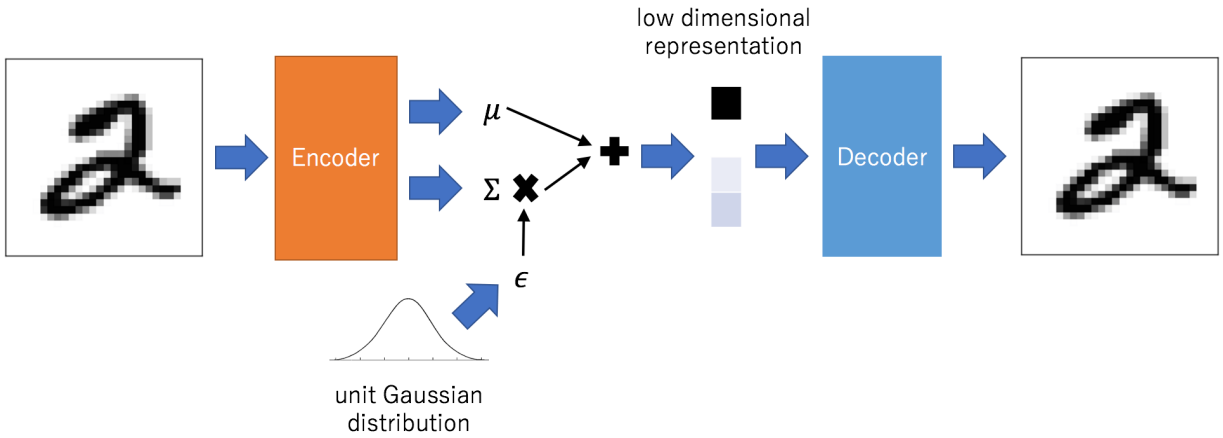
Problem 1 - Variational Auto-Encoder (VAE)

Open in Colab

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



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

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

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

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

1. MNIST Dataset

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

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

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

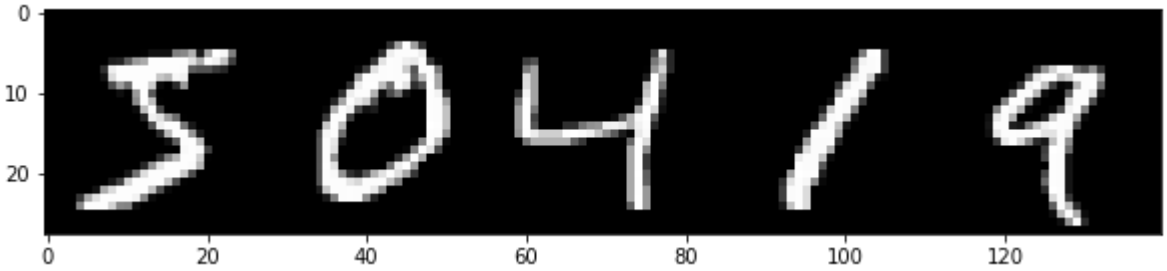
In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline

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

In [ ]:
```
import torch
import torchvision

# this will automatically download the MNIST training set
mnist_train = torchvision.datasets.MNIST(root='./data',
                                         train=True,
                                         download=True,
                                         transform=torchvision.transforms.ToTens
print("\n Download complete! Downloaded {} training examples!".format(len(mnist_
```

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

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

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

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

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

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

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

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


 Download complete! Downloaded 60000 training examples!
```

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np

# Let's display some of the training samples.
sample_images = []
mnist_it = iter(mnist_train)  # create simple iterator, later we will use proper
for _ in range(5):
```

```
  sample = next(mnist_it)     # samples a tuple (image, label)
  sample_images.append(sample[0][0].data.cpu().numpy())

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



2. Auto-Encoder

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

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

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

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

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

Defining the Auto-Encoder Architecture [6pt]

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

# Let's define encoder and decoder networks
#########################################################################
# Encoder Architecture:                                                 #
#   - Conv2d, hidden units: 32, output resolution: 14x14, kernel: 4     #
#   - LeakyReLU                                                         #
#   - Conv2d, hidden units: 64, output resolution: 7x7, kernel: 4       #
#   - BatchNorm2d                                                       #
#   - LeakyReLU                                                         #
#   - Conv2d, hidden units: 128, output resolution: 3x3, kernel: 3      #
#   - BatchNorm2d                                                       #
#   - LeakyReLU                                                         #
#   - Conv2d, hidden units: 256, output resolution: 1x1, kernel: 3      #
#   - BatchNorm2d                                                       #
#   - LeakyReLU                                                         #
#   - Flatten                                                           #
#   - Linear, output units: nz (= representation dimensionality)        #
#########################################################################

class Encoder(nn.Module):
  def __init__(self, nz):
    super().__init__()
    ################################## TODO ##########################################
    # Create the network architecture using a nn.Sequential module wrapper.
    # All convolutional layers should also learn a bias.
    # HINT: use the given information to compute stride and padding
    #       for each convolutional layer. Verify the shapes of intermediate laye
    #       by running partial networks (with the next cell) and visualizing the
    #       output shapes.
    ##################################################################################
    self.net = nn.Sequential(
        # add your network layers here
        # ...
        nn.Conv2d(in_channels=1, out_channels=32, kernel_size=4, stride=2, paddi
        nn.LeakyReLU(),
        nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2, padd
        nn.BatchNorm2d(64),
        nn.LeakyReLU(),
        nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=2, pad
        nn.BatchNorm2d(128),
        nn.LeakyReLU(),
        nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=2, pa
        nn.BatchNorm2d(256),
        nn.LeakyReLU(),
        nn.Flatten(),
        nn.Linear(in_features=256, out_features=nz, bias=True)
    )
    ################################## END TODO ######################################

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


#########################################################################
# Decoder Architecture (mirrors encoder architecture):                  #
#   - Linear, output units: 256                                         #
#   - Reshape, output shape: (256, 1, 1)                                #
#   - BatchNorm2d                                                       #
#   - LeakyReLU                                                         #
```

```
#   - ConvT2d, hidden units: 128, output resolution: 3x3, kernel: 3 #
#   - BatchNorm2d                                                   #
#   - LeakyReLU                                                     #
#   - ConvT2d, hidden units: 64, output resolution: 7x7, kernel: 3  #
#   - ...                                                           #
#   - ...                                                           #
#   - ConvT2d, output units: 1, output resolution: 28x28, kernel: 4 #
#   - Sigmoid (to limit output in range [0...1])                    #
#####################################################################

class Decoder(nn.Module):
  def __init__(self, nz):
    super().__init__()
    ################################## TODO ##########################################
    # Create the network architecture using a nn.Sequential module wrapper.
    # Again, all (transposed) convolutional layers should also learn a bias.
    # We need to separate the intial linear layer into a separate variable since
    # nn.Sequential does not support reshaping. Instead the "Reshape" is perform
    # in the forward() function below and does not need to be added to self.net
    # HINT: use the class nn.ConvTranspose2d for the transposed convolutions.
    #       Verify the shapes of intermediate layers by running partial networks
    #       (using the next cell) and visualizing the output shapes.
    ##################################################################################
    self.map = nn.Linear(in_features=nz, out_features=256, bias=True)   # for in
    self.net = nn.Sequential(
        # add your network layers here
        # ...
        nn.BatchNorm2d(256),
        nn.LeakyReLU(),
        nn.ConvTranspose2d(in_channels=256, out_channels=128, kernel_size=3, str
        nn.BatchNorm2d(128),
        nn.LeakyReLU(),
        nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=3, stri
        nn.BatchNorm2d(64),
        nn.LeakyReLU(),
        nn.ConvTranspose2d(in_channels=64, out_channels=32, kernel_size=4, strid
        nn.LeakyReLU(),
        nn.ConvTranspose2d(in_channels=32, out_channels=1, kernel_size=4, stride
        nn.Sigmoid()
    )
    ################################## END TODO ######################################

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

Testing the Auto-Encoder Forward Pass [1pt]

In [ ]:

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

######################################## TODO ######################################
# Create a PyTorch DataLoader object for efficiently generating training batches
# Make sure that the data loader automatically shuffles the training dataset.
# HINT: The DataLoader wraps the MNIST dataset class we created earlier.
#       Use the given batch_size and number of data loading workers when creatin
```

```
#       the DataLoader.
################################################################################
mnist_data_loader = torch.utils.data.DataLoader(mnist_train, batch_size=batch_si
################################### END TODO ###################################

# now we can run a forward pass for encoder and decoder and check the produced s
nz = 64          # dimensionality of the learned embedding
encoder = Encoder(nz)
decoder = Decoder(nz)
for sample_img, sample_label in mnist_data_loader:
  enc = encoder(sample_img)
  print("Shape of encoding vector (should be [batch_size, nz]): {}".format(enc.s
  dec = decoder(enc)
  print("Shape of decoded image (should be [batch_size, 1, 28, 28]): {}".format(
  break
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserW
arning: This DataLoader will create 4 worker processes in total. Our suggested m
ax number of worker in current system is 2, which is smaller than what this Data
Loader is going to create. Please be aware that excessive worker creation might
get DataLoader running slow or even freeze, lower the worker number to avoid pot
ential slowness/freeze if necessary.
  cpuset_checked))
Shape of encoding vector (should be [batch_size, nz]): torch.Size([64, 64])
Shape of decoded image (should be [batch_size, 1, 28, 28]): torch.Size([64, 1, 2
8, 28])
```

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

In [ ]:

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

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

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

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

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

In [ ]:

