Embedding Space Interpolation [3pt]

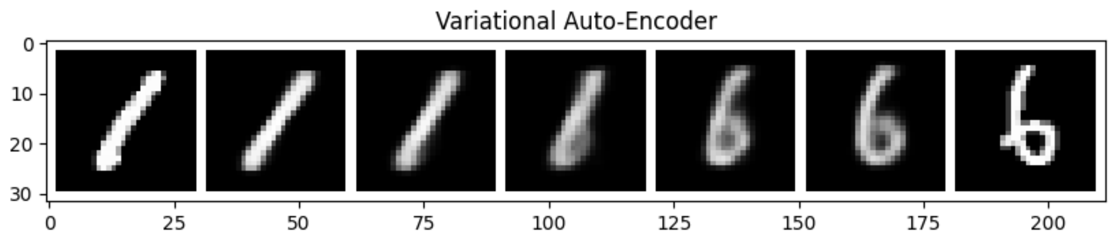
As mentioned in the introduction, AEs and VAEs cannot only be used to generate images, but also to learn low-dimensional representations of their inputs. In this final section we will investigate the representations we learned with both models by **interpolating in embedding space** between different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

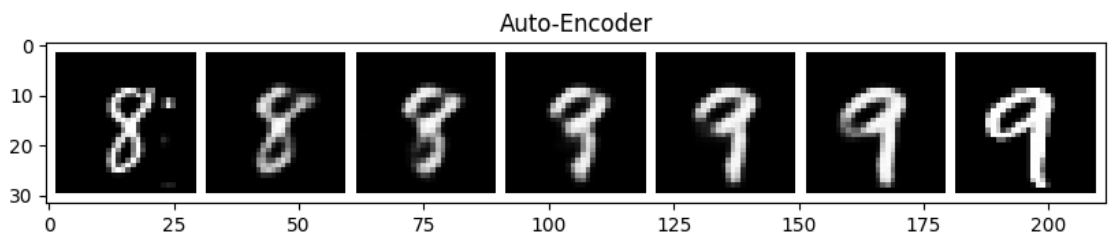
[ ]:
```
# Prob1-7
nz=32

def get_image_with_label(target_label):
  """Returns a random image from the training set with the requested digit."""
```

