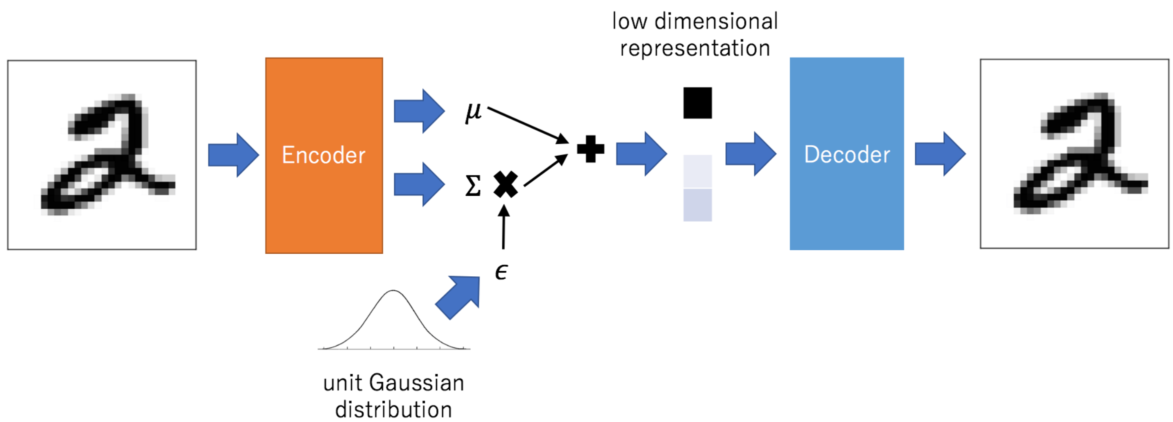
Problem 1 - Variational Auto-Encoder (VAE)

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.

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.

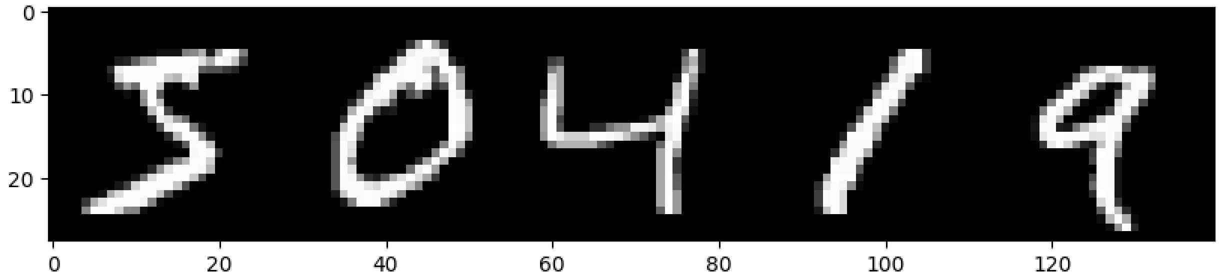
Let's start by downloading the data and visualizing some samples.

In [1]:
```
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
```

In [2]:
```
import torch
import torchvision
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')   # use GPU i
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: cpu

 Download complete! Downloaded 60000 training examples!
```

In [3]:
```
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 {sam
  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()
```



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.

Defining the Auto-Encoder Architecture [6pt]

In [4]:
```
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.      #
    ################################################################################

    # Here input_size = 28*28 = 784 ; output_dim = nz = 32
    hidden_dim1 = 256
    hidden_dim2 = 64
    self.net = nn.Sequential(
          nn.Linear(self.input_size, hidden_dim1),
          nn.ReLU(),
          nn.Linear(hidden_dim1, hidden_dim2),
          nn.ReLU(),
          nn.Linear(hidden_dim2, 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                                                         #
    ################################################################################

    # Here nz = 32 and output_size = 28*28
    hidden_dim1 = 64
    hidden_dim2 = 256
    self.net = nn.Sequential(
          nn.Linear(nz, hidden_dim1),
          nn.ReLU(),
          nn.Linear(hidden_dim1, hidden_dim2),
          nn.ReLU(),
          nn.Linear(hidden_dim2,output_size),
          nn.Sigmoid()
    )
    ################################# END TODO ########################################

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

Testing the Auto-Encoder Forward Pass

In [5]:

```
# 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 errors.                #
# 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
  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=tor
ch.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, 78
4]).
```

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.

In [6]:

```
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
```

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

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

In [7]:
```
# 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 updat

# 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.            #
####################################################################################
class Loss(nn.Module):
  def __init__(self, net):
      super().__init__()
      self.net = net

  def cross_entropy(self,x,x_hat):
      """
      Given a batch of images, this function returns the reconstruction loss (Binary C
      Inputs:
      - x_hat: Reconstruced input data of shape (N, 1, H*W)
      - x: Input data for this timestep of shape (N, 1, H, W)
      Returns:
      - loss: Tensor containing the scalar loss
      """
      rec_term = nn.functional.binary_cross_entropy(x_hat, x)
      return rec_term

optimizer = torch.optim.Adam(ae_model.parameters(), lr=learning_rate)
###################################### 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.   #
  ##################################################################################
  #Define models
  loss_func          = Loss(ae_model)

  #Main training loop
  for data, labels in mnist_data_loader:
    optimizer.zero_grad()
    x = data.reshape([batch_size,1,-1])  #[64,1,784] input
    x_hat = ae_model.forward(x)                 # [64,1,784] as decoded output
    loss = loss_func.cross_entropy(x,x_hat)
    rec_loss = loss.item()
    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
```

```
Device available cpu
Run Epoch 0
It 0: Reconstruction Loss: 0.6938329935073853
It 100: Reconstruction Loss: 0.2569555640220642
It 200: Reconstruction Loss: 0.23635277152061462
It 300: Reconstruction Loss: 0.1860174983739853
It 400: Reconstruction Loss: 0.16457697749137878
It 500: Reconstruction Loss: 0.15991303324699402
It 600: Reconstruction Loss: 0.15143071115016937
It 700: Reconstruction Loss: 0.13905277848243713
It 800: Reconstruction Loss: 0.13941501080989838
It 900: Reconstruction Loss: 0.13531626760959625
Run Epoch 1
It 1000: Reconstruction Loss: 0.12519817054271698
It 1100: Reconstruction Loss: 0.12607747316360474
It 1200: Reconstruction Loss: 0.12638314068317413
It 1300: Reconstruction Loss: 0.12222416698932648
It 1400: Reconstruction Loss: 0.11951999366283417
It 1500: Reconstruction Loss: 0.12242441624403
It 1600: Reconstruction Loss: 0.11918717622756958
It 1700: Reconstruction Loss: 0.11149801313877106
It 1800: Reconstruction Loss: 0.1054108738899231
Run Epoch 2
It 1900: Reconstruction Loss: 0.1114017516374588
It 2000: Reconstruction Loss: 0.11107499152421951
It 2100: Reconstruction Loss: 0.11277510225772858
It 2200: Reconstruction Loss: 0.1165136992931366
It 2300: Reconstruction Loss: 0.11966025084257126
It 2400: Reconstruction Loss: 0.11350910365581512
It 2500: Reconstruction Loss: 0.10243100672960281
It 2600: Reconstruction Loss: 0.10380048304796219
It 2700: Reconstruction Loss: 0.10798954218626022
It 2800: Reconstruction Loss: 0.10299589484930038
Run Epoch 3
It 2900: Reconstruction Loss: 0.1141364797949791
It 3000: Reconstruction Loss: 0.10195007920265198
It 3100: Reconstruction Loss: 0.10179024934768677
It 3200: Reconstruction Loss: 0.09840816259384155
It 3300: Reconstruction Loss: 0.10996390134096146
It 3400: Reconstruction Loss: 0.0971972793340683
It 3500: Reconstruction Loss: 0.09546881169080734
It 3600: Reconstruction Loss: 0.09931895136833191
It 3700: Reconstruction Loss: 0.0938466489315033
Run Epoch 4
It 3800: Reconstruction Loss: 0.10071701556444168
It 3900: Reconstruction Loss: 0.10130659490823746
It 4000: Reconstruction Loss: 0.09417625516653061
It 4100: Reconstruction Loss: 0.09656966477632523
It 4200: Reconstruction Loss: 0.10197492688894272
It 4300: Reconstruction Loss: 0.09621604532003403
It 4400: Reconstruction Loss: 0.10513295978307724
It 4500: Reconstruction Loss: 0.09467465430498123
It 4600: Reconstruction Loss: 0.09195762127637863
Run Epoch 5
It 4700: Reconstruction Loss: 0.09589675813913345
It 4800: Reconstruction Loss: 0.09865595400333405
It 4900: Reconstruction Loss: 0.08860144019126892
It 5000: Reconstruction Loss: 0.08932796865701675
It 5100: Reconstruction Loss: 0.09967068582773209
It 5200: Reconstruction Loss: 0.09360353648662567
```

```
It 5300: Reconstruction Loss: 0.09894923865795135
It 5400: Reconstruction Loss: 0.09478701651096344
It 5500: Reconstruction Loss: 0.09014914184808731
It 5600: Reconstruction Loss: 0.09137824177742004
Run Epoch 6
It 5700: Reconstruction Loss: 0.08713827282190323
It 5800: Reconstruction Loss: 0.08904056251049042
It 5900: Reconstruction Loss: 0.0988946482539177
It 6000: Reconstruction Loss: 0.08909577131271362
It 6100: Reconstruction Loss: 0.10124988853931427
It 6200: Reconstruction Loss: 0.09208624809980392
It 6300: Reconstruction Loss: 0.09246266633272171
It 6400: Reconstruction Loss: 0.0932762399315834
It 6500: Reconstruction Loss: 0.09367182105779648
Run Epoch 7
It 6600: Reconstruction Loss: 0.08903134614229202
It 6700: Reconstruction Loss: 0.09674311429262161
It 6800: Reconstruction Loss: 0.0953303724527359
It 6900: Reconstruction Loss: 0.08966638892889023
It 7000: Reconstruction Loss: 0.09215637296438217
It 7100: Reconstruction Loss: 0.08817680180072784
It 7200: Reconstruction Loss: 0.09264498203992844
It 7300: Reconstruction Loss: 0.09817483276128769
It 7400: Reconstruction Loss: 0.08968212455511093
Run Epoch 8
It 7500: Reconstruction Loss: 0.08880729228258133
It 7600: Reconstruction Loss: 0.09190203994512558
It 7700: Reconstruction Loss: 0.08737900853157043
It 7800: Reconstruction Loss: 0.09107953310012817
It 7900: Reconstruction Loss: 0.08999734371900558
It 8000: Reconstruction Loss: 0.08645826578140259
It 8100: Reconstruction Loss: 0.09110664576292038
It 8200: Reconstruction Loss: 0.09122484922409058
It 8300: Reconstruction Loss: 0.08559893816709518
It 8400: Reconstruction Loss: 0.09122334420681
Run Epoch 9
It 8500: Reconstruction Loss: 0.08359403163194656
It 8600: Reconstruction Loss: 0.08975856006145477
It 8700: Reconstruction Loss: 0.08736152201890945
It 8800: Reconstruction Loss: 0.09081952273845673
It 8900: Reconstruction Loss: 0.08784164488315582
It 9000: Reconstruction Loss: 0.08497586846351624
It 9100: Reconstruction Loss: 0.08587465435266495
It 9200: Reconstruction Loss: 0.08558985590934753
It 9300: Reconstruction Loss: 0.09142599254846573
Done!
```

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.**

In [8]:
```
# visualize test data reconstructions
def vis_reconstruction(model, randomize=False):
  # download MNIST test set + build Dataset object
  mnist_test = torchvision.datasets.MNIST(root='./data',
```

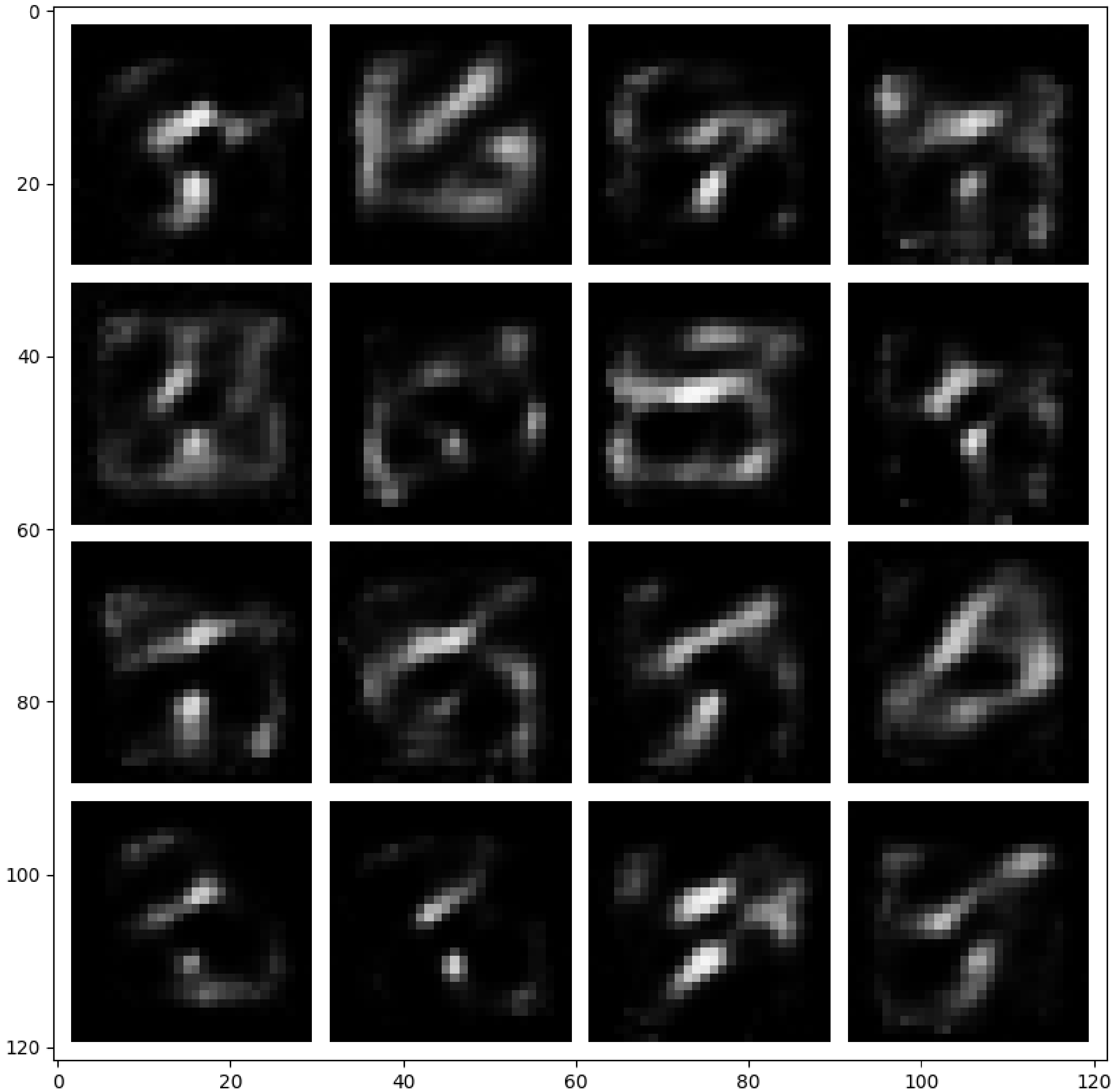
```
                                          train=False,
                                          download=True,
                                          transform=torchvision.transforms.ToTensor()
  model.eval()      # set model in evalidation mode (eg freeze batchnorm params)
  num_samples = 5
  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_train[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), cr
  plt.show()