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

# build AE model
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')   # use
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
```

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

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

    opt.zero_grad()
    target = ae_model.forward(sample_img)
    rec_loss = criterion(sample_img, target)
    rec_loss.backward()
    opt.step()
    ###################################### END TODO #################################

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

print("Done!")
```

```
Run Epoch 0
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserW
arning: This DataLoader will create 4 worker processes in total. Our suggested m
ax number of worker in current system is 2, which is smaller than what this Data
Loader is going to create. Please be aware that excessive worker creation might
get DataLoader running slow or even freeze, lower the worker number to avoid pot
ential slowness/freeze if necessary.
  cpuset_checked))
It 0: Reconstruction Loss: 0.29177120327949524
It 100: Reconstruction Loss: 0.02578936330974102
It 200: Reconstruction Loss: 0.020548971369862556
It 300: Reconstruction Loss: 0.014111995697021484
It 400: Reconstruction Loss: 0.011732138693332672
It 500: Reconstruction Loss: 0.012748383916914463
It 600: Reconstruction Loss: 0.010603846050798893
```

```
It 700: Reconstruction Loss: 0.010678308084607124
It 800: Reconstruction Loss: 0.008142291568219662
It 900: Reconstruction Loss: 0.008616521023213863
Run Epoch 1
It 1000: Reconstruction Loss: 0.008870997466146946
It 1100: Reconstruction Loss: 0.009555907920002937
It 1200: Reconstruction Loss: 0.009623277001082897
It 1300: Reconstruction Loss: 0.007725004572421312
It 1400: Reconstruction Loss: 0.0072948825545609
It 1500: Reconstruction Loss: 0.0073433928191661835
It 1600: Reconstruction Loss: 0.007074765395373106
It 1700: Reconstruction Loss: 0.0069890934973955154
It 1800: Reconstruction Loss: 0.007780701853334904
Run Epoch 2
It 1900: Reconstruction Loss: 0.006261726375669241
It 2000: Reconstruction Loss: 0.006312916520982981
It 2100: Reconstruction Loss: 0.006624008063226938
It 2200: Reconstruction Loss: 0.006727360188961029
It 2300: Reconstruction Loss: 0.00625053234398365
It 2400: Reconstruction Loss: 0.00694984570145607
It 2500: Reconstruction Loss: 0.0068231429904699326
It 2600: Reconstruction Loss: 0.006343253422528505
It 2700: Reconstruction Loss: 0.006886878050863743
It 2800: Reconstruction Loss: 0.006354279350489378
Run Epoch 3
It 2900: Reconstruction Loss: 0.006388189736753702
It 3000: Reconstruction Loss: 0.005587352439761162
It 3100: Reconstruction Loss: 0.005983870010823011
It 3200: Reconstruction Loss: 0.006008658558130264
It 3300: Reconstruction Loss: 0.006128655280917883
It 3400: Reconstruction Loss: 0.0058621070347726345
It 3500: Reconstruction Loss: 0.005859750788658857
It 3600: Reconstruction Loss: 0.006796752568334341
It 3700: Reconstruction Loss: 0.006117346230894327
Run Epoch 4
It 3800: Reconstruction Loss: 0.005634102039039135
It 3900: Reconstruction Loss: 0.004877128172665834
It 4000: Reconstruction Loss: 0.005274308379739523
It 4100: Reconstruction Loss: 0.0047994013875722885
It 4200: Reconstruction Loss: 0.005949391983449459
It 4300: Reconstruction Loss: 0.0054855686612427235
It 4400: Reconstruction Loss: 0.004812504630535841
It 4500: Reconstruction Loss: 0.004748622886836529
It 4600: Reconstruction Loss: 0.004370357375591993
Run Epoch 5
It 4700: Reconstruction Loss: 0.005479232873767614
It 4800: Reconstruction Loss: 0.005228730849921703
It 4900: Reconstruction Loss: 0.004014940001070499
It 5000: Reconstruction Loss: 0.004477561451494694
It 5100: Reconstruction Loss: 0.005650628358125687
It 5200: Reconstruction Loss: 0.0050682746805250645
It 5300: Reconstruction Loss: 0.004934049677103758
It 5400: Reconstruction Loss: 0.004681332502514124
It 5500: Reconstruction Loss: 0.004286372568458319
It 5600: Reconstruction Loss: 0.005523474887013435
Run Epoch 6
It 5700: Reconstruction Loss: 0.004918365739285946
It 5800: Reconstruction Loss: 0.0046380930580198765
It 5900: Reconstruction Loss: 0.004395829513669014
It 6000: Reconstruction Loss: 0.0048562632873654366
```

```
It 6100: Reconstruction Loss: 0.004582577850669622
It 6200: Reconstruction Loss: 0.004858891945332289
It 6300: Reconstruction Loss: 0.004552130587399006
It 6400: Reconstruction Loss: 0.004578228108584881
It 6500: Reconstruction Loss: 0.0047793095000088215
Run Epoch 7
It 6600: Reconstruction Loss: 0.004723196849226952
It 6700: Reconstruction Loss: 0.004422673024237156
It 6800: Reconstruction Loss: 0.004474954679608345
It 6900: Reconstruction Loss: 0.004708188585937023
It 7000: Reconstruction Loss: 0.004184214398264885
It 7100: Reconstruction Loss: 0.004332628566771746
It 7200: Reconstruction Loss: 0.004429869819432497
It 7300: Reconstruction Loss: 0.005063371267169714
It 7400: Reconstruction Loss: 0.004203329794108868
It 7500: Reconstruction Loss: 0.005644906312227249
Run Epoch 8
It 7600: Reconstruction Loss: 0.004139378201216459
It 7700: Reconstruction Loss: 0.0049005853943526745
It 7800: Reconstruction Loss: 0.0042317709885537624
It 7900: Reconstruction Loss: 0.005367446690797806
It 8000: Reconstruction Loss: 0.005605527199804783
It 8100: Reconstruction Loss: 0.004418946336954832
It 8200: Reconstruction Loss: 0.00427576620131731
It 8300: Reconstruction Loss: 0.004392936360090971
It 8400: Reconstruction Loss: 0.003684197785332799
Run Epoch 9
It 8500: Reconstruction Loss: 0.003888723673298955
It 8600: Reconstruction Loss: 0.004150626249611378
It 8700: Reconstruction Loss: 0.004330557771027088
It 8800: Reconstruction Loss: 0.004583722446113825
It 8900: Reconstruction Loss: 0.004036941099911928
It 9000: Reconstruction Loss: 0.004215131048113108
It 9100: Reconstruction Loss: 0.003929841332137585
It 9200: Reconstruction Loss: 0.004221653565764427
It 9300: Reconstruction Loss: 0.004226346500217915
Done!
```

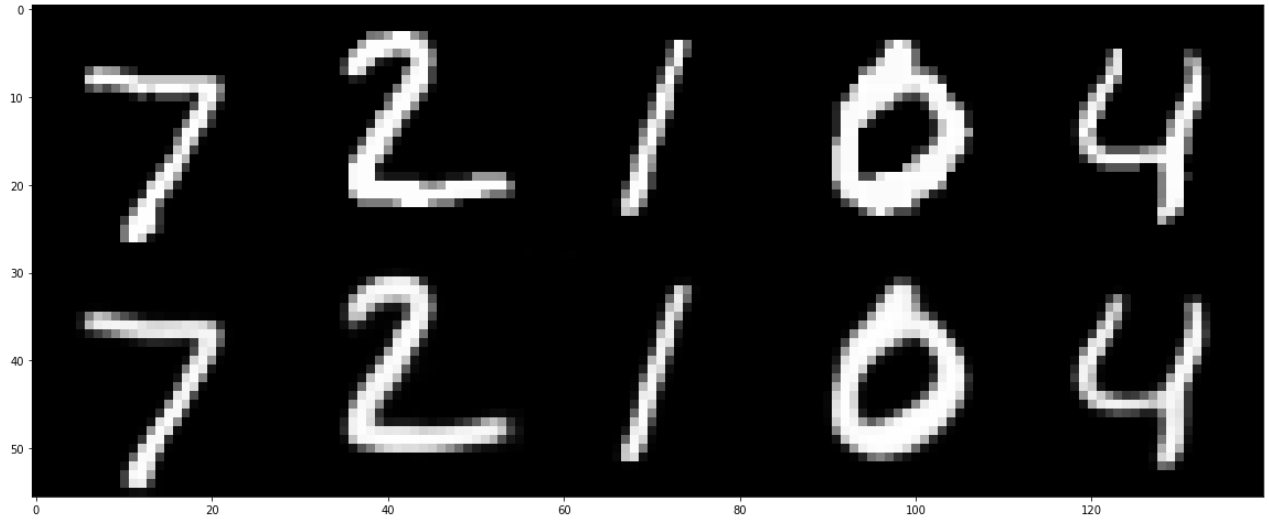
Verifying reconstructions [0pt]

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

In [ ]:

```
# visualize test data reconstructions
def vis_reconstruction(model):
  # download MNIST test set + build Dataset object
  mnist_test = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          download=True,
                                          transform=torchvision.transforms.ToTen
  mnist_test_iter = iter(mnist_test)
  model.eval()      # set model in evalidation mode (eg freeze batchnorm params)
  input_imgs, test_reconstructions = [], []
  for _ in range(5):
    input_img = np.asarray(next(mnist_test_iter)[0])
    reconstruction = model.reconstruct(torch.tensor(input_img[None], device=devi
```

```
    input_imgs.append(input_img[0])
    test_reconstructions.append(reconstruction[0, 0].data.cpu().numpy())

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

vis_reconstruction(ae_model)
```



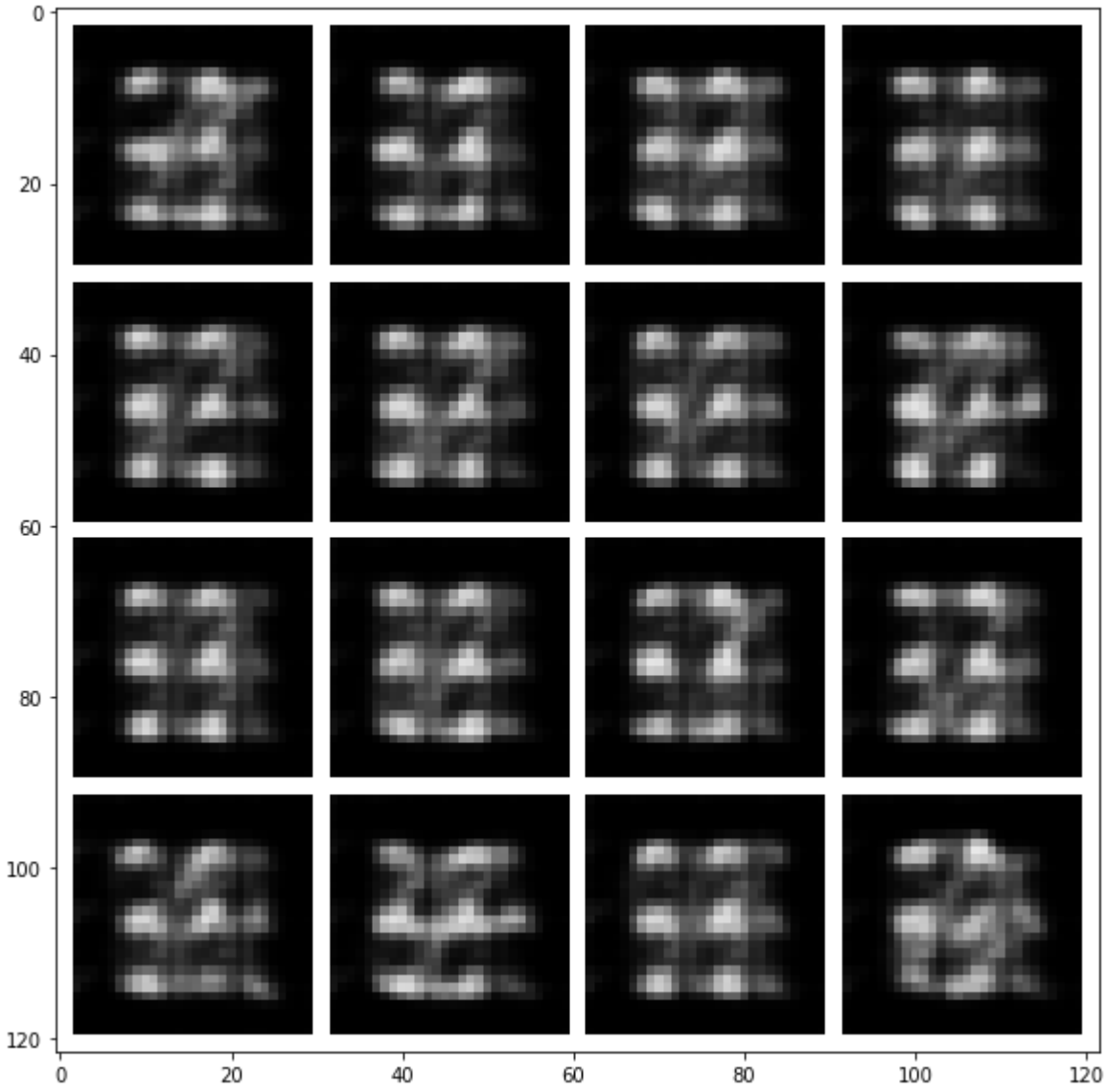
Sampling from the Auto-Encoder [2pt]