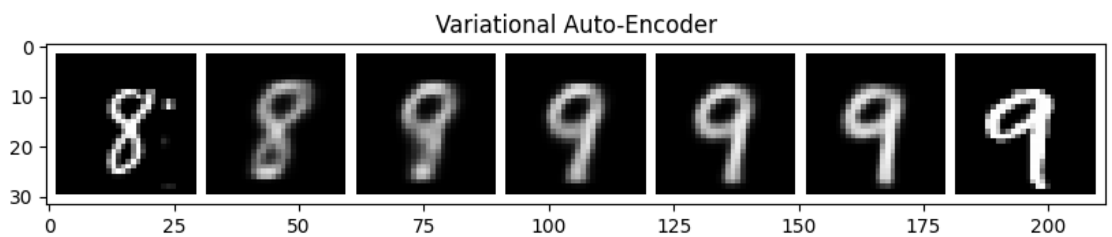
```
  for img_batch, label_batch in mnist_data_loader:
    for img, label in zip(img_batch, label_batch):
      if label == target_label:
        return img.to(device)

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].reshape(1,784))[..., :nz]
  z_end = model.encoder(end_img[None].reshape(1,784))[..., :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))
  img_inter = img_inter.reshape(-1, 28, 28)

  # reshape result and display interpolation
  vis_imgs = torch.cat([start_img, img_inter, end_img]).reshape(-1,1,28,28)
  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()


### Interpolation 1
START_LABEL = 1 # ... TODO CHOOSE
END_LABEL = 6 # ... TODO CHOOSE
# 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)

### Interpolation 2
START_LABEL = 8# ... TODO CHOOSE
END_LABEL = 9# ... TODO CHOOSE
```

```
# 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.

> **Prob1-7 continued: Inline Question: Repeat the interpolation experiment with different start / end labels and multiple samples. Describe your observations! [2 pt]** 1. How do AE and VAE embedding space interpolations differ?
>
> **Answer**:
>
> In contrast to an AE embedding space, the interpolation carried out in a VAE embedding space seems to be smoother and more linear. This is because the VAE has learned an embedding space that is closer to the prior distribution due to KL loss. Hence, the VAE is able to interpolate between two points in the embedding space in a more linear fashion.
>
> We can see in the above experiment that while interpolating between 1 and 6, AE passes through a state where the generated images looks like a broken 8. This is because the AE embedding space is not close to the prior distribution. Hence, we may observe random shapes. In contrast, in both examples, VAE interpolates smoothly between 1 and 6. We expect to see more recognizable numbers in the intermediate generated images.
>
> 2. How do you expect these differences to affect the usefulness of the learned representation for downstream learning? (max 300 words)
>
> **Answer**:
>
> The VAE embedding space is closer to the prior distribution. Hence, the VAE is able to interpolate between two points in the embedding space in a more linear fashion. This is useful for downstream learning tasks that involve interpolation or extrapolation.
>
> For example, if we want to generate new images by interpolating between two existing images, we can simply interpolate between their embeddings in the VAE embedding space and then decode the interpolated embeddings to generate the new image. Because the VAE embedding space is smooth and continuous, this interpolation will be more linear and natural-looking than if we were to interpolate directly between the raw image pixels or using AutoEncoders. This use case can be extended to Data Augmentation.
>
> On the other hand, the absence of continuity and structure in the AE embedding space might lead to representations that are less useful for downstream learning tasks and more susceptible to noise.
>
> In addition, the regularized structure of the VAE embedding space reduces the likelihood of overfitting to the training set and improves generalization to new data.

6 5. Conditional VAE

Let us now try a Conditional VAE Now we will try to create a Conditional VAE, where we can condition the encoder and decoder of the VAE on the label `c`.

6.1 Defining the conditional Encoder, Decoder, and VAE models [5 pt]

Prob1-8. We create a separate encoder and decoder class that take in an additional argument `c` in their forward pass, and then build our CVAE model on top of it. Note that the encoder and decoder just need to append `c` to the standard inputs to these modules.

[ ]:
```
def idx2onehot(idx, n):
    """Converts a batch of indices to a one-hot representation."""
    assert torch.max(idx).item() < n
    if idx.dim() == 1:
        idx = idx.unsqueeze(1)
    onehot = torch.zeros(idx.size(0), n).to(idx.device)
    onehot.scatter_(1, idx, 1)

    return onehot

# Let's define encoder and decoder networks

class CVAEEncoder(nn.Module):
  def __init__(self, nz, input_size, conditional, num_labels):
    super().__init__()
    self.input_size = input_size + num_labels if conditional else input_size
    self.num_labels = num_labels
    self.conditional = conditional

    ################################# TODO␣
 ↪########################################
    # Create the network architecture using a nn.Sequential module wrapper.     ␣
 ↪   #
    # Encoder Architecture:                                                     ␣
 ↪   #
    # - input_size -> 256                                                       ␣
 ↪   #
    # - ReLU                                                                    ␣
 ↪   #
    # - 256 -> 64                                                               ␣
 ↪   #
    # - ReLU                                                                    ␣
 ↪   #
    # - 64 -> nz                                                                ␣
 ↪   #
    # HINT: Verify the shapes of intermediate layers by running partial␣
 ↪networks    #
    #         (with the next notebook cell) and visualizing the output shapes.  ␣
 ↪   #
   ␣
 ↪###############################################################################
```