vis_reconstruction(ae_model, randomize=False) # set randomize to False for debugging
```

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.

In [9]:
```
# 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 th
```

```
    # 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.               #
    ###########################################################################
    #num_samples = Batch size, embedding_dim = nz
    z= torch.randn(batch_size, nz)

    # Scale and shift the tensor for each sample to obtain samples from a diagonal unit
    sample_embeddings =z#/ torch.sqrt(torch.tensor(nz, dtype=torch.float32)).unsqueeze(6

    #[64, 1, 784] -> [64, 1, 28,28]
    decoded_samples = model.decoder(sample_embeddings)
    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:

The observation of the above sampling is that the auto-encoder can generate new data samples by sampling from a Gaussian prior distribution and then decoding the samples through the decoder network. By using the auto-encoder as a generative model, we can evaluate its ability to learn and represent the underlying data distribution in the latent space, and generate new samples that are similar to the original data distribution.

When sampling embeddings from a diagonal unit Gaussian distribution and decoding them using the auto-encoder model, we can make the following observations:

> 1. The generated samples will be similar to the original data: Since the auto-encoder is trained to reconstruct the input data from the encoded embeddings, the generated samples will be similar to the original data in terms of their features and structure.
>
> 2. The generated samples may lack diversity: Since we are using a diagonal unit Gaussian distribution to sample embeddings, we are assuming that the dimensions of the latent space are independent of each other. This may result in generated samples that lack diversity, as there may be limited variation in the embeddings along each dimension.
>
> 3. The generated samples may not capture the true data distribution: The diagonal unit Gaussian distribution may not accurately represent the true probability distribution of the data in the latent space. As a result, the generated samples may not capture the full range of possible samples that could be generated from the true distribution.
>
> Overall, while sampling embeddings from a diagonal unit Gaussian distribution and decoding them using the auto-encoder model can provide some insights into the auto-encoder's ability to generate new data samples, it may not be the most effective way to evaluate its generative capabilities. Alternative approaches such as using Variational Autoencoders (VAEs) or Generative Adversarial Networks (GANs) may be more suitable for generating diverse and novel samples that better capture the true distribution of the data.

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):

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

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

$$\mathcal{L}_{\text{VAE}} = \sum_{x\sim\mathcal{D}} (x - \hat{x})^2 - \beta \cdot D_{\text{KL}}\big(\mathcal{N}(\mu_q, \sigma_q), \mathcal{N}(0, I)\big)$$

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

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

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

Reparametrization Trick

The sampling procedure inside the VAE's forward pass for obtaining a sample z from the posterior distribution $q(z|x)$, 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.

Defining the VAE Model [7pt]

In [10]:
```
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=2*nz, input_size=in_size) #in_size = 28*28
    self.decoder = Decoder(nz=nz, output_size=out_size) #out_size = 28*28
    ###################################### 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
    ######################################################################################

    # 1. Encode input x (input image) => Output shape is: (N, H_d)
    q = self.encoder(x)

    # Get the posterior mu and log of standard deviation from the encoder's output. Si
    mu, logsigma = torch.chunk(q, 2, dim=-1)

    # Convert the "log of the standard deviation" to "sigma" (standard deviation).
    sigma = torch.exp(logsigma)

    # Reparametrize to compute the latent vector "z", of shape (N, Z)
    z = sigma*torch.randn_like(mu)  + mu

    # Pass "z" through the decoder to resconstruct "x" => "x_hat".
    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)
    ######################################################################################
    q , reconstruction = outputs['q'], outputs['rec']

    mu_q, logsigma_q = torch.chunk(q, 2, dim=-1)

    # Scale and shift the tensor for each sample to obtain samples from a diagonal uni
```

```
    # Epsilon is a Tensor that contains random samples from a standard normal distribu
    mu_p, logsigma_p = torch.zeros(mu_q.shape), torch.zeros(logsigma_q.shape)

    # Compute the reconstruction loss term : Binary Cross Entropy
    rec_loss = nn.functional.binary_cross_entropy(reconstruction, x)

    # Compute the KL divergence term
    kl_loss= torch.mean(kl_divergence(mu_q, logsigma_q, mu_p, logsigma_p ))
    #################################### 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 obtair
    #       the reconstruction.
    ###################################################################################

    recon_img = self.forward(x)['rec']
    image = recon_img.reshape(-1, 28, 28)

    #################################### END TODO ####################################
    return image
```

Setting up the VAE Training Loop [4pt]

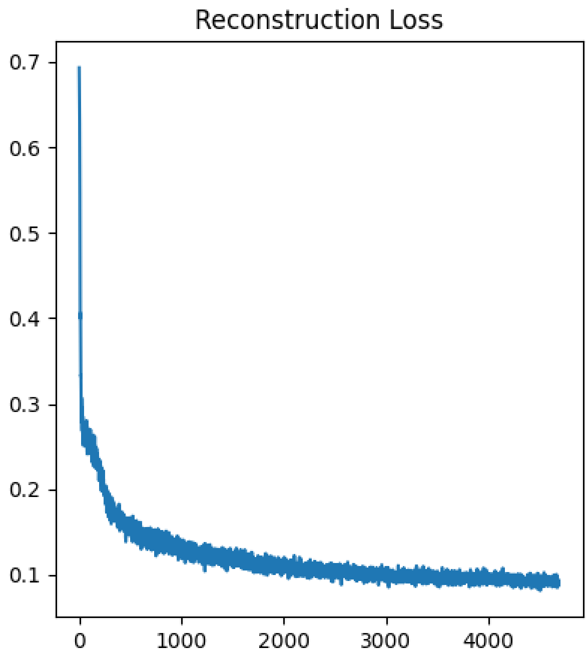
Let's start training the VAE model! We will first verify our implementation by setting $\beta = 0$.