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

In [ ]:

```
# we will sample N embeddings, then decode and visualize them
def vis_samples(model):
  ######################################## TODO ###################################
  # Sample embeddings from a diagonal unit Gaussian distribution and decode them
  # using the model.
  # HINT: The sampled embeddings should have shape [batch_size, nz]. Diagonal un
  #       Gaussians have mean 0 and a covariance matrix with ones on the diagona
  #       and zeros everywhere else.
  # HINT: If you are unsure whether you sampled the correct distribution, you ca
  #       sample a large batch and compute the empirical mean and variance using
  #       .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
  decoded_samples = model.decoder(sampled_embeddings)       # decoder output ima
  ################################### 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
              .data.cpu().numpy().transpose(1, 2, 0), cmap='gray')
```

```
  plt.show()

vis_samples(ae_model)
```



> **Inline Question: Describe your observations, why do you think they occur? [2pt]** \ (please limit your answer to <150 words) \ **Answer:** The images generated by the model are not distinct. To be specific the model is trying to generate digits but it is not clearly able to generate a single digit. It seems that two or more digits which are partially generated are overlapping. For example I can see, traces of 3 and 5 overlapping, or may be 4 is converting to 7 or vice versa. Another example is first image in second row has traces of 3 and 6. It can also be noted that the images generated are not in shape and are very blurry.

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):

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

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

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

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

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

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

Reparametrization Trick

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

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

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

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

Note: While in the equations above the encoder network parametrizes the standard deviation σ_q of the Gaussian posterior distribution, in practice we usually parametrize the **logarithm of the**

standard deviation $\log \sigma_q$ for numerical stability. Before sampling z we will then exponentiate the network's output to obtain σ_q.

Defining the VAE Model [7pt]

In [ ]:

```
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 * torch.exp(log_sigma2) ** 2) - 0.5


class VAE(nn.Module):
  def __init__(self, nz, beta=1.0):
    super().__init__()
    self.beta = beta          # factor trading off between two loss components
    ######################################## TODO ##################################
    # Instantiate Encoder and Decoder.
    # HINT: Remember that the encoder is now parametrizing a Gaussian distributi
    #       mean and log_sigma, so the dimensionality of the output embedding ne
    #       double.
    ################################################################################
    self.encoder = Encoder(nz * 2)
    self.decoder = Decoder(nz)
    self.nz = nz
    ###################################### END TODO ################################

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

    # sample latent variable z with reparametrization
    mu = q[:, : self.nz]
    log_sigma = q[:, self.nz: ]

    prior = torch.randn_like(log_sigma)
    z = mu + (prior * log_sigma)       # batch of sampled embeddings

    # compute reconstruction
    reconstruction = self.decoder(z)   # decoder reconstruction from embedding
    ###################################### END TODO ################################

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

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

```
    #          2. compute the KL divergence loss between the inferred posterior
    #             distribution and a unit Gaussian prior; you can use the provid
    #             function above for computing the KL divergence between two Gau
    #             parametrized by mean and log_sigma
    # HINT: Make sure to compute the KL divergence in the correct order since it
    #       not symmetric, ie. KL(p, q) != KL(q, p)!
    ###############################################################################
    # compute reconstruction loss
    criterion = nn.MSELoss()
    rec_loss = criterion(x, outputs['rec'])

    # compute KL divergence loss
    q = outputs['q']
    mu1 = q[:, : self.nz]
    log_sigma1 = q[:, self.nz: ]

    # Create a unit gaussian prior
    mu2 = torch.zeros_like(mu1)
    log_sigma2 = torch.zeros_like(log_sigma1)

    kl_loss = kl_divergence(mu1, log_sigma1, mu2, log_sigma2).sum(dim = 1).mean(
    ################################### END TODO ##################################

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

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

    mu1 = q[:, : self.nz]

    reconstruction = self.decoder(mu1)
    ################################### END TODO ##################################
    return reconstruction
```

Setting up the VAE Training Loop [4pt]

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

In [ ]:

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

###################################### TODO ###################################
# Tune the beta parameter to obtain good VAE training results. However, for the
# initial experiments leave beta = 0 in order to verify our implementation.
###############################################################################
epochs = 20         # using 5 epochs is sufficient for the first two experiments
                    # for the experiment where you tune beta, 20 epochs are appro
```

```
beta = 1e-3
################################### 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 g

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

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

    # move to device
    sample_img = sample_img.to(device)

    opt.zero_grad()
    outputs = vae_model(sample_img)
    total_loss, losses = vae_model.loss(sample_img, outputs)

    losses['rec_loss'] = losses['rec_loss'].cpu().item()
    losses['kl_loss'] = losses['kl_loss'].cpu().item()

    total_loss.backward()
    opt.step()
    ################################### END TODO ###################################

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

print("Done!")

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

```
ax2 = plt.subplot(122)
ax2.plot(kl_loss)
ax2.title.set_text("KL Loss")
plt.show()
```

```
Run Epoch 0
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserW
arning: This DataLoader will create 4 worker processes in total. Our suggested m
ax number of worker in current system is 2, which is smaller than what this Data
Loader is going to create. Please be aware that excessive worker creation might
get DataLoader running slow or even freeze, lower the worker number to avoid pot
ential slowness/freeze if necessary.
  cpuset_checked))
It 0: Total Loss: 0.26922595500946045,   Rec Loss: 0.2501518428325653,   KL Los
s: 19.074108123779297
It 100: Total Loss: 0.03411846235394478,         Rec Loss: 0.03188098967075348,
KL Loss: 2.237471580505371
It 200: Total Loss: 0.02382558397948742,         Rec Loss: 0.021717246621847153,
KL Loss: 2.108336925506592
It 300: Total Loss: 0.02047553099691868,         Rec Loss: 0.0181252583861351,
KL Loss: 2.3502731323242188
It 400: Total Loss: 0.01757904328405857,         Rec Loss: 0.01583368144929409,
KL Loss: 1.7453625202178955
It 500: Total Loss: 0.015067837201058865,        Rec Loss: 0.013248082250356674,
KL Loss: 1.8197544813156128
It 600: Total Loss: 0.016139546409249306,        Rec Loss: 0.014799226075410843,
KL Loss: 1.3403207063674927
It 700: Total Loss: 0.013345640152692795,        Rec Loss: 0.01235753670334816,
KL Loss: 0.988103449344635
It 800: Total Loss: 0.011660315096378326,        Rec Loss: 0.01073085144162178,
KL Loss: 0.9294639229774475
It 900: Total Loss: 0.012558987364172935,        Rec Loss: 0.01145233865827322,
KL Loss: 1.1066490411758423
Run Epoch 1
It 1000: Total Loss: 0.010426237247884274,       Rec Loss: 0.009481783024966717,
KL Loss: 0.9444541335105896
It 1100: Total Loss: 0.010139930061995983,       Rec Loss: 0.009146393276751041,
KL Loss: 0.9935365915298462
It 1200: Total Loss: 0.011834647506475449,       Rec Loss: 0.01092914305627346,
KL Loss: 0.9055041074752808
It 1300: Total Loss: 0.011315342038869858,       Rec Loss: 0.010607816278934479,
KL Loss: 0.7075258493423462
It 1400: Total Loss: 0.01010518055409193,        Rec Loss: 0.009292833507061005,
KL Loss: 0.8123472332954407
It 1500: Total Loss: 0.010183797217905521,       Rec Loss: 0.009600727818906307,
KL Loss: 0.583069384098053
It 1600: Total Loss: 0.01091449148952961,        Rec Loss: 0.010364306159317493,
KL Loss: 0.550184965133667
It 1700: Total Loss: 0.009113453328609467,       Rec Loss: 0.00856081023812294,
KL Loss: 0.552642822265625
It 1800: Total Loss: 0.00835987832397225,        Rec Loss: 0.007901393808424473,
KL Loss: 0.4584846496582031
Run Epoch 2
It 1900: Total Loss: 0.00907213520258665,        Rec Loss: 0.008518517017364502,
KL Loss: 0.553618311882019
It 2000: Total Loss: 0.00788838043808937,        Rec Loss: 0.007360632997006178,
KL Loss: 0.5277476906776428
It 2100: Total Loss: 0.008453278802335262,       Rec Loss: 0.008074895478785038,
KL Loss: 0.3783830404281616
```