```
    self.net = nn.Sequential(
        nn.Linear(self.input_size, 256),
        nn.ReLU(),
        nn.Linear(256, 64),
        nn.ReLU(),
        nn.Linear(64, nz)
    )
    ################################ END TODO␣
↪#######################################

  def forward(self, x, c=None):
    ################################# TODO␣
↪##########################################
    # If using conditional VAE, concatenate x and a onehot version of c to␣
↪create  #
    # the full input. Use function idx2onehot above.                      ␣
↪   #
  ␣
↪##################################################################################
    if self.conditional:
        x = torch.cat([x, idx2onehot(c, self.num_labels)], dim=1)
  ␣
↪##################################################################################
    return self.net(x)


class CVAEDecoder(nn.Module):
  def __init__(self, nz, output_size, conditional, num_labels):
    super().__init__()
    self.output_size = output_size
    self.conditional = conditional
    self.num_labels = num_labels
    if self.conditional:
        nz = nz + num_labels
    ################################# TODO␣
↪##########################################
    # Create the network architecture using a nn.Sequential module wrapper.  ␣
↪   #
    # Decoder Architecture (mirrors encoder architecture):                 ␣
↪   #
    # - nz -> 64                                                            ␣
↪   #
    # - ReLU                                                                ␣
↪   #
    # - 64 -> 256                                                           ␣
↪   #
```

```
    # - ReLU                                                              ␣
↪   #
    # - 256 -> output_size                                                ␣
↪   #
   ␣
↪################################################################################
    self.net = nn.Sequential(
        nn.Linear(nz, 64),
        nn.ReLU(),
        nn.Linear(64, 256),
        nn.ReLU(),
        nn.Linear(256, output_size),
        nn.Sigmoid()
    )
    ################################ END TODO␣
↪#######################################

  def forward(self, z, c=None):
    ################################# TODO␣
↪#########################################
    # If using conditional VAE, concatenate z and a onehot version of c to␣
↪create  #
    # the full embedding. Use function idx2onehot above.                  ␣
↪   #
   ␣
↪################################################################################
    if self.conditional:
        z = torch.cat([z, idx2onehot(c, self.num_labels)], dim=1)
    ################################ END TODO␣
↪#######################################

    return self.net(z).reshape(-1, 1, self.output_size)


class CVAE(nn.Module):
    def __init__(self, nz, beta=1.0, conditional=False, num_labels=0):
        super().__init__()
        if conditional:
            assert num_labels > 0
        self.beta = beta
        self.encoder = CVAEEncoder(2*nz, input_size=in_size,␣
↪conditional=conditional, num_labels=num_labels)
        self.decoder = CVAEDecoder(nz, output_size=out_size,␣
↪conditional=conditional, num_labels=num_labels)

    def forward(self, x, c=None):
```

```
        if x.dim() > 2:
            x = x.view(-1, 28*28)

        q = self.encoder(x,c)
        mu, log_sigma = torch.chunk(q, 2, dim=-1)

        # sample latent variable z with reparametrization
        eps = torch.normal(mean=torch.zeros_like(mu), std=torch.
↪ones_like(log_sigma))
        # eps = torch.randn_like(mu) # Alternatively use this
        z = mu + eps * torch.exp(log_sigma)

        # compute reconstruction
        reconstruction = self.decoder(z, c)

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

    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 loss␣
↪above      #
        #          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)                      ␣
↪           #
     ␣
↪##############################################################################
        rec_loss = F.binary_cross_entropy(outputs['rec'].reshape(-1,784), x,␣
↪reduction='mean')
        mu, log_sigma = torch.chunk(outputs['q'], 2, dim=-1)
        kl_loss = kl_divergence(mu, log_sigma, torch.zeros_like(mu), torch.
↪zeros_like(log_sigma)).mean()
```

```
        ##################################### END TODO␣
↪######################################

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

    def reconstruct(self, x, c=None):
        """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, c)
        mu, log_sigma = torch.chunk(q, 2, dim=-1)
        z = mu
        image = self.decoder(z, c)
        ##################################### END TODO␣
↪######################################
        return image
```

6.2 Setting up the CVAE Training loop