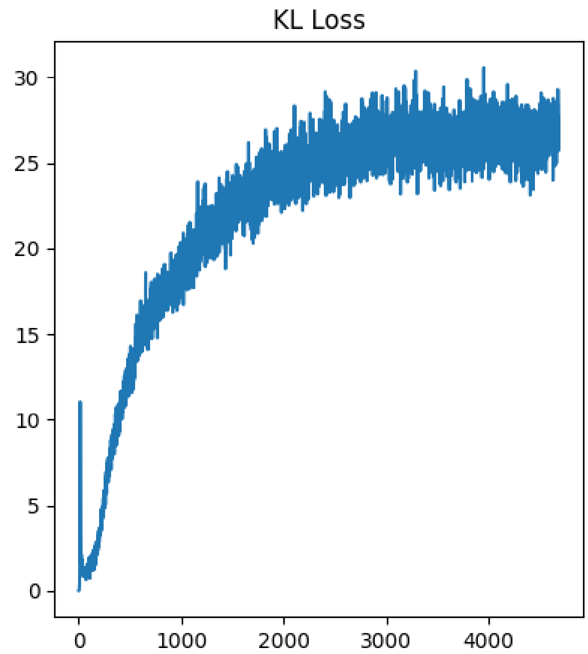
In [11]:
```
# Prob1-5 VAE training loop
learning_rate = 1e-3
nz = 32
beta = 0

######################################## TODO ######################################
epochs = 5      # 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 upd

# 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 data, labels in mnist_data_loader:

    optimizer.zero_grad()

    x = data.reshape([batch_size,1,-1])
    outputs = vae_model.forward(x)
    total_loss, losses = vae_model.loss(x,outputs)

    total_loss.backward()
    optimizer.step()

    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 5 epochs with beta=0
Run Epoch 0
It 0: Total Loss: 0.693147599697113,     Rec Loss: 0.693147599697113,    KL Loss: 0.0
11819479987025261
It 100: Total Loss: 0.2433088719844818,         Rec Loss: 0.2433088719844818,   KL L
oss: 1.7700715065002441
It 200: Total Loss: 0.233688086271286,   Rec Loss: 0.233688086271286,    KL Loss: 2.8
968505859375
It 300: Total Loss: 0.17495886981487274,        Rec Loss: 0.17495886981487274,  KL L
oss: 6.944118976593018
It 400: Total Loss: 0.1512487381696701,         Rec Loss: 0.1512487381696701,   KL L
oss: 10.237713813781738
It 500: Total Loss: 0.1569766104221344,         Rec Loss: 0.1569766104221344,   KL L
oss: 12.905401229858398
It 600: Total Loss: 0.14225216209888458,        Rec Loss: 0.14225216209888458,  KL L
oss: 15.840085983276367
It 700: Total Loss: 0.14923934638500214,        Rec Loss: 0.14923934638500214,  KL L
oss: 14.74007511138916
It 800: Total Loss: 0.144846111536026,   Rec Loss: 0.144846111536026,    KL Loss: 17.
253496170043945
It 900: Total Loss: 0.13558420538902283,        Rec Loss: 0.13558420538902283,  KL L
oss: 19.087926864624023
Run Epoch 1
It 1000: Total Loss: 0.13687224686145782,       Rec Loss: 0.13687224686145782,  KL L
oss: 18.518619537353516
It 1100: Total Loss: 0.13799136877059937,       Rec Loss: 0.13799136877059937,  KL L
oss: 17.934894561767578
It 1200: Total Loss: 0.1292702704668045,        Rec Loss: 0.1292702704668045,   KL L
oss: 21.42696762084961
It 1300: Total Loss: 0.12063372135162354,       Rec Loss: 0.12063372135162354,  KL L
oss: 20.878293991088867
It 1400: Total Loss: 0.11643146723508835,       Rec Loss: 0.11643146723508835,  KL L
oss: 21.251583099365234
It 1500: Total Loss: 0.1143510714173317,        Rec Loss: 0.1143510714173317,   KL L
oss: 22.459924697875977
It 1600: Total Loss: 0.11115910857915878,       Rec Loss: 0.11115910857915878,  KL L
oss: 22.796112060546875
It 1700: Total Loss: 0.10386842489242554,       Rec Loss: 0.10386842489242554,  KL L
oss: 22.842941284179688
It 1800: Total Loss: 0.11153236776590347,       Rec Loss: 0.11153236776590347,  KL L
oss: 24.47545623779297
Run Epoch 2
It 1900: Total Loss: 0.1055745780467987,        Rec Loss: 0.1055745780467987,   KL L
oss: 25.31749153137207
It 2000: Total Loss: 0.10218067467212677,       Rec Loss: 0.10218067467212677,  KL L
oss: 25.219438552856445
It 2100: Total Loss: 0.12267471849918365,       Rec Loss: 0.12267471849918365,  KL L
oss: 24.03156280517578
It 2200: Total Loss: 0.10882527381181717,       Rec Loss: 0.10882527381181717,  KL L
oss: 24.9440975189209
It 2300: Total Loss: 0.10048309713602066,       Rec Loss: 0.10048309713602066,  KL L
oss: 24.68692398071289
It 2400: Total Loss: 0.10352138429880142,       Rec Loss: 0.10352138429880142,  KL L
oss: 26.606435775756836
It 2500: Total Loss: 0.10721655935049057,       Rec Loss: 0.10721655935049057,  KL L
oss: 26.444252014160156
It 2600: Total Loss: 0.10174262523651123,       Rec Loss: 0.10174262523651123,  KL L
oss: 24.888578414916992
It 2700: Total Loss: 0.09354066103696823,       Rec Loss: 0.09354066103696823,  KL L
oss: 25.4385986328125
```

```
It 2800: Total Loss: 0.1008419468998909,        Rec Loss: 0.1008419468998909,   KL L
oss: 27.260311126708984
Run Epoch 3
It 2900: Total Loss: 0.10200774669647217,       Rec Loss: 0.10200774669647217,  KL L
oss: 26.01701545715332
It 3000: Total Loss: 0.09535600990056992,       Rec Loss: 0.09535600990056992,  KL L
oss: 25.99920654296875
It 3100: Total Loss: 0.09757409989833832,       Rec Loss: 0.09757409989833832,  KL L
oss: 26.165878295898438
It 3200: Total Loss: 0.09380381554365158,       Rec Loss: 0.09380381554365158,  KL L
oss: 24.523958206176758
It 3300: Total Loss: 0.09243538230657578,       Rec Loss: 0.09243538230657578,  KL L
oss: 25.77347183227539
It 3400: Total Loss: 0.09652618318796158,       Rec Loss: 0.09652618318796158,  KL L
oss: 26.147188186645508
It 3500: Total Loss: 0.09168478846549988,       Rec Loss: 0.09168478846549988,  KL L
oss: 25.7030029296875
It 3600: Total Loss: 0.09531252086162567,       Rec Loss: 0.09531252086162567,  KL L
oss: 24.55748748779297
It 3700: Total Loss: 0.09894564002752304,       Rec Loss: 0.09894564002752304,  KL L
oss: 27.554988861083984
Run Epoch 4
It 3800: Total Loss: 0.09562414884567261,       Rec Loss: 0.09562414884567261,  KL L
oss: 26.60307502746582
It 3900: Total Loss: 0.09242966026067734,       Rec Loss: 0.09242966026067734,  KL L
oss: 26.090505599975586
It 4000: Total Loss: 0.10001926124095917,       Rec Loss: 0.10001926124095917,  KL L
oss: 26.964391708374023
It 4100: Total Loss: 0.08997836709022522,       Rec Loss: 0.08997836709022522,  KL L
oss: 25.46681785583496
It 4200: Total Loss: 0.0920863226056099,        Rec Loss: 0.0920863226056099,   KL L
oss: 27.5040283203125
It 4300: Total Loss: 0.09425146877765656,       Rec Loss: 0.09425146877765656,  KL L
oss: 26.26022720336914
It 4400: Total Loss: 0.09243783354759216,       Rec Loss: 0.09243783354759216,  KL L
oss: 27.782285690307617
It 4500: Total Loss: 0.08945124596357346,       Rec Loss: 0.08945124596357346,  KL L
oss: 26.522281646728516
It 4600: Total Loss: 0.08883402496576309,       Rec Loss: 0.08883402496576309,  KL L
oss: 27.33452606201172
Done!
```

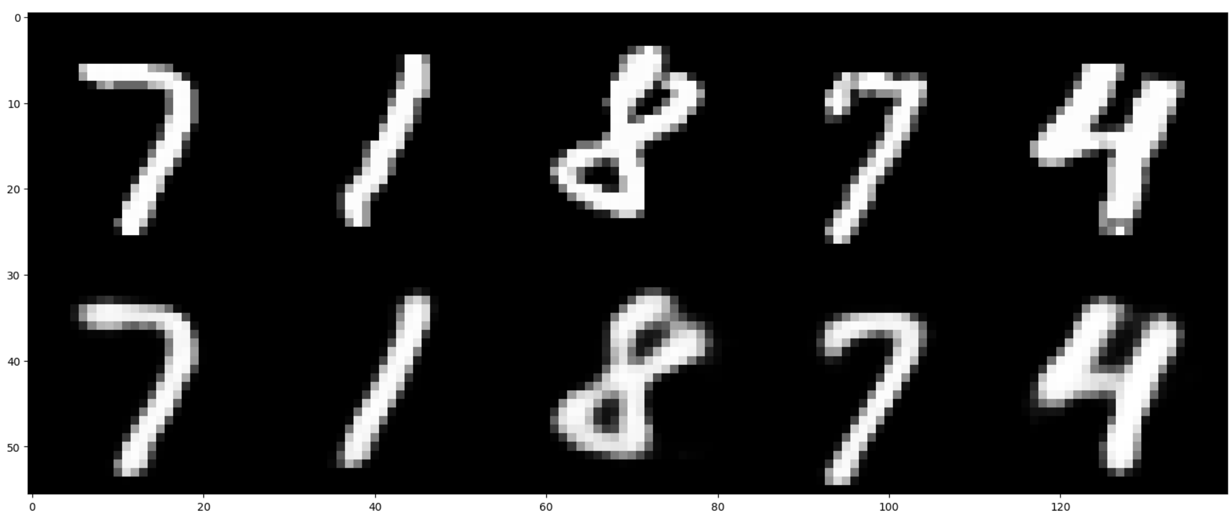


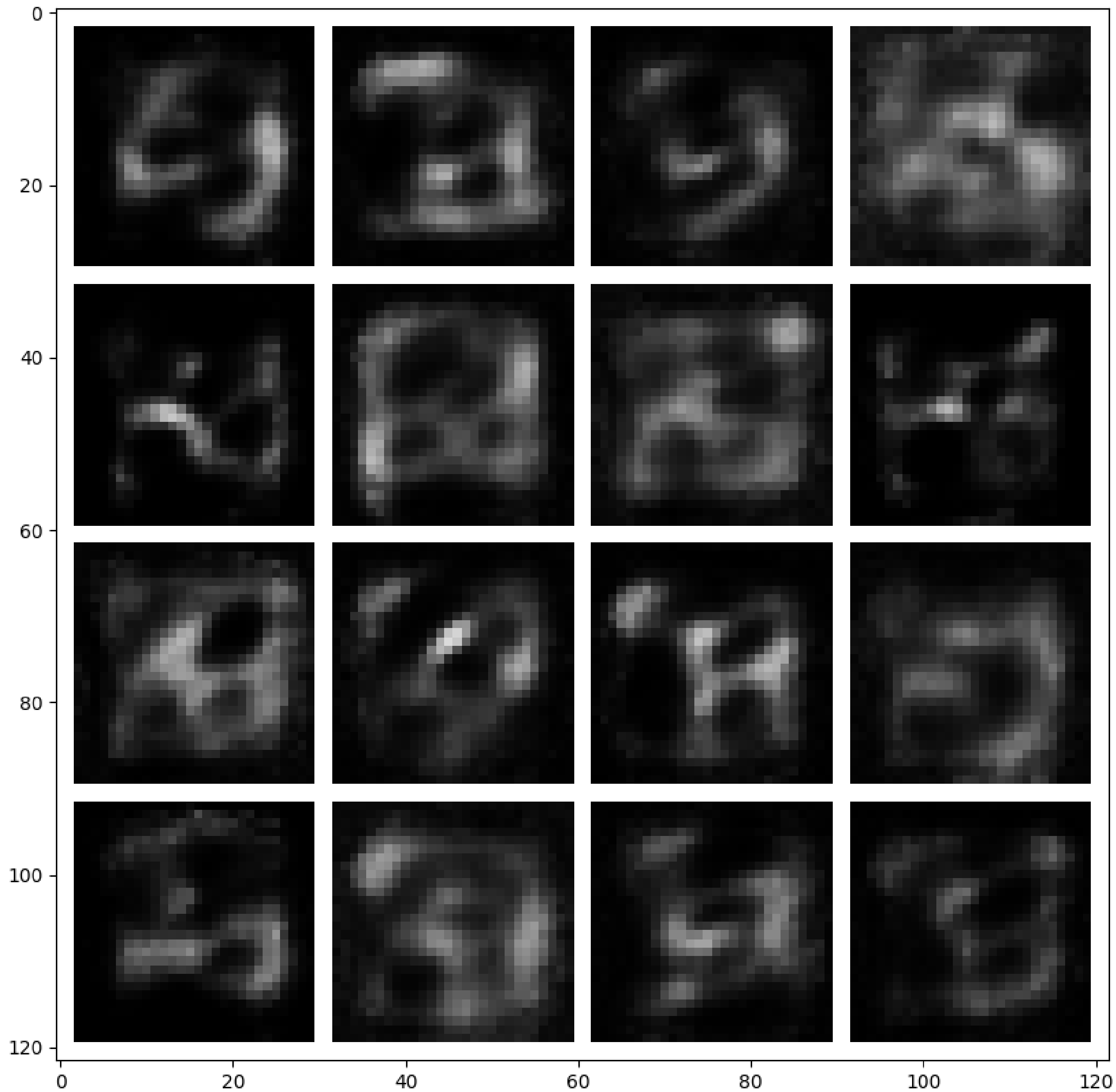


Let's look at some reconstructions and decoded embedding samples!

In [12]:
```
# visualize VAE reconstructions and samples from the generative model
print("beta = ", beta)
vis_reconstruction(vae_model, randomize=True)
vis_samples(vae_model)
```

```
beta =  0
```





Tweaking the loss function β [2pt]

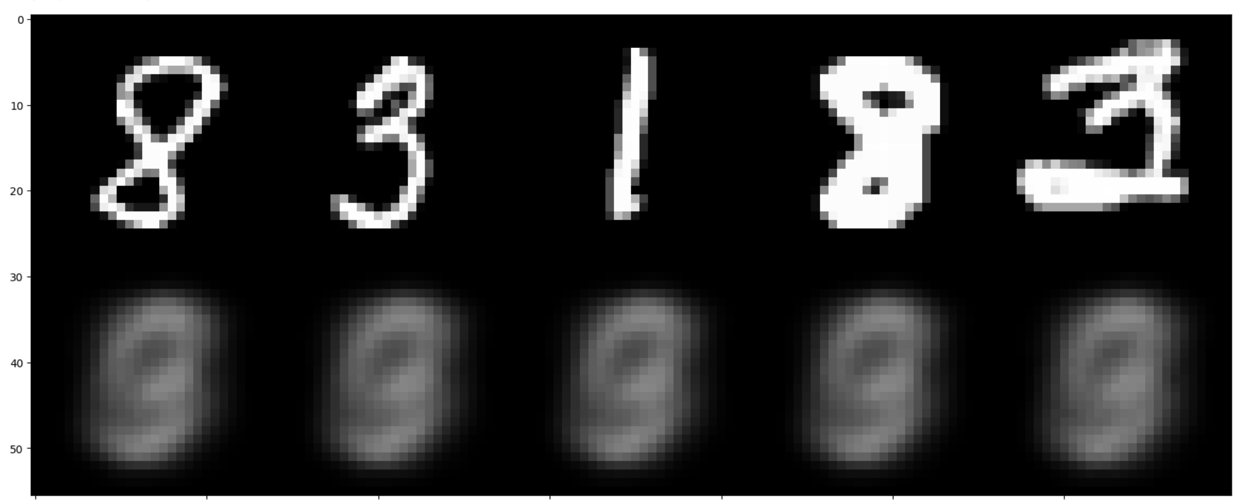
Prob1-6: Let's repeat the same experiment for $\beta = 10$, a very high value for the coefficient.

