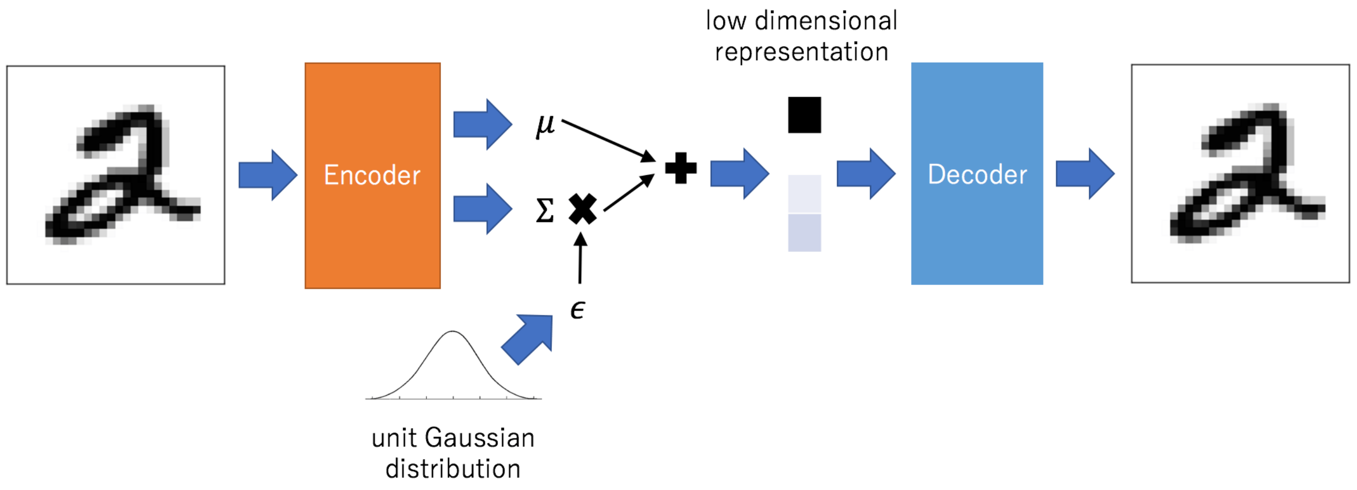
Problem 1 - Variational Auto-Encoder (VAE)

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.

✓ 0s completed at 4:14 PM

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.

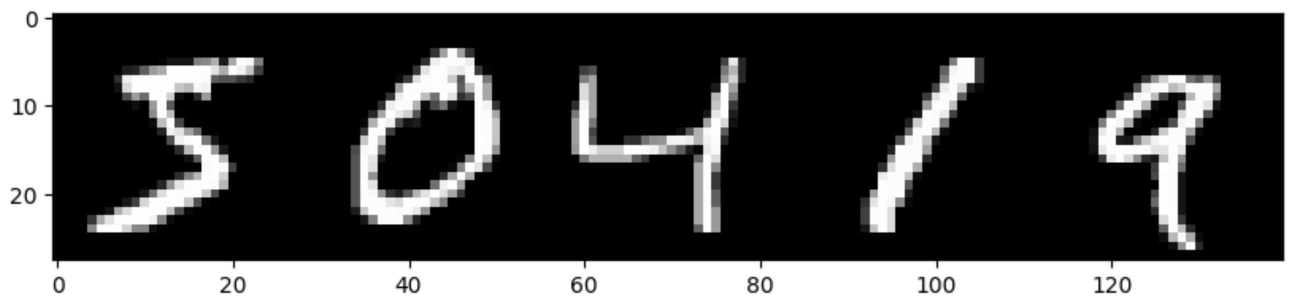
Let's start by downloading the data and visualizing some samples.

```
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
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_tra
```

```
Using device: cuda:0
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 ./(
100%                                        9912422/9912422 [00:00<00:00, 143290405.80it/s]
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 ./(
100%                                        28881/28881 [00:00<00:00, 580694.11it/s]
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 ./da
100%                                        1648877/1648877 [00:00<00:00, 3265917.51it/s]
```

```
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 ./da
100%                                                4542/4542 [00:00<00:00, 105894.51it/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw


 Download complete! Downloaded 60000 training examples!
```

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

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

for idx in sample_idxs:
  sample = mnist_train[idx]
  # print(f"Tensor w/ shape {sample[0][0].detach().cpu().numpy().shape} and label {
  sample_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
# 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 shap
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(
  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.sl
    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,
```

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 up

# 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.
################################################################################
bce_loss = nn.BCELoss()
optimizer = torch.optim.Adam(ae_model.parameters(),lr = learning_rate)
##################################### END TODO #################################

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

```
    for sample_img,sample_label in mnist_data_loader:

      input = sample_img.reshape([batch_size, in_size])
      optimizer.zero_grad()

      input_cuda = input.to(device)
      prediction = ae_model.forward(input_cuda)
      prediction = prediction.reshape([batch_size,in_size])

      rec_loss = bce_loss(prediction,input_cuda)
      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
```

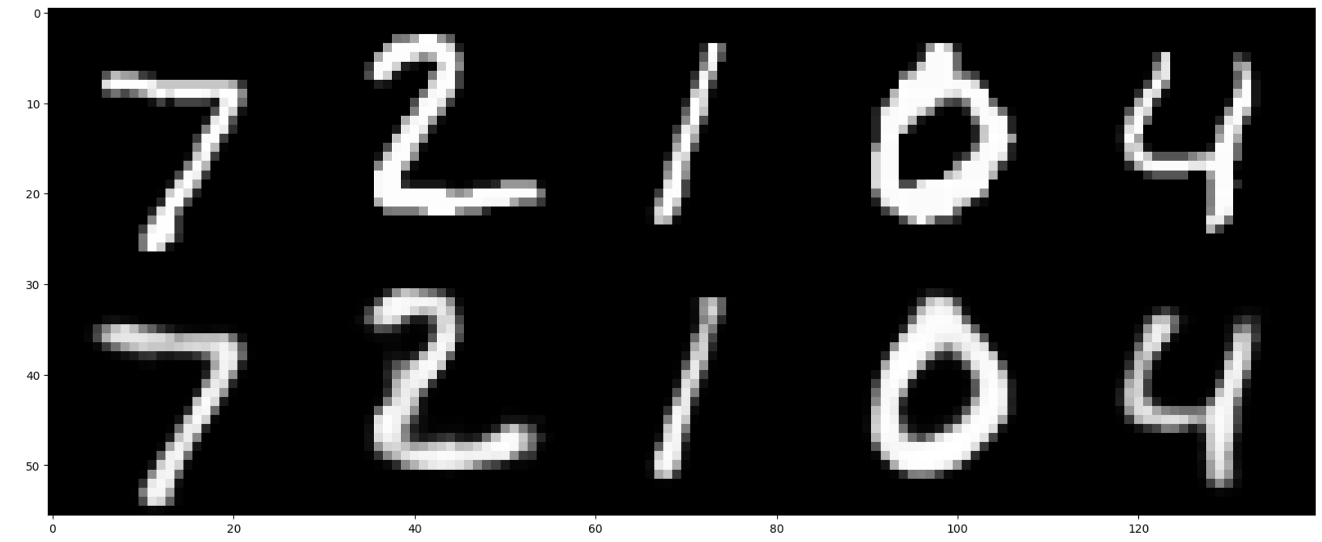
```
Device available cuda:0
Run Epoch 0
It 0: Reconstruction Loss: 0.6934277415275574
It 100: Reconstruction Loss: 0.26246997714042664
It 200: Reconstruction Loss: 0.21563392877578735
It 300: Reconstruction Loss: 0.20175373554229736
It 400: Reconstruction Loss: 0.16304413974285126
It 500: Reconstruction Loss: 0.1616559624671936
It 600: Reconstruction Loss: 0.15916962921619415
It 700: Reconstruction Loss: 0.1557304561138153
It 800: Reconstruction Loss: 0.12541331350803375
It 900: Reconstruction Loss: 0.14705593883991241
Run Epoch 1
It 1000: Reconstruction Loss: 0.1291280835866928
It 1100: Reconstruction Loss: 0.1393081694841385
It 1200: Reconstruction Loss: 0.12752269208431244
It 1300: Reconstruction Loss: 0.13263387978076935
It 1400: Reconstruction Loss: 0.1288689225912094
It 1500: Reconstruction Loss: 0.11984479427337646
It 1600: Reconstruction Loss: 0.1195826530456543
It 1700: Reconstruction Loss: 0.11513250321149826
It 1800: Reconstruction Loss: 0.12415891140699387
Run Epoch 2
It 1900: Reconstruction Loss: 0.11693749576807022
It 2000: Reconstruction Loss: 0.11451063305139542
It 2100: Reconstruction Loss: 0.12673673033714294
It 2200: Reconstruction Loss: 0.11206132173538208
It 2300: Reconstruction Loss: 0.11955691874027252
It 2400: Reconstruction Loss: 0.11562681943178177
It 2500: Reconstruction Loss: 0.11289265751838684
It 2600: Reconstruction Loss: 0.11465832591056824
It 2700: Reconstruction Loss: 0.11078079789876938
It 2800: Reconstruction Loss: 0.11443022814430237
It 2800: Reconstruction Loss: 0.1144303381443023/
Run Epoch 3
It 2900: Reconstruction Loss: 0.11637833714485168
It 3000: Reconstruction Loss: 0.10653810948133469
It 3100: Reconstruction Loss: 0.10616770386695862
It 3200: Reconstruction Loss: 0.1066799908876419
It 3300: Reconstruction Loss: 0.10290299355983734
It 3400: Reconstruction Loss: 0.10540199279785156
It 3500: Reconstruction Loss: 0.1065022274851799
It 3600: Reconstruction Loss: 0.10695852339267731
It 3700: Reconstruction Loss: 0.10068052262067795
Run Epoch 4
It 3800: Reconstruction Loss: 0.09857209771871567
It 3900: Reconstruction Loss: 0.10542745143175125
It 4000: Reconstruction Loss: 0.10304061323404312
It 4100: Reconstruction Loss: 0.09818131476640701
It 4200: Reconstruction Loss: 0.10154187679290771
It 4300: Reconstruction Loss: 0.09710821509361267
It 4400: Reconstruction Loss: 0.09591236710548401
It 4500: Reconstruction Loss: 0.09673869609832764
It 4600: Reconstruction Loss: 0.10290904343128204
Run Epoch 5
It 4700: Reconstruction Loss: 0.09846751391887665
It 4800: Reconstruction Loss: 0.10225614160299301
It 4900: Reconstruction Loss: 0.09372911602258682
It 5000: Reconstruction Loss: 0.09644916653633118
```

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])
```

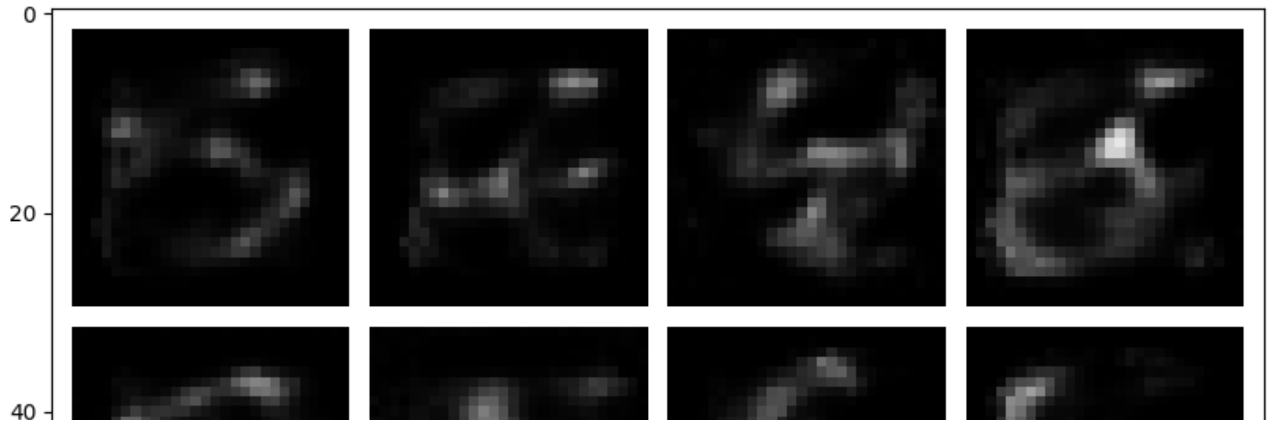
```
      input_img = np.asarray(sample[0])
      input_flat = input_img.reshape(784)
      reconstruction = model.reconstruct(torch.tensor(input_flat, device=device))

      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),
    plt.show()

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

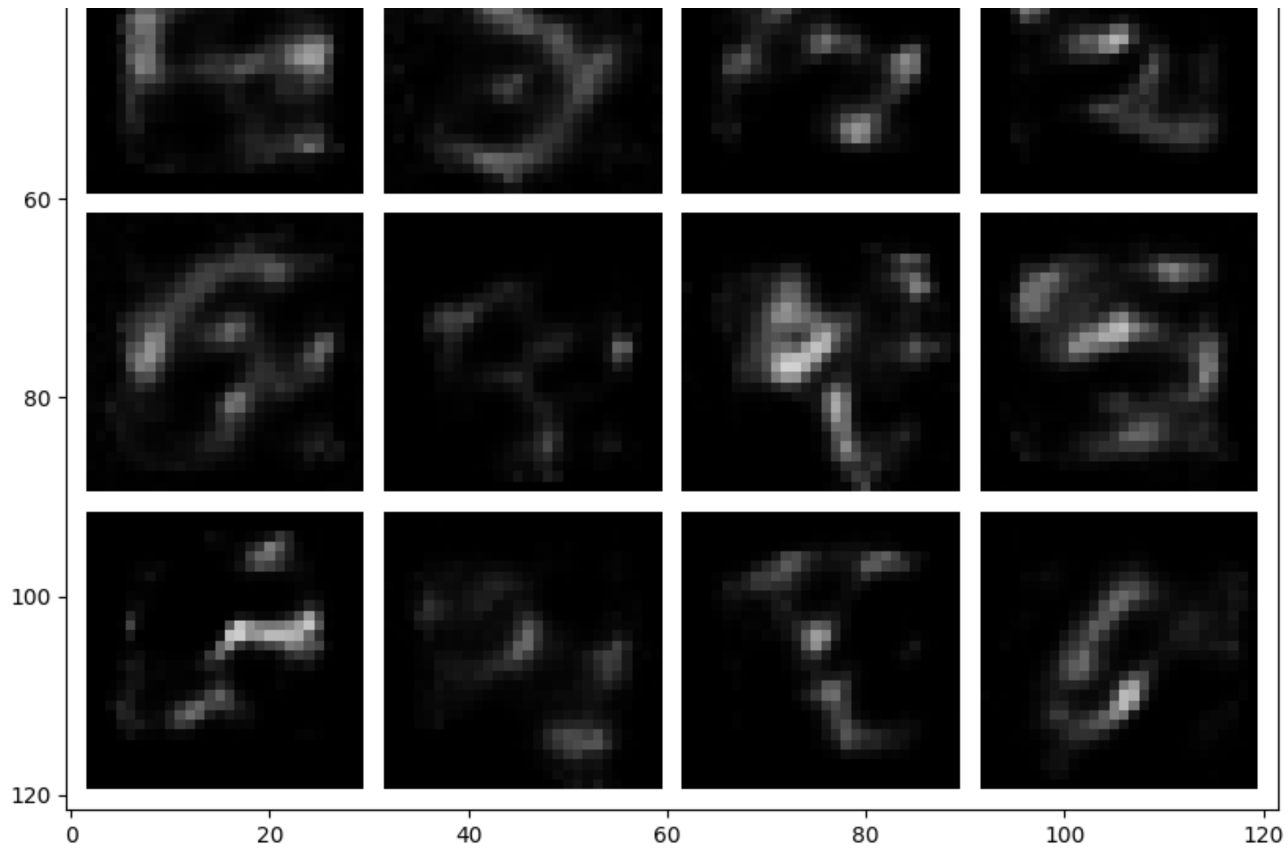


Sampling from the Auto-Encoder [2pt]

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
  # 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 th
  #       .mean() and .var() functions.
  # HINT: You can directly use model.decoder() to decode the samples.
  ##################################################################################
  samples = torch.randn(batch_size, nz).to(device)
  #diag_gaussian = np.random.multivariate_normal(0,1,(batch_size,nz))
  #samples = torch.from_numpy(diag_gaussian).to(device)
  decoded_samples = model.decoder(samples).view(batch_size,1,28,28)
  ##################################### END TODO ######################################

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

vis_samples(ae_model)
```





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

Answer: In general, the decoder expects a prior that comes from an encoded image of a number. By using a seed to generate the encoded nz, the decoder recieves an input it does not expect

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 input. This sampling is required to optimize the VAE objective also known as the evidence lower bound (ELBO):

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

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:

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

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:

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

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

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'
    #       mean and log_sigma, so the dimensionality of the output needs to
    #       double. The decoder works with an embedding sampled from this output.
    ################################################################################
    self.encoder = Encoder(nz=2*nz, input_size=in_size)
    self.decoder = Decoder(nz=nz, output_size=out_size)
    self.bce_loss = nn.BCELoss()

    ##################################### 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 Gaussia
    #          2. sample z from inferred posterior distribution using
    #             reparametrization trick
    #          3. decode the sampled z to obtain the reconstructed image
    ################################################################################
    q = self.encoder(x) # 1. (encode input x)
    mu, logsig = torch.chunk(q,2,dim=-1) #1. (split encoding)

    eps = torch.randn_like(mu)
    z = mu + eps * torch.exp(logsig) #2.

    # compute reconstruction
```

```
      reconstruction = self.decoder(z) #3.

      #################################### 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 Gaussi
      #             parametrized by mean and log_sigma
      # HINT: Make sure to compute the KL divergence in the correct order since it is
      #       not symmetric!!  ie. KL(p, q) != KL(q, p)
      ################################################################################
      rec_loss = self.bce_loss(outputs,x) #should be bce(output,input)

      q = self.encoder(x)
      q_mu, q_logsig = torch.chunk(q,2,dim=-1)

      # In p = N(0,1) p_mu=0, p_logsig=log(1)=0
      p_mu, p_logsig = torch.zeros(q_mu.shape).to(device),torch.zeros(q_logsig.shape)
      # KL(q,p) as specified by ELBO loss
      kl_loss = kl_divergence(q_mu,q_logsig,p_mu,p_logsig)
      kl_loss = torch.sum(kl_loss)/batch_size


      #################################### 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 th
      # 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 obt
      #       the reconstruction.
      ################################################################################
      q = self.encoder(x)
      q_mu, q_logsig = torch.chunk(q,2,dim=-1)
      flattened = self.decoder(q_mu)
      image = flattened.reshape(-1, 28, 28)
    #################################### END TODO ###################################
    return image
```

Setting up the VAE Training Loop [4pt]

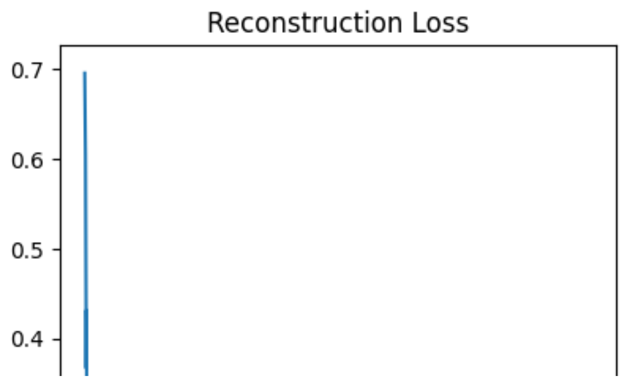
Let's start training the VAE model! We will first verify our implementation by setting $\beta = 0$.

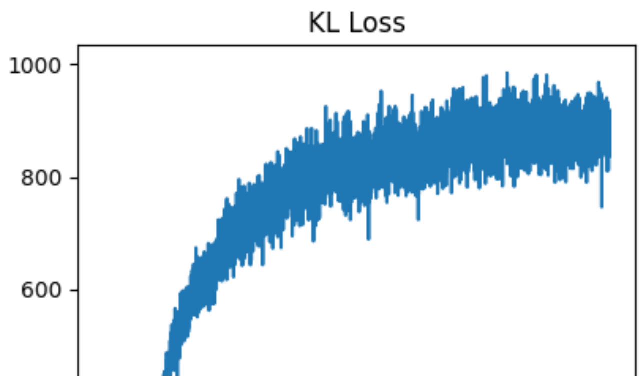
```
# Prob1-5 VAE training loop
learning_rate = 1e-3
nz = 32
beta = 0

######################################## 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

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

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

    input = sample_img.reshape([batch_size, in_size])
```

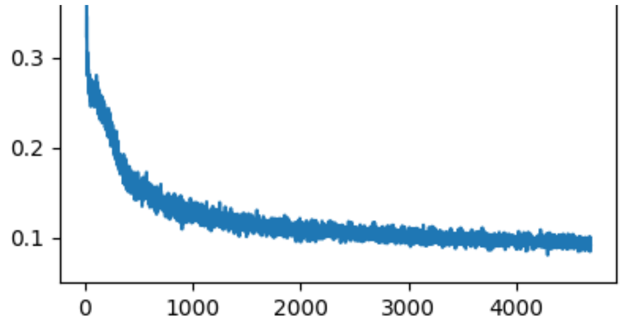
```
        optimizer.zero_grad()

        input_cuda = input.to(device)
        prediction = vae_model.forward(input_cuda)
        q, rec = prediction['q'], prediction['rec']
        #prediction = prediction.reshape([batch_size,in_size])
        rec = rec.reshape([batch_size,in_size])

        total_loss,losses = vae_model.loss(input_cuda,rec)
        #print(total_loss)
        #print(total_loss.shape)
        #print(total_loss)
        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()
```

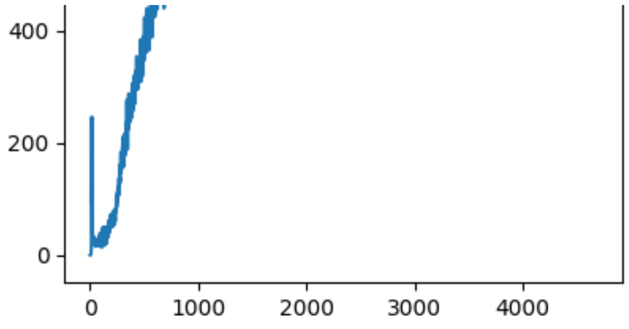
```
Running 5 epochs with beta=0
Run Epoch 0
It 0: Total Loss: 0.695181667804718,      Rec Loss: 0.695181667804718,     KL
It 100: Total Loss: 0.25128671526908875,     Rec Loss: 0.25128671526908875,
It 200: Total Loss: 0.2391764372587204,   Rec Loss: 0.2391764372587204,    KL
It 300: Total Loss: 0.20591431856155396,     Rec Loss: 0.20591431856155396,
It 400: Total Loss: 0.17898251116275787,     Rec Loss: 0.17898251116275787,
It 500: Total Loss: 0.1551220864057541,   Rec Loss: 0.1551220864057541,    KL
It 600: Total Loss: 0.1509290188550949,   Rec Loss: 0.1509290188550949,    KL
It 700: Total Loss: 0.14379703998565674,     Rec Loss: 0.14379703998565674,
It 800: Total Loss: 0.14535599946975708,     Rec Loss: 0.14535599946975708,
It 900: Total Loss: 0.1373644322156906,   Rec Loss: 0.1373644322156906,    KL
```

```
Run Epoch 1
It 1000: Total Loss: 0.1211744174361229,     Rec Loss: 0.1211744174361229,
It 1100: Total Loss: 0.12230156362056732,    Rec Loss: 0.12230156362056732,
It 1200: Total Loss: 0.12985381484031677,    Rec Loss: 0.12985381484031677,
It 1300: Total Loss: 0.11963105201721191,    Rec Loss: 0.11963105201721191,
It 1400: Total Loss: 0.11387616395950317,    Rec Loss: 0.11387616395950317,
It 1500: Total Loss: 0.123427614569664,   Rec Loss: 0.123427614569664,     KL
It 1600: Total Loss: 0