```
[ ]: learning_rate = 1e-3
nz = 32

######################################## TODO␣
 ↪#######################################
# Tune the beta parameter to obtain good training results. However, for the    #
# initial experiments leave beta = 0 in order to verify our implementation.    ␣
 ↪   #
####################################################################################
epochs = 5 # works with fewer epochs than AE, VAE. we only test conditional␣
 ↪samples.
beta = 0.2
```

```
####################################### END TODO␣
↪########################################

# build CVAE model
conditional = True
cvae_model = CVAE(nz, beta, conditional=conditional, num_labels=10).to(device) ␣
↪  # transfer model to GPU if available
cvae_model = cvae_model.train()   # set model in train mode (eg batchnorm␣
↪params get updated)

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

train_it = 0
rec_loss, kl_loss = [], []
print(f"Running {epochs} epochs with {beta=}")
for ep in range(epochs):
  print(f"Run Epoch {ep}")
  ######################################## TODO␣
  ↪########################################
  # Implement the main training loop for the model.                             ␣
  ↪     #
  # If using conditional VAE, remember to pass the conditional variable c to␣
  ↪the     #
  # forward pass                                                                 ␣
  ↪     #
  # HINT: Your training loop should sample batches from the data loader, run␣
  ↪the     #
  #       forward pass of the model, 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: As before, 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 sample_images, sample_labels in mnist_data_loader:
    # reshape images to (batch_size, 784)
    images = sample_images.reshape([batch_size, in_size])
    images = images.to(device)
    labels = sample_labels.to(device)
    # reset gradients
    optimizer.zero_grad()

    # forward pass
    outputs = cvae_model(images, labels)

    # compute loss
    loss_dict = cvae_model.loss(images, outputs)
    total_loss = loss_dict[0]
    losses = loss_dict[1]
    # print(total_loss)

    # backward pass
    total_loss.backward()

    # perform one optimization step
    optimizer.step()

    # save losses for plotting
    rec_loss.append(losses['rec_loss']); kl_loss.append(losses['kl_loss'])



    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
  #################################### END TODO␣
↪####################################

print("Done!")

rec_loss_plotdata = [foo.detach().cpu() for foo in rec_loss]
kl_loss_plotdata = [foo.detach().cpu() for foo in kl_loss]
```

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

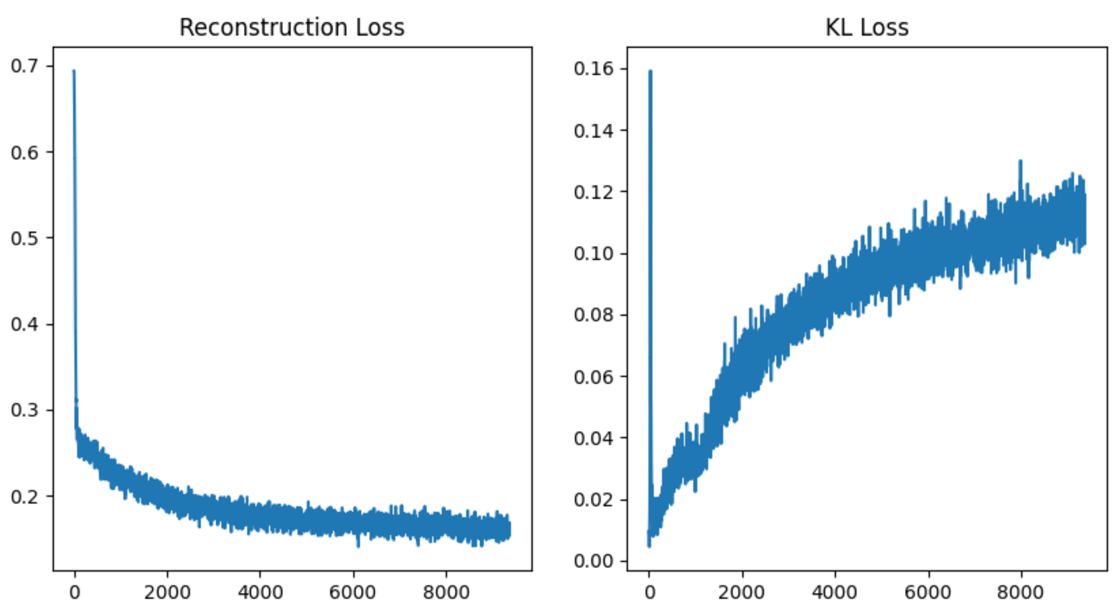
```
Running 5 epochs with beta=0.2
Run Epoch 0
```