```
It 2200: Total Loss: 0.007354896515607834,       Rec Loss: 0.00701980572193861,
KL Loss: 0.3350909352302551
It 2300: Total Loss: 0.008476541377604008,       Rec Loss: 0.008059632033109665,
KL Loss: 0.41690927743911743
It 2400: Total Loss: 0.009219754487276077,       Rec Loss: 0.008842402137815952,
KL Loss: 0.3773522973060608
It 2500: Total Loss: 0.008052741177380085,       Rec Loss: 0.007743803784251213,
KL Loss: 0.3089371919631958
It 2600: Total Loss: 0.007283627986907959,       Rec Loss: 0.006973492912948131
6,       KL Loss: 0.3101351857185364
It 2700: Total Loss: 0.00784380454570055,        Rec Loss: 0.007517438381910324,
KL Loss: 0.3263661861419678
It 2800: Total Loss: 0.00732382433488965,        Rec Loss: 0.007047029677778482
4,       KL Loss: 0.27679455280303955
Run Epoch 3
It 2900: Total Loss: 0.007837085984647274,       Rec Loss: 0.007455257698893547,
KL Loss: 0.381828635931015
It 3000: Total Loss: 0.006999272387474775,       Rec Loss: 0.006747105624526739,
KL Loss: 0.25216686725616455
It 3100: Total Loss: 0.008299864828586578,       Rec Loss: 0.007871262729167938,
KL Loss: 0.4286017417907715
It 3200: Total Loss: 0.008080116473138332,       Rec Loss: 0.007766071241348982,
KL Loss: 0.3140455484390259
It 3300: Total Loss: 0.008381279185414314,       Rec Loss: 0.00804158579558134,
KL Loss: 0.33969372510910034
It 3400: Total Loss: 0.007047781255096197,       Rec Loss: 0.006696682889014482
5,       KL Loss: 0.35109853744506836
It 3500: Total Loss: 0.007154163438826799,       Rec Loss: 0.006811557803303003,
KL Loss: 0.34260547161102295
It 3600: Total Loss: 0.006584944203495979,       Rec Loss: 0.006239618640393019,
KL Loss: 0.34532561898231506
It 3700: Total Loss: 0.00685651320964098,        Rec Loss: 0.006617627572268248,
KL Loss: 0.2388855218887329
Run Epoch 4
It 3800: Total Loss: 0.00842901412397623,        Rec Loss: 0.007972311228513718,
KL Loss: 0.45670250058174133
It 3900: Total Loss: 0.008081781677901745,       Rec Loss: 0.007800160441547632,
KL Loss: 0.28162091970443726
It 4000: Total Loss: 0.007010988425463438,       Rec Loss: 0.006710661109536886,
KL Loss: 0.3003273010253906
It 4100: Total Loss: 0.006465846206992865,       Rec Loss: 0.006216596346348524,
KL Loss: 0.24924972653388977
It 4200: Total Loss: 0.006356330588459969,       Rec Loss: 0.006043963134288788,
KL Loss: 0.3123674988746643
It 4300: Total Loss: 0.0066890292800962925,      Rec Loss: 0.006408725399523973
5,       KL Loss: 0.2803040146827698
It 4400: Total Loss: 0.005983876995742321,       Rec Loss: 0.005675582680851221,
KL Loss: 0.30829453468322754
It 4500: Total Loss: 0.006211401894688606,       Rec Loss: 0.005849447567015886,
KL Loss: 0.36195406317710876
It 4600: Total Loss: 0.006675940006971359,       Rec Loss: 0.006391089875251055,
KL Loss: 0.2848500609397888
Run Epoch 5
It 4700: Total Loss: 0.005995573475956917,       Rec Loss: 0.005724702961742878,
KL Loss: 0.27087074518203735
It 4800: Total Loss: 0.0066544427536427975,      Rec Loss: 0.006296738982200622
6,       KL Loss: 0.3577035665512085
It 4900: Total Loss: 0.006212788634002209,       Rec Loss: 0.005808989517390728,
KL Loss: 0.4037991166114807
It 5000: Total Loss: 0.006538514047861099,       Rec Loss: 0.006168239284306765,
```

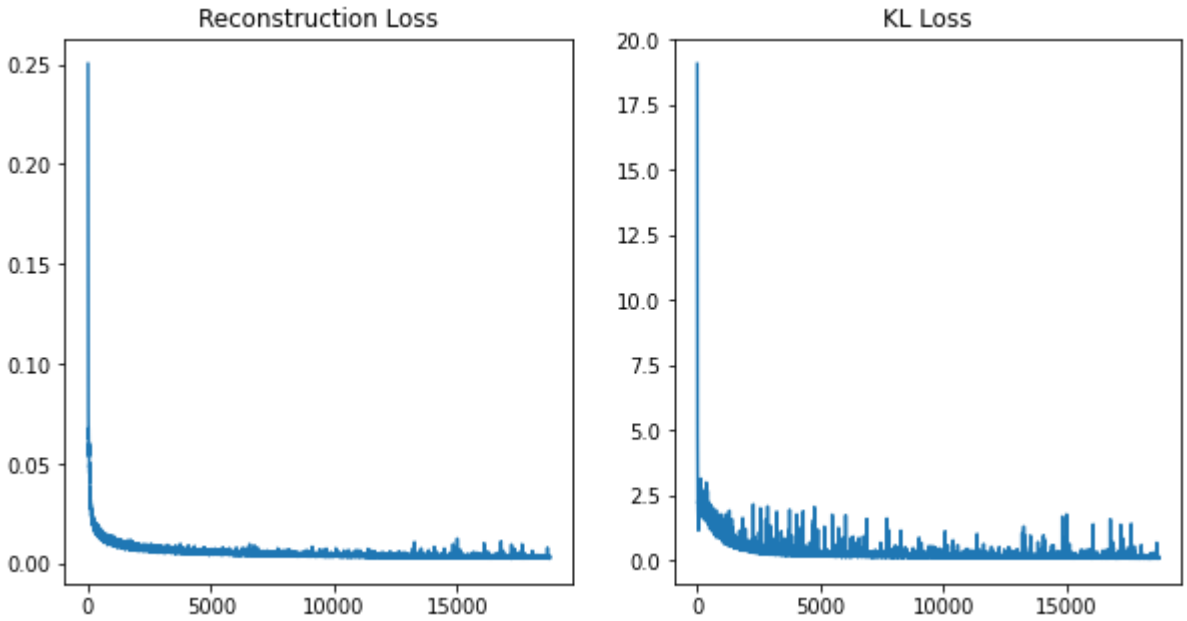
```
KL Loss: 0.37027496099472046
It 5100: Total Loss: 0.006713804788887501,       Rec Loss: 0.006417501717805862,
KL Loss: 0.29630327224731445
It 5200: Total Loss: 0.00603457260876894,        Rec Loss: 0.005725190974771976
5,       KL Loss: 0.3093816637992859
It 5300: Total Loss: 0.005794647615402937,       Rec Loss: 0.005564499646425247,
KL Loss: 0.2301478385925293
It 5400: Total Loss: 0.0062414915300905704,      Rec Loss: 0.006008253432810306
5,       KL Loss: 0.23323827981948853
It 5500: Total Loss: 0.006253019906580448,       Rec Loss: 0.005987999495118856,
KL Loss: 0.26502060890197754
It 5600: Total Loss: 0.005829576868563890 5,     Rec Loss: 0.005617179907858372,
KL Loss: 0.21239681541919708
Run Epoch 6
It 5700: Total Loss: 0.007204785477370024,       Rec Loss: 0.006697153206914663,
KL Loss: 0.5076322555541992
It 5800: Total Loss: 0.006873822305351496,       Rec Loss: 0.006656263954937458,
KL Loss: 0.21755832433700562
It 5900: Total Loss: 0.005388494580984116,       Rec Loss: 0.005110898520797491,
KL Loss: 0.27759605646133423
It 6000: Total Loss: 0.005756568629294634,       Rec Loss: 0.005548966117203236,
KL Loss: 0.2076023817062378
It 6100: Total Loss: 0.005699420813471079,       Rec Loss: 0.005487451795488596,
KL Loss: 0.2119690477848053
It 6200: Total Loss: 0.006244759075343609,       Rec Loss: 0.005976343527436256,
KL Loss: 0.26841533184051514
It 6300: Total Loss: 0.006266389042139053,       Rec Loss: 0.006005870178341865
5,       KL Loss: 0.2605190873146057
It 6400: Total Loss: 0.005618091206997633,       Rec Loss: 0.005334649700671434,
KL Loss: 0.2834414839744568
It 6500: Total Loss: 0.005489204544574022,       Rec Loss: 0.005244406405836344,
KL Loss: 0.24479815363883972
Run Epoch 7
It 6600: Total Loss: 0.004794036038219929,       Rec Loss: 0.004562241025269031
5,       KL Loss: 0.23179477453231812
It 6700: Total Loss: 0.0053237434476614,         Rec Loss: 0.005093781277537346,
KL Loss: 0.2299623191356659
It 6800: Total Loss: 0.005610467866063118,       Rec Loss: 0.005326656624674797,
KL Loss: 0.2838112711906433
It 6900: Total Loss: 0.005354320630431175,       Rec Loss: 0.005130419041961431
5,       KL Loss: 0.22390159964561462
It 7000: Total Loss: 0.005086187738925219,       Rec Loss: 0.004865820053964853,
KL Loss: 0.22036775946617126
It 7100: Total Loss: 0.005275968462228775,       Rec Loss: 0.004962440114468336,
KL Loss: 0.31352850794792175
It 7200: Total Loss: 0.005146760959178209,       Rec Loss: 0.004943015985190868,
KL Loss: 0.2037450075149536
It 7300: Total Loss: 0.005543584935367107,       Rec Loss: 0.00535433366894722,
KL Loss: 0.1892511546611786
It 7400: Total Loss: 0.004886536858975887,       Rec Loss: 0.004671472124755382
5,       KL Loss: 0.21506449580192566
It 7500: Total Loss: 0.005213099531829357,       Rec Loss: 0.004954899195581675,
KL Loss: 0.2582004964351654
Run Epoch 8
It 7600: Total Loss: 0.005079042632132769,       Rec Loss: 0.004828375298529863,
KL Loss: 0.25066736340522766
It 7700: Total Loss: 0.005445661023259163,       Rec Loss: 0.005233791191130876
5,       KL Loss: 0.21187004446983337
It 7800: Total Loss: 0.004865939728915691,       Rec Loss: 0.004696831107139587,
KL Loss: 0.16910848021507263
```