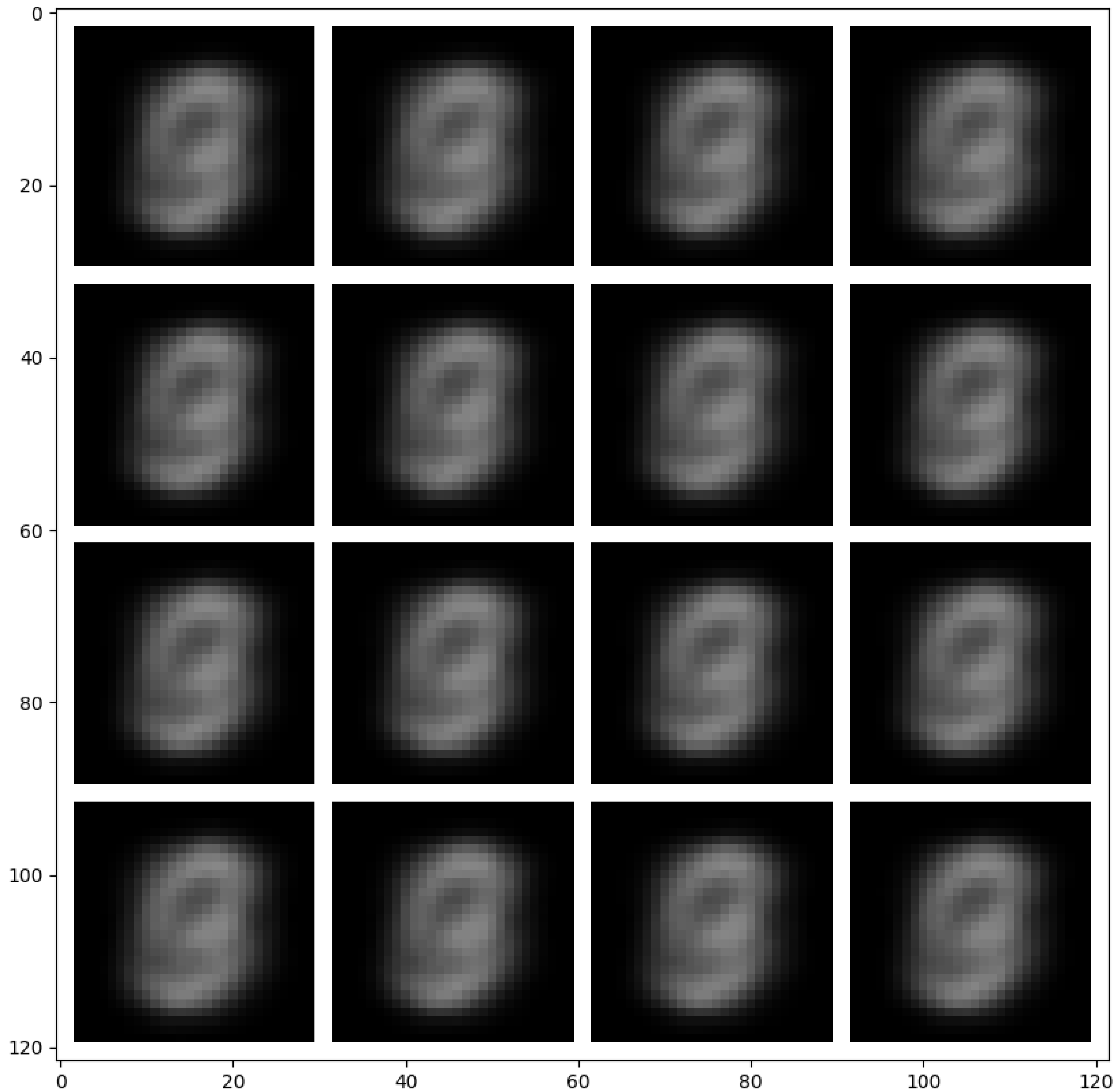
In [13]:
```
# VAE training loop
learning_rate = 1e-3
nz = 32
beta = 10

######################################## TODO ########################################
epochs = 5      # 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 upd

# 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 data, labels in mnist_data_loader:

    optimizer.zero_grad()

    x = data.reshape([batch_size,1,-1])
    outputs = vae_model(x)

    total_loss, losses = vae_model.loss(x,outputs)
    total_loss.backward()

    optimizer.step()

    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

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

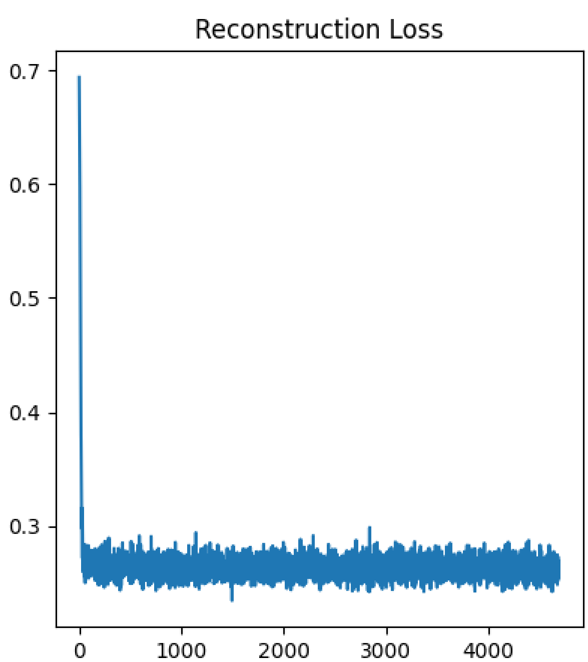
```
Running 5 epochs with beta=10
Run Epoch 0
It 0: Total Loss: 0.781637966632843,     Rec Loss: 0.6936955451965332,   KL Loss: 0.0
08794243447482586
It 100: Total Loss: 0.26054847240448,    Rec Loss: 0.25999507308006287,  KL Loss: 5.5
34125375561416e-05
It 200: Total Loss: 0.2572806477546692,          Rec Loss: 0.2570207417011261,   KL L
oss: 2.5991248548962176e-05
It 300: Total Loss: 0.2733444273471832,          Rec Loss: 0.27316153049468994,  KL L
oss: 1.829025859478861e-05
It 400: Total Loss: 0.2715872824192047,          Rec Loss: 0.27147042751312256,  KL L
oss: 1.1684998753480613e-05
It 500: Total Loss: 0.27069008350372314,         Rec Loss: 0.2705797851085663,   KL L
oss: 1.1030249879695475e-05
It 600: Total Loss: 0.2749125361442566,          Rec Loss: 0.27483218908309937,  KL L
oss: 8.03446164354682e-06
It 700: Total Loss: 0.2598475217819214,          Rec Loss: 0.25977659225463867,  KL L
oss: 7.093287422321737e-06
It 800: Total Loss: 0.25552621483802795,         Rec Loss: 0.25546643137931824,  KL L
oss: 5.978974513709545e-06
It 900: Total Loss: 0.27761054039001465,         Rec Loss: 0.27753862738609314,  KL L
oss: 7.191323675215244e-06
Run Epoch 1
It 1000: Total Loss: 0.25601157546043396,        Rec Loss: 0.25595182180404663,  KL L
oss: 5.9743470046669245e-06
It 1100: Total Loss: 0.2716078758239746,         Rec Loss: 0.2715624272823334,   KL L
oss: 4.545712727122009e-06
It 1200: Total Loss: 0.2715733051300049,         Rec Loss: 0.2715413570404053,   KL L
oss: 3.194261807948351e-06
It 1300: Total Loss: 0.26294994354248047,        Rec Loss: 0.2629200518131256,   KL L
oss: 2.9890798032283783e-06
It 1400: Total Loss: 0.2680308222770691,         Rec Loss: 0.2680012881755829,   KL L
oss: 2.953413059003651e-06
It 1500: Total Loss: 0.2602635324001312,         Rec Loss: 0.2602294683456421,   KL L
oss: 3.405657480470836e-06
It 1600: Total Loss: 0.25746241211891174,        Rec Loss: 0.2574446499347687,   KL L
oss: 1.7765996744856238e-06
It 1700: Total Loss: 0.2533991038799286,         Rec Loss: 0.25337961316108704,  KL L
oss: 1.948923454619944e-06
It 1800: Total Loss: 0.2586655616760254,         Rec Loss: 0.25864270329475403,  KL L
oss: 2.285698428750038e-06
Run Epoch 2
It 1900: Total Loss: 0.27191388607025146,        Rec Loss: 0.271893173456192,    KL L
oss: 2.0700827008113265e-06
It 2000: Total Loss: 0.2718668580055237,         Rec Loss: 0.2718465328216553,   KL L
oss: 2.031141775660217e-06
It 2100: Total Loss: 0.27126815915107727,        Rec Loss: 0.27125176787376404,  KL L
oss: 1.6380945453420281e-06
It 2200: Total Loss: 0.27224189043045044,        Rec Loss: 0.27223101258277893,  KL L
oss: 1.0871590347960591e-06
It 2300: Total Loss: 0.26582589745521545,        Rec Loss: 0.26581501960754395,  KL L
oss: 1.0882504284381866e-06
It 2400: Total Loss: 0.2638525664806366,         Rec Loss: 0.2638431787490845,   KL L
oss: 9.383948054164648e-07
It 2500: Total Loss: 0.279774934053421,          Rec Loss: 0.27976590394973755,  KL L
oss: 9.022187441587448e-07
It 2600: Total Loss: 0.25447311997413635,        Rec Loss: 0.25446251034736633,  KL L
oss: 1.061373041011393e-06
It 2700: Total Loss: 0.252629816532135,          Rec Loss: 0.25262120366096497,  KL L
oss: 8.618226274847984e-07
```

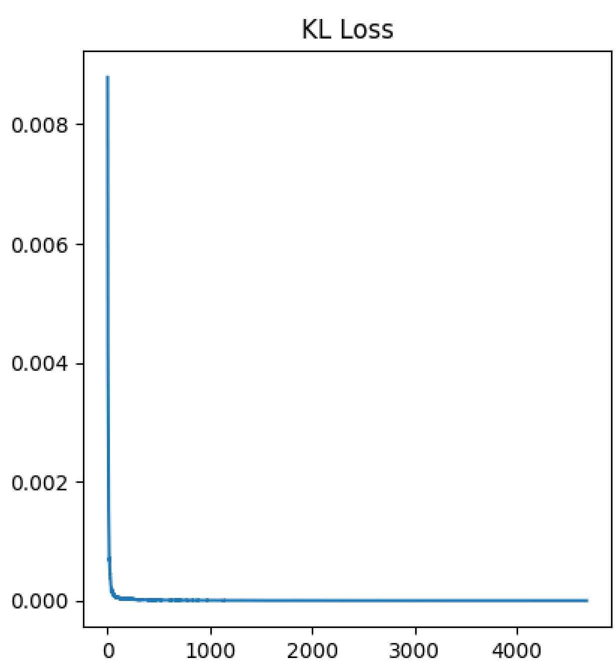
```
It 2800: Total Loss: 0.2691068947315216,         Rec Loss: 0.2690988779067993,   KL L
oss: 8.019414963200688e-07
Run Epoch 3
It 2900: Total Loss: 0.2620745897293091,         Rec Loss: 0.26206910610198975,  KL L
oss: 5.471811164170504e-07
It 3000: Total Loss: 0.26083722710609436,        Rec Loss: 0.260831356048584,    KL L
oss: 5.875190254300833e-07
It 3100: Total Loss: 0.2677065432071686,         Rec Loss: 0.26770132780075073,  KL L
oss: 5.214387783780694e-07
It 3200: Total Loss: 0.27675727009773254,        Rec Loss: 0.2767491936683655,   KL L
oss: 8.064962457865477e-07
It 3300: Total Loss: 0.2697741389274597,         Rec Loss: 0.2697668969631195,   KL L
oss: 7.237686077132821e-07
It 3400: Total Loss: 0.26040735840797424,        Rec Loss: 0.2604040801525116,   KL L
oss: 3.290333552286029e-07
It 3500: Total Loss: 0.2558209300041199,         Rec Loss: 0.25581347942352295,  KL L
oss: 7.456546882167459e-07
It 3600: Total Loss: 0.2675812542438507,         Rec Loss: 0.26757732033729553,  KL L
oss: 3.945315256714821e-07
It 3700: Total Loss: 0.2500292956829071,         Rec Loss: 0.2500261068344116,   KL L
oss: 3.1964736990630627e-07
Run Epoch 4
It 3800: Total Loss: 0.25322484970092773,        Rec Loss: 0.2532195746898651,   KL L
oss: 5.278561729937792e-07
It 3900: Total Loss: 0.2575134336948395,         Rec Loss: 0.2575107514858246,   KL L
oss: 2.6807538233697414e-07
It 4000: Total Loss: 0.25669270753860474,        Rec Loss: 0.2566896677017212,   KL L
oss: 3.047025529667735e-07
It 4100: Total Loss: 0.27762237191200256,        Rec Loss: 0.27761802077293396,  KL L
oss: 4.3559703044593334e-07
It 4200: Total Loss: 0.271106481552124,          Rec Loss: 0.27110281586647034,  KL L
oss: 3.6675191950052977e-07
It 4300: Total Loss: 0.2586595118045807,         Rec Loss: 0.25865620374679565,  KL L
oss: 3.309687599539757e-07
It 4400: Total Loss: 0.2688199579715729,         Rec Loss: 0.2688162922859192,   KL L
oss: 3.670429578050971e-07
It 4500: Total Loss: 0.2835359573364258,         Rec Loss: 0.28353211283683777,  KL L
oss: 3.8536381907761097e-07
It 4600: Total Loss: 0.25813570618629456,        Rec Loss: 0.25813326239585876,  KL L
oss: 2.4515611585229635e-07
beta =  10
```





Done!


Reconstruction Loss
0.7
0.6
0.5
0.4
0.3
0
1000
2000
3000
4000


KL Loss
0.008
0.006
0.004
0.002
0.000
0
1000
2000
3000
4000

> **Inline Question: What can you observe when setting $\beta = 0$ and $\beta = 10$? Explain your observations! [2pt]** (max 200 words)
>
> **Answer**:
>
> In a Variational Autoencoder (VAE), the value of the hyperparameter beta controls the trade-off between the reconstruction loss and the KL divergence loss. Specifically, the total loss function of the VAE is defined as the sum of the reconstruction loss and the KL divergence loss multiplied by beta. Therefore, setting beta to different values can affect the behavior of the VAE in terms of how it balances the importance of reconstruction accuracy and the regularization of the latent space.
>
> If we set beta to 0, the VAE becomes equivalent to a standard Autoencoder (AE), as the KL divergence loss is effectively turned off. In this case, the VAE will only optimize for reconstruction accuracy and will not constrain the latent space to follow a particular probability distribution. As a result, the generated samples will be similar to the original data but may lack diversity and novelty.
>
> On the other hand, setting beta to a higher value such as 10 will prioritize the regularization of the latent space over reconstruction accuracy. In this case, the VAE will aim to ensure that the latent space follows a prior distribution, typically a standard normal distribution. This will lead to more diverse and novel generated samples that better capture the underlying probability distribution of the data. However, it may also result in slightly lower reconstruction accuracy as the VAE sacrifices some accuracy in favor of regularizing the latent space.
>
> Overall, setting beta to different values in a VAE can have a significant impact on its behavior and the quality of generated samples. Choosing the right value for beta depends on the specific application and the desired trade-off between reconstruction accuracy and regularization of the latent space.

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**:
>
> In a Variational Autoencoder (VAE), a "good result" would typically involve both high reconstruction accuracy for the input data and high quality for the

generated samples. Specifically, we would expect the following:

> 1. High Reconstruction Accuracy: The VAE should be able to accurately reconstruct the input data with low reconstruction error. This means that the VAE should be able to capture the key features and structure of the input data and generate a reconstruction that is similar to the original data.
>
> 2. High Sample Quality: The VAE should be able to generate high-quality samples that are diverse, novel, and follow the underlying probability distribution of the data in the latent space. This means that the generated samples should be visually similar to the original data and exhibit similar statistical properties such as mean and variance.

When tuning the beta value in a VAE, we should aim to find a value that balances the importance of reconstruction accuracy and regularization of the latent space to achieve both high reconstruction accuracy and high sample quality. A good result would therefore involve finding a beta value that achieves both high reconstruction accuracy for the input data and high quality for the generated samples. We can evaluate the quality of generated samples using metrics such as visual inspection, distributional statistics, and perceptual metrics such as Inception Score or Frechet Inception Distance.

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 *0.105* [2pt]

In [215…

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

######################################## TODO ########################################
epochs = 10      # recommended 5-20 epochs
beta = 0.105 # 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 upd

# 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.                        #
####################################################################################
```