```
/var/folders/8y/gs8783k968bbsmv5m7dmmh7h0000gn/T/ipykernel_1237/1883098713.py:3:
UserWarning: MPS: no support for int64 min/max ops, casting it to int32
(Triggered internally at /Users/runner/work/pytorch/pytorch/pytorch/aten/src/ATe
n/native/mps/operations/ReduceOps.mm:1271.)
  assert torch.max(idx).item() < n
```

```
It 0: Total Loss: 0.6956194043159485,    Rec Loss: 0.6937246322631836,   KL
Loss: 0.00947391614317894
It 100: Total Loss: 0.260721892118454,   Rec Loss: 0.25669094920158386,  KL
Loss: 0.020154759287834167
It 200: Total Loss: 0.24837006628513336,         Rec Loss: 0.24243195354938507,
KL Loss: 0.029690533876419067
It 300: Total Loss: 0.2511320114135742,          Rec Loss: 0.24587726593017578,
KL Loss: 0.02627367526292801
It 400: Total Loss: 0.23478983342647552,         Rec Loss: 0.22749966382980347,
KL Loss: 0.0364508256316185
It 500: Total Loss: 0.23287999629974365,         Rec Loss: 0.2259892076253891,
KL Loss: 0.03445395454764366
It 600: Total Loss: 0.21981805562973022,         Rec Loss: 0.212930828332901,
KL Loss: 0.034436099231243134
It 700: Total Loss: 0.2230454683303833,          Rec Loss: 0.21325422823429108,
KL Loss: 0.04895617812871933
It 800: Total Loss: 0.22043979167938232,         Rec Loss: 0.21030935645103455,
KL Loss: 0.050652191042900085
It 900: Total Loss: 0.20771117508411407,         Rec Loss: 0.19788426160812378,
KL Loss: 0.049134545028209686
Run Epoch 1
It 1000: Total Loss: 0.2132871448993683,         Rec Loss: 0.20169280469417572,
KL Loss: 0.05797170475125313
It 1100: Total Loss: 0.21025697886943817,        Rec Loss: 0.19775144755840302,
KL Loss: 0.06252765655517578
It 1200: Total Loss: 0.20517143607139587,        Rec Loss: 0.19091524183750153,
KL Loss: 0.07128100097179413
It 1300: Total Loss: 0.20742833614349365,        Rec Loss: 0.19163766503334045,
```