```
It 7900: Total Loss: 0.0044798534363508224,      Rec Loss: 0.004296416416764259,
KL Loss: 0.18343693017959595
It 8000: Total Loss: 0.004869313910603523,       Rec Loss: 0.004699136596173048,
KL Loss: 0.17017747461795807
It 8100: Total Loss: 0.004903698805719614,       Rec Loss: 0.004731287714093924,
KL Loss: 0.1724109798669815
It 8200: Total Loss: 0.0077116722241044044,      Rec Loss: 0.007172245532274246,
KL Loss: 0.5394265055656433
It 8300: Total Loss: 0.006022926419973373,       Rec Loss: 0.005849214736372232
4,       KL Loss: 0.17371153831481934
It 8400: Total Loss: 0.005353226326406002,       Rec Loss: 0.005163110326975584,
KL Loss: 0.19011589884757996
Run Epoch 9
It 8500: Total Loss: 0.004906961694359779,       Rec Loss: 0.004719962365925312,
KL Loss: 0.18699952960014343
It 8600: Total Loss: 0.004826475400477648,       Rec Loss: 0.004607649054378271,
KL Loss: 0.2188263088464737
It 8700: Total Loss: 0.004693531431257725,       Rec Loss: 0.004461982753127813,
KL Loss: 0.23154866695404053
It 8800: Total Loss: 0.005027307663112879,       Rec Loss: 0.004823827650398016,
KL Loss: 0.203480064868927
It 8900: Total Loss: 0.0050328923389315605,      Rec Loss: 0.004799007903784513
5,       KL Loss: 0.23388434946537018
It 9000: Total Loss: 0.004924063570797443,       Rec Loss: 0.004717043600976467,
KL Loss: 0.20701992511749268
It 9100: Total Loss: 0.004492474719882011,       Rec Loss: 0.004301968496292829
5,       KL Loss: 0.1905062347650528
It 9200: Total Loss: 0.004527377896010876,       Rec Loss: 0.004320426378399134,
KL Loss: 0.20695146918296814
It 9300: Total Loss: 0.00458685215562582,        Rec Loss: 0.004435384180396795,
KL Loss: 0.15146812796592712
Run Epoch 10
It 9400: Total Loss: 0.004491895437240601,       Rec Loss: 0.004301084671169519,
KL Loss: 0.19081082940101624
It 9500: Total Loss: 0.0050077359192073345,      Rec Loss: 0.004851294215768576,
KL Loss: 0.1564418375492096
It 9600: Total Loss: 0.004699022509157658,       Rec Loss: 0.004528558813035488,
KL Loss: 0.17046375572681427
It 9700: Total Loss: 0.004514545667916536,       Rec Loss: 0.004316950216889381,
KL Loss: 0.1975955218076706
It 9800: Total Loss: 0.004843468312174082,       Rec Loss: 0.004701549187302589,
KL Loss: 0.14191901683807373
It 9900: Total Loss: 0.004742530174553394,       Rec Loss: 0.004542327951639891,
KL Loss: 0.20020221173763275
It 10000: Total Loss: 0.005038567818701267,      Rec Loss: 0.00486414460465312,
KL Loss: 0.1744231879711151
It 10100: Total Loss: 0.005223837681114674,      Rec Loss: 0.005059842020273209,
KL Loss: 0.16399571299552917
It 10200: Total Loss: 0.005947913508862257,      Rec Loss: 0.005772929172962904,
KL Loss: 0.1749841421842575
It 10300: Total Loss: 0.005577289033681154,      Rec Loss: 0.005396690219640732,
KL Loss: 0.18059873580932617
Run Epoch 11
It 10400: Total Loss: 0.004154666792601347,      Rec Loss: 0.004013857338577509,
KL Loss: 0.14080962538719177
It 10500: Total Loss: 0.0046446602791547775,     Rec Loss: 0.004500629380345344
5,       KL Loss: 0.14403092861175537
It 10600: Total Loss: 0.004689822904765606,      Rec Loss: 0.004511961247771978,
KL Loss: 0.17786185443401337
It 10700: Total Loss: 0.004872195888310671,      Rec Loss: 0.004680278711020946
```

