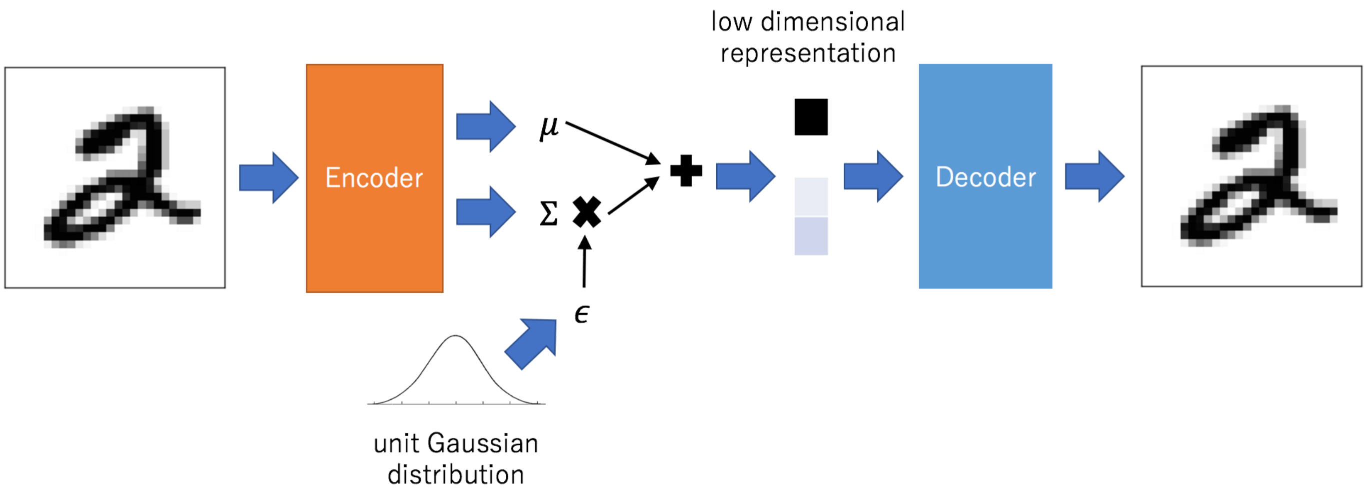
Problem 1 - Variational Auto-Encoder (VAE)

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.

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

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

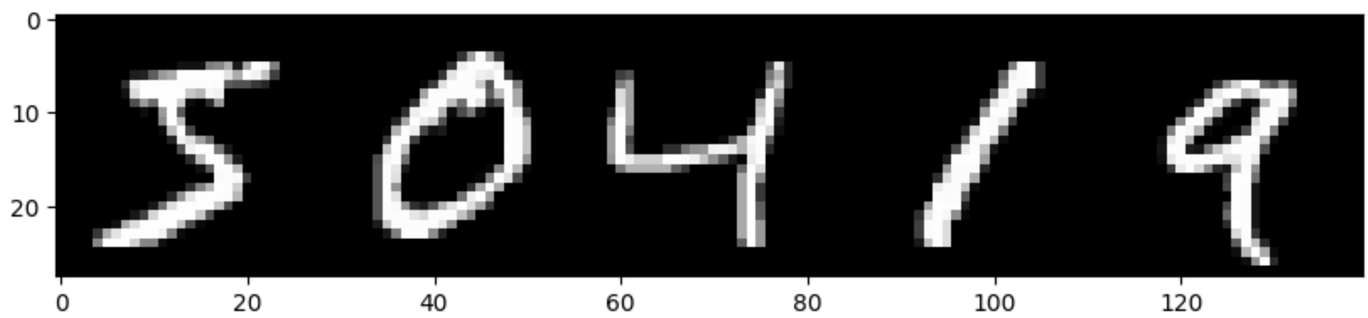
```
import torch
import torchvision
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')   # use GPU if
available
print(f"Using device: {device}")

# 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: cuda:0

 Download complete! Downloaded 60000 training examples!
```

```
/home/jingmin/miniconda3/envs/adapter/lib/python3.8/site-
packages/torchvision/datasets/mnist.py:498: UserWarning: The given NumPy array is not
writeable, and PyTorch does not support non-writeable tensors. This means you can write
to the underlying (supposedly non-writeable) NumPy array using the tensor. You may want
to copy the array to protect its data or make it writeable before converting it to a
tensor. This type of warning will be suppressed for the rest of this program.
(Triggered internally at  /pytorch/torch/csrc/utils/tensor_numpy.cpp:180.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

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

# Let's display some of the training samples.
sample_images = []
randomize = False # set to False for debugging
num_samples = 5 # simple data sampling for now, later we will use proper DataLoader
```

```
if randomize:
    sample_idxs = np.random.randint(low=0,high=len(mnist_train), size=num_samples)
else:
    sample_idxs = list(range(num_samples))

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

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



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]

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

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


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

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

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

Testing the Auto-Encoder Forward Pass

```
# To test your encoder/decoder, let's encode/decode some sample images
# first, make a PyTorch DataLoader object to sample data batches
batch_size = 64
nworkers = 2        # number of workers used for efficient data loading

####################################################################################
# Create a PyTorch DataLoader object for efficiently generating training batches.  #
# Make sure that the data loader automatically shuffles the training dataset.      #
# Consider only *full* batches of data, to avoid torch errrors.              #
# The DataLoader wraps the MNIST dataset class we created earlier.           #
#       Use the given batch_size and number of data loading workers when creating  #
#       the DataLoader. https://pytorch.org/docs/stable/data.html                  #
####################################################################################
mnist_data_loader = torch.utils.data.DataLoader(mnist_train,
                                                batch_size=batch_size,
                                                shuffle=True,
                                                num_workers=nworkers,
                                                drop_last=True)
####################################################################################

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

```
    print(f"Shape of encoding vector (should be [batch_size, nz]): {enc.shape}")
    dec = decoder(enc)
    print("Shape of decoded image (should be [batch_size, 1, out_size]):
{}.".format(dec.shape))
    break

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

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

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

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

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

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

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

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

```
# Prob1-2
epochs = 10
learning_rate = 1e-3

# build AE model
print(f'Device available {device}')
ae_model = AutoEncoder(nz).to(device)    # transfer model to GPU if available
ae_model = ae_model.train()   # set model in train mode (eg batchnorm params get
updated)
```

```
# build optimizer and loss function
####################################### TODO #######################################
# Build the optimizer and loss classes. For the loss you can use a loss layer      #
# from the torch.nn package. We recommend binary cross entropy.                    #
# HINT: We will use the Adam optimizer (learning rate given above, otherwise       #
#       default parameters).                                                       #
# NOTE: We could also use alternative losses like MSE and cross entropy, depending #
#       on the assumptions we are making about the output distribution.            #
####################################################################################
optimizer = torch.optim.Adam(ae_model.parameters(), learning_rate)
loss_func = nn.BCELoss(reduction='mean')
# loss_func = nn.MSELoss()
#################################### END TODO #######################################

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

 ####################################################################################
    for sample_img, _ in mnist_data_loader:  # _ refers to y_label
        sample_img = sample_img.to(device)
        sample_img = sample_img.reshape([batch_size, in_size])
        optimizer.zero_grad()
        # falten X_train: [64, 28, 28] -> [64, 784]
        y_pred = ae_model(sample_img)
        # print(y_pred.shape, X_train.unsqueeze(1).shape)
        # loss between reconstructed img and original img
        # rec_loss = loss_func(y_pred, X_train.unsqueeze(1))
        rec_loss = loss_func(y_pred.view(batch_size, -1), sample_img) # unsqueeze
X_train: [64, 784] -> [64, 1, 784]
        rec_loss.backward()
        optimizer.step()
        if train_it % 100 == 0:
            print("It {}: Reconstruction Loss: {}".format(train_it, rec_loss))
        train_it += 1
#################################### END TODO ######################################
```

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

```
Device available cuda:0
Run Epoch 0
It 0: Reconstruction Loss: 0.6941973567008972
It 100: Reconstruction Loss: 0.2600642442703247
It 200: Reconstruction Loss: 0.21377862989902496
It 300: Reconstruction Loss: 0.19295649230480194
It 400: Reconstruction Loss: 0.1773861050605774
It 500: Reconstruction Loss: 0.15200963616371155
It 600: Reconstruction Loss: 0.15025803446769714
It 700: Reconstruction Loss: 0.1468232423067093
It 800: Reconstruction Loss: 0.15159788727760315
It 900: Reconstruction Loss: 0.12887603044509888
Run Epoch 1
It 1000: Reconstruction Loss: 0.13041232526302338
It 1100: Reconstruction Loss: 0.13362430036067963
It 1200: Reconstruction Loss: 0.12781736254692078
It 1300: Reconstruction Loss: 0.12402553856372833
It 1400: Reconstruction Loss: 0.12981130182743073
It 1500: Reconstruction Loss: 0.11978445202112198
It 1600: Reconstruction Loss: 0.11579222232103348
It 1700: Reconstruction Loss: 0.11232109367847443
It 1800: Reconstruction Loss: 0.10930512845516205
Run Epoch 2
It 1900: Reconstruction Loss: 0.12100382894277573
It 2000: Reconstruction Loss: 0.11269942671060562
It 2100: Reconstruction Loss: 0.11609923839569092
It 2200: Reconstruction Loss: 0.11043394356966019
It 2300: Reconstruction Loss: 0.1102084144949913
It 2400: Reconstruction Loss: 0.1063244491815567
It 2500: Reconstruction Loss: 0.10522106289863586
It 2600: Reconstruction Loss: 0.10785263031721115
It 2700: Reconstruction Loss: 0.1129796952009201
It 2800: Reconstruction Loss: 0.11421039700508118
Run Epoch 3
It 2900: Reconstruction Loss: 0.10506419837474823
It 3000: Reconstruction Loss: 0.10360829532146454
It 3100: Reconstruction Loss: 0.10624536126852036
It 3200: Reconstruction Loss: 0.10837209969758987
It 3300: Reconstruction Loss: 0.10888055711984634
It 3400: Reconstruction Loss: 0.1086772009730339
It 3500: Reconstruction Loss: 0.10291092842817307
It 3600: Reconstruction Loss: 0.09583210200071335
It 3700: Reconstruction Loss: 0.10236454010009766
Run Epoch 4
It 3800: Reconstruction Loss: 0.09569605439901352
It 3900: Reconstruction Loss: 0.10292566567659378
```

```
It 4000: Reconstruction Loss: 0.09966414421796799
It 4100: Reconstruction Loss: 0.10311181098222733
It 4200: Reconstruction Loss: 0.10047150403261185
It 4300: Reconstruction Loss: 0.10418491810560226
It 4400: Reconstruction Loss: 0.09745653718709946
It 4500: Reconstruction Loss: 0.09582802653312683
It 4600: Reconstruction Loss: 0.0952438935637474
Run Epoch 5
It 4700: Reconstruction Loss: 0.09557317942380905
It 4800: Reconstruction Loss: 0.09920232743024826
It 4900: Reconstruction Loss: 0.09611573070287704
It 5000: Reconstruction Loss: 0.10116905719041824
It 5100: Reconstruction Loss: 0.09177839010953903
It 5200: Reconstruction Loss: 0.0999147817492485
It 5300: Reconstruction Loss: 0.09217223525047302
It 5400: Reconstruction Loss: 0.09530997276306152
It 5500: Reconstruction Loss: 0.0955212265253067
It 5600: Reconstruction Loss: 0.09844547510147095
Run Epoch 6
It 5700: Reconstruction Loss: 0.08832038193941116
It 5800: Reconstruction Loss: 0.09406760334968567
It 5900: Reconstruction Loss: 0.0942428931593895
It 6000: Reconstruction Loss: 0.09220677614212036
It 6100: Reconstruction Loss: 0.09558732062578201
It 6200: Reconstruction Loss: 0.08982578665018082
It 6300: Reconstruction Loss: 0.09608685225248337
It 6400: Reconstruction Loss: 0.09362149238586426
It 6500: Reconstruction Loss: 0.09448808431625366
Run Epoch 7
It 6600: Reconstruction Loss: 0.09203379601240158
It 6700: Reconstruction Loss: 0.09271685779094696
It 6800: Reconstruction Loss: 0.08378888666629791
It 6900: Reconstruction Loss: 0.08798205107450485
It 7000: Reconstruction Loss: 0.08749934285879135
It 7100: Reconstruction Loss: 0.08796916157007217
It 7200: Reconstruction Loss: 0.09372701495885849
It 7300: Reconstruction Loss: 0.0945957824587822
It 7400: Reconstruction Loss: 0.0908859372138977
Run Epoch 8
It 7500: Reconstruction Loss: 0.08930474519729614
It 7600: Reconstruction Loss: 0.09202592819929123
It 7700: Reconstruction Loss: 0.0878760814666748
It 7800: Reconstruction Loss: 0.08705627173185349
It 7900: Reconstruction Loss: 0.09490896761417389
It 8000: Reconstruction Loss: 0.08935634046792984
It 8100: Reconstruction Loss: 0.09234100580215454
It 8200: Reconstruction Loss: 0.09026658535003662
It 8300: Reconstruction Loss: 0.0871456116437912
It 8400: Reconstruction Loss: 0.09165652841329575
```

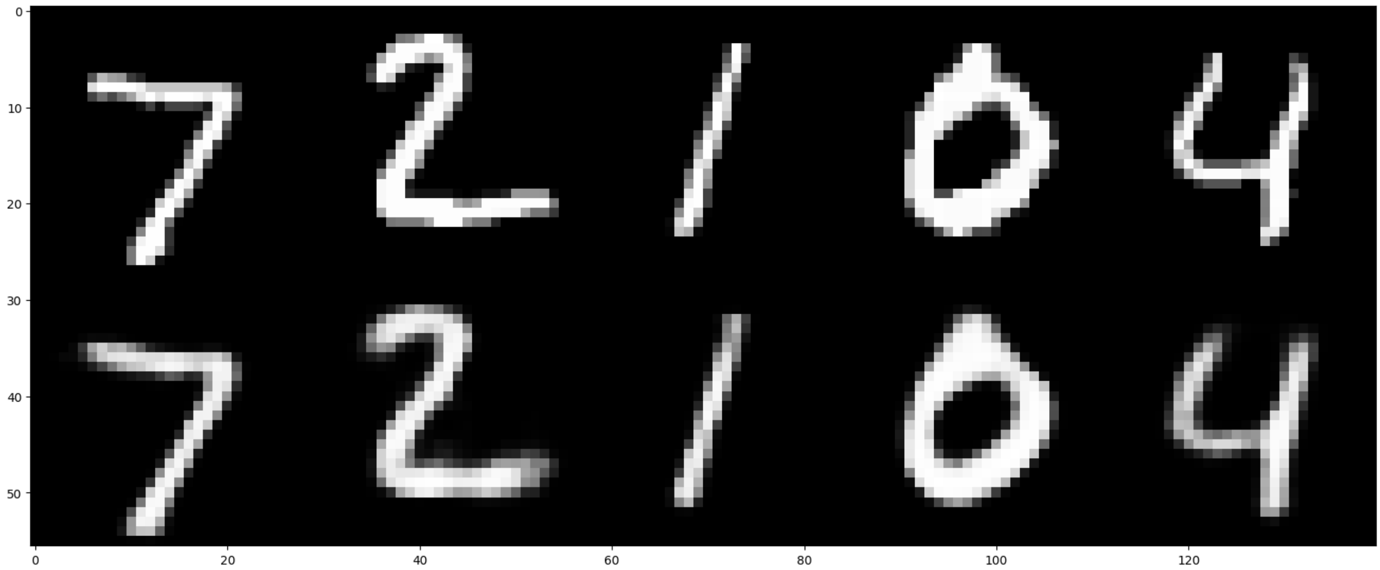
```
Run Epoch 9
It 8500: Reconstruction Loss: 0.08753928542137146
It 8600: Reconstruction Loss: 0.09677305072546005
It 8700: Reconstruction Loss: 0.08558925241231918
It 8800: Reconstruction Loss: 0.08861494809389114
It 8900: Reconstruction Loss: 0.0951785147190094
It 9000: Reconstruction Loss: 0.083978570997715
It 9100: Reconstruction Loss: 0.08792658895254135
It 9200: Reconstruction Loss: 0.09171804040670395
It 9300: Reconstruction Loss: 0.08485647290945053
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.**