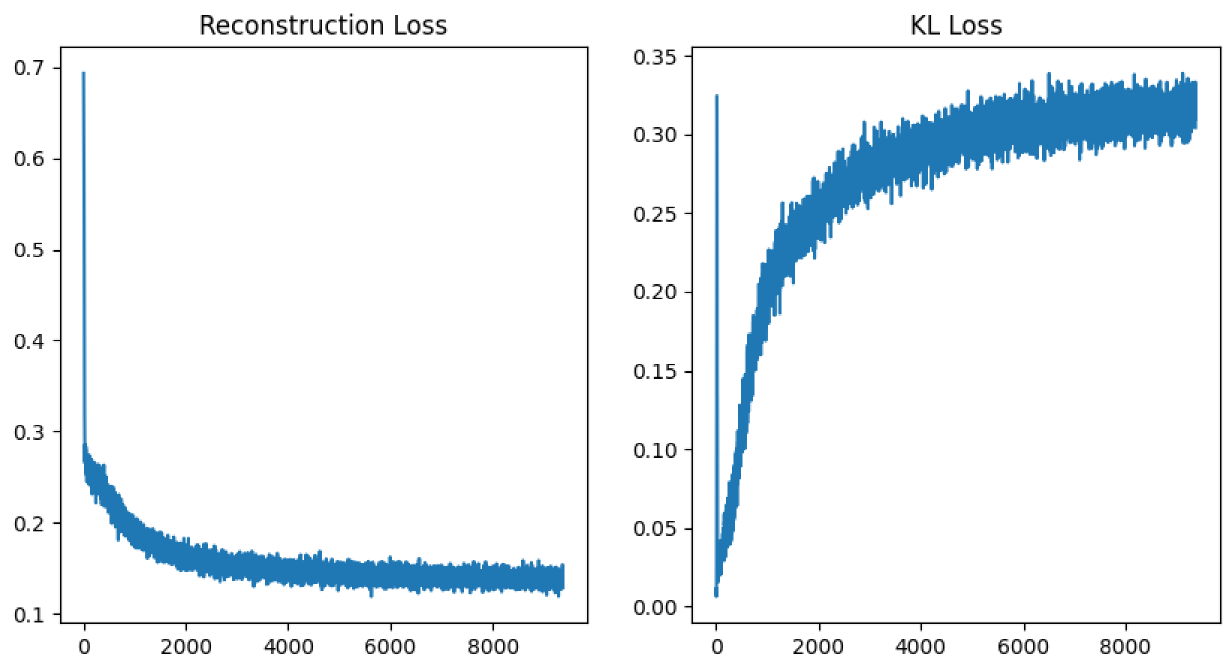
```
# same as AE
optimizer = torch.optim.Adam(vae_model.parameters(), lr=learning_rate)
#################################### END TODO ######################################

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 data, labels in mnist_data_loader:
    optimizer.zero_grad()

    x = data.reshape([batch_size,1,-1])
    outputs = vae_model.forward(x)

    total_loss, losses = vae_model.loss(x,outputs)

    total_loss.backward()
    optimizer.step()

    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=0.105
Run Epoch 0
It 0: Total Loss: 0.6947729587554932,    Rec Loss: 0.6935595273971558,   KL Loss: 0.0
11556386947631836
It 100: Total Loss: 0.26539313793182373,        Rec Loss: 0.26232996582984924,  KL L
oss: 0.02917313575744629
It 200: Total Loss: 0.2624681293964386,         Rec Loss: 0.2580614984035492,   KL L
oss: 0.04196779429912567
It 300: Total Loss: 0.24857638776302338,        Rec Loss: 0.24259699881076813,  KL L
oss: 0.056946516036987305
It 400: Total Loss: 0.25790169835090637,        Rec Loss: 0.2497626692056656,   KL L
oss: 0.07751446962356567
It 500: Total Loss: 0.24484653770923615,        Rec Loss: 0.23335643112659454,  KL L
oss: 0.10942957550287247
It 600: Total Loss: 0.2336249202489853,         Rec Loss: 0.21857701241970062,  KL L
oss: 0.1433134227991104
It 700: Total Loss: 0.2312140166759491,         Rec Loss: 0.2140575349330902,   KL L
oss: 0.1633950024843216
It 800: Total Loss: 0.22285595536231995,        Rec Loss: 0.20307643711566925,  KL L
oss: 0.1883763074874878
It 900: Total Loss: 0.2074926495552063,         Rec Loss: 0.18825627863407135,  KL L
oss: 0.18320345878601074
Run Epoch 1
It 1000: Total Loss: 0.2182980179786682,        Rec Loss: 0.19827449321746826,  KL L
oss: 0.19070029258728027
It 1100: Total Loss: 0.20528140664100647,       Rec Loss: 0.18377067148685455,  KL L
oss: 0.20486418902873993
It 1200: Total Loss: 0.20563651621341705,       Rec Loss: 0.1821991503238678,   KL L
oss: 0.22321301698684692
It 1300: Total Loss: 0.20531684160232544,       Rec Loss: 0.18147334456443787,  KL L
oss: 0.22708086669445038
It 1400: Total Loss: 0.21465875208377838,       Rec Loss: 0.189167320728302,    KL L
oss: 0.24277551472187042
It 1500: Total Loss: 0.20047657191753387,       Rec Loss: 0.17568503320217133,  KL L
oss: 0.23610985279083252
It 1600: Total Loss: 0.19690248370170593,       Rec Loss: 0.17084628343582153,  KL L
oss: 0.24815428256988525
It 1700: Total Loss: 0.2046859860420227,        Rec Loss: 0.17975609004497528,  KL L
oss: 0.2374275177717209
It 1800: Total Loss: 0.1886475384235382,        Rec Loss: 0.16228057444095612,  KL L
oss: 0.2511139214038849
Run Epoch 2
It 1900: Total Loss: 0.18695074319839478,       Rec Loss: 0.16275297105312347,  KL L
oss: 0.2304549515247345
It 2000: Total Loss: 0.19091655313968658,       Rec Loss: 0.16453781723976135,  KL L
oss: 0.2512260973453522
It 2100: Total Loss: 0.18897069990634918,       Rec Loss: 0.16350415349006653,  KL L
oss: 0.24253857135772705
It 2200: Total Loss: 0.19438309967517853,       Rec Loss: 0.16831892728805542,  KL L
oss: 0.2482302039861679
It 2300: Total Loss: 0.1735401302576065,        Rec Loss: 0.14652349054813385,  KL L
oss: 0.25730136036872864
It 2400: Total Loss: 0.18948282301425934,       Rec Loss: 0.16173920035362244,  KL L
oss: 0.264225035905838
It 2500: Total Loss: 0.1867908388376236,        Rec Loss: 0.16010749340057373,  KL L
oss: 0.2541271150112152
It 2600: Total Loss: 0.1845022737979889,        Rec Loss: 0.15566572546958923,  KL L
oss: 0.2746337950229645
It 2700: Total Loss: 0.19447340071201324,       Rec Loss: 0.16438250243663788,  KL L
oss: 0.28657999634742737
```

```
It 2800: Total Loss: 0.18700000643730164,        Rec Loss: 0.1572657972574234,   KL L
oss: 0.2831829786300659
Run Epoch 3
It 2900: Total Loss: 0.1929275244474411,         Rec Loss: 0.1626623123884201,   KL L
oss: 0.28824010491371155
It 3000: Total Loss: 0.1889939159154892,         Rec Loss: 0.15945708751678467,  KL L
oss: 0.28130319714546204
It 3100: Total Loss: 0.19310711324214935,        Rec Loss: 0.16363754868507385,  KL L
oss: 0.280662477016449
It 3200: Total Loss: 0.17821112275123596,        Rec Loss: 0.14858406782150269,  KL L
oss: 0.28216245770454407
It 3300: Total Loss: 0.17307689785957336,        Rec Loss: 0.1453087329864502,   KL L
oss: 0.26445868611335754
It 3400: Total Loss: 0.1795249730348587,         Rec Loss: 0.1494489163160324,   KL L
oss: 0.2864386737346649
It 3500: Total Loss: 0.1810847520828247,         Rec Loss: 0.1508813053369522,   KL L
oss: 0.28765183687210083
It 3600: Total Loss: 0.1821560114622116,         Rec Loss: 0.15164567530155182,  KL L
oss: 0.29057469964027405
It 3700: Total Loss: 0.18574221432209015,        Rec Loss: 0.15543296933174133,  KL L
oss: 0.2886595129966736
Run Epoch 4
It 3800: Total Loss: 0.17822830379009247,        Rec Loss: 0.14794720709323883,  KL L
oss: 0.2883913815021515
It 3900: Total Loss: 0.17182351648807526,        Rec Loss: 0.1413903534412384,   KL L
oss: 0.2898397147655487
It 4000: Total Loss: 0.18732376396656036,        Rec Loss: 0.1554793268442154,   KL L
oss: 0.3032803237438202
It 4100: Total Loss: 0.1830521523952484,         Rec Loss: 0.15146200358867645,  KL L
oss: 0.30085858702659607
It 4200: Total Loss: 0.1773264855146408,         Rec Loss: 0.14613354206085205,  KL L
oss: 0.29707562923431396
It 4300: Total Loss: 0.1790957748889923,         Rec Loss: 0.1480710208415985,   KL L
oss: 0.2954738140106201
It 4400: Total Loss: 0.17582957446575165,        Rec Loss: 0.14366847276687622,  KL L
oss: 0.3062962293624878
It 4500: Total Loss: 0.18983609974384308,        Rec Loss: 0.15937910974025726,  KL L
oss: 0.29006654024124146
It 4600: Total Loss: 0.17447131872177124,        Rec Loss: 0.14371457695960999,  KL L
oss: 0.29292136430740356
Run Epoch 5
It 4700: Total Loss: 0.17911115288734436,        Rec Loss: 0.14662782847881317,  KL L
oss: 0.30936500430107117
It 4800: Total Loss: 0.17851482331752777,        Rec Loss: 0.1450544148683548,   KL L
oss: 0.3186706006526947
It 4900: Total Loss: 0.1761751025915146,         Rec Loss: 0.1444154679775238,   KL L
oss: 0.3024727404117584
It 5000: Total Loss: 0.1762675791978836,         Rec Loss: 0.14405550062656403,  KL L
oss: 0.30678173899650574
It 5100: Total Loss: 0.16433344781398773,        Rec Loss: 0.1344289630651474,   KL L
oss: 0.28480467200279236
It 5200: Total Loss: 0.17125950753688812,        Rec Loss: 0.1397213488817215,   KL L
oss: 0.3003634512424469
It 5300: Total Loss: 0.17964240908622742,        Rec Loss: 0.1484673023223877,   KL L
oss: 0.2969057261943817
It 5400: Total Loss: 0.17154061794281006,        Rec Loss: 0.1401151567697525,   KL L
oss: 0.29929018020629883
It 5500: Total Loss: 0.1648334562778473,         Rec Loss: 0.13295023143291473,  KL L
oss: 0.3036497235298157
It 5600: Total Loss: 0.1798655241727829,         Rec Loss: 0.14902101457118988,  KL L
```

```
oss: 0.29375728964805603
Run Epoch 6
It 5700: Total Loss: 0.17178088426589966,        Rec Loss: 0.14167046546936035,  KL L
oss: 0.2867659032344818
It 5800: Total Loss: 0.17004644870758057,        Rec Loss: 0.13765232264995575,  KL L
oss: 0.3085155189037323
It 5900: Total Loss: 0.16069133579730988,        Rec Loss: 0.13044193387031555,  KL L
oss: 0.2880895137786865
It 6000: Total Loss: 0.17665761709213257,        Rec Loss: 0.14594340324401855,  KL L
oss: 0.2925163209438324
It 6100: Total Loss: 0.18519560992717743,        Rec Loss: 0.1529051661491394,   KL L
oss: 0.307528018951416
It 6200: Total Loss: 0.18425869941711426,        Rec Loss: 0.1524551957845688,   KL L
oss: 0.30289044976234436
It 6300: Total Loss: 0.19036740064620972,        Rec Loss: 0.15690098702907562,  KL L
oss: 0.3187278211116791
It 6400: Total Loss: 0.15654109418392181,        Rec Loss: 0.12482138723134995,  KL L
oss: 0.30209243297576904
It 6500: Total Loss: 0.1796177476644516,         Rec Loss: 0.14658954739570618,  KL L
oss: 0.31455427408218384
Run Epoch 7
It 6600: Total Loss: 0.1788819134235382,         Rec Loss: 0.14575131237506866,  KL L
oss: 0.3155295252799988
It 6700: Total Loss: 0.16820290684700012,        Rec Loss: 0.1368684321641922,   KL L
oss: 0.2984236478805542
It 6800: Total Loss: 0.16998827457427979,        Rec Loss: 0.1378902792930603,   KL L
oss: 0.30569523572921753
It 6900: Total Loss: 0.1737610399723053,         Rec Loss: 0.14152278006076813,  KL L
oss: 0.3070310354232788
It 7000: Total Loss: 0.1828971952199936,         Rec Loss: 0.1490914523601532,   KL L
oss: 0.3219594359397888
It 7100: Total Loss: 0.17596597969532013,        Rec Loss: 0.14479300379753113,  KL L
oss: 0.29688552021980286
It 7200: Total Loss: 0.1709160953760147,         Rec Loss: 0.13878381252288818,  KL L
oss: 0.3060217499732971
It 7300: Total Loss: 0.17114217579364777,        Rec Loss: 0.13798516988754272,  KL L
oss: 0.3157810568809509
It 7400: Total Loss: 0.1689058393239975,         Rec Loss: 0.1359989494085312,   KL L
oss: 0.31339898705482483
Run Epoch 8
It 7500: Total Loss: 0.17262078821659088,        Rec Loss: 0.14075133204460144,  KL L
oss: 0.3035186529159546
It 7600: Total Loss: 0.16992604732513428,        Rec Loss: 0.13802731037139893,  KL L
oss: 0.30379748344421387
It 7700: Total Loss: 0.1639455109834671,         Rec Loss: 0.1320890188217163,   KL L
oss: 0.30339518189430237
It 7800: Total Loss: 0.16765017807483673,        Rec Loss: 0.13491809368133545,  KL L
oss: 0.3117341697216034
It 7900: Total Loss: 0.17559337615966797,        Rec Loss: 0.14230957627296448,  KL L
oss: 0.31698858737945557
It 8000: Total Loss: 0.17672297358512878,        Rec Loss: 0.14335516095161438,  KL L
oss: 0.3177887797355652
It 8100: Total Loss: 0.16876664757728577,        Rec Loss: 0.13569602370262146,  KL L
oss: 0.314958393573761
It 8200: Total Loss: 0.16149374842643738,        Rec Loss: 0.1299077719449997,   KL L
oss: 0.3008188009262085
It 8300: Total Loss: 0.16358311474323273,        Rec Loss: 0.13051775097846985,  KL L
oss: 0.3149082660675049
It 8400: Total Loss: 0.1803998202085495,         Rec Loss: 0.1465911716222763,   KL L
oss: 0.321987122297287
```