```
KL Loss: 0.0789533257484436
It 1400: Total Loss: 0.2130509912967682,        Rec Loss: 0.1980966478586197,
KL Loss: 0.0747716873884201
It 1500: Total Loss: 0.2011607140302658,        Rec Loss: 0.18484999239444733,
KL Loss: 0.08155359327793121
It 1600: Total Loss: 0.19429148733615875,       Rec Loss: 0.17874468863010406,
KL Loss: 0.07773397117853165
It 1700: Total Loss: 0.19536611437797546,       Rec Loss: 0.17875538766384125,
KL Loss: 0.08305363357067108
It 1800: Total Loss: 0.19059112668037415,       Rec Loss: 0.1740662157535553,
KL Loss: 0.08262457698583603
Run Epoch 2
It 1900: Total Loss: 0.19311463832855225,       Rec Loss: 0.17617517709732056,
KL Loss: 0.08469727635383606
It 2000: Total Loss: 0.19398333132266998,       Rec Loss: 0.17691615223884583,
KL Loss: 0.08533588796854019
It 2100: Total Loss: 0.21196426451206207,       Rec Loss: 0.19332461059093475,
KL Loss: 0.09319829940795898
It 2200: Total Loss: 0.19123513996601105,       Rec Loss: 0.17097462713718414,
KL Loss: 0.10130259394645691
It 2300: Total Loss: 0.19420577585697174,       Rec Loss: 0.1740705668926239,
KL Loss: 0.10067601501941681
It 2400: Total Loss: 0.20001065731048584,       Rec Loss: 0.18047143518924713,
KL Loss: 0.09769611060619354
It 2500: Total Loss: 0.18178808689117432,       Rec Loss: 0.16433598101139069,
KL Loss: 0.08726052939891815
It 2600: Total Loss: 0.18268899619579315,       Rec Loss: 0.16284224390983582,
KL Loss: 0.09923376888036728
It 2700: Total Loss: 0.19304151833057404,       Rec Loss: 0.1736820787191391,
KL Loss: 0.09679718315601349
It 2800: Total Loss: 0.19333922863006592,       Rec Loss: 0.17563340067863464,
KL Loss: 0.08852913975715637
Run Epoch 3
It 2900: Total Loss: 0.18051007390022278,       Rec Loss: 0.16208194196224213,
KL Loss: 0.09214069694280624
It 3000: Total Loss: 0.1791294813156128,        Rec Loss: 0.1583447903394699,
KL Loss: 0.10392343997955322
It 3100: Total Loss: 0.18459992110729218,       Rec Loss: 0.1644851118326187,
KL Loss: 0.10057403147220612
It 3200: Total Loss: 0.18928878009319305,       Rec Loss: 0.16749079525470734,
KL Loss: 0.10898995399475098
It 3300: Total Loss: 0.18060684204101562,       Rec Loss: 0.161444753408432,
KL Loss: 0.09581047296524048
It 3400: Total Loss: 0.1778506636619568,        Rec Loss: 0.15643210709095,
KL Loss: 0.10709281265735626
It 3500: Total Loss: 0.19502829015254974,       Rec Loss: 0.17234575748443604,
KL Loss: 0.11341264098882675
It 3600: Total Loss: 0.17903289198875427,       Rec Loss: 0.15822066366672516,
```