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

transform=torchvision.transforms.ToTensor())
    model.eval()      # set model in evalidation mode (eg freeze batchnorm params)
    num_samples = 5
    if randomize:
        sample_idxs = np.random.randint(low=0,high=len(mnist_test), size=num_samples)
    else:
        sample_idxs = list(range(num_samples))

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

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

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

```
    plt.show()

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

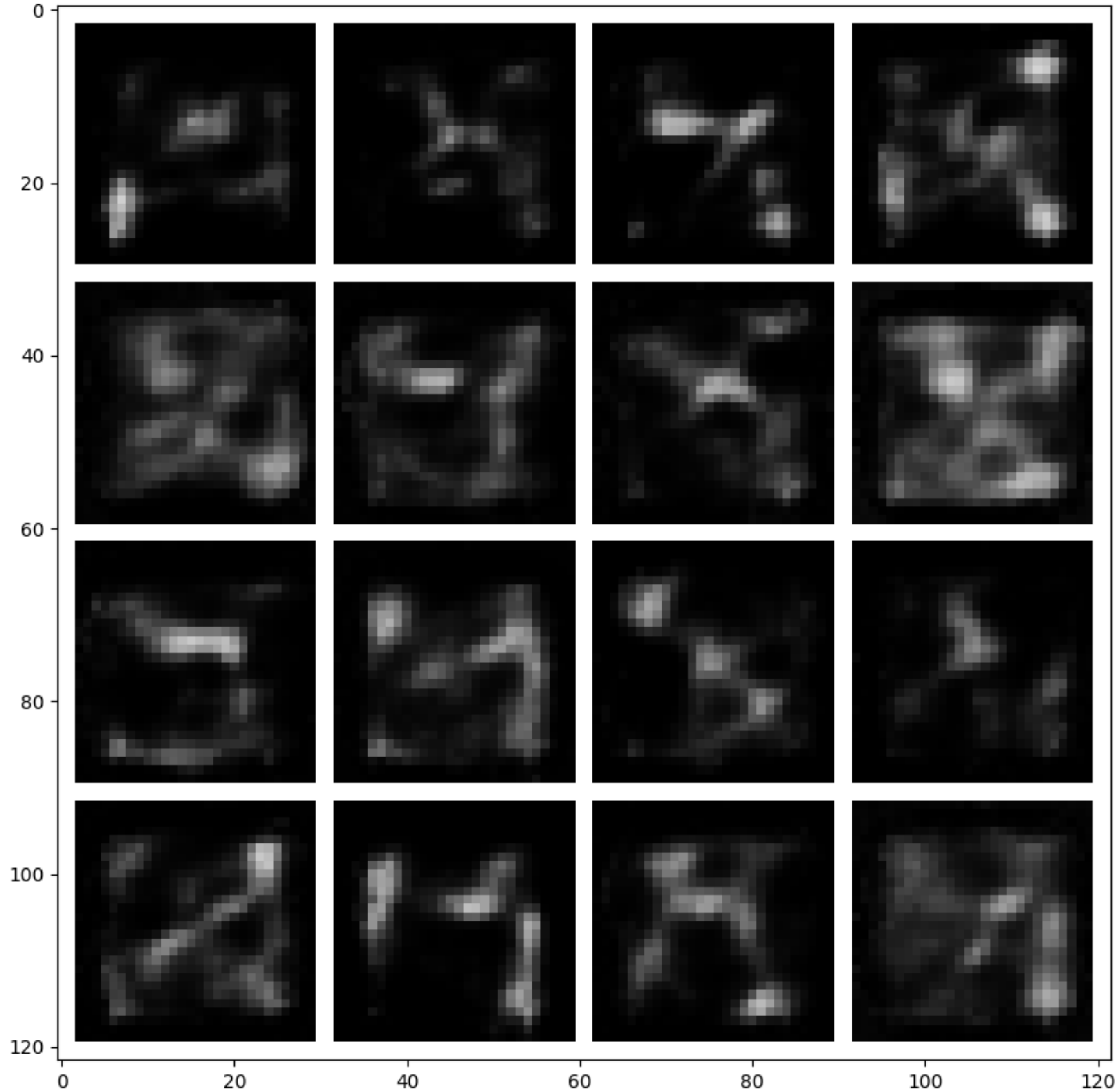


Sampling from the Auto-Encoder [2pt]

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

```
# we will sample N embeddings, then decode and visualize them
def vis_samples(model):
    ####################################### TODO
#######################################
    # Prob1-3 Sample embeddings from a diagonal unit Gaussian distribution and decode
them     #
    # using the model.
#
    # HINT: The sampled embeddings should have shape [batch_size, nz]. Diagonal unit
#
    #       Gaussians have mean 0 and a covariance matrix with ones on the diagonal
#
    #       and zeros everywhere else.
#
    # HINT: If you are unsure whether you sampled the correct distribution, you can
#
    #       sample a large batch and compute the empirical mean and variance using the
#
    #       .mean() and .var() functions.
#
```

```
    # HINT: You can directly use model.decoder() to decode the samples.
#

 ####################################################################################
    sampled_embeddings = torch.randn((batch_size, nz)).to(device)  # sample batch of
embedding from prior
    # print(sampled_embeddings.mean(), sampled_embeddings.var())
    # decoder output images for sampled embeddings
    decoded_samples = model.decoder(sampled_embeddings).reshape(-1, 1, 28, 28)
    # print(decoded_samples.shape)
    #################################### 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 encoded latent space may not be regular, which can make it difficult for AE model to perform smooth and meaningful interpolations between different points in the space. This irregularity and discontinuity can result from various factors such as the distribution of data, model architecture, and the dimension of the latent space. Furthermore, AE cannot learn the underlying distribution of the input data, which can limit their ability to generate high-quality samples from the latent space. Thus, the results decoded from Gaussain distribution are low-quality.

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):

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

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

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

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

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

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

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

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]

```
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.nz = nz
        self.encoder = Encoder(2 * self.nz, in_size)
        self.decoder = Decoder(self.nz, out_size)
        ##################################### 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
    #

####################################################################################
        q = self.encoder(x)  # encode x
```

```
        mu = q[:, :self.nz]  # split encoding
        log_sigma = q[:, self.nz:]
        # mu, log_sigma = torch.chunk(q, 2, dim=-1)
        # reparametrization: z = mu + sigma * epsilon
        eps = torch.randn(log_sigma.shape).to(device)
        # eps = torch.normal(mean=torch.zeros_like(mu), std=torch.ones_like(log_sigma))
        # sample latent variable z with reparametrization
        z = mu + torch.exp(log_sigma) * eps
        reconstruction = self.decoder(z)  # decode 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)
   #

####################################################################################
        loss_func = nn.BCELoss(reduction='mean')
        # loss_func = nn.MSELoss()
        # x: [64, 784], y: [64, 1, 784] -> [64, 784]
        rec_loss = loss_func(outputs['rec'].view(batch_size, -1), x)

        # print(outputs['q'].shape)
        q = outputs['q']
        # mu1 = q[:, :self.nz]  # [batch_size, nz] = [64, 32]
        # log_sigma1 = q[:, self.nz:]
        mu1, log_sigma1 = torch.chunk(q, 2, dim=1)
        mu2 = torch.zeros_like(mu1).to(device)
```

```
        log_sigma2 = torch.zeros_like(log_sigma1).to(device)
        # make it as an scalar instead of a array
        # kl_loss = torch.mean(torch.sum(kl_divergence(mu1, log_sigma1, mu2,
log_sigma2), dim=1), dim=0)
        kl_loss = kl_divergence(mu1, log_sigma1, mu2,
log_sigma2).sum(dim=1).mean(dim=0)
        #################################### END TODO
######################################

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

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

#####################################################################################
        mu = self.encoder(x)[:self.nz]
        # encoder x, split mean, then decode mean
        image = self.decoder(mu).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 $\beta = 0$.

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

####################################### TODO #######################################
epochs = 5      # recommended 5-20 epochs
#################################### END TODO ######################################
```

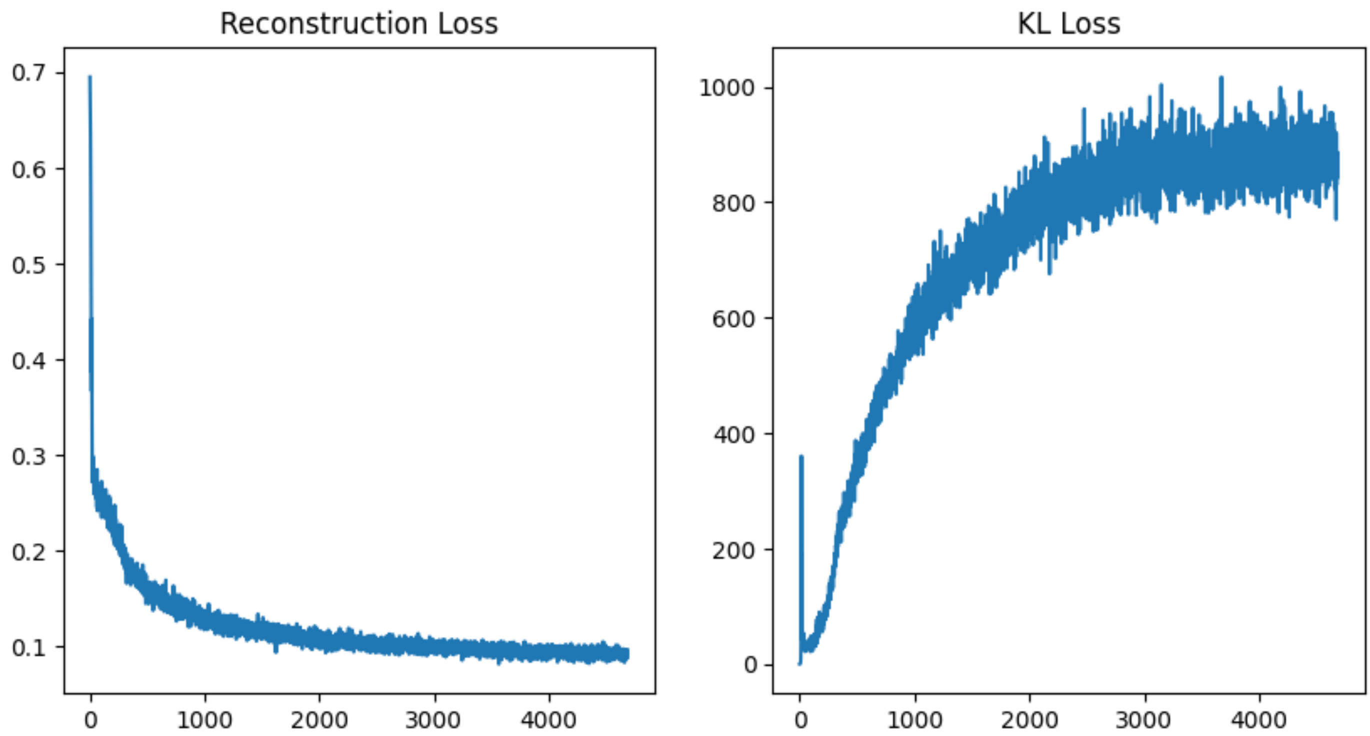
```
# build VAE model
vae_model = VAE(nz, beta).to(device)    # transfer model to GPU if available
vae_model = vae_model.train()   # set model in train mode (eg batchnorm params get
updated)

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

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

 ####################################################################################
    for X_train, y_train in mnist_data_loader:
        X_train, y_train = X_train.to(device), y_train.to(device)
        optimizer.zero_grad()
        X_train = X_train.reshape([batch_size, in_size])
        # falten X_train: [64, 28, 28] -> [64, 784]
        y_pred = vae_model(X_train)
        # print(y_pred['rec'].shape, X_train.shape)
        total_loss, losses = vae_model.loss(X_train, y_pred)
        # losses['rec_loss'] = losses['rec_loss'].detach().cpu()
```

```
        # losses['kl_loss'] = losses['kl_loss'].detach().cpu()
        # print(total_loss.shape)

        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: 0.6949584484100342,    Rec Loss: 0.6949584484100342,   KL Loss:
0.30301231145858765
It 100: Total Loss: 0.25568607449531555,     Rec Loss: 0.25568607449531555,  KL Loss:
34.07611846923828
It 200: Total Loss: 0.236419677734375,   Rec Loss: 0.236419677734375,    KL Loss:
66.10553741455078
It 300: Total Loss: 0.20365391671657562,     Rec Loss: 0.20365391671657562,  KL Loss:
167.48959350585938
It 400: Total Loss: 0.16944284737110138,     Rec Loss: 0.16944284737110138,  KL Loss:
268.63836669921875
It 500: Total Loss: 0.16152262687683105,     Rec Loss: 0.16152262687683105,  KL Loss:
330.0137634277344
It 600: Total Loss: 0.15133586525917053,     Rec Loss: 0.15133586525917053,  KL Loss:
409.5831604003906
It 700: Total Loss: 0.147485613822937,   Rec Loss: 0.147485613822937,    KL Loss:
427.9168701171875
It 800: Total Loss: 0.14304688572883606,     Rec Loss: 0.14304688572883606,  KL Loss:
481.8742980957031
```

```
It 900: Total Loss: 0.13400740921497345,    Rec Loss: 0.13400740921497345,  KL Loss:
537.3521728515625
Run Epoch 1
It 1000: Total Loss: 0.13504794239997864,   Rec Loss: 0.13504794239997864,  KL Loss:
589.2233276367188
It 1100: Total Loss: 0.11404438316822052,   Rec Loss: 0.11404438316822052,  KL Loss:
637.029052734375
It 1200: Total Loss: 0.11407783627510071,   Rec Loss: 0.11407783627510071,  KL Loss:
616.8056030273438
It 1300: Total Loss: 0.12522414326667786,   Rec Loss: 0.12522414326667786,  KL Loss:
671.4837036132812
It 1400: Total Loss: 0.11785966157913208,   Rec Loss: 0.11785966157913208,  KL Loss:
676.5065307617188
It 1500: Total Loss: 0.11737716943025589,   Rec Loss: 0.11737716943025589,  KL Loss:
728.6448974609375
It 1600: Total Loss: 0.11114905774593353,   Rec Loss: 0.11114905774593353,  KL Loss:
718.3271484375
It 1700: Total Loss: 0.11836127936840057,   Rec Loss: 0.11836127936840057,  KL Loss:
699.9386596679688
It 1800: Total Loss: 0.12070836871862411,   Rec Loss: 0.12070836871862411,  KL Loss:
734.30224609375
Run Epoch 2
It 1900: Total Loss: 0.10240508615970612,   Rec Loss: 0.10240508615970612,  KL Loss:
776.1109619140625
It 2000: Total Loss: 0.10611988604068756,   Rec Loss: 0.10611988604068756,  KL Loss:
818.9705810546875
It 2100: Total Loss: 0.10807178914546967,   Rec Loss: 0.10807178914546967,  KL Loss:
837.394775390625
It 2200: Total Loss: 0.10127409547567368,   Rec Loss: 0.10127409547567368,  KL Loss:
838.2537231445312
It 2300: Total Loss: 0.1084408089518547,    Rec Loss: 0.1084408089518547,   KL Loss:
813.2244873046875
It 2400: Total Loss: 0.10154508799314499,   Rec Loss: 0.10154508799314499,  KL Loss:
779.3074951171875
It 2500: Total Loss: 0.10442092269659042,   Rec Loss: 0.10442092269659042,  KL Loss:
850.5172119140625
It 2600: Total Loss: 0.1011001393198967,    Rec Loss: 0.1011001393198967,   KL Loss:
791.435302734375
It 2700: Total Loss: 0.0972280353307724,    Rec Loss: 0.0972280353307724,   KL Loss:
817.681396484375
It 2800: Total Loss: 0.09908905625343323,   Rec Loss: 0.09908905625343323,  KL Loss:
875.5213012695312
Run Epoch 3
It 2900: Total Loss: 0.10294036567211151,   Rec Loss: 0.10294036567211151,  KL Loss:
882.3240356445312
It 3000: Total Loss: 0.09609825909137726,   Rec Loss: 0.09609825909137726,  KL Loss:
857.989013671875
It 3100: Total Loss: 0.09500861167907715,   Rec Loss: 0.09500861167907715,  KL Loss:
912.4680786132812
```