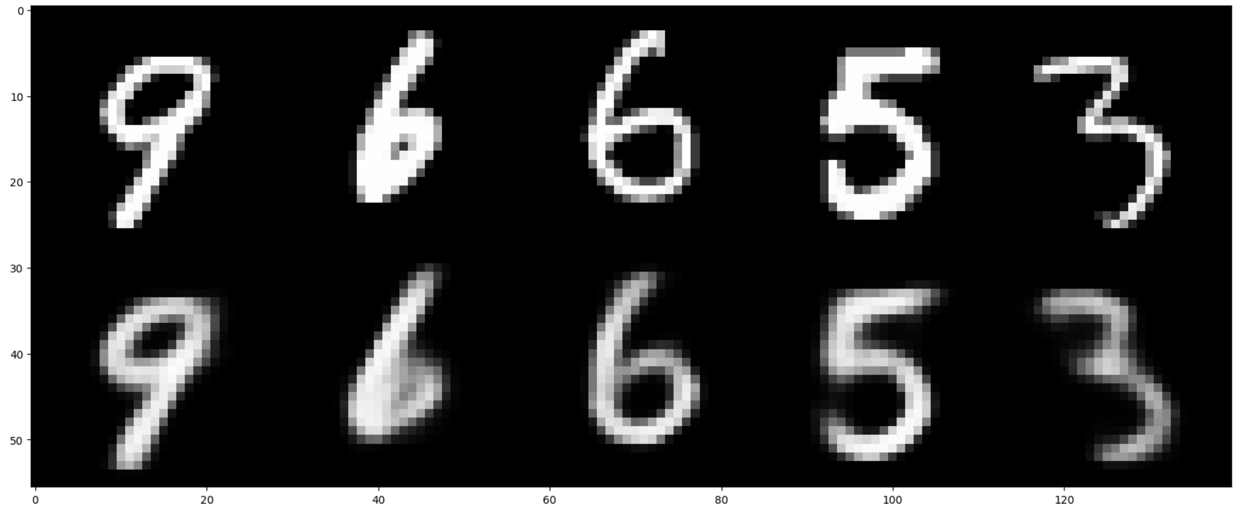
```
Run Epoch 9
It 8500: Total Loss: 0.170094296336174,         Rec Loss: 0.13828183710575104,  KL L
oss: 0.3029758334159851
It 8600: Total Loss: 0.16914066672325134,        Rec Loss: 0.13656945526599884,  KL L
oss: 0.3102019727230072
It 8700: Total Loss: 0.17067426443099976,        Rec Loss: 0.13730870187282562,  KL L
oss: 0.31776732206344604
It 8800: Total Loss: 0.16982831060886383,        Rec Loss: 0.13728845119476318,  KL L
oss: 0.30990347266197205
It 8900: Total Loss: 0.15619505941867828,        Rec Loss: 0.1254684031009674,   KL L
oss: 0.29263484477996826
It 9000: Total Loss: 0.17090395092964172,        Rec Loss: 0.13762030005455017,  KL L
oss: 0.31698718667030334
It 9100: Total Loss: 0.16210998594760895,        Rec Loss: 0.1297767460346222,   KL L
oss: 0.30793559551239014
It 9200: Total Loss: 0.1689661145210266,         Rec Loss: 0.13540203869342804,  KL L
oss: 0.3196578621864319
It 9300: Total Loss: 0.167435884475708,          Rec Loss: 0.13366548717021942,  KL L
oss: 0.3216228783130646
Done!
```