```
KL Loss: 0.10406111180782318
It 3700: Total Loss: 0.18378674983978271,       Rec Loss: 0.164883092045784,
KL Loss: 0.09451830387115479
Run Epoch 4
It 3800: Total Loss: 0.18141210079193115,       Rec Loss: 0.16053706407546997,
KL Loss: 0.10437516123056412
It 3900: Total Loss: 0.17425134778022766,       Rec Loss: 0.1522674709558487,
KL Loss: 0.10991936922073364
It 4000: Total Loss: 0.18029645085334778,       Rec Loss: 0.1582668274641037,
KL Loss: 0.11014814674854279
It 4100: Total Loss: 0.1889621615409851,        Rec Loss: 0.16683948040008545,
KL Loss: 0.11061344295740128
It 4200: Total Loss: 0.19541144371032715,       Rec Loss: 0.1730339676141739,
KL Loss: 0.11188739538192749
It 4300: Total Loss: 0.17937952280044556,       Rec Loss: 0.15887252986431122,
KL Loss: 0.1025349497795105
It 4400: Total Loss: 0.18996360898017883,       Rec Loss: 0.16852012276649475,
KL Loss: 0.10721741616725922
It 4500: Total Loss: 0.19127579033374786,       Rec Loss: 0.1691390424966812,
KL Loss: 0.11068373918533325
It 4600: Total Loss: 0.17645497620105743,       Rec Loss: 0.15517325699329376,
KL Loss: 0.10640860348939896
Done!
```

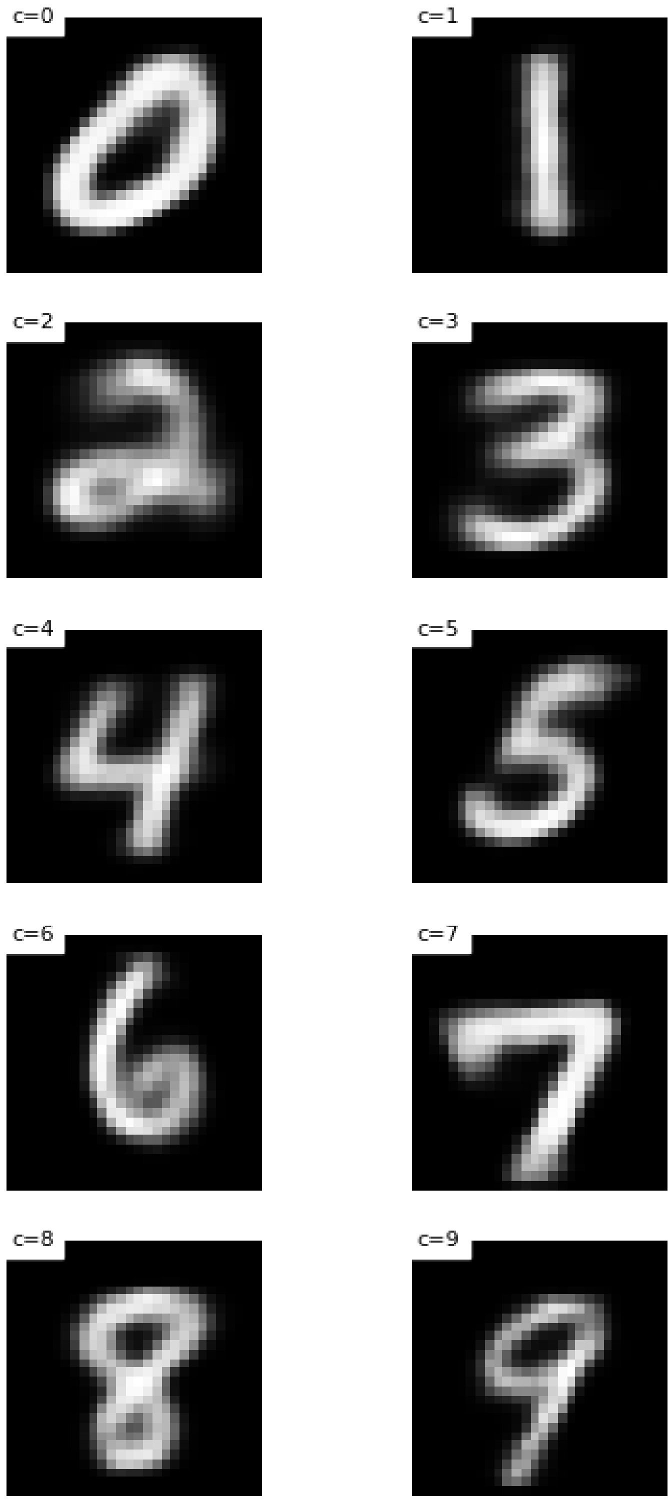


6.2.1 Verifying conditional samples from CVAE [6 pt]

Now let us generate samples from the trained model, conditioned on all the labels.

[ ]:
```
# Prob1-9
if conditional:
    c = torch.arange(0, 10).long().unsqueeze(1).to(device)
    z = torch.randn([10, nz]).to(device)
    x = cvae_model.decoder(z, c=c)
else:
    z = torch.randn([10, nz]).to(device)
    x = cvae_model.decoder(z)

plt.figure()
plt.figure(figsize=(5, 10))
for p in range(10):
    plt.subplot(5, 2, p+1)
    if conditional:
        plt.text(
            0, 0, "c={:d}".format(c[p].item()), color='black',
            backgroundcolor='white', fontsize=8)
    plt.imshow(x[p].view(28, 28).cpu().data.numpy(), cmap='gray')
    plt.axis('off')
```

```
<Figure size 640x480 with 0 Axes>
```


c=0
c=1
c=2
c=3
c=4
c=5
c=6
c=7
c=8
c=9

7 Submission Instructions

You need to submit this jupyter notebook and a PDF. See Piazza for detailed submission instructions.

```
[ ]:
```