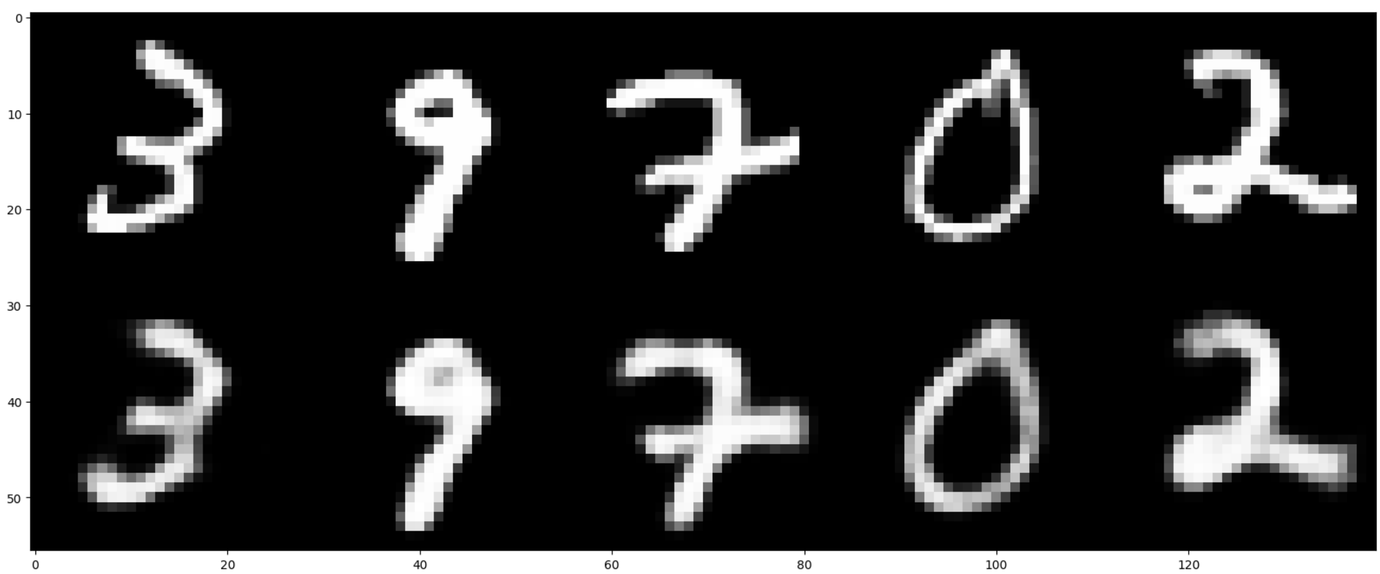
```
It 3200: Total Loss: 0.09884800016880035,    Rec Loss: 0.09884800016880035,  KL Loss:
897.725830078125
It 3300: Total Loss: 0.10174050182104111,    Rec Loss: 0.10174050182104111,  KL Loss:
846.115478515625
It 3400: Total Loss: 0.09637000411748886,    Rec Loss: 0.09637000411748886,  KL Loss:
898.1522216796875
It 3500: Total Loss: 0.09257102012634277,    Rec Loss: 0.09257102012634277,  KL Loss:
826.9158325195312
It 3600: Total Loss: 0.08987121284008026,    Rec Loss: 0.08987121284008026,  KL Loss:
824.9658203125
It 3700: Total Loss: 0.09394235163927078,    Rec Loss: 0.09394235163927078,  KL Loss:
841.6760864257812
Run Epoch 4
It 3800: Total Loss: 0.09188032150268555,    Rec Loss: 0.09188032150268555,  KL Loss:
938.658203125
It 3900: Total Loss: 0.09344063699245453,    Rec Loss: 0.09344063699245453,  KL Loss:
872.4083862304688
It 4000: Total Loss: 0.10003844648599625,    Rec Loss: 0.10003844648599625,  KL Loss:
875.1181640625
It 4100: Total Loss: 0.09494234621524811,    Rec Loss: 0.09494234621524811,  KL Loss:
863.547607421875
It 4200: Total Loss: 0.09472823143005371,    Rec Loss: 0.09472823143005371,  KL Loss:
909.6565551757812
It 4300: Total Loss: 0.08830751478672028,    Rec Loss: 0.08830751478672028,  KL Loss:
878.493408203125
It 4400: Total Loss: 0.09269534796476364,    Rec Loss: 0.09269534796476364,  KL Loss:
855.6065673828125
It 4500: Total Loss: 0.08940350264310837,    Rec Loss: 0.08940350264310837,  KL Loss:
824.0811767578125
It 4600: Total Loss: 0.10150223225355148,    Rec Loss: 0.10150223225355148,  KL Loss:
834.7205200195312
Done!
```

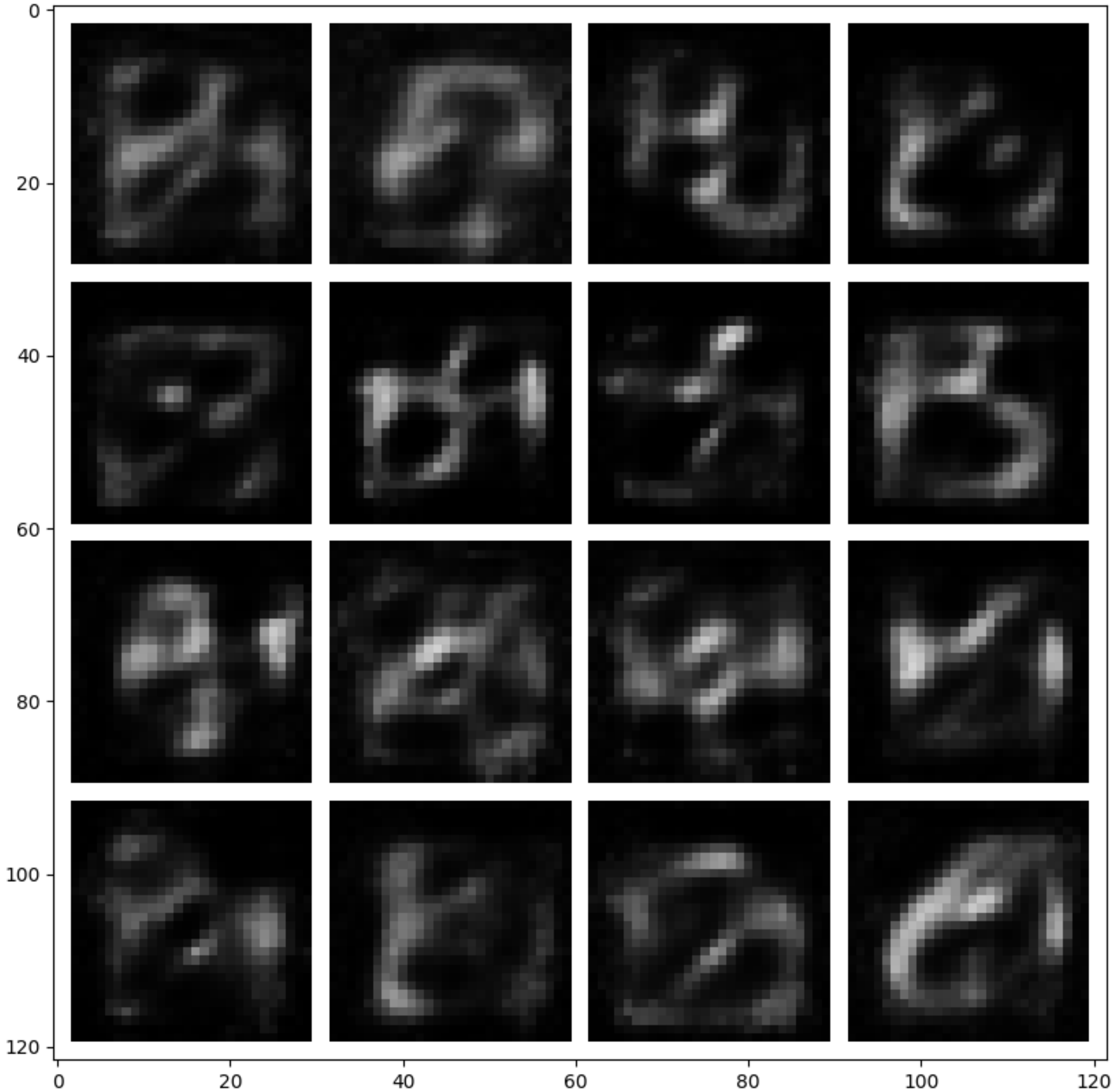


Let's look at some reconstructions and decoded embedding samples!

```
# 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.

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

####################################### TODO #######################################
epochs = 5      # recommended 5-20 epochs
#################################### END TODO ######################################
```

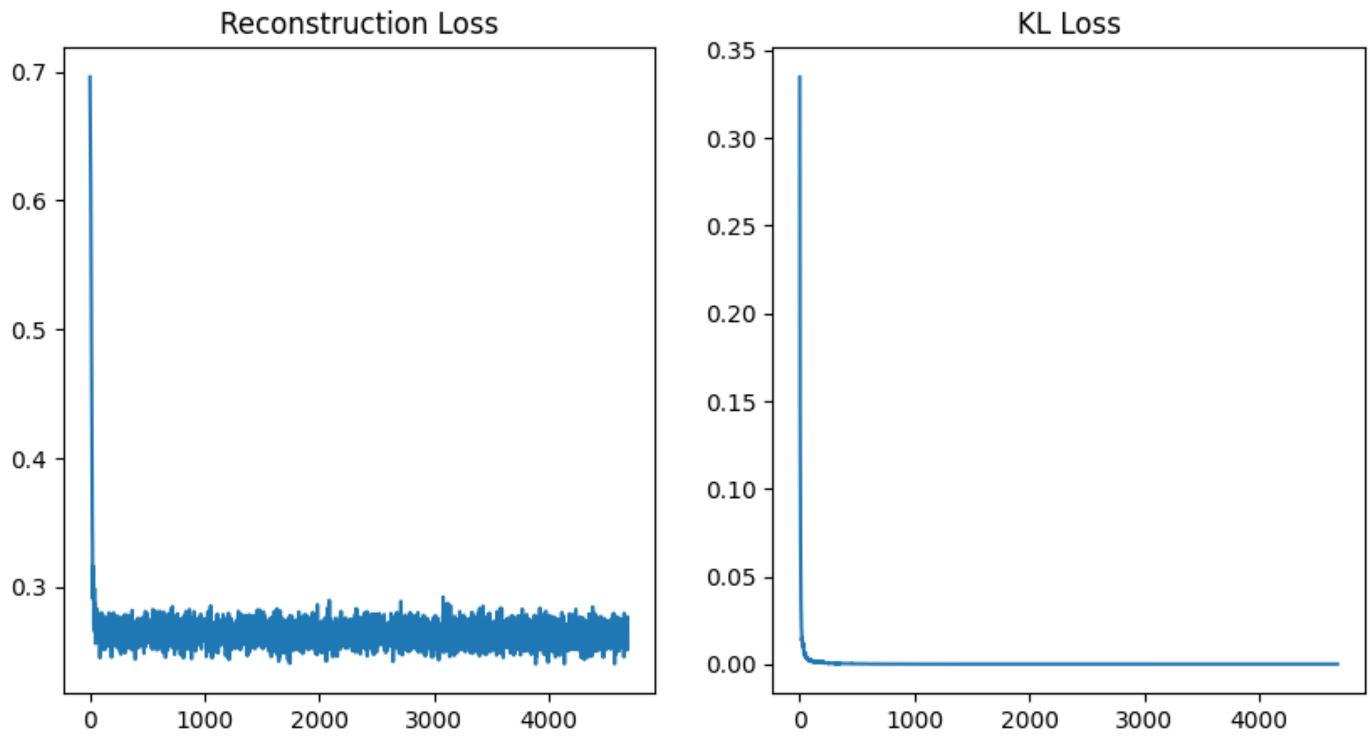
```python
# build VAE model
vae_model = VAE(nz, beta).to(device)    # transfer model to GPU if available
vae_model = vae_model.train()   # set model in train mode (eg batchnorm params get
updated)

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

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

 ####################################################################################
    for X_train, y_train in mnist_data_loader:
        X_train, y_train = X_train.to(device), y_train.to(device)
        optimizer.zero_grad()
        X_train = X_train.reshape([batch_size, in_size])
        # falten X_train: [64, 28, 28] -> [64, 784]
        y_pred = vae_model(X_train)
        # print(y_pred['rec'].shape, X_train.shape)
        total_loss, losses = vae_model.loss(X_train, y_pred)
        # losses['rec_loss'] = losses['rec_loss'].detach().cpu()
        # losses['kl_loss'] = losses['kl_loss'].detach().cpu()
```

```
        # print(total_loss.shape)

        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=10
Run Epoch 0
It 0: Total Loss: 4.043502330780029,     Rec Loss: 0.6957789659500122,   KL Loss:
0.3347723186016083
It 100: Total Loss: 0.2881004810333252,      Rec Loss: 0.2702191472053528,   KL Loss:
0.0017881318926811218
It 200: Total Loss: 0.2783050537109375,      Rec Loss: 0.2672809064388275,   KL Loss:
0.0011024139821529388
It 300: Total Loss: 0.26480501890182495,     Rec Loss: 0.26092588901519775,  KL Loss:
0.0003879130817949772
It 400: Total Loss: 0.2728104293346405,      Rec Loss: 0.26955360174179077,  KL Loss:
0.0003256821073591709
It 500: Total Loss: 0.2586118280887604,      Rec Loss: 0.2566627264022827,   KL Loss:
0.0001949113793671131
It 600: Total Loss: 0.2576855421066284,      Rec Loss: 0.2564143240451813,   KL Loss:
0.00012712227180600166
It 700: Total Loss: 0.2710407078266144,      Rec Loss: 0.27025744318962097,  KL Loss:
7.832655683159828e-05
It 800: Total Loss: 0.2611941695213318,      Rec Loss: 0.2604992389678955,   KL Loss:
6.949156522750854e-05
```