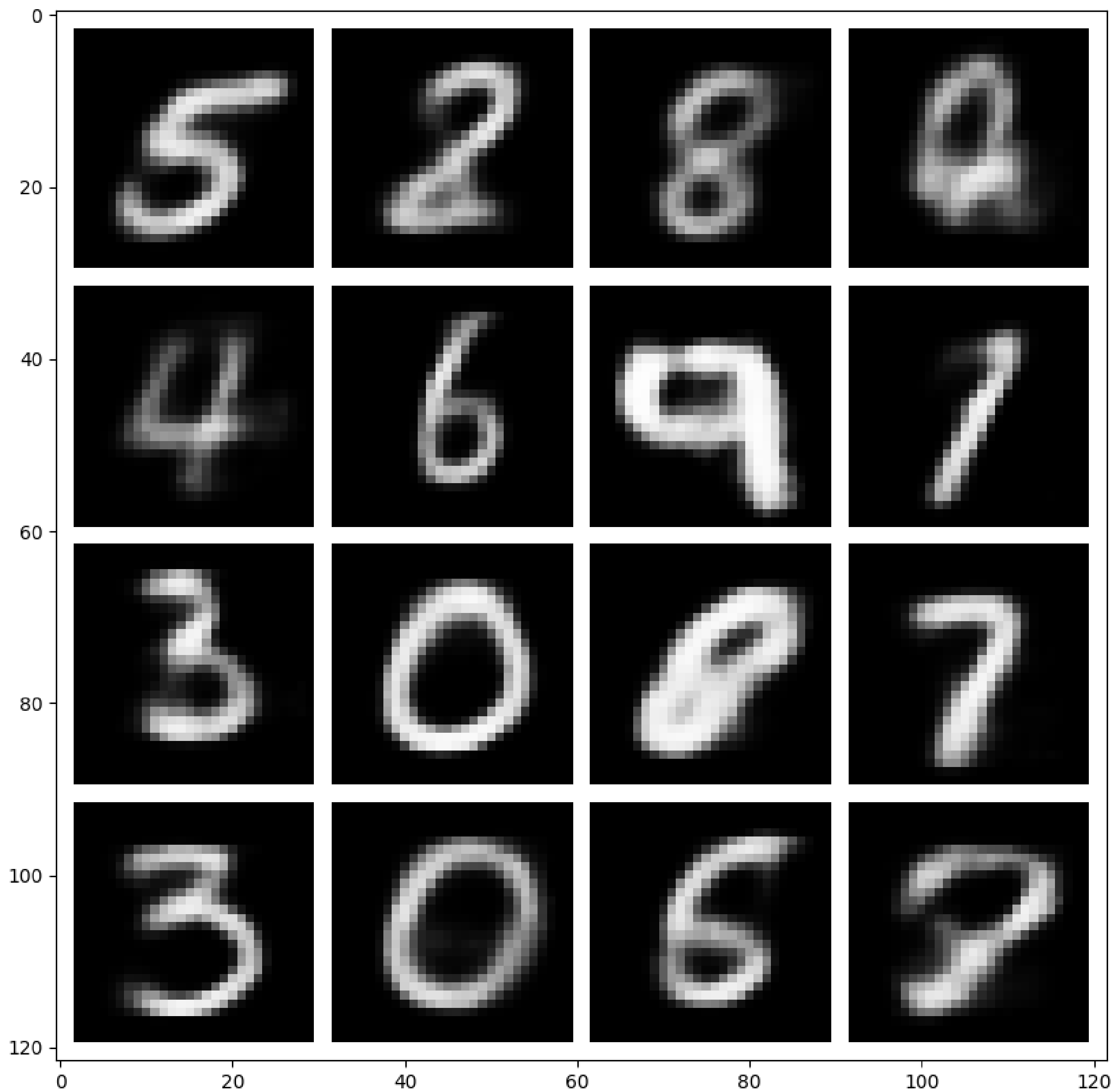


In [222…

```
# visualize VAE reconstructions and sampling for VAE for optimal Beta
print("beta = ", beta)
vis_reconstruction(vae_model, randomize=True)
vis_samples(vae_model)
```

```
beta =  0.105
```





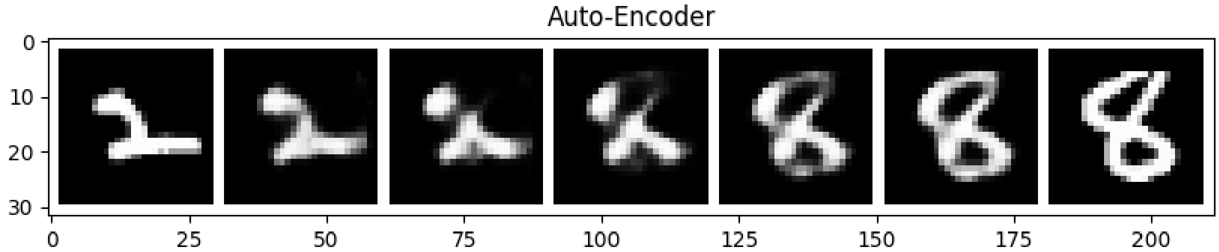
4. Embedding Space Interpolation [3pt]

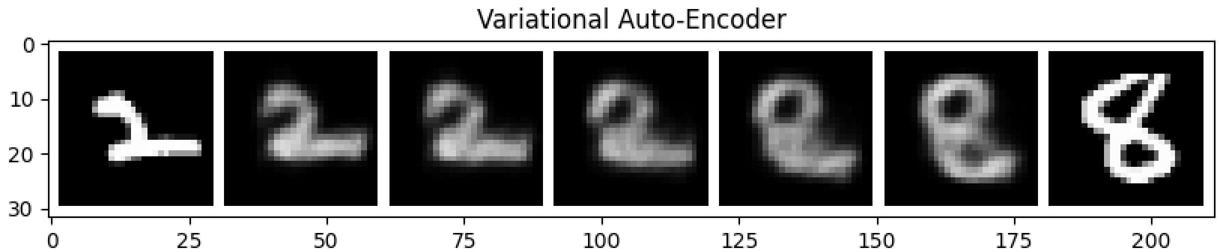
As mentioned in the introduction, AEs and VAEs cannot only be used to generate images, but also to learn low-dimensional representations of their inputs. In this final section we will investigate the representations we learned with both models by **interpolating in embedding space** between different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

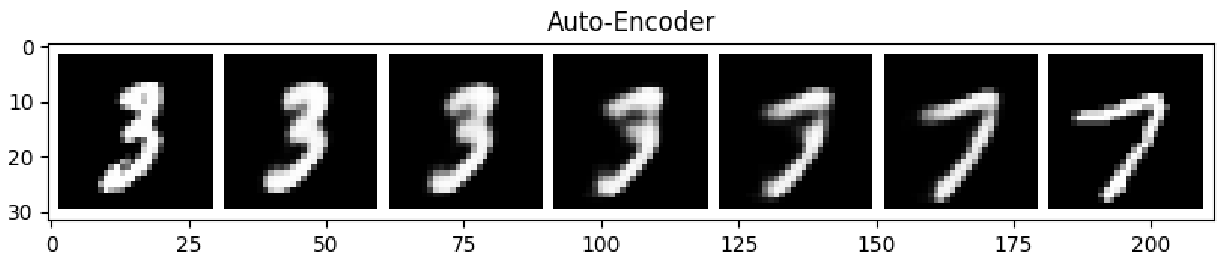
In [175…

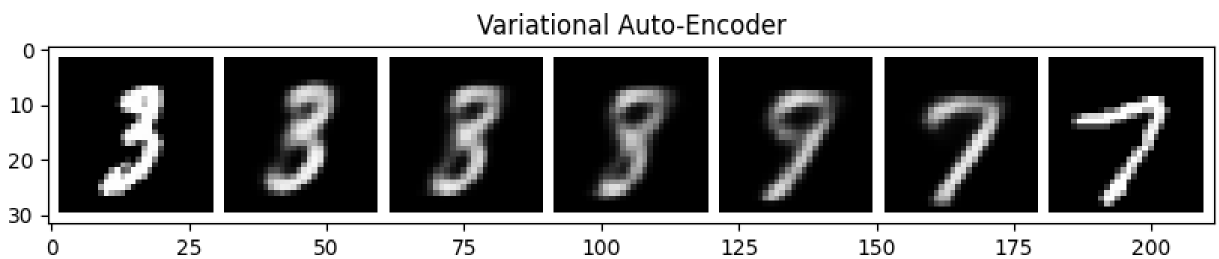
```
# Prob1-7
nz=32

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_STEF

  # 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 = 2# ... TODO CHOOSE
END_LABEL = 8# ... 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 = 3# ... TODO CHOOSE
END_LABEL = 7# ... 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**:
>
> The embeddings learned by Autoencoder (AE) and Variational Autoencoder (VAE) differ in the way they represent the input data.
>
> In an AE, the encoder maps the input data to a low-dimensional latent space representation, which is then fed into the decoder to reconstruct the input data.

The latent space is learned through a deterministic mapping, which means that the same input data will always result in the same embedding. When interpolating between two embeddings in an AE, we can simply linearly interpolate between the two embeddings and feed the resulting embeddings into the decoder to generate intermediate reconstructions. However, since the latent space in an AE is not constrained to follow any particular probability distribution, the interpolations may not follow any particular pattern or have any meaningful semantic relationships between them.

On the other hand, in a VAE, the encoder maps the input data to a distribution in the latent space rather than a deterministic embedding. Specifically, the encoder outputs the mean and standard deviation of a multivariate Gaussian distribution in the latent space. The embedding is then sampled from this distribution using the reparameterization trick. This sampling process ensures that the latent space follows a specific probability distribution, typically a standard normal distribution. As a result, the embeddings learned by a VAE have a meaningful structure and can capture semantic relationships between different inputs.

When interpolating between two embeddings in a VAE, we cannot simply linearly interpolate between the two embeddings as this may not follow the underlying probability distribution. Instead, we typically sample intermediate embeddings from the latent space distribution between the mean embeddings of the two input images. These intermediate embeddings can then be fed into the decoder to generate intermediate reconstructions that follow the underlying probability distribution. This allows us to generate meaningful interpolations between different images, such as smoothly transitioning from one facial expression to another. Overall, the embedding space interpolations in a VAE are more meaningful and follow a structured latent space distribution compared to an AE.

1. How do you expect these differences to affect the usefulness of the learned representation for downstream learning? (max 300 words)

Answer:

The differences in embedding space interpolations between an Autoencoder (AE) and a Variational Autoencoder (VAE) can have an impact on the usefulness of the learned representation for downstream learning tasks.

Since the embeddings learned by a VAE follow a structured latent space distribution, they are more likely to capture meaningful and semantically relevant information about the input data. This can make them more useful for downstream learning tasks such as classification or clustering. Additionally, since the VAE embeddings are learned through a probabilistic framework, they are more robust to variations and noise in the input data, which can be beneficial for tasks such as anomaly detection or data denoising.

> In contrast, the embeddings learned by an AE are not constrained to follow any particular probability distribution, and the interpolations between embeddings may not have any meaningful semantic relationships. This can limit the usefulness of the learned representation for downstream learning tasks that require semantically meaningful embeddings. However, AEs can still be useful for certain downstream learning tasks such as feature extraction or dimensionality reduction.
>
> Overall, the usefulness of the learned representation for downstream learning tasks will depend on the specific requirements of the task, and both AE and VAE can have their own strengths and weaknesses depending on the context.

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`.

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.

In [85]:
```python
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.      #
    ################################################################################
    hidden_dim1 = 256
    hidden_dim2 = 64
    self.net = nn.Sequential(
          nn.Linear(self.input_size, hidden_dim1),
          nn.ReLU(),
          nn.Linear(hidden_dim1, hidden_dim2),
          nn.ReLU(),
          nn.Linear(hidden_dim2, 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.                               #
    ################################################################################
    c_onehot = idx2onehot(c, self.num_labels) #One hot of classes
    x_flat = torch.flatten(x, start_dim=1, end_dim=-1)  #Flattened image : (N, H*W)
    x_cond = torch.cat((x_flat, c_onehot), dim=1) #concatenated image + classes : (N,
    x = x_cond if self.conditional else x
    ################################################################################
    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                                                         #
    ################################################################################
    hidden_dim1 = 256
    hidden_dim2 = 64
    self.net = nn.Sequential(
          nn.Linear(nz, hidden_dim2),
          nn.ReLU(),
          nn.Linear(hidden_dim2, hidden_dim1),
          nn.ReLU(),
          nn.Linear(hidden_dim1,self.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.                    #
        ################################################################################
        c_onehot = idx2onehot(c, self.num_labels)
        z_cond = torch.cat([z, c_onehot], dim=1)
        z = z_cond if self.conditional else z
        ################################ 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,
        self.decoder = CVAEDecoder(nz, output_size=out_size, conditional=conditional,

    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 Gauss
        #             parametrized by mean and log_sigma
        # HINT: Make sure to compute the KL divergence in the correct order since it i
        #       not symmetric!!  ie. KL(p, q) != KL(q, p)
        ################################################################################
        (mu_q, logsigma_q) , reconstruction = torch.chunk(outputs['q'],2,dim=1), outpu
        mu_p, logsigma_p = torch.zeros(mu_q.shape), torch.zeros(logsigma_q.shape)

        rec_loss = nn.functional.binary_cross_entropy(reconstruction, x)

        kl_loss= torch.mean(kl_divergence(mu_q, logsigma_q, mu_p, logsigma_p ))