```
5,       KL Loss: 0.1919173300266266
It 10800: Total Loss: 0.004354311153292656,      Rec Loss: 0.004208287224173546,
KL Loss: 0.14602407813072205
It 10900: Total Loss: 0.00449323607608676,       Rec Loss: 0.004235249944031238
6,       KL Loss: 0.25798630714416504
It 11000: Total Loss: 0.004760143347084522,      Rec Loss: 0.004623852204531431,
KL Loss: 0.13629105687141418
It 11100: Total Loss: 0.00419662008062005,       Rec Loss: 0.004013105761259794,
KL Loss: 0.18351438641548157
It 11200: Total Loss: 0.004025398753583431,      Rec Loss: 0.003896188922226429,
KL Loss: 0.12920965254306793
Run Epoch 12
It 11300: Total Loss: 0.004341209307312965,      Rec Loss: 0.004198967013508081
4,       KL Loss: 0.1422424167394638
It 11400: Total Loss: 0.004449472296983004,      Rec Loss: 0.004317832179367542,
KL Loss: 0.13164007663726807
It 11500: Total Loss: 0.004147522617131472,      Rec Loss: 0.003974417690187693,
KL Loss: 0.17310509085655212
It 11600: Total Loss: 0.004113809671252966,      Rec Loss: 0.003957874607294798,
KL Loss: 0.15593484044075012
It 11700: Total Loss: 0.004583938978612423,      Rec Loss: 0.004443798214197159,
KL Loss: 0.1401406228542328
It 11800: Total Loss: 0.004726773593574762,      Rec Loss: 0.004596525803208351,
KL Loss: 0.13024777173995972
It 11900: Total Loss: 0.0043541742488741875,     Rec Loss: 0.004205956589430571,
KL Loss: 0.1482176035642624
It 12000: Total Loss: 0.00435516657307744,       Rec Loss: 0.004213053267449140
5,       KL Loss: 0.1421130895614624
It 12100: Total Loss: 0.0043882569298148155,     Rec Loss: 0.004220366477966309,
KL Loss: 0.16789057850837708
Run Epoch 13
It 12200: Total Loss: 0.004231785424053669,      Rec Loss: 0.004099682904779911,
KL Loss: 0.13210274279117584
It 12300: Total Loss: 0.004699204117059708,      Rec Loss: 0.004516805056482553
5,       KL Loss: 0.18239903450012207
It 12400: Total Loss: 0.0042648171074688435,     Rec Loss: 0.004122994840145111,
KL Loss: 0.14182233810424805
It 12500: Total Loss: 0.004478238057345152,      Rec Loss: 0.004318239632993936
5,       KL Loss: 0.15999841690063477
It 12600: Total Loss: 0.004258602857589722,      Rec Loss: 0.004126457497477531,
KL Loss: 0.13214528560638428
It 12700: Total Loss: 0.004293706268072128,      Rec Loss: 0.004172015935182571,
KL Loss: 0.12169034779071808
It 12800: Total Loss: 0.004178192466497421,      Rec Loss: 0.004036800935864448
5,       KL Loss: 0.14139163494110107
It 12900: Total Loss: 0.004668023902922869,      Rec Loss: 0.004542313050478697,
KL Loss: 0.12571075558662415
It 13000: Total Loss: 0.003966711461544037,      Rec Loss: 0.003811499103903770
4,       KL Loss: 0.1552121490240097
It 13100: Total Loss: 0.004216627683490515,      Rec Loss: 0.00411172304302454,
KL Loss: 0.1049046441912651
Run Epoch 14
It 13200: Total Loss: 0.004215450957417488,      Rec Loss: 0.003965407609939575,
KL Loss: 0.2500433027744293
It 13300: Total Loss: 0.007781679276376963,      Rec Loss: 0.006810007616877556,
KL Loss: 0.971671462059021
It 13400: Total Loss: 0.00410824129357934,       Rec Loss: 0.00394505076110363,
KL Loss: 0.16319060325622559
It 13500: Total Loss: 0.0044427490793168545,     Rec Loss: 0.004285346250981092
5,       KL Loss: 0.15740303695201874
```

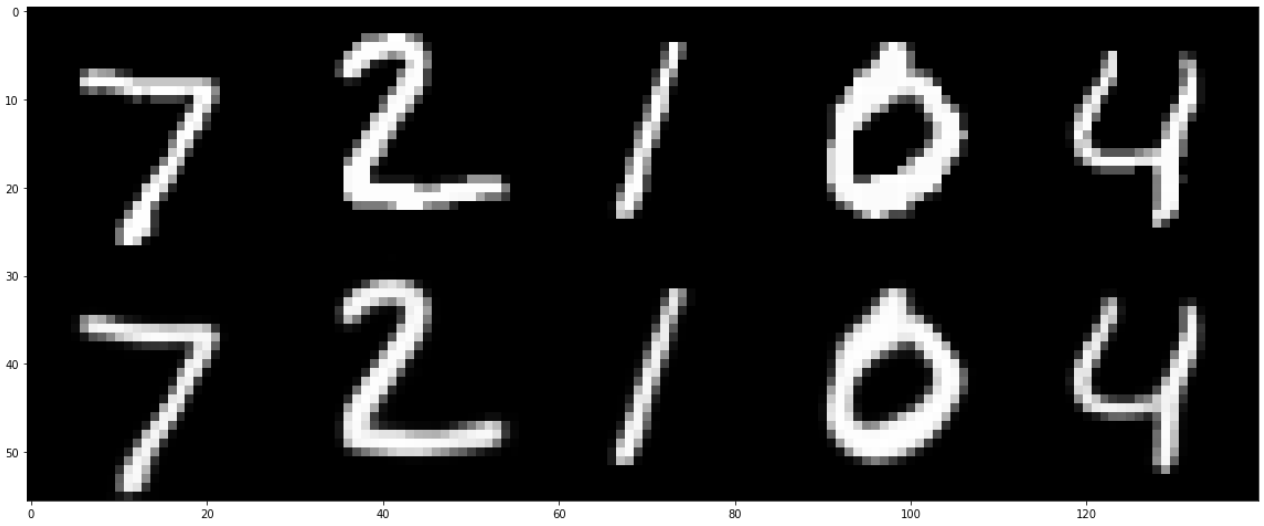
```
It 13600: Total Loss: 0.004037093371152878,      Rec Loss: 0.003902042284607887
3,       KL Loss: 0.13505128026008606
It 13700: Total Loss: 0.004159770905971527,      Rec Loss: 0.004031369462609291,
KL Loss: 0.12840153276920319
It 13800: Total Loss: 0.004537696484476328,      Rec Loss: 0.004427261184900999,
KL Loss: 0.11043539643287659
It 13900: Total Loss: 0.003716464592933655,      Rec Loss: 0.003599512856453657,
KL Loss: 0.116933673620224
It 14000: Total Loss: 0.004380689933896065,      Rec Loss: 0.004235985223203897
5,       KL Loss: 0.14470461010932922
Run Epoch 15
It 14100: Total Loss: 0.004517771303653717,      Rec Loss: 0.004362531937658787,
KL Loss: 0.15523958206176758
It 14200: Total Loss: 0.003998449537903070 4,     Rec Loss: 0.003884807461872697,
KL Loss: 0.11364200711250305
It 14300: Total Loss: 0.004094869364053011,      Rec Loss: 0.003957539796829224,
KL Loss: 0.13732975721359253
It 14400: Total Loss: 0.003834463655948639,      Rec Loss: 0.00368153746239841,
KL Loss: 0.1529262512922287
It 14500: Total Loss: 0.004533988423645496,      Rec Loss: 0.004396525677293539,
KL Loss: 0.13746283948421478
It 14600: Total Loss: 0.004096277058124542,      Rec Loss: 0.003978016320616007,
KL Loss: 0.1182607114315033
It 14700: Total Loss: 0.003954428713768721,      Rec Loss: 0.003832349320873618,
KL Loss: 0.12207929790019989
It 14800: Total Loss: 0.003856675699353218,      Rec Loss: 0.003757359925657510
8,       KL Loss: 0.09931586682796478
It 14900: Total Loss: 0.004534353502094746,      Rec Loss: 0.004301533102989197,
KL Loss: 0.23282018303871155
It 15000: Total Loss: 0.004169418476521969,      Rec Loss: 0.004038522019982338,
KL Loss: 0.13089659810066223
Run Epoch 16
It 15100: Total Loss: 0.003758917795494199,      Rec Loss: 0.003603243734687566
8,       KL Loss: 0.15567395091056824
It 15200: Total Loss: 0.004951199050992727,      Rec Loss: 0.004784936551004648,
KL Loss: 0.16626238822937012
It 15300: Total Loss: 0.003766810754314065,      Rec Loss: 0.003629133338108659,
KL Loss: 0.1376773715019226
It 15400: Total Loss: 0.003732908517122268 7,     Rec Loss: 0.003611580701544881,
KL Loss: 0.12132792919874191
It 15500: Total Loss: 0.003582943696528673,      Rec Loss: 0.003475998295471072,
KL Loss: 0.1069454699754715
It 15600: Total Loss: 0.006114284507930279,      Rec Loss: 0.005695669911801815,
KL Loss: 0.4186146855354309
It 15700: Total Loss: 0.003543287515640259,      Rec Loss: 0.00342406053096056,
KL Loss: 0.11922699213027954
It 15800: Total Loss: 0.003680086694657802 6,     Rec Loss: 0.003585251513868570
3,       KL Loss: 0.09483512490987778
It 15900: Total Loss: 0.003734204452484846,      Rec Loss: 0.003584013320505619,
KL Loss: 0.15019114315509796
Run Epoch 17
It 16000: Total Loss: 0.004053917713463306,      Rec Loss: 0.003952860366553068,
KL Loss: 0.10105755925178528
It 16100: Total Loss: 0.003821081249043345 5,     Rec Loss: 0.003678833832964301,
KL Loss: 0.1422474980354309
It 16200: Total Loss: 0.003761612810194492 3,     Rec Loss: 0.003659768961369991
3,       KL Loss: 0.10184374451637268
It 16300: Total Loss: 0.003894043387845158 6,     Rec Loss: 0.003796720877289772,
KL Loss: 0.09732260555028915