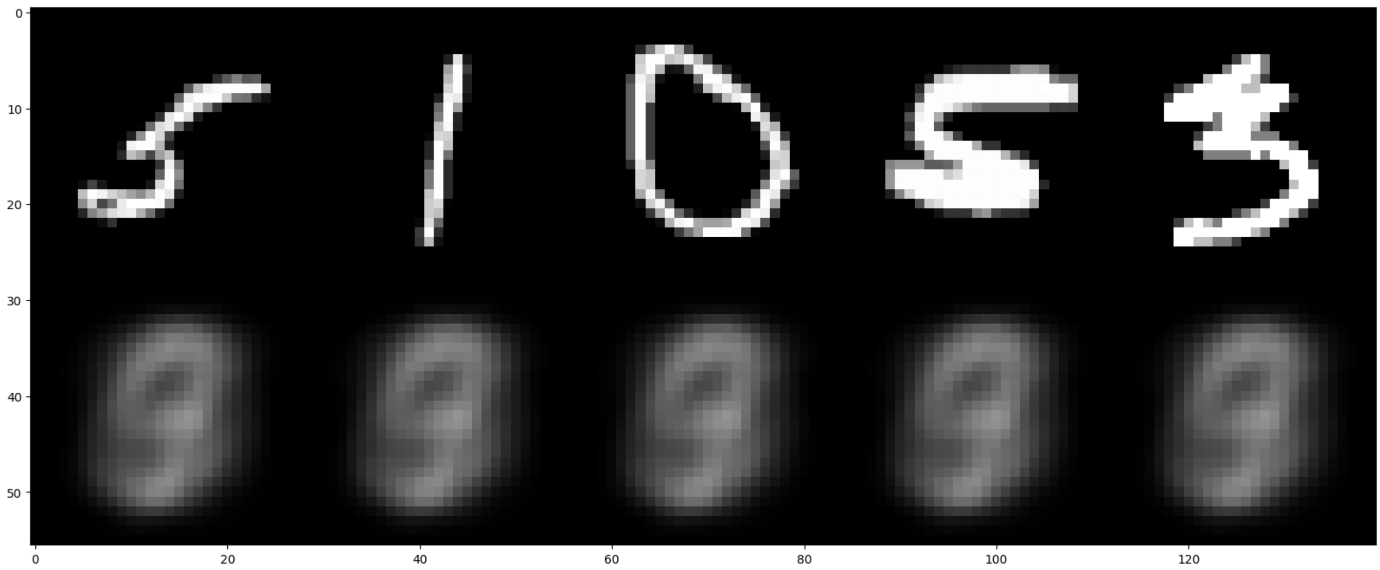
```
It 900: Total Loss: 0.2668711841106415,     Rec Loss: 0.26644134521484375,  KL Loss:
4.298286512494087e-05
Run Epoch 1
It 1000: Total Loss: 0.27138444781303406,    Rec Loss: 0.2710864841938019,   KL Loss:
2.9797665774822235e-05
It 1100: Total Loss: 0.2624884843826294,     Rec Loss: 0.26222851872444153,  KL Loss:
2.599647268652916e-05
It 1200: Total Loss: 0.26039522886276245,    Rec Loss: 0.26023051142692566,  KL Loss:
1.6470439732074738e-05
It 1300: Total Loss: 0.2614837884902954,     Rec Loss: 0.261352092027664 2,   KL Loss:
1.3169366866350174e-05
It 1400: Total Loss: 0.2620086967945099,     Rec Loss: 0.26190704107284546,  KL Loss:
1.0164454579353333e-05
It 1500: Total Loss: 0.27102693915367126,    Rec Loss: 0.27092647552490234,  KL Loss:
1.0045245289802551e-05
It 1600: Total Loss: 0.2748018205165863,     Rec Loss: 0.27472493052482605,  KL Loss:
7.689464837312698e-06
It 1700: Total Loss: 0.26623067259788513,    Rec Loss: 0.26614588499069214,  KL Loss:
8.478760719299316e-06
It 1800: Total Loss: 0.26382920145988464,    Rec Loss: 0.26377955079078674,  KL Loss:
4.9658119678497314e-06
Run Epoch 2
It 1900: Total Loss: 0.25132864713668823,    Rec Loss: 0.2512858510017395,   KL Loss:
4.280358552932739e-06
It 2000: Total Loss: 0.25856828689575195,    Rec Loss: 0.2585291564464569,   KL Loss:
3.9138831198215485e-06
It 2100: Total Loss: 0.25332406163215637,    Rec Loss: 0.2532922625541687,   KL Loss:
3.180932253599167e-06
It 2200: Total Loss: 0.27400606870651245,    Rec Loss: 0.2739742696285248,   KL Loss:
3.1813979148864746e-06
It 2300: Total Loss: 0.2571167051792145,     Rec Loss: 0.2570798695087433,   KL Loss:
3.6847777664661407e-06
It 2400: Total Loss: 0.26613470911979675,    Rec Loss: 0.2661021053791046,   KL Loss:
3.259163349866867e-06
It 2500: Total Loss: 0.27010685205459595,    Rec Loss: 0.2700801193714142,   KL Loss:
2.671964466571808e-06
It 2600: Total Loss: 0.2720736265182495,     Rec Loss: 0.2720469534397125,   KL Loss:
2.6668421924114227e-06
It 2700: Total Loss: 0.2632483243942261,     Rec Loss: 0.26319634914398193,  KL Loss:
5.198176950216293e-06
It 2800: Total Loss: 0.2606072425842285,     Rec Loss: 0.2605708837509155,   KL Loss:
3.636348992586136e-06
Run Epoch 3
It 2900: Total Loss: 0.27058422565460205,    Rec Loss: 0.2705676853656769,   KL Loss:
1.655425876379013e-06
It 3000: Total Loss: 0.2606358528137207,     Rec Loss: 0.26061490178108215,  KL Loss:
2.096407115459442e-06
It 3100: Total Loss: 0.2866145968437195,     Rec Loss: 0.2866026759147644,   KL Loss:
1.191161572933197e-06
```

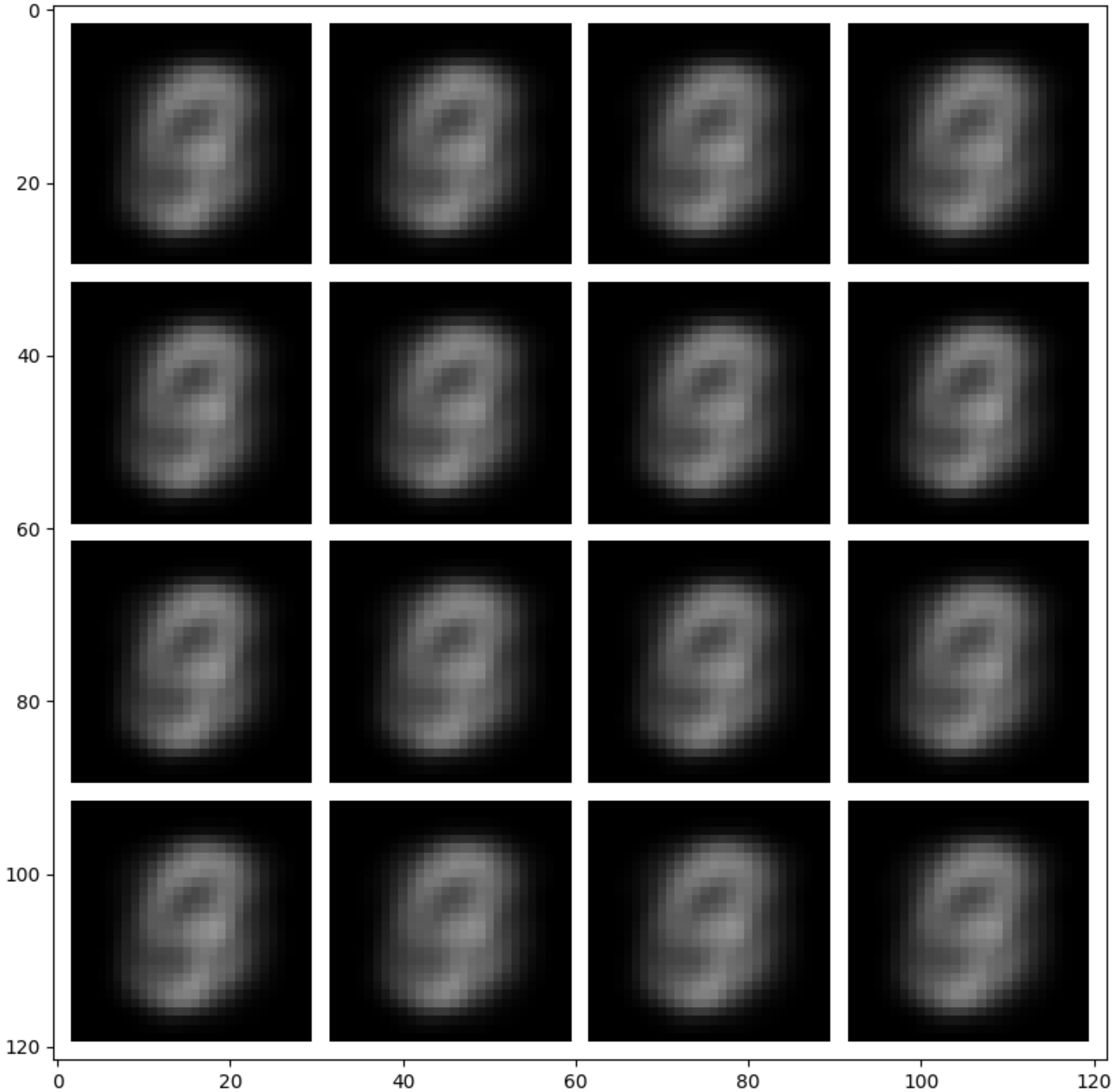
```
It 3200: Total Loss: 0.2566582262516022,     Rec Loss: 0.25663626194000244,  KL Loss:
2.1960586309432983e-06
It 3300: Total Loss: 0.275623083114624,      Rec Loss: 0.2756001651287079,   KL Loss:
2.291519194841385e-06
It 3400: Total Loss: 0.26475316286087036,    Rec Loss: 0.2647143304347992,   KL Loss:
3.884546458721161e-06
It 3500: Total Loss: 0.2548610270023346,     Rec Loss: 0.25484466552734375,  KL Loss:
1.6363337635993958e-06
It 3600: Total Loss: 0.26295459270477295,    Rec Loss: 0.26291799545288086,  KL Loss:
3.661029040813446e-06
It 3700: Total Loss: 0.2668231725692749,     Rec Loss: 0.2668111324310303,   KL Loss:
1.2042000889778137e-06
Run Epoch 4
It 3800: Total Loss: 0.2698535621166229,     Rec Loss: 0.26984137296676636,  KL Loss:
1.2186355888843536e-06
It 3900: Total Loss: 0.2766888439655304,     Rec Loss: 0.2766738831996918,   KL Loss:
1.4961697161197662e-06
It 4000: Total Loss: 0.2591097354888916,     Rec Loss: 0.25910162925720215,  KL Loss:
8.121132850646973e-07
It 4100: Total Loss: 0.2572685778141022,     Rec Loss: 0.2572603225708008,   KL Loss:
8.246861398220062e-07
It 4200: Total Loss: 0.2516791820526123,     Rec Loss: 0.25166958570480347,  KL Loss:
9.583309292793274e-07
It 4300: Total Loss: 0.2628110349178314,     Rec Loss: 0.262800008058548,    KL Loss:
1.1031515896320343e-06
It 4400: Total Loss: 0.2500588595867157,     Rec Loss: 0.2500489354133606,   KL Loss:
9.918585419654846e-07
It 4500: Total Loss: 0.27144747972488403,    Rec Loss: 0.27142757177352905,  KL Loss:
1.991167664527893e-06
It 4600: Total Loss: 0.26362940669059753,    Rec Loss: 0.26361724734306335,  KL Loss:
1.2149102985858917e-06
Done!
```




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

```
beta =  10
```





> **Inline Question: What can you observe when setting $\beta = 0$ and $\beta = 10$? Explain your observations! [2pt]**
> (max 200 words)
>
> **Answer**:
>
> When setting $\beta = 0$ in VAE, the KL divergence term in the loss function is completely removed, and the model only considers the reconstruction loss. In this case, the VAE model behaves like an AE model, and is without the regularization term β on the latent space distribution. From the figure, the reconstruction image is the same as the AE model before. Therefore, the VAE with $\beta = 0$ doesn't have a good generative process. And after decoding, sample distribution cannot produce clear results.

> When setting $\beta = 10$ in VAE, the weight of the KL divergence term is heavily increased. This encourages the VAE model to make the latent space more continuous and smooth, reducing the difference between the reconstructed images and the latent variable. If $\beta > 0$, a stronger constraint will be applied to latent space. However, if β is set too large (like 10), the VAE model may become over-constrained, it make too strong constraint at the cost of reconstruction quality. As we can see in the figure, the reconstructions and the samples are blurred. If we constrain a very low KL loss, the VAE's reconstructed quality will also decreases
>
> Thus, it's essential for us to make trade-off between latent space amd reconstructions. Choosing an appropriate value of β can help the model better balance the weight of reconstruction loss and KL divergence term, leading to better latent variable representation and generative ability.

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**:
> A well-tuned VAE is expected to have:
>
> 1. High reconstructions quality. The reconstruced images shoud look like the original images with low blurred level, i.e., The VAE should be able to accurately reconstruct the input data from the latent space representation.
> 2. Low reconstruction error. The reconstruction error, i.e., the difference between the input data and its reconstruction, should be low.
> 3. Consistency between reconstructions & samples. The reconstructed images and the generated samples should be consistent with the learned data distribution. This indicates that the VAE is able to capture the underlying data distribution accurately.
> 4. Smooth and continuous latent space representation. Neighboring points in the latent space are corresponding to similar input data points. Compared to AE, VAE can generate high-quality samples from the learned latent space distribution.

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 0.005 [2pt]

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

######################################## TODO ########################################
epochs = 20       # recommended 5-20 epochs
beta = 0.005 # Tune this for best results
```

```
#################################### END TODO ######################################

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

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

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

 ####################################################################################
    for X_train, y_train in mnist_data_loader:
        X_train, y_train = X_train.to(device), y_train.to(device)
        optimizer.zero_grad()
        X_train = X_train.reshape([batch_size, in_size])
        # falten X_train: [64, 28, 28] -> [64, 784]
        y_pred = vae_model(X_train)
        # print(y_pred['rec'].shape, X_train.shape)
        total_loss, losses = vae_model.loss(X_train, y_pred)
```

```
        # losses['rec_loss'] = losses['rec_loss'].detach().cpu()
        # losses['kl_loss'] = losses['kl_loss'].detach().cpu()
        # print(total_loss.shape)

        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 20 epochs with beta=0.005
Run Epoch 0
It 0: Total Loss: 0.6951301097869873,    Rec Loss: 0.6934783458709717,   KL Loss:
0.33035051822662354
It 100: Total Loss: 0.2443913072347641,      Rec Loss: 0.24127808213233948,  KL Loss:
0.6226447224617004
It 200: Total Loss: 0.24876989424228668,     Rec Loss: 0.24097298085689545,  KL Loss:
1.559383749961853
It 300: Total Loss: 0.24794985353946686,     Rec Loss: 0.24170154333114624,  KL Loss:
1.2496623992919922
It 400: Total Loss: 0.2551601231098175,      Rec Loss: 0.24916376173496246,  KL Loss:
1.1992732286453247
It 500: Total Loss: 0.2513655722141266,      Rec Loss: 0.24168802797794342,  KL Loss:
1.9355098009109497
It 600: Total Loss: 0.24870824813842773,     Rec Loss: 0.23745959997177124,  KL Loss:
2.249729633331299
It 700: Total Loss: 0.2450023591518402,      Rec Loss: 0.23487970232963562,  KL Loss:
2.0245306491851807
```

```
It 800: Total Loss: 0.24700921773910522,     Rec Loss: 0.23252926766872406,  KL Loss:
2.895991325378418
It 900: Total Loss: 0.2514171600341797,      Rec Loss: 0.23447592556476593,  KL Loss:
3.3882479667663574
Run Epoch 1
It 1000: Total Loss: 0.22184057533740997,    Rec Loss: 0.2044508010149002,   KL Loss:
3.4779553413391113
It 1100: Total Loss: 0.24142208695411682,    Rec Loss: 0.22059844434261322,  KL Loss:
4.164727687835693
It 1200: Total Loss: 0.23203182220458984,    Rec Loss: 0.20698562264442444,  KL Loss:
5.009239196777344
It 1300: Total Loss: 0.22673968970775604,    Rec Loss: 0.20143674314022064,  KL Loss:
5.0605902671813965
It 1400: Total Loss: 0.2066623717546463,     Rec Loss: 0.18062065541744232,  KL Loss:
5.208342552185059
It 1500: Total Loss: 0.22205908596515656,    Rec Loss: 0.19776885211467743,  KL Loss:
4.8580474853515625
It 1600: Total Loss: 0.2141190469264984,     Rec Loss: 0.18736711144447327,  KL Loss:
5.350385665893555
It 1700: Total Loss: 0.21713028848171234,    Rec Loss: 0.1909298598766327,   KL Loss:
5.240086078643799
It 1800: Total Loss: 0.194880872964859,      Rec Loss: 0.16739700734615326,  KL Loss:
5.496772766113281
Run Epoch 2
It 1900: Total Loss: 0.19650927186012268,    Rec Loss: 0.16791556775569916,  KL Loss:
5.7187395095825195
It 2000: Total Loss: 0.20267681777477264,    Rec Loss: 0.17228631675243378,  KL Loss:
6.078100204467773
It 2100: Total Loss: 0.19628649950027466,    Rec Loss: 0.16787561774253845,  KL Loss:
5.68217658996582
It 2200: Total Loss: 0.196811243891716,      Rec Loss: 0.16782918572425842,  KL Loss:
5.796411514282227
It 2300: Total Loss: 0.1915660798549652,     Rec Loss: 0.1595163345336914,   KL Loss:
6.40994930267334
It 2400: Total Loss: 0.1947268694639206,     Rec Loss: 0.16579195857048035,  KL Loss:
5.786983489990234
It 2500: Total Loss: 0.19760264456272125,    Rec Loss: 0.16682474315166473,  KL Loss:
6.155581474304199
It 2600: Total Loss: 0.203748419880867,      Rec Loss: 0.17177671194076538,  KL Loss:
6.394340991973877
It 2700: Total Loss: 0.2055603563785553,     Rec Loss: 0.17226076126098633,  KL Loss:
6.659918308258057
It 2800: Total Loss: 0.20353147387504578,    Rec Loss: 0.1708575189113617,   KL Loss:
6.534791946411133
Run Epoch 3
It 2900: Total Loss: 0.1967518925666809,     Rec Loss: 0.165424183011055,    KL Loss:
6.265541076660156
It 3000: Total Loss: 0.2000911384820938,     Rec Loss: 0.16593904793262482,  KL Loss:
6.830418586730957
```

```
It 3100: Total Loss: 0.20262224972248077,    Rec Loss: 0.17109079658985138,  KL Loss:
6.3062896728515625
It 3200: Total Loss: 0.19437962770462036,    Rec Loss: 0.16230979561805725,  KL Loss:
6.413968086242676
It 3300: Total Loss: 0.21200965344905853,    Rec Loss: 0.17782606184482574,  KL Loss:
6.8367180824279785
It 3400: Total Loss: 0.20770622789859772,    Rec Loss: 0.17445023357868195,  KL Loss:
6.651199817657471
It 3500: Total Loss: 0.19349583983421326,    Rec Loss: 0.16077196598052979,  KL Loss:
6.544774055480957
It 3600: Total Loss: 0.20763127505779266,    Rec Loss: 0.17201201617717743,  KL Loss:
7.123852729797363
It 3700: Total Loss: 0.20930856466293335,    Rec Loss: 0.17470797896385193,  KL Loss:
6.920116901397705
Run Epoch 4
It 3800: Total Loss: 0.19993086159229279,    Rec Loss: 0.16449172794818878,  KL Loss:
7.087825775146484
It 3900: Total Loss: 0.20996065437793732,    Rec Loss: 0.17486898601055145,  KL Loss:
7.01833438873291
It 4000: Total Loss: 0.20386317372322083,    Rec Loss: 0.16952234506607056,  KL Loss:
6.868167400360107
It 4100: Total Loss: 0.1966114193201065,     Rec Loss: 0.16228628158569336,  KL Loss:
6.865027904510498
It 4200: Total Loss: 0.1865125596523285,     Rec Loss: 0.15318986773490906,  KL Loss:
6.664539337158203
It 4300: Total Loss: 0.19731616973876953,    Rec Loss: 0.16393496096134186,  KL Loss:
6.676242828369141
It 4400: Total Loss: 0.19014900922775269,    Rec Loss: 0.15601840615272522,  KL Loss:
6.826122283935547
It 4500: Total Loss: 0.19030043482780457,    Rec Loss: 0.1569850593805313,   KL Loss:
6.663074493408203
It 4600: Total Loss: 0.19880101084709167,    Rec Loss: 0.16299062967300415,  KL Loss:
7.162077903747559
Run Epoch 5
It 4700: Total Loss: 0.19488291442394257,    Rec Loss: 0.1600218117237091,   KL Loss:
6.972219944000244
It 4800: Total Loss: 0.1990889012813568,     Rec Loss: 0.1648281216621399,   KL Loss:
6.852155685424805
It 4900: Total Loss: 0.19571277499198914,    Rec Loss: 0.16137830913066864,  KL Loss:
6.866893768310547
It 5000: Total Loss: 0.1889776885509491,     Rec Loss: 0.15373916923999786,  KL Loss:
7.04770565032959
It 5100: Total Loss: 0.20280897617340088,    Rec Loss: 0.16757342219352722,  KL Loss:
7.047110557556152
It 5200: Total Loss: 0.18687373399734497,    Rec Loss: 0.1526944488286972,   KL Loss:
6.8358564376831055
It 5300: Total Loss: 0.20477578043937683,    Rec Loss: 0.16930599510669708,  KL Loss:
7.093958854675293
```

```
It 5400: Total Loss: 0.19899554550647736,    Rec Loss: 0.1621955931186676,   KL Loss:
7.359989643096924
It 5500: Total Loss: 0.18705078959465027,    Rec Loss: 0.1515687257051468,   KL Loss:
7.096412658691406
It 5600: Total Loss: 0.18706177175045013,    Rec Loss: 0.15283788740634918,  KL Loss:
6.844776630401611
Run Epoch 6
It 5700: Total Loss: 0.1935669332742691,     Rec Loss: 0.15728919208049774,  KL Loss:
7.255548477172852
It 5800: Total Loss: 0.1895630657672882,     Rec Loss: 0.15292860567569733,  KL Loss:
7.326891899108887
It 5900: Total Loss: 0.18698394298553467,    Rec Loss: 0.1517072170972824,   KL Loss:
7.055345058441162
It 6000: Total Loss: 0.18950411677360535,    Rec Loss: 0.15435409545898438,  KL Loss:
7.030004978179932
It 6100: Total Loss: 0.18968701362609863,    Rec Loss: 0.15467776358127594,  KL Loss:
7.001849174499512
It 6200: Total Loss: 0.19093656539916992,    Rec Loss: 0.15323297679424286,  KL Loss:
7.540716171264648
It 6300: Total Loss: 0.1920289546251297,     Rec Loss: 0.15543164312839508,  KL Loss:
7.319461822509766
It 6400: Total Loss: 0.19125127792358398,    Rec Loss: 0.1552550494670868,   KL Loss:
7.199246406555176
It 6500: Total Loss: 0.19423648715019226,    Rec Loss: 0.1582861691713333,   KL Loss:
7.190062046051025
Run Epoch 7
It 6600: Total Loss: 0.18539518117904663,    Rec Loss: 0.1491309553384781,   KL Loss:
7.252843856811523
It 6700: Total Loss: 0.18688955903053284,    Rec Loss: 0.15019963681697845,  KL Loss:
7.337984085083008
It 6800: Total Loss: 0.18603327870368958,    Rec Loss: 0.15092241764068604,  KL Loss:
7.022171497344971
It 6900: Total Loss: 0.18600872159004211,    Rec Loss: 0.1500854790210724,   KL Loss:
7.1846489906311035
It 7000: Total Loss: 0.1882767677307129,     Rec Loss: 0.15044063329696655,  KL Loss:
7.567228317260742
It 7100: Total Loss: 0.1843862235546112,     Rec Loss: 0.14625440537929535,  KL Loss:
7.626364707946777
It 7200: Total Loss: 0.1825878918170929,     Rec Loss: 0.145768940448761,    KL Loss:
7.363788604736328
It 7300: Total Loss: 0.1900421679019928,     Rec Loss: 0.1529398411512375,   KL Loss:
7.420465469360352
It 7400: Total Loss: 0.19006752967834473,    Rec Loss: 0.1529003083705902,   KL Loss:
7.433444499969482
Run Epoch 8
It 7500: Total Loss: 0.19654998183250427,    Rec Loss: 0.1584579348564148,   KL Loss:
7.618408203125
It 7600: Total Loss: 0.1849372833967209,     Rec Loss: 0.1475396603345871,   KL Loss:
7.479524612426758
```

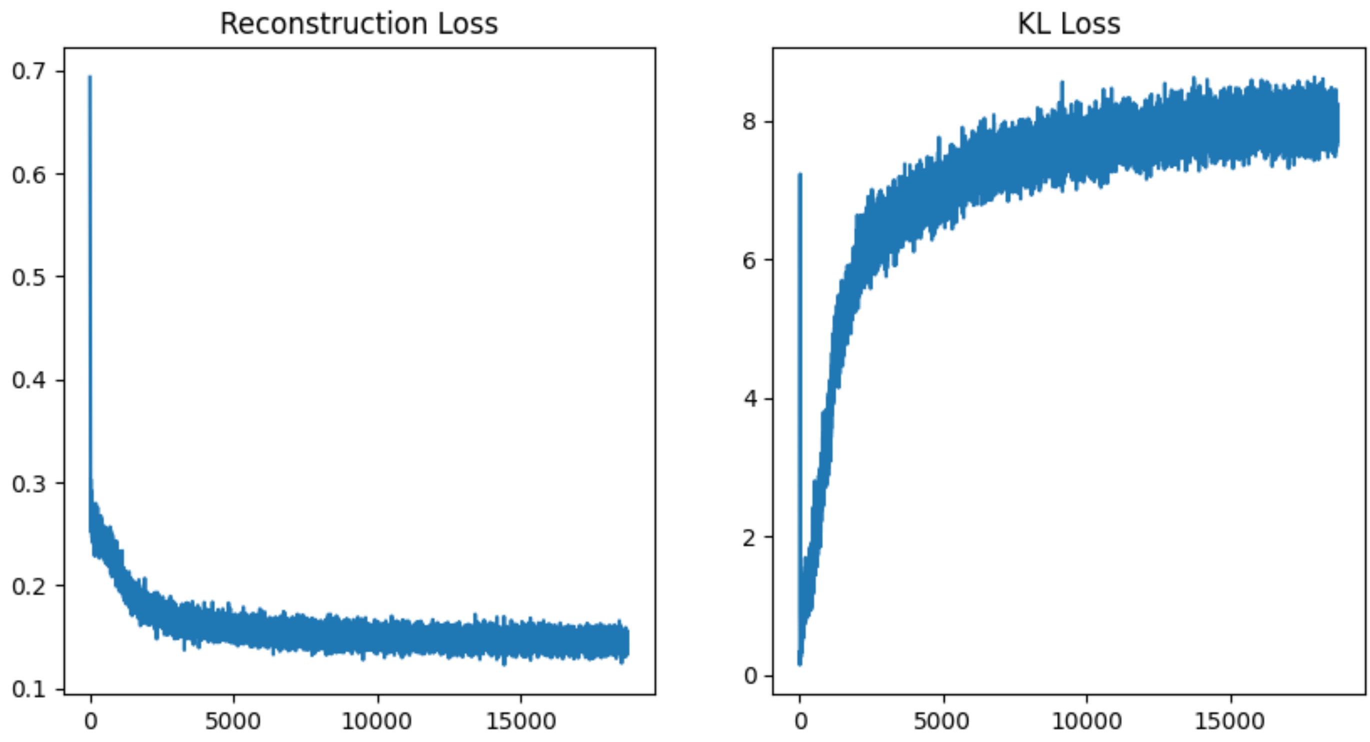
```
It 7700: Total Loss: 0.18161731958389282,    Rec Loss: 0.14452072978019714,  KL Loss: 7.419319152832031
It 7800: Total Loss: 0.19282472133636475,    Rec Loss: 0.15506255626678467,  KL Loss: 7.552432537078857
It 7900: Total Loss: 0.19747202098369598,    Rec Loss: 0.16059567034244537,  KL Loss: 7.375269412994385
It 8000: Total Loss: 0.19068089127540588,    Rec Loss: 0.1514756828546524,   KL Loss: 7.841041564941406
It 8100: Total Loss: 0.19213226437568665,    Rec Loss: 0.15354064106941223,  KL Loss: 7.718325614929199
It 8200: Total Loss: 0.20216801762580872,    Rec Loss: 0.16320323944091797,  KL Loss: 7.792954444885254
It 8300: Total Loss: 0.17858733236789703,    Rec Loss: 0.14160911738872528,  KL Loss: 7.3956427574157715
It 8400: Total Loss: 0.18532830476760864,    Rec Loss: 0.14582575857639313,  KL Loss: 7.900510787963867
Run Epoch 9
It 8500: Total Loss: 0.18583723902702332,    Rec Loss: 0.14698615670204163,  KL Loss: 7.770216464996338
It 8600: Total Loss: 0.1969892382621765,     Rec Loss: 0.15844395756721497,  KL Loss: 7.7090559005737305
It 8700: Total Loss: 0.19546076655387878,    Rec Loss: 0.1580272763967514,   KL Loss: 7.486700057983398
It 8800: Total Loss: 0.18930882215499878,    Rec Loss: 0.14961139857769012,  KL Loss: 7.939485549926758
It 8900: Total Loss: 0.18677033483982086,    Rec Loss: 0.14624272286891937,  KL Loss: 8.105522155761719
It 9000: Total Loss: 0.18644224107265472,    Rec Loss: 0.15007558465003967,  KL Loss: 7.2733306884765625
It 9100: Total Loss: 0.18947285413742065,    Rec Loss: 0.15077179670333862,  KL Loss: 7.740213394165039
It 9200: Total Loss: 0.1850229799747467,     Rec Loss: 0.1461508721113205,   KL Loss: 7.7744221687316895
It 9300: Total Loss: 0.19828584790229797,    Rec Loss: 0.16042104363441467,  KL Loss: 7.572961807250977
Run Epoch 10
It 9400: Total Loss: 0.18824121356010437,    Rec Loss: 0.14896312355995178,  KL Loss: 7.855619430541992
It 9500: Total Loss: 0.18680152297019958,    Rec Loss: 0.15108616650104523,  KL Loss: 7.143072128295898
It 9600: Total Loss: 0.18067589402198792,    Rec Loss: 0.14411959052085876,  KL Loss: 7.3112592697143555
It 9700: Total Loss: 0.18653658032417297,    Rec Loss: 0.1474369317293167,   KL Loss: 7.819928169250488
It 9800: Total Loss: 0.18674400448799133,    Rec Loss: 0.14878571033477783,  KL Loss: 7.591657638549805
It 9900: Total Loss: 0.1909576803445816,     Rec Loss: 0.15217049419879913,  KL Loss: 7.757437705993652
```

```
It 10000: Total Loss: 0.18401560187339783,   Rec Loss: 0.14439243078231812,  KL Loss:
7.9246344566345215
It 10100: Total Loss: 0.18373335897922516,   Rec Loss: 0.14645099639892578,  KL Loss:
7.456473350524902
It 10200: Total Loss: 0.18784180283546448,   Rec Loss: 0.14841406047344208,  KL Loss:
7.8855485916137695
It 10300: Total Loss: 0.18322569131851196,   Rec Loss: 0.14702613651752472,  KL Loss:
7.239912033081055
Run Epoch 11
It 10400: Total Loss: 0.18788081407546997,   Rec Loss: 0.1488001048564911,   KL Loss:
7.816140174865723
It 10500: Total Loss: 0.18667256832122803,   Rec Loss: 0.1491824984550476,   KL Loss:
7.498013496398926
It 10600: Total Loss: 0.19172543287277222,   Rec Loss: 0.15500469505786896,  KL Loss:
7.344147205352783
It 10700: Total Loss: 0.1872260868549347,    Rec Loss: 0.1488967388868332,   KL Loss:
7.66586971282959
It 10800: Total Loss: 0.17188550531864166,   Rec Loss: 0.13446079194545746,  KL Loss:
7.4849419593811035
It 10900: Total Loss: 0.19570723176002502,   Rec Loss: 0.15637187659740448,  KL Loss:
7.86707067489624
It 11000: Total Loss: 0.1935248076915741,    Rec Loss: 0.15481413900852203,  KL Loss:
7.7421345710754395
It 11100: Total Loss: 0.18665218353271484,   Rec Loss: 0.14725801348686218,  KL Loss:
7.8788347244262695
It 11200: Total Loss: 0.189425989985466,     Rec Loss: 0.15134811401367188,  KL Loss:
7.615576267242432
Run Epoch 12
It 11300: Total Loss: 0.1935732662677765,    Rec Loss: 0.1556502729654312,   KL Loss:
7.584600448608398
It 11400: Total Loss: 0.19450336694717407,   Rec Loss: 0.15464948117733002,  KL Loss:
7.970778942108154
It 11500: Total Loss: 0.18322308361530304,   Rec Loss: 0.14320829510688782,  KL Loss:
8.002958297729492
It 11600: Total Loss: 0.1875043362379074,    Rec Loss: 0.14917130768299103,  KL Loss:
7.666605472564697
It 11700: Total Loss: 0.19429203867912292,   Rec Loss: 0.15310420095920563,  KL Loss:
8.237565994262695
It 11800: Total Loss: 0.1964062750339508,    Rec Loss: 0.15767161548137665,  KL Loss:
7.7469305992126465
It 11900: Total Loss: 0.18954935669898987,   Rec Loss: 0.15130889415740967,  KL Loss:
7.6480913162231445
It 12000: Total Loss: 0.19058331847190857,   Rec Loss: 0.15023036301136017,  KL Loss:
8.070592880249023
It 12100: Total Loss: 0.1869863122701645,    Rec Loss: 0.1487039029598236,   KL Loss:
7.656482696533203
Run Epoch 13
It 12200: Total Loss: 0.17617058753967285,   Rec Loss: 0.13721297681331635,  KL Loss:
7.791522979736328
```

```
It 12300: Total Loss: 0.18486186861991882,   Rec Loss: 0.14609679579734802,  KL Loss:
7.753015518188477
It 12400: Total Loss: 0.2003490924835205,    Rec Loss: 0.16176839172840118,  KL Loss:
7.71613883972168
It 12500: Total Loss: 0.19093936681747437,   Rec Loss: 0.1543717384338379,   KL Loss:
7.313527584075928
It 12600: Total Loss: 0.19128888845443726,   Rec Loss: 0.1521652191877365,   KL Loss:
7.824732780456543
It 12700: Total Loss: 0.190524622797966,     Rec Loss: 0.15048272907733917,  KL Loss:
8.008378982543945
It 12800: Total Loss: 0.18168240785598755,   Rec Loss: 0.1427675485610962,   KL Loss:
7.7829718589782715
It 12900: Total Loss: 0.18937534093856812,   Rec Loss: 0.15112897753715515,  KL Loss:
7.649273872375488
It 13000: Total Loss: 0.18815547227859497,   Rec Loss: 0.14947466552257538,  KL Loss:
7.736161231994629
It 13100: Total Loss: 0.19273564219474792,   Rec Loss: 0.15276360511779785,  KL Loss:
7.994407653808594
Run Epoch 14
It 13200: Total Loss: 0.18535305559635162,   Rec Loss: 0.1487305462360382,   KL Loss:
7.324501991271973
It 13300: Total Loss: 0.183826744556427,     Rec Loss: 0.14499539136886597,  KL Loss:
7.766269207000732
It 13400: Total Loss: 0.18496474623680115,   Rec Loss: 0.14537641406059265,  KL Loss:
7.917668342590332
It 13500: Total Loss: 0.19278588891029358,   Rec Loss: 0.1535452902317047,   KL Loss:
7.848118782043457
It 13600: Total Loss: 0.18447203934192657,   Rec Loss: 0.14727067947387695,  KL Loss:
7.440271377563477
It 13700: Total Loss: 0.18202924728393555,   Rec Loss: 0.14262284338474274,  KL Loss:
7.881281852722168
It 13800: Total Loss: 0.18698722124099731,   Rec Loss: 0.1460842788219452,   KL Loss:
8.18058967590332
It 13900: Total Loss: 0.18116974830627441,   Rec Loss: 0.1441998928785324,   KL Loss:
7.393972396850586
It 14000: Total Loss: 0.18770824372768402,   Rec Loss: 0.14802145957946777,  KL Loss:
7.937355995178223
Run Epoch 15
It 14100: Total Loss: 0.1852949857711792,    Rec Loss: 0.146331787109375,    KL Loss:
7.792640209197998
It 14200: Total Loss: 0.1930757611989975,    Rec Loss: 0.15263445675373077,  KL Loss:
8.088260650634766
It 14300: Total Loss: 0.18235264718532562,   Rec Loss: 0.14123766124248505,  KL Loss:
8.222996711730957
It 14400: Total Loss: 0.19115066528320312,   Rec Loss: 0.1514677256345749,   KL Loss:
7.936586380004883
It 14500: Total Loss: 0.1819249540567398,    Rec Loss: 0.14475977420806885,  KL Loss:
7.433035373687744
```

```
It 14600: Total Loss: 0.18414416909217834,   Rec Loss: 0.14455077052116394,  KL Loss:
7.9186787605285645
It 14700: Total Loss: 0.18654689192771912,   Rec Loss: 0.14815469086170197,  KL Loss:
7.678439140319824
It 14800: Total Loss: 0.1933647245168686,    Rec Loss: 0.15246760845184326,  KL Loss:
8.179423332214355
It 14900: Total Loss: 0.19114679098129272,   Rec Loss: 0.1503664255142212,   KL Loss:
8.156071662902832
Run Epoch 16
It 15000: Total Loss: 0.18181112408638,      Rec Loss: 0.14116209745407104,  KL Loss:
8.129806518554688
It 15100: Total Loss: 0.1729908287525177,    Rec Loss: 0.13639989495277405,  KL Loss:
7.318188667297363
It 15200: Total Loss: 0.1822843998670578,    Rec Loss: 0.14202356338500977,  KL Loss:
8.052167892456055
It 15300: Total Loss: 0.1919952630996704,    Rec Loss: 0.15038636326789856,  KL Loss:
8.32178020477295
It 15400: Total Loss: 0.17490413784980774,   Rec Loss: 0.13368433713912964,  KL Loss:
8.243961334228516
It 15500: Total Loss: 0.18474510312080383,   Rec Loss: 0.14500203728675842,  KL Loss:
7.948615074157715
It 15600: Total Loss: 0.18608561158180237,   Rec Loss: 0.14678214490413666,  KL Loss:
7.860692977905273
It 15700: Total Loss: 0.1844157725572586,    Rec Loss: 0.14610742032527924,  KL Loss:
7.661670684814453
It 15800: Total Loss: 0.18887898325920105,   Rec Loss: 0.14954546093940735,  KL Loss:
7.866703987121582
It 15900: Total Loss: 0.18216419219970703,   Rec Loss: 0.14187881350517273,  KL Loss:
8.057074546813965
Run Epoch 17
It 16000: Total Loss: 0.18552523851394653,   Rec Loss: 0.14477647840976715,  KL Loss:
8.149752616882324
It 16100: Total Loss: 0.1823066622018814,    Rec Loss: 0.14397001266479492,  KL Loss:
7.667329788208008
It 16200: Total Loss: 0.19213439524173737,   Rec Loss: 0.15020732581615448,  KL Loss:
8.385414123535156
It 16300: Total Loss: 0.17049941420555115,   Rec Loss: 0.13015002012252808,  KL Loss:
8.069877624511719
It 16400: Total Loss: 0.18184056878089905,   Rec Loss: 0.1432226300239563,   KL Loss:
7.7235894203186035
It 16500: Total Loss: 0.1807950735092163,    Rec Loss: 0.14196515083312988,  KL Loss:
7.7659831047058105
It 16600: Total Loss: 0.191180020570755,     Rec Loss: 0.1513635814189911,   KL Loss:
7.963287353515625
It 16700: Total Loss: 0.1910543441772461,    Rec Loss: 0.1501229852437973,   KL Loss:
8.186273574829102
It 16800: Total Loss: 0.1856914460659027,    Rec Loss: 0.14592380821704865,  KL Loss:
7.953529357910156
Run Epoch 18
```

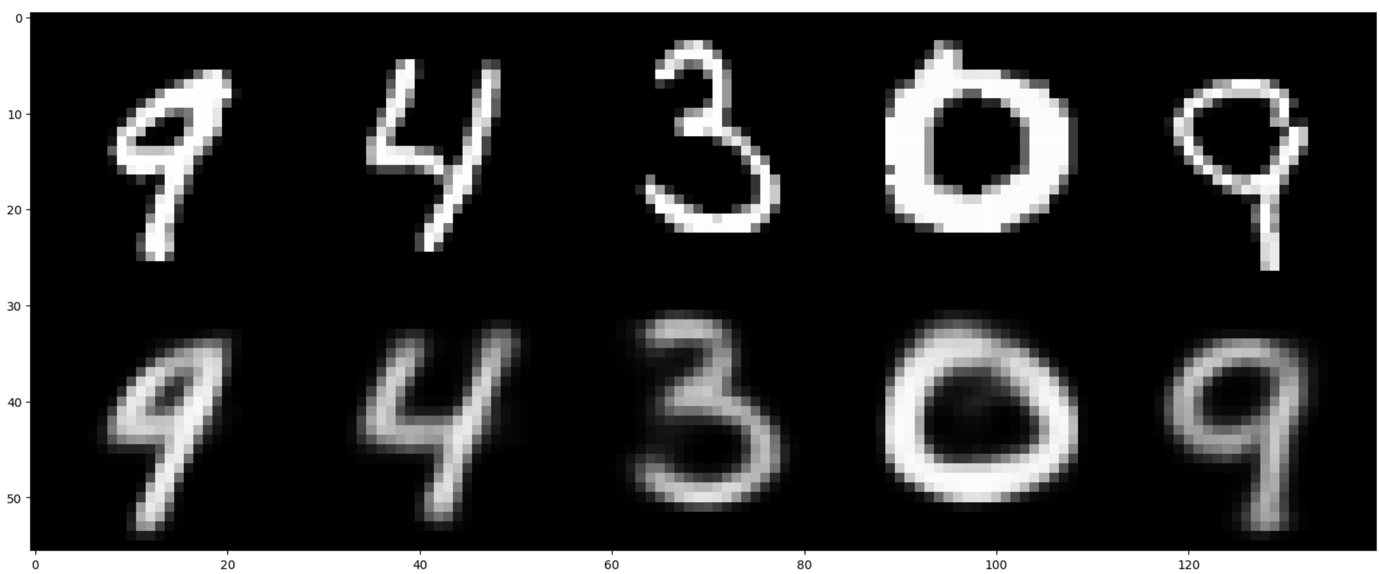
```
It 16900: Total Loss: 0.1933879554271698,    Rec Loss: 0.15386168658733368,  KL Loss:
7.905253887176514
It 17000: Total Loss: 0.1883453130722046,    Rec Loss: 0.14708679914474487,  KL Loss:
8.251703262329102
It 17100: Total Loss: 0.19285333156585693,   Rec Loss: 0.15068215131759644,  KL Loss:
8.434236526489258
It 17200: Total Loss: 0.18289020657539368,   Rec Loss: 0.1438358873128891,   KL Loss:
7.8108625411987305
It 17300: Total Loss: 0.18287375569343567,   Rec Loss: 0.14295001327991486,  KL Loss:
7.984746932983398
It 17400: Total Loss: 0.1826770305633545,    Rec Loss: 0.1426956057548523,   KL Loss:
7.996283531188965
It 17500: Total Loss: 0.20027422904968262,   Rec Loss: 0.15813352167606354,  KL Loss:
8.428141593933105
It 17600: Total Loss: 0.18573778867721558,   Rec Loss: 0.14833498001098633,  KL Loss:
7.480560302734375
It 17700: Total Loss: 0.18565452098846436,   Rec Loss: 0.14438177645206451,  KL Loss:
8.254549026489258
It 17800: Total Loss: 0.1775231510400772,    Rec Loss: 0.13850897550582886,  KL Loss:
7.802834510803223
Run Epoch 19
It 17900: Total Loss: 0.18767033517360687,   Rec Loss: 0.14698205888271332,  KL Loss:
8.137656211853027
It 18000: Total Loss: 0.1843641847372055,    Rec Loss: 0.14634205400943756,  KL Loss:
7.604427337646484
It 18100: Total Loss: 0.17245778441429138,   Rec Loss: 0.1327248513698578,   KL Loss:
7.9465861320495605
It 18200: Total Loss: 0.18513017892837524,   Rec Loss: 0.1458715945482254,   KL Loss:
7.8517165184021
It 18300: Total Loss: 0.1836102306842804,    Rec Loss: 0.1436929702758789,   KL Loss:
7.983452320098877
It 18400: Total Loss: 0.17901290953159332,   Rec Loss: 0.13977617025375366,  KL Loss:
7.847348213195801
It 18500: Total Loss: 0.18653157353401184,   Rec Loss: 0.1466200351715088,   KL Loss:
7.982309341430664
It 18600: Total Loss: 0.18460577726364136,   Rec Loss: 0.1445745974779129,   KL Loss:
8.00623607635498
It 18700: Total Loss: 0.19640396535396576,   Rec Loss: 0.15679343044757843,  KL Loss:
7.922106742858887
Done!
```

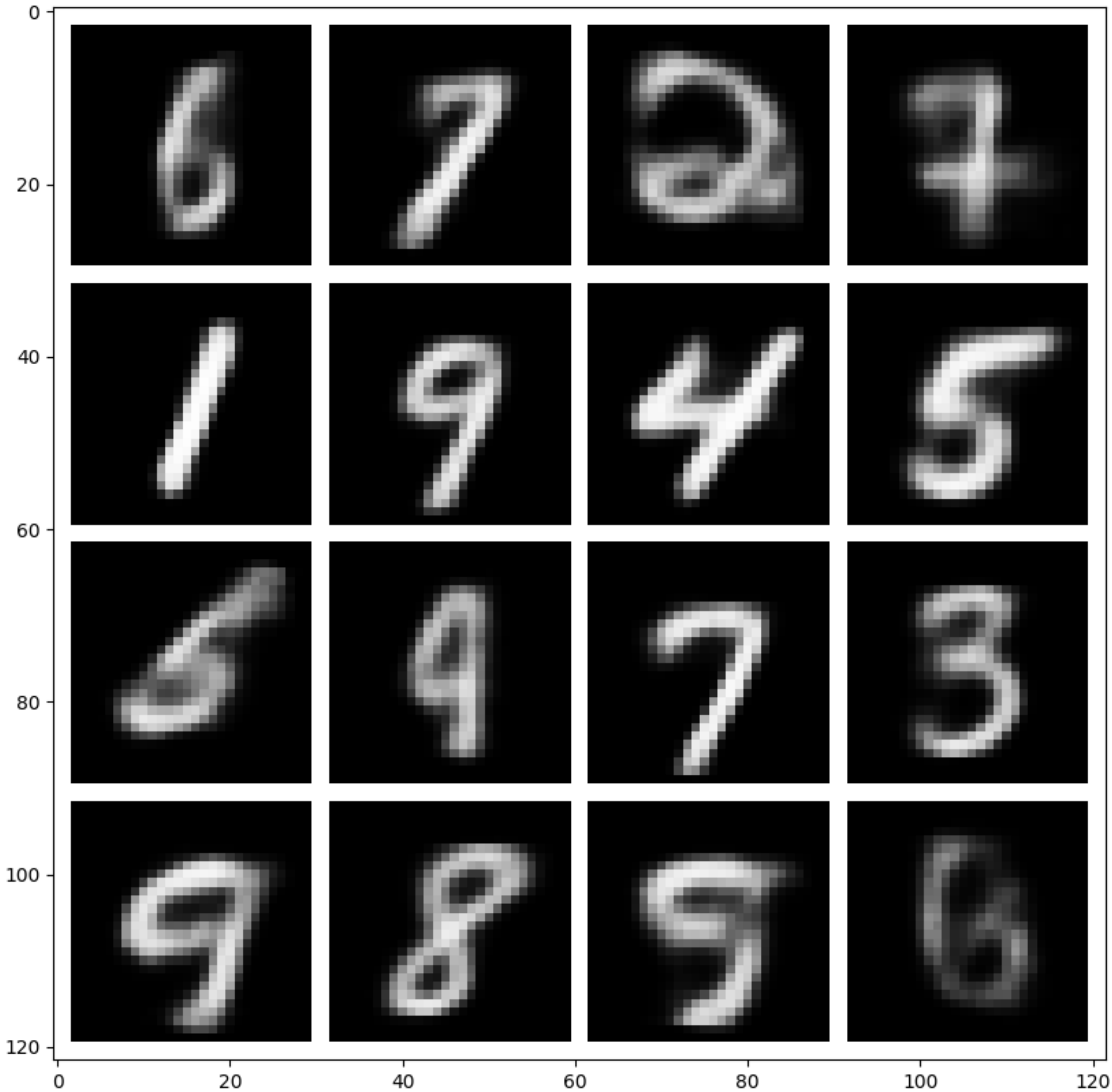


Let's look at some reconstructions and decoded embedding samples for this beta!

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

```
BEST beta =  0.005
```





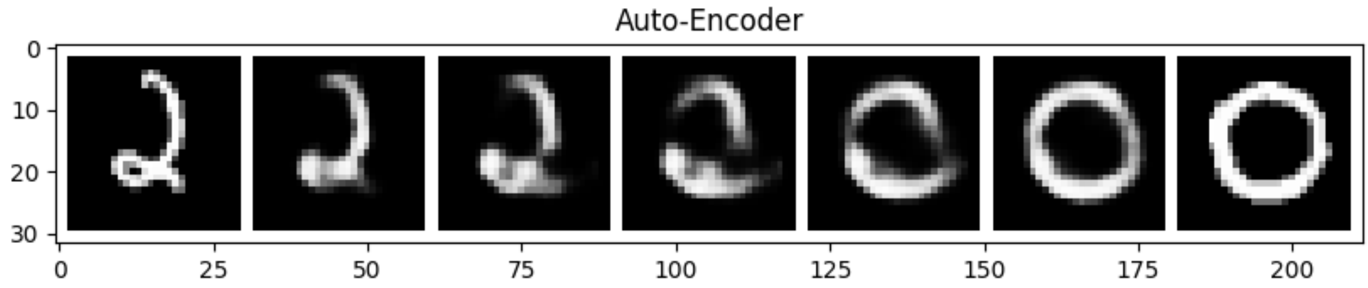
4. Embedding Space Interpolation [3pt]

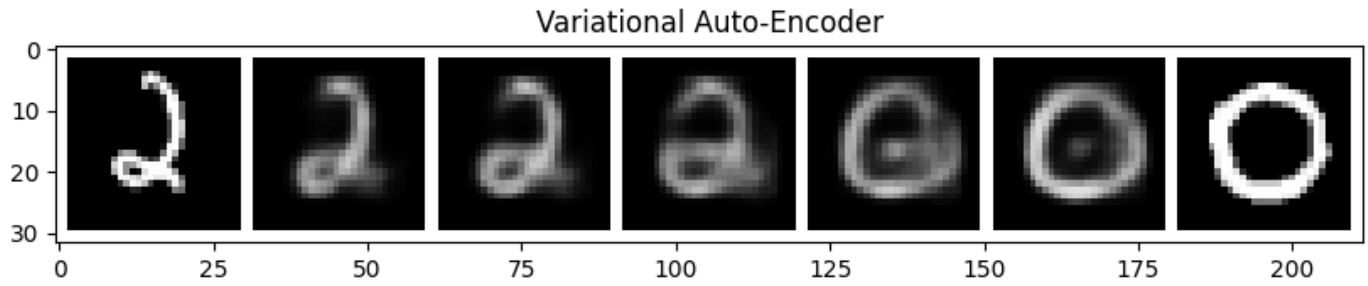
As mentioned in the introduction, AEs and VAEs cannot only be used to generate images, but also to learn low-dimensional representations of their inputs. In this final section we will investigate the representations we learned with both models by **interpolating in embedding space** between different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

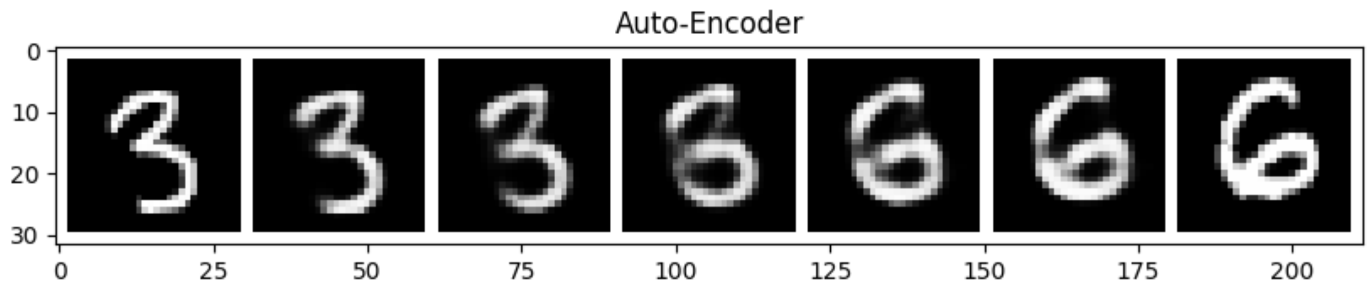
```
# Prob1-7
nz=32

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

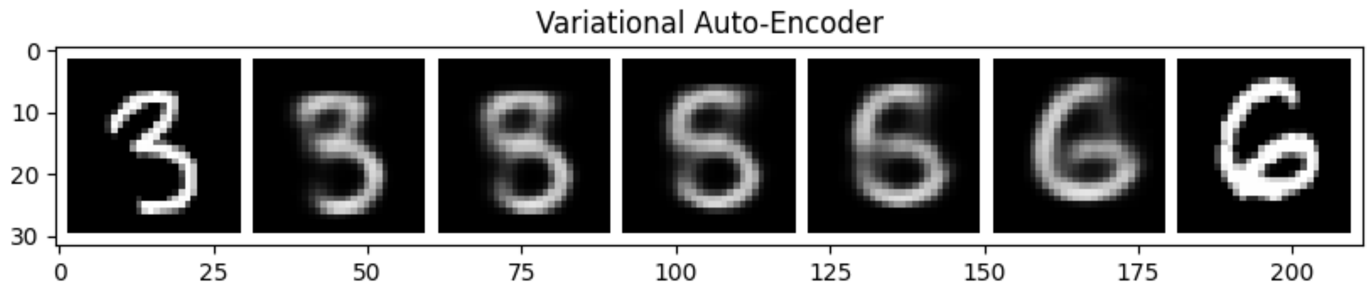
```
    for img_batch, label_batch in mnist_data_loader:
        for img, label in zip(img_batch, label_batch):
            if label == target_label:
                return img.to(device)

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

    # encode both images into embeddings (use posterior mean for interpolation)
    z_start = model.encoder(start_img[None].reshape(1,784))[..., :nz]
    z_end = model.encoder(end_img[None].reshape(1,784))[..., :nz]

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

    # decode interpolated embeddings (as a single batch)
    img_inter = model.decoder(torch.cat(z_inter))
    img_inter = img_inter.reshape(-1, 28, 28)

    # reshape result and display interpolation
    vis_imgs = torch.cat([start_img, img_inter, end_img]).reshape(-1,1,28,28)
    fig = plt.figure(figsize = (10, 10))
    ax1 = plt.subplot(111)
    ax1.imshow(torchvision.utils.make_grid(vis_imgs, nrow=N_INTER_STEPS+2,
pad_value=1.)\
                    .data.cpu().numpy().transpose(1, 2, 0), cmap='gray')
    plt.title(tag)
    plt.show()


### Interpolation 1
START_LABEL = 2 # ... TODO CHOOSE
END_LABEL = 0 # ... 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 = 6 # ... TODO CHOOSE
# sample two training images with given labels
start_img = get_image_with_label(START_LABEL)
end_img = get_image_with_label(END_LABEL)
```

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









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?
> 2. How do you expect these differences to affect the usefulness of the learned representation for downstream learning?
>   (max 300 words)
>
> **Answer**:
>
> 1. From the visulization results, the interpolations in the AE embedding space were generally more linear and less smooth compared to those in the VAE embedding space. Besides, the edges of AE around the digit are not continuous. The VAE embedding space interpolations were more continuous and better followed the underlying data distribution.
> 2. VAE can get better results than AE in most of the time. In general, VAEs may be more suitable for tasks with complex and difficult-to-model data distributions, while AEs may be more appropriate for simpler, more linear data distributions. Besdies, VAE is better suited for learning the underlying data distribution and for generating new data samples due to their ability to sample from the learned latent space distribution during decoding.

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.

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

    return onehot

# Let's define encoder and decoder networks

class CVAEEncoder(nn.Module):
    def __init__(self, nz, input_size, conditional, num_labels):
        super().__init__()
```

```
        self.input_size = input_size + num_labels if conditional else input_size
        self.num_labels = num_labels
        self.conditional = conditional

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

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

    def forward(self, x, c=None):
        ################################# TODO
#########################################
        # If using conditional VAE, concatenate x and a onehot version of c to create
#
        # the full input. Use function idx2onehot above.
#

 ################################################################################
        if self.conditional:
            c_onehot = idx2onehot(c, self.num_labels)  # convert to one-hot
            x = torch.cat([x, c_onehot], dim=-1)
```

```
 ###############################################################################
        return self.net(x)


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

 ###############################################################################
        self.net = nn.Sequential(
            nn.Linear(nz, 64),
            nn.ReLU(),
            nn.Linear(64, 256),
            nn.ReLU(),
            nn.Linear(256, output_size),
            nn.Sigmoid()
        )
        ################################ END TODO
#######################################

    def forward(self, z, c=None):
        ################################# TODO
#########################################
        # If using conditional VAE, concatenate z and a onehot version of c to create
#
        # the full embedding. Use function idx2onehot above.
#
```

```
 ################################################################################
        if self.conditional:
            c_onehot = idx2onehot(c, self.num_labels)  # convert to one-hot
            z = torch.cat([z, c_onehot], dim=-1)
        ################################ END TODO
#######################################

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


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

    def forward(self, x, c=None):
        if x.dim() > 2:
            x = x.view(-1, 28*28)

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

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

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

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

    def loss(self, x, outputs):
        ####################################### TODO
#######################################
        # Implement the loss computation of the VAE.
    #
        # HINT: Your code should implement the following steps:
   #
        #          1. compute the image reconstruction loss, similar to AE loss above
   #
```

```
        #          2. compute the KL divergence loss between the inferred posterior
   #
        #             distribution and a unit Gaussian prior; you can use the provided
    #
        #             function above for computing the KL divergence between two
Gaussians #
        #             parametrized by mean and log_sigma
    #
        # HINT: Make sure to compute the KL divergence in the correct order since it is
   #
        #       not symmetric!!  ie. KL(p, q) != KL(q, p)
   #

 ####################################################################################
        loss_func = nn.BCELoss(reduction='mean')
        # loss_func = nn.MSELoss()
        # x: [64, 784], y: [64, 1, 784] -> [64, 784]
        rec_loss = loss_func(outputs['rec'].squeeze(1), x)

        # print(outputs['q'].shape)
        mu1 = outputs['q'][:, :self.nz]  # [batch_size, nz] = [64, 32]
        log_sigma1 = outputs['q'][:, self.nz:]
        mu2 = torch.zeros_like(mu1).to(device)
        log_sigma2 = torch.zeros_like(log_sigma1).to(device)
        # make it as an scalar instead of a array
        kl_loss = torch.mean(torch.sum(kl_divergence(mu1, log_sigma1, mu2, log_sigma2),
dim=1), dim=0)
        #################################### END TODO
######################################

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

    def reconstruct(self, x, c=None):
        """Use mean of posterior estimate for visualization reconstruction."""
        ####################################### TODO
#######################################
        # This function is used for visualizing reconstructions of our VAE model. To
    #
        # obtain the maximum likelihood estimate we bypass the sampling procedure of
the   #
        # inferred latent and instead directly use the mean of the inferred posterior.
    #
        # HINT: encode the input image and then decode the mean of the posterior to
obtain #
        #       the reconstruction.
   #
```

```
 ####################################################################################
        q = self.encoder(x, c)
        mu, log_sigma = torch.chunk(q, 2, dim=-1)
        image = self.decoder(mu, c)
        #################################### END TODO
######################################
        return image
```

Setting up the CVAE Training loop

```
learning_rate = 1e-3
nz = 32

####################################### TODO #######################################
# Tune the beta parameter to obtain good training results. However, for the    #
# initial experiments leave beta = 0 in order to verify our implementation.        #
####################################################################################
epochs = 5 # works with fewer epochs than AE, VAE. we only test conditional samples.
beta = 0.005
#################################### END TODO ######################################

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

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

train_it = 0
rec_loss, kl_loss = [], []
print(f"Running {epochs} epochs with {beta=}")
for ep in range(epochs):
    print(f"Run Epoch {ep}")
####################################### TODO #######################################
# Implement the main training loop for the model.                                  #
# If using conditional VAE, remember to pass the conditional variable c to the     #
# forward pass                                                                     #
# HINT: Your training loop should sample batches from the data loader, run the     #
```

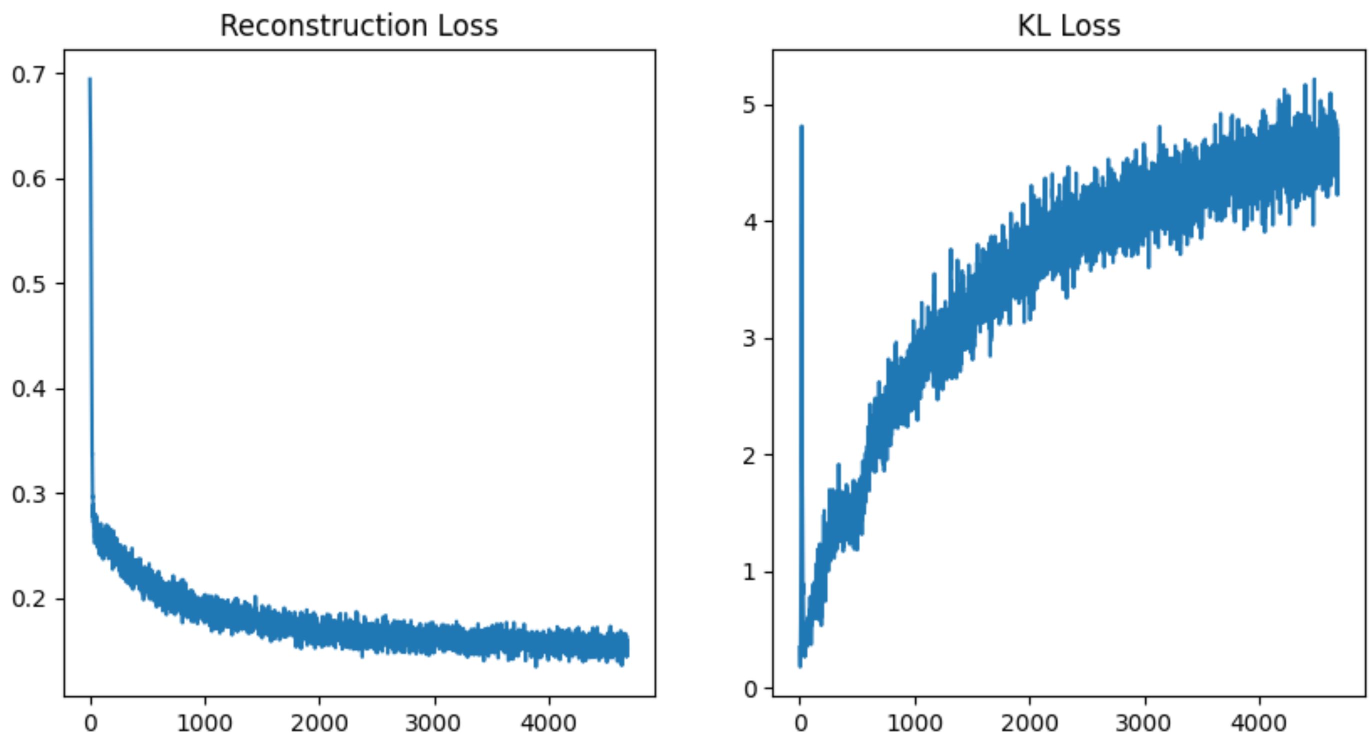
```
#       forward pass of the model, compute the loss, perform the backward pass and #
#       perform one gradient step with the optimizer.                              #
# HINT: Don't forget to erase old gradients before performing the backward pass.   #
# HINT: As before, we will use the loss() function of our model for computing the  #
#       training loss. It outputs the total training loss and a dict containing    #
#       the breakdown of reconstruction and KL loss.                               #
####################################################################################
    for X_train, y_train in mnist_data_loader:
        X_train, y_train = X_train.to(device), y_train.to(device)
        optimizer.zero_grad()
        X_train = X_train.reshape([batch_size, in_size])
        # falten X_train: [64, 28, 28] -> [64, 784]
        # print(y_train.shape)
        y_pred = cvae_model(X_train, y_train)
        # print(y_pred['rec'].shape, X_train.shape)
        total_loss, losses = cvae_model.loss(X_train, y_pred)
        # losses['rec_loss'] = losses['rec_loss'].detach().cpu()
        # losses['kl_loss'] = losses['kl_loss'].detach().cpu()
        # print(total_loss.shape)

        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=0.005
Run Epoch 0
```

```
It 0: Total Loss: 0.6961221098899841,    Rec Loss: 0.6943913698196411,   KL Loss:
0.3461534380912781
It 100: Total Loss: 0.26270538568496704,     Rec Loss: 0.260428249835968,    KL Loss:
0.4554291367530823
It 200: Total Loss: 0.23876087367534637,     Rec Loss: 0.23477250337600708,  KL Loss:
0.797675371170044
It 300: Total Loss: 0.2432468831539154,      Rec Loss: 0.23577070236206055,  KL Loss:
1.4952359199523926
It 400: Total Loss: 0.22452087700366974,     Rec Loss: 0.21745862066745758,  KL Loss:
1.4124525785446167
It 500: Total Loss: 0.22840912640094757,     Rec Loss: 0.2225133180618286,   KL Loss:
1.1791630983352661
It 600: Total Loss: 0.21754541993141174,     Rec Loss: 0.20786292850971222,  KL Loss:
1.9364993572235107
It 700: Total Loss: 0.20954541862010956,     Rec Loss: 0.19906261563301086,  KL Loss:
2.0965616703033447
It 800: Total Loss: 0.2016667127609253,      Rec Loss: 0.19087526202201843,  KL Loss:
2.1582891941070557
It 900: Total Loss: 0.20380091667175293,     Rec Loss: 0.1916627287864685,   KL Loss:
2.427638530731201
Run Epoch 1
It 1000: Total Loss: 0.1933600753545761,     Rec Loss: 0.18044421076774597,  KL Loss:
2.5831735134124756
It 1100: Total Loss: 0.1996302604675293,     Rec Loss: 0.18462316691875458,  KL Loss:
3.0014185905456543
It 1200: Total Loss: 0.190623477101326,      Rec Loss: 0.17829325795173645,  KL Loss:
2.4660449028015137
It 1300: Total Loss: 0.1867140382528305,     Rec Loss: 0.17216433584690094,  KL Loss:
2.9099395275115967
It 1400: Total Loss: 0.18920038640499115,    Rec Loss: 0.17551429569721222,  KL Loss:
2.737218141555786
It 1500: Total Loss: 0.17712968587875366,    Rec Loss: 0.1616891324520111,   KL Loss:
3.08811092376709
It 1600: Total Loss: 0.19290322065353394,    Rec Loss: 0.1765541434288025,   KL Loss:
3.269814968109131
It 1700: Total Loss: 0.19574669003486633,    Rec Loss: 0.17641183733940125,  KL Loss:
3.866971254348755
It 1800: Total Loss: 0.1918659508228302,     Rec Loss: 0.17484483122825623,  KL Loss:
3.4042232036590576
Run Epoch 2
It 1900: Total Loss: 0.1924508512020111,     Rec Loss: 0.17469161748886108,  KL Loss:
3.551847457885742
It 2000: Total Loss: 0.18140479922294617,    Rec Loss: 0.16491581499576569,  KL Loss:
3.297797203063965
It 2100: Total Loss: 0.18916717171669006,    Rec Loss: 0.17058850824832916,  KL Loss:
3.715733289718628
It 2200: Total Loss: 0.19339223206043243,    Rec Loss: 0.17240621149539948,  KL Loss:
4.19720458984375
```

```
It 2300: Total Loss: 0.18865463137626648,    Rec Loss: 0.1688929945230484,   KL Loss:
3.9523258209228516
It 2400: Total Loss: 0.1980040967464447,     Rec Loss: 0.177912637591362,    KL Loss:
4.018292427062988
It 2500: Total Loss: 0.19335682690143585,    Rec Loss: 0.17468349635601044,  KL Loss:
3.734665870666504
It 2600: Total Loss: 0.18068158626556396,    Rec Loss: 0.16160601377487183,  KL Loss:
3.8151142597198486
It 2700: Total Loss: 0.18145106732845306,    Rec Loss: 0.16110093891620636,  KL Loss:
4.070026397705078
It 2800: Total Loss: 0.18490146100521088,    Rec Loss: 0.1643911898136139,   KL Loss:
4.102055549621582
Run Epoch 3
It 2900: Total Loss: 0.18923825025558472,    Rec Loss: 0.16879847645759583,  KL Loss:
4.087955474853516
It 3000: Total Loss: 0.17653420567512512,    Rec Loss: 0.1568654477596283,   KL Loss:
3.9337525367736816
It 3100: Total Loss: 0.17404583096504211,    Rec Loss: 0.15342868864536285,  KL Loss:
4.123428821563721
It 3200: Total Loss: 0.17988714575767517,    Rec Loss: 0.15814469754695892,  KL Loss:
4.348489761352539
It 3300: Total Loss: 0.17844687402248383,    Rec Loss: 0.15773390233516693,  KL Loss:
4.142595291137695
It 3400: Total Loss: 0.17906738817691803,    Rec Loss: 0.15721042454242706,  KL Loss:
4.371393203735352
It 3500: Total Loss: 0.18766286969184875,    Rec Loss: 0.16648314893245697,  KL Loss:
4.235945224761963
It 3600: Total Loss: 0.1857607513666153,     Rec Loss: 0.1640034019947052,   KL Loss:
4.351469039916992
It 3700: Total Loss: 0.18224720656871796,    Rec Loss: 0.16147513687610626,  KL Loss:
4.154415130615234
Run Epoch 4
It 3800: Total Loss: 0.186319962143898,      Rec Loss: 0.16309502720832825,  KL Loss:
4.644988059997559
It 3900: Total Loss: 0.17574700713157654,    Rec Loss: 0.15413954854011536,  KL Loss:
4.3214921951293945
It 4000: Total Loss: 0.19297140836715698,    Rec Loss: 0.1698864847421646,   KL Loss:
4.616983890533447
It 4100: Total Loss: 0.17652323842048645,    Rec Loss: 0.15436682105064392,  KL Loss:
4.431282043457031
It 4200: Total Loss: 0.18403884768486023,    Rec Loss: 0.1607266217470169,   KL Loss:
4.662446022033691
It 4300: Total Loss: 0.1708582639694214,     Rec Loss: 0.1481492668390274,   KL Loss:
4.541799545288086
It 4400: Total Loss: 0.18357664346694946,    Rec Loss: 0.15911288559436798,  KL Loss:
4.892752647399902
It 4500: Total Loss: 0.17335020005702972,    Rec Loss: 0.15160426497459412,  KL Loss:
4.349187850952148
```

```
It 4600: Total Loss: 0.17443019151687622,    Rec Loss: 0.1515093594789505,    KL Loss: 4.58416748046875
Done!
```

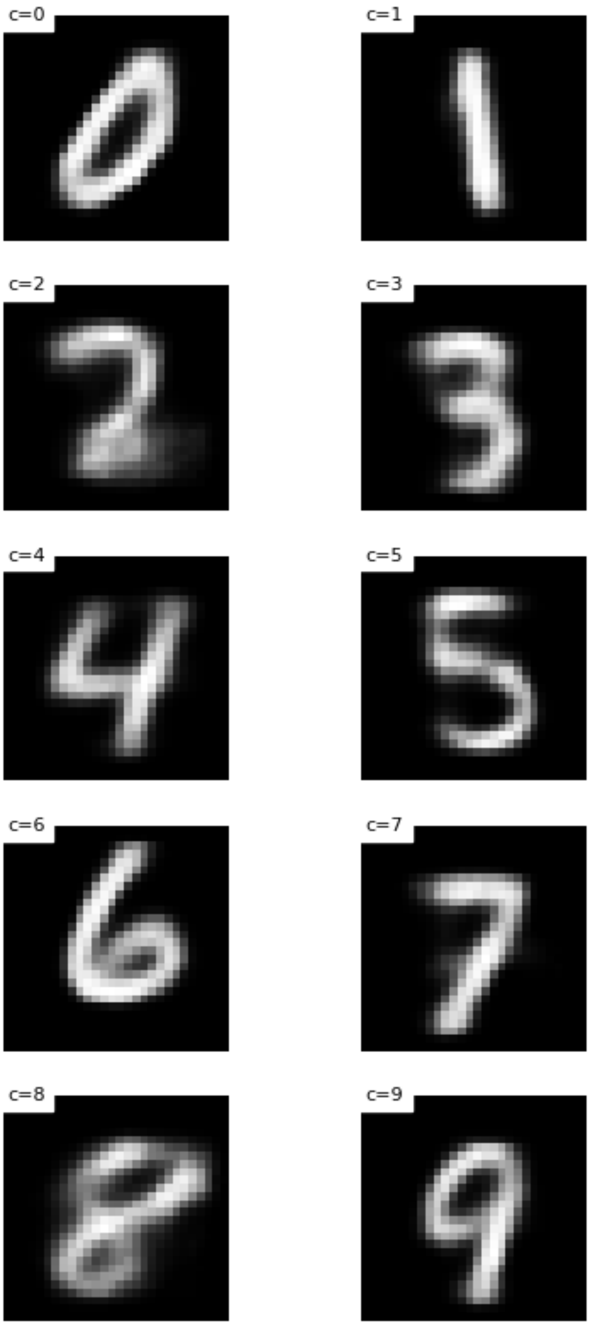


Verifying conditional samples from CVAE [6 pt]

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

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

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

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



Submission Instructions

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