        ################################## 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 t
        # 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 ob
        #       the reconstruction.
        ################################################################################
        recon_img = self.forward(x)['rec']
        image = recon_img.reshape(-1, 28, 28)
        #################################### END TODO ##################################
        return image
```

Setting up the CVAE Training loop

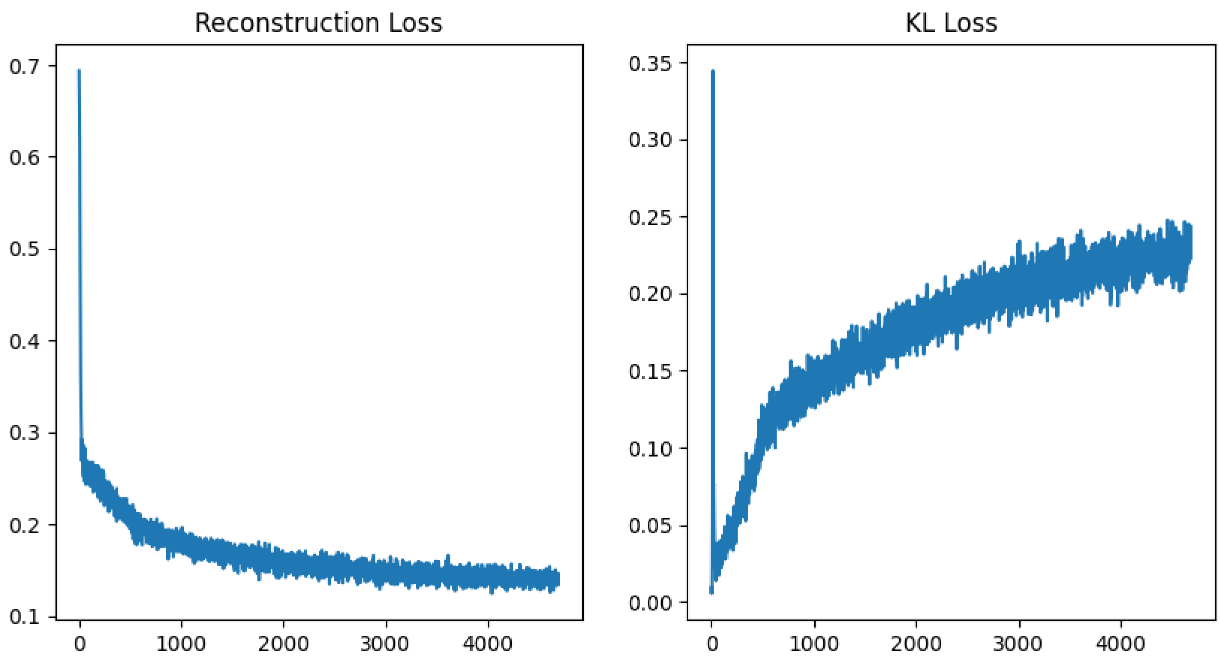
In [83]:

```
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 = 0.105
#################################### END TODO ######################################

# build CVAE model
conditional = True
cvae_model = CVAE(nz, beta, conditional=conditional, num_labels=10).to(device)    # tr
cvae_model = cvae_model.train()   # set model in train mode (eg batchnorm params get u

# 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 x, labels in mnist_data_loader:
    optimizer.zero_grad()

    x = x.reshape([batch_size,1,-1])
    outputs = cvae_model.forward(x, labels)

    total_loss, losses = cvae_model.loss(x,outputs)
    total_loss.backward()

    optimizer.step()

    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 5 epochs with beta=0.105
Run Epoch 0
It 0: Total Loss: 0.6948158740997314,    Rec Loss: 0.6938651204109192,   KL Loss: 0.0
09054571390151978
It 100: Total Loss: 0.2649941146373749,          Rec Loss: 0.2622748911380768,   KL L
oss: 0.025897307321429253
It 200: Total Loss: 0.25664496421813965,         Rec Loss: 0.2528477907180786,   KL L
oss: 0.036163508892059326
It 300: Total Loss: 0.23512998223304749,         Rec Loss: 0.22830787301063538,  KL L
oss: 0.06497249007225037
It 400: Total Loss: 0.2315811961889267,          Rec Loss: 0.22191806137561798,  KL L
oss: 0.0920298844575882
It 500: Total Loss: 0.2094084769487381,          Rec Loss: 0.19923977553844452,  KL L
oss: 0.09684472531080246
It 600: Total Loss: 0.2105351835489273,          Rec Loss: 0.19707940518856049,  KL L
oss: 0.12815020978450775
It 700: Total Loss: 0.20036396384239197,         Rec Loss: 0.18714174628257751,  KL L
oss: 0.12592582404613495
It 800: Total Loss: 0.20167873799800873,         Rec Loss: 0.18932119011878967,  KL L
oss: 0.11769090592861176
It 900: Total Loss: 0.1984962373971939,          Rec Loss: 0.18537688255310059,  KL L
oss: 0.1249462142586708
Run Epoch 1
It 1000: Total Loss: 0.20235675573349,   Rec Loss: 0.18686234951019287,  KL Loss: 0.1
4756575226783752
It 1100: Total Loss: 0.1933654099702835,         Rec Loss: 0.17773737013339996,  KL L
oss: 0.14883846044540405
It 1200: Total Loss: 0.18565481901168823,        Rec Loss: 0.17033880949020386,  KL L
oss: 0.1458667516708374
It 1300: Total Loss: 0.18042656779289246,        Rec Loss: 0.16515275835990906,  KL L
oss: 0.14546482264995575
It 1400: Total Loss: 0.19916559755802155,        Rec Loss: 0.1820301115512848,   KL L
oss: 0.16319505870342255
It 1500: Total Loss: 0.18197162449359894,        Rec Loss: 0.16378390789031982,  KL L
oss: 0.17321640253067017
It 1600: Total Loss: 0.18239036202430725,        Rec Loss: 0.16406555473804474,  KL L
oss: 0.1745220124721527
It 1700: Total Loss: 0.181539386510849,          Rec Loss: 0.16288615763187408,  KL L
oss: 0.1776498556137085
It 1800: Total Loss: 0.17550309002399445,        Rec Loss: 0.1573994904756546,   KL L
oss: 0.17241525650024414
Run Epoch 2
It 1900: Total Loss: 0.18644703924655914,        Rec Loss: 0.16810530424118042,  KL L
oss: 0.17468318343162537
It 2000: Total Loss: 0.17760151624679565,        Rec Loss: 0.15929709374904633,  KL L
oss: 0.17432783544063568
It 2100: Total Loss: 0.18703186511993408,        Rec Loss: 0.1674419641494751,   KL L
oss: 0.1865704357624054
It 2200: Total Loss: 0.17535345256328583,        Rec Loss: 0.15562903881072998,  KL L
oss: 0.18785160779953003
It 2300: Total Loss: 0.18024872243404388,        Rec Loss: 0.16058674454689026,  KL L
oss: 0.18725688755512238
It 2400: Total Loss: 0.17952921986579895,        Rec Loss: 0.15881194174289703,  KL L
oss: 0.19730746746063232
It 2500: Total Loss: 0.18631364405155182,        Rec Loss: 0.16438062489032745,  KL L
oss: 0.20888587832450867
It 2600: Total Loss: 0.1749994456768036,         Rec Loss: 0.15318405628204346,  KL L
oss: 0.20776566863059998
It 2700: Total Loss: 0.17646947503089905,        Rec Loss: 0.15570785105228424,  KL L
oss: 0.1977297067642212
```

```
It 2800: Total Loss: 0.17723329365253448,        Rec Loss: 0.15610693395137787,  KL L
oss: 0.20120345056056976
Run Epoch 3
It 2900: Total Loss: 0.17058566212654114,        Rec Loss: 0.14935904741287231,  KL L
oss: 0.20215818285942078
It 3000: Total Loss: 0.1624581664800644,         Rec Loss: 0.14074444770812988,  KL L
oss: 0.2067972719669342
It 3100: Total Loss: 0.18063484132289886,        Rec Loss: 0.15774229168891907,  KL L
oss: 0.21802429854869843
It 3200: Total Loss: 0.1778496503829956,         Rec Loss: 0.1546842604875564 6,  KL L
oss: 0.220622718334198
It 3300: Total Loss: 0.1647314727306366,         Rec Loss: 0.1436770111322403,   KL L
oss: 0.20051871240139008
It 3400: Total Loss: 0.16798365116119385,        Rec Loss: 0.14573705196380615,  KL L
oss: 0.2118724286556244
It 3500: Total Loss: 0.16919830441474915,        Rec Loss: 0.14710643887519836,  KL L
oss: 0.21039873361587524
It 3600: Total Loss: 0.1750793159008026,         Rec Loss: 0.15192589163780212,  KL L
oss: 0.22050876915454865
It 3700: Total Loss: 0.16740354895591736,        Rec Loss: 0.14354413747787476,  KL L
oss: 0.22723254561424255
Run Epoch 4
It 3800: Total Loss: 0.17040421068668365,        Rec Loss: 0.14640668034553528,  KL L
oss: 0.22854793071746826
It 3900: Total Loss: 0.16430538892745972,        Rec Loss: 0.14322790503501892,  KL L
oss: 0.20073802769184113
It 4000: Total Loss: 0.1689513921737671,         Rec Loss: 0.14477583765983582,  KL L
oss: 0.23024335503578186
It 4100: Total Loss: 0.168458491563797,          Rec Loss: 0.14616873860359192,  KL L
oss: 0.2122834175825119
It 4200: Total Loss: 0.16507993638515472,        Rec Loss: 0.142886221408844,    KL L
oss: 0.21136876940727234
It 4300: Total Loss: 0.16078682243824005,        Rec Loss: 0.13760749995708466,  KL L
oss: 0.22075539827346802
It 4400: Total Loss: 0.17576828598976135,        Rec Loss: 0.15245427191257477,  KL L
oss: 0.22203825414180756
It 4500: Total Loss: 0.1637858748435974,         Rec Loss: 0.14143535494804382,  KL L
oss: 0.21286214888095856
It 4600: Total Loss: 0.16477108001708984,        Rec Loss: 0.1406160295009613,   KL L
oss: 0.2300480604171753
Done!
```

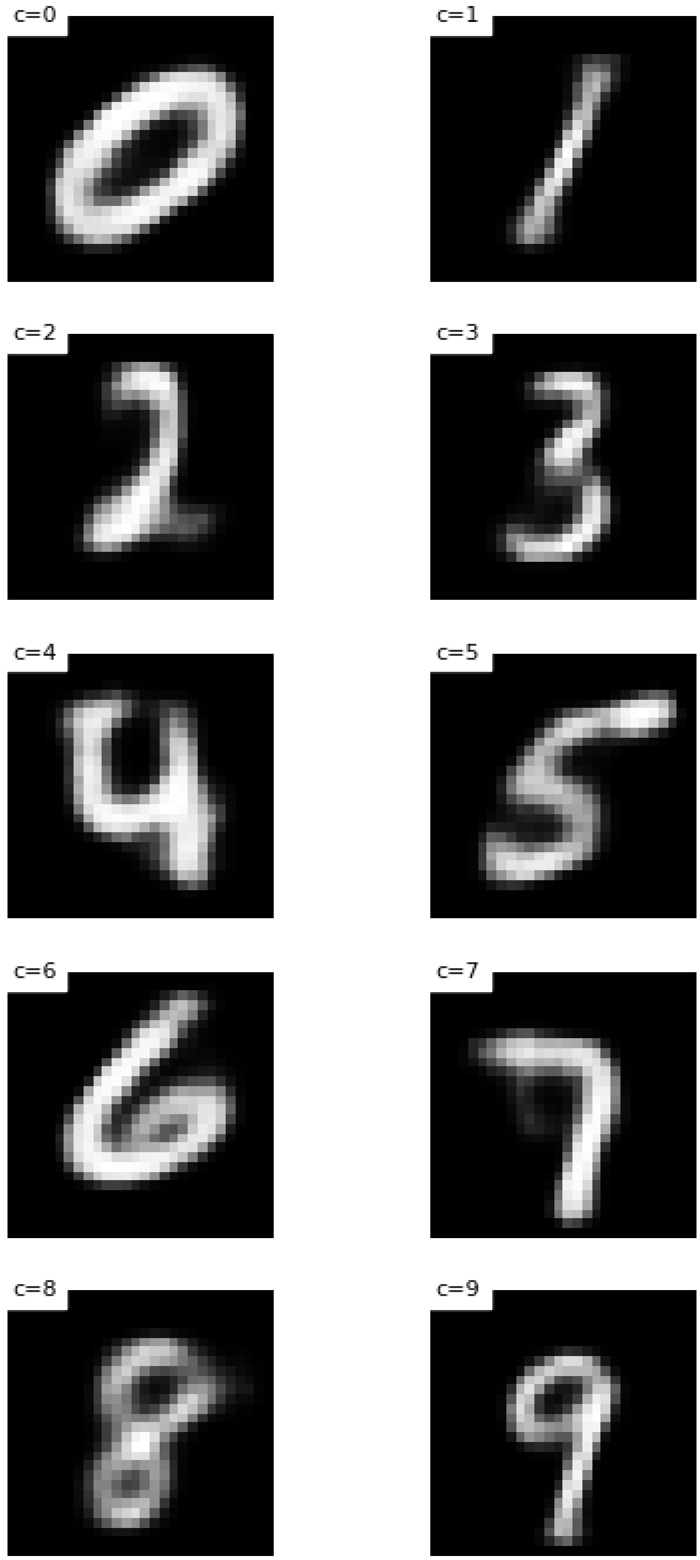


Verifying conditional samples from CVAE [6 pt]

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

In [22]:
```
# 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>
```



Submission Instructions

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