It 16400: Total Loss: 0.003909083431756496,      Rec Loss: 0.003800529753789305
```

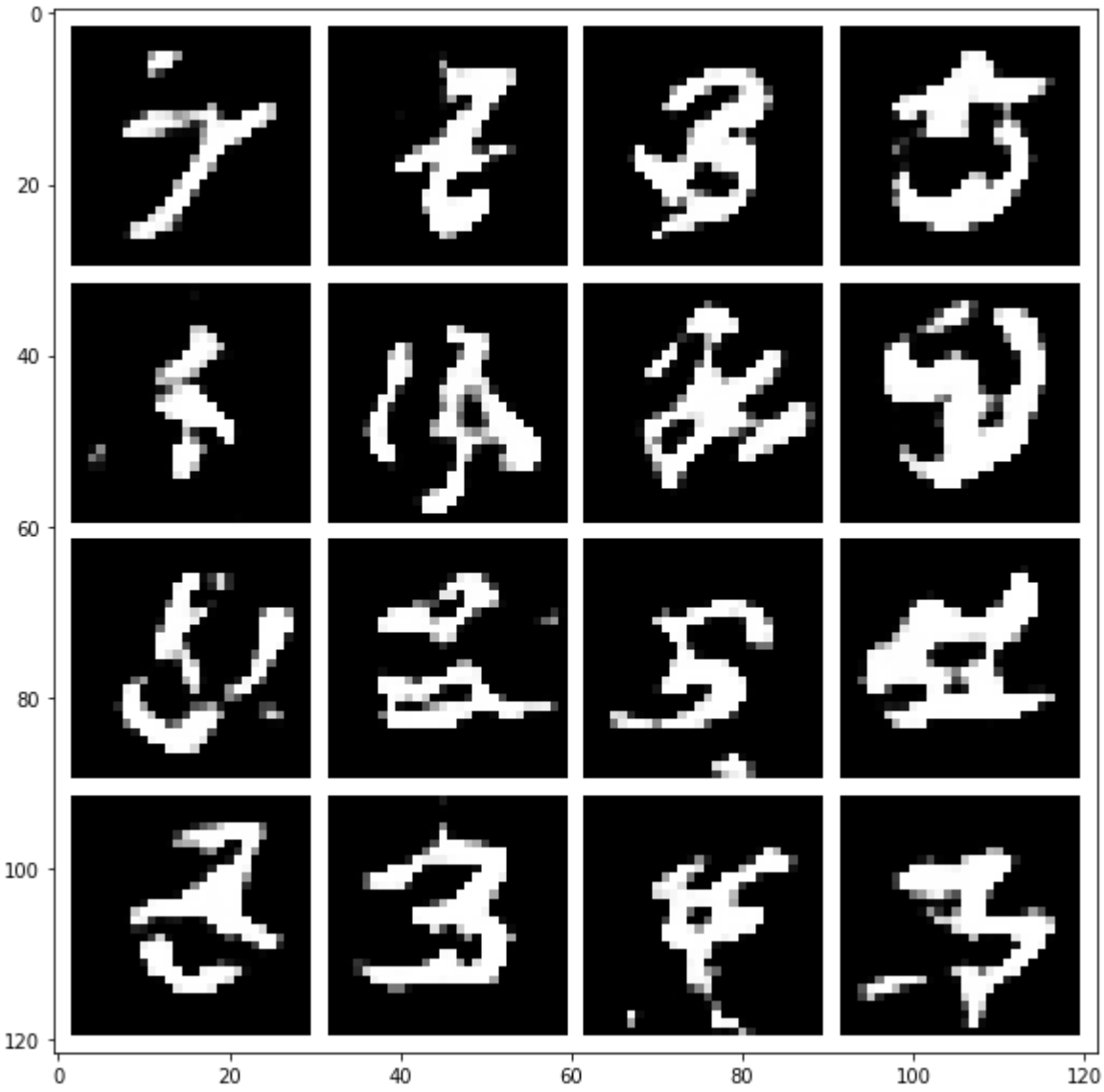
```
7,       KL Loss: 0.10845388472080231
It 16500: Total Loss: 0.0037310095503926277,     Rec Loss: 0.003643732285127043
7,       KL Loss: 0.08727714419364929
It 16600: Total Loss: 0.004633539356291294,      Rec Loss: 0.004509482532739639,
KL Loss: 0.12405681610107422
It 16700: Total Loss: 0.003495992161333561,      Rec Loss: 0.003390134312212467,
KL Loss: 0.10585778206586838
It 16800: Total Loss: 0.004326010122895241,      Rec Loss: 0.004201371222734451,
KL Loss: 0.12463894486427307
Run Epoch 18
It 16900: Total Loss: 0.004572106525301933,      Rec Loss: 0.004442562814801931,
KL Loss: 0.12954355776309967
It 17000: Total Loss: 0.0037856611888855696,     Rec Loss: 0.00368088623508811,
KL Loss: 0.10477486252784729
It 17100: Total Loss: 0.003986068069934845,      Rec Loss: 0.003861297620460391,
KL Loss: 0.12477042526006699
It 17200: Total Loss: 0.004126640502363443,      Rec Loss: 0.003966519143432379,
KL Loss: 0.16012153029441833
It 17300: Total Loss: 0.004082903731614351,      Rec Loss: 0.003900779644027352
3,       KL Loss: 0.1821240782737732
It 17400: Total Loss: 0.003652399405837059,      Rec Loss: 0.003549639834091067
3,       KL Loss: 0.10275952517986298
It 17500: Total Loss: 0.004304452333599329,      Rec Loss: 0.004114850424230099,
KL Loss: 0.18960201740264893
It 17600: Total Loss: 0.0033025802113115788,     Rec Loss: 0.003223259467631578
4,       KL Loss: 0.07932080328464508
It 17700: Total Loss: 0.003961729351431131,      Rec Loss: 0.003817604854702949
5,       KL Loss: 0.1441245973110199
It 17800: Total Loss: 0.003955003805458546,      Rec Loss: 0.003842065343633294,
KL Loss: 0.11293867230415344
Run Epoch 19
It 17900: Total Loss: 0.0033943899907171726,     Rec Loss: 0.003306794678792357
4,       KL Loss: 0.0875953882932663
It 18000: Total Loss: 0.003698650049045682,      Rec Loss: 0.003570934990420937
5,       KL Loss: 0.12771503627300262
It 18100: Total Loss: 0.003660994814708829,      Rec Loss: 0.003561631077900529,
KL Loss: 0.09936362504959106
It 18200: Total Loss: 0.004299158230423927,      Rec Loss: 0.004186541307717562,
KL Loss: 0.1126168817281723
It 18300: Total Loss: 0.00365262757986784,       Rec Loss: 0.003565923776477575
3,       KL Loss: 0.08670385926961899
It 18400: Total Loss: 0.004048938862979412,      Rec Loss: 0.003942064009606838,
KL Loss: 0.1068747490644455
It 18500: Total Loss: 0.0032608420588076115,     Rec Loss: 0.003154038684442639
4,       KL Loss: 0.10680335760116577
It 18600: Total Loss: 0.0032568590249866247,     Rec Loss: 0.003145406721159816,
KL Loss: 0.11145234853029251
It 18700: Total Loss: 0.0032666351180523634,     Rec Loss: 0.003160976804792881,
KL Loss: 0.10565830767154694
Done!
```



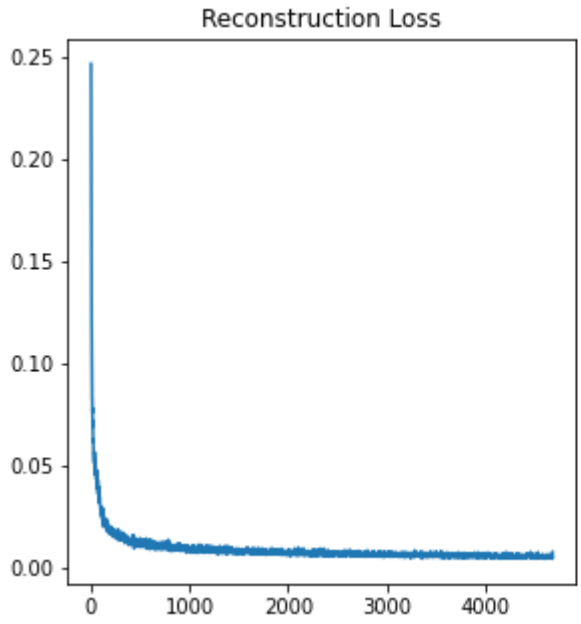
Let's look at some reconstructions and decoded embedding samples!

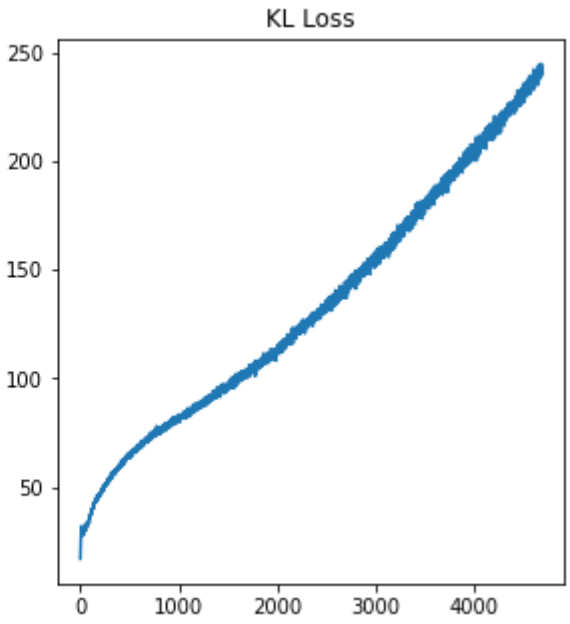
In [ ]:
```
# visualize VAE reconstructions and samples from the generative model
vis_reconstruction(vae_model)
vis_samples(vae_model)
```

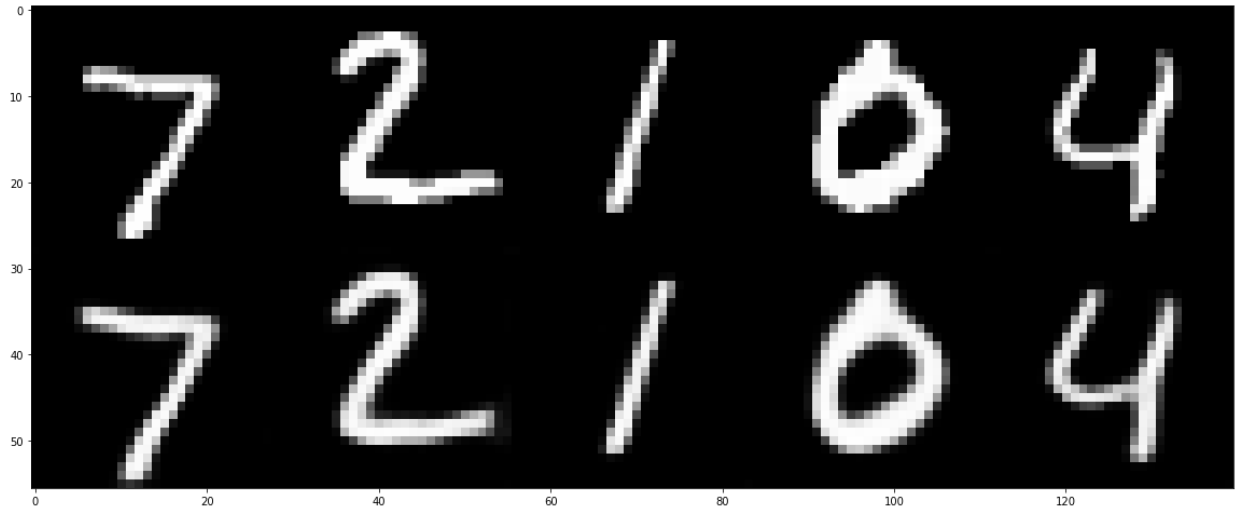


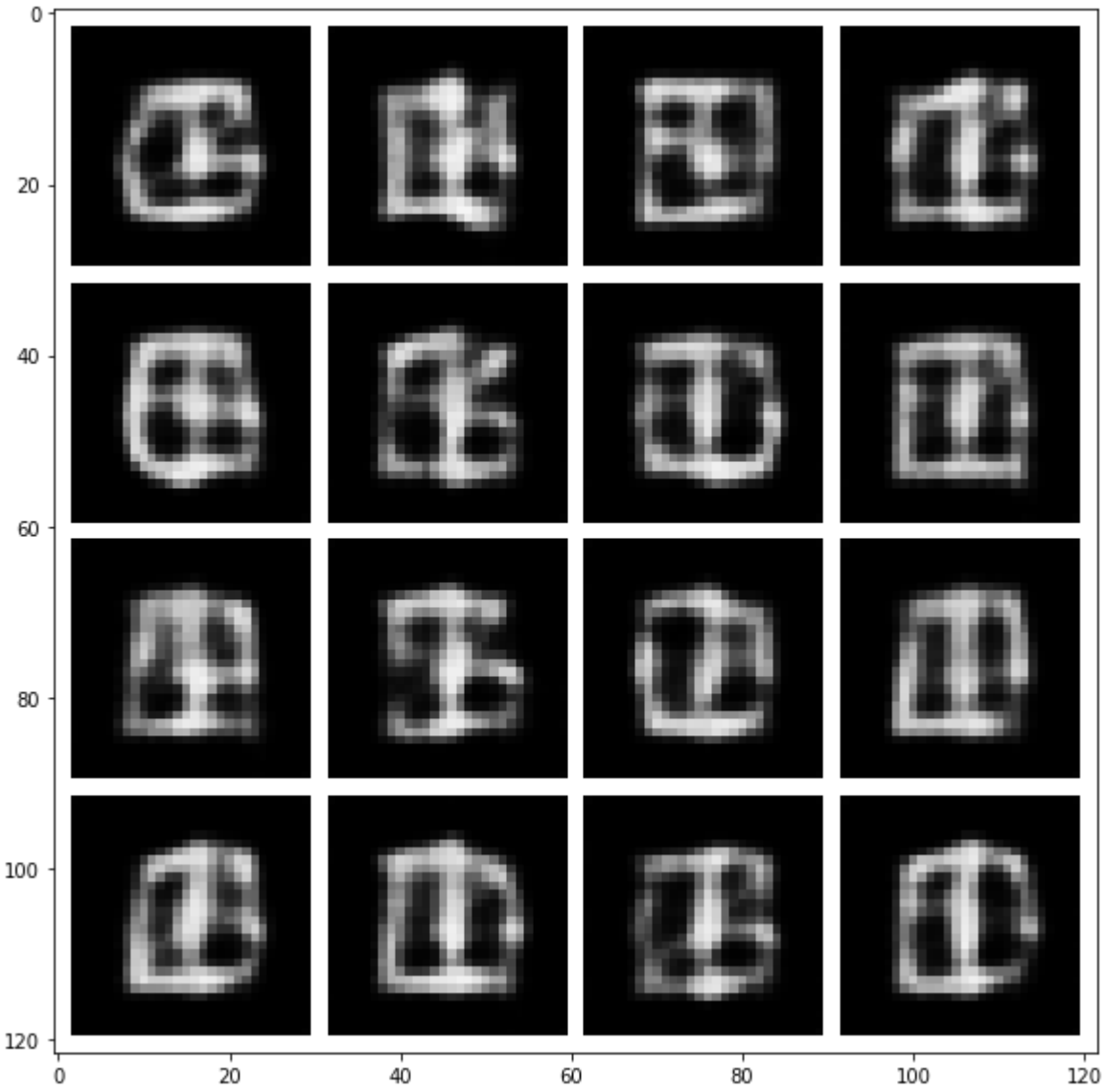


> **Inline Question: What can you observe when setting $\beta = 0$? Explain your observations! [3pt]** \ (please limit your answer to <150 words) \ **Answer:** The model generates the original input almost precisely except some blurriness in the edges. I think it is almost similar to an autoencoder. For the generative part of the model where we sample from unit diagonal gaussian, the images generated are slightly more distinct and clear than the one generated by autoencoders. It seems as if the model is trying to generated some digits but most of them are deformed/skewed.


Reconstruction Loss
0.25
0.20
0.15
0.10
0.05
0.00
0
1000
2000
3000
4000

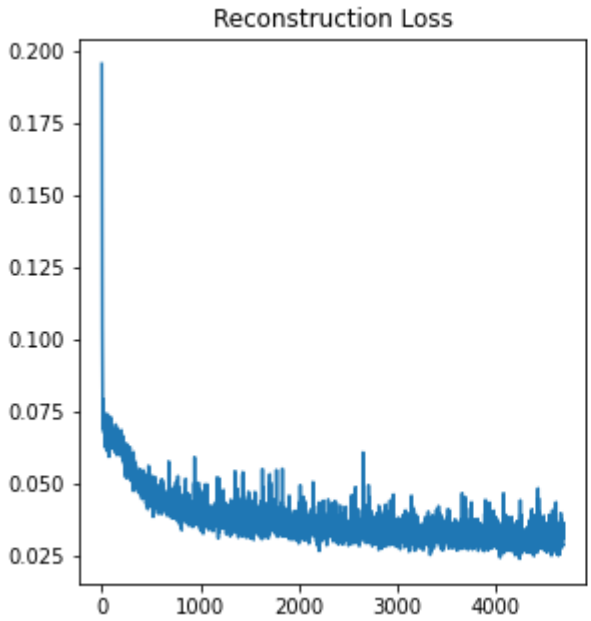

KL Loss
250
200
150
100
50
0
1000
2000
3000
4000

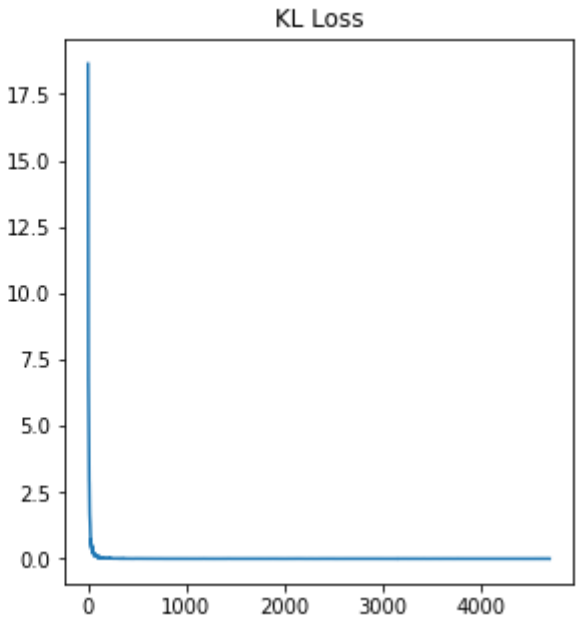

0
10
20
30
40
50
0
20
40
60
80
100
120



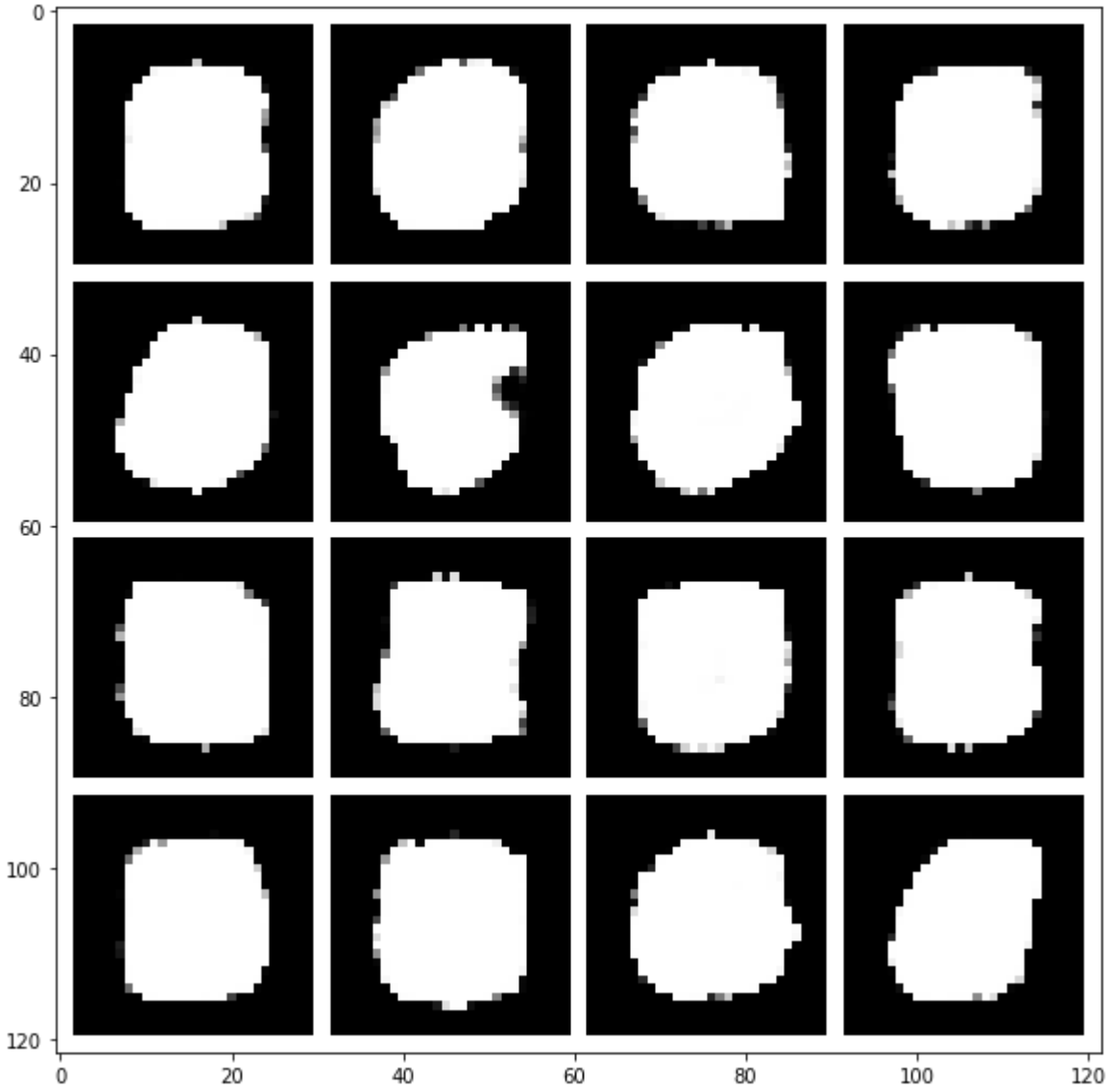
Let's repeat the same experiment for $\beta = 10$, a very high value for the coefficient. You can modify the β value in the cell above and rerun it (it is okay to overwrite the outputs of the previous experiment, but **make sure to copy the visualizations of training curves, reconstructions and samples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

> **Inline Question: What can you observe when setting $\beta = 10$? Explain your observations! [3pt]** \ (please limit your answer to <200 words) \ **Answer**: It seems like at `beta=10` , the digits in the reconstructed image are getting very blurry. While the overall digits are almost the same, we can see additional pixels around the digits and the shapes getting a bit off from the origitnal. Moroever, the samples of generation are just white blogs. One possible reason may be that the model is trying to fit the randomly generated models into the latent space and generate digits.


Reconstruction Loss
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0
1000
2000
3000
4000


KL Loss
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
1000
2000
3000
4000


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



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

> **Inline Question: Characterize what properties you would expect for reconstructions (1pt) and samples (2pt) of a well-tuned VAE! [3pt]** \ (please limit your answer to <200 words) \ **Answer**: A well-tuned VAE should have sharper reconstructions and should have almost negligible blur around the digits. For the samples that are generated by a VAE, they should be more distinct and clear with respect to what digit they represent. Moreover, the skewness of the digits should not be too much that they are unreadable or unbelievable that a human would have written it.

Tuning the β-factor [5pt]

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

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

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

4. Embedding Space Interpolation [3pt]

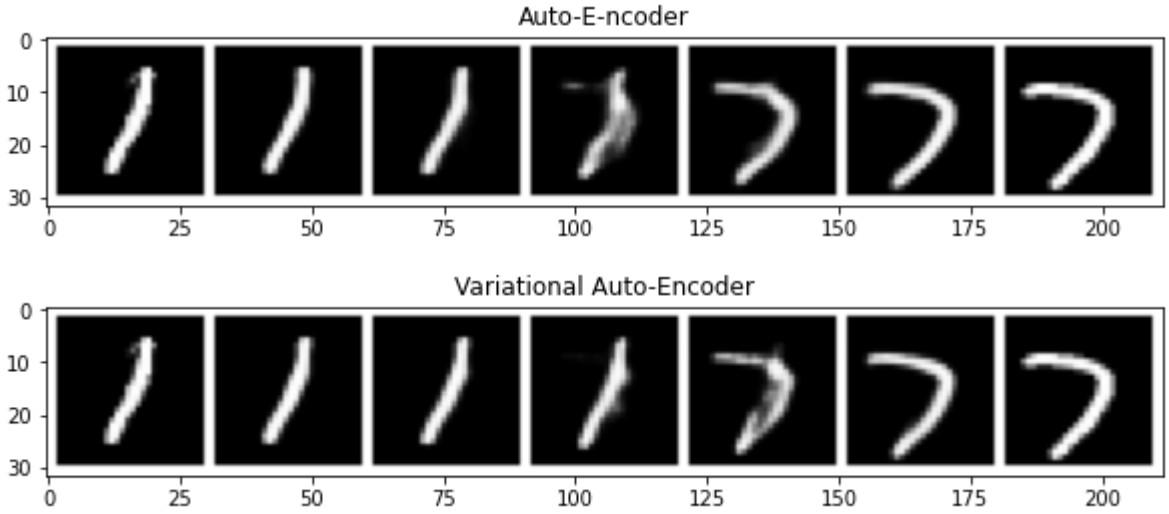
As mentioned in the introduction, AEs and VAEs cannot only be used to generate images, but also to learn low-dimensional representations of their inputs. In this final section we will investigate the representations we learned with both models by **interpolating in embedding space** between different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

In [ ]:

```
START_LABEL = 1
END_LABEL = 7
nz=64

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

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

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

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

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

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


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

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

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserW
arning: This DataLoader will create 4 worker processes in total. Our suggested m
ax number of worker in current system is 2, which is smaller than what this Data
Loader is going to create. Please be aware that excessive worker creation might
get DataLoader running slow or even freeze, lower the worker number to avoid pot
ential slowness/freeze if necessary.
  cpuset_checked))
```



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

> **Inline Question: Repeat the interpolation experiment with different start / end labels and multiple samples. Describe your observations! Focus on**: \
>
> 1. **How do AE and VAE embedding space interpolations differ?** \
> 2. **How do you expect these differences to affect the usefulness of the learned representation for downstream learning?** \ (please limit your answer to <300 words)
>
> **Answer**:
>
> 1. The interpolations of AE have more randomness/blurriness compared to the embedding space interpolations from VAE. The interpolation sample at 100 shows that VAE is able to generate a better image even if there is some variance added to the embedding vector, which implies that is more robust. Moreover, the images generated by VAE are sharper and more inline with the MNIST data than their counterpart generated by AE.
> 2. Using AE and VAE in downstream learning will provide drastically different results because a random value in case of VAE generates more meaningful output than AE in somecases where is feels like AE is trying to morph one character into another or the latent space is overlapping.

Submission PDF

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

1. Auto-encoder samples and AE sampling inline question answer.
2. VAE training curves, reconstructions and samples for:
   - $\beta = 0$
   - $\beta = 10$
   - your tuned β (also listing the tuned value for β)
3. Answers to all inline questions in VAE section (ie 4 inline questions).
4. Three representative interpolation comparisons that show AE and VAE embedding interpolation between the same images.
5. Answer to interpolation inline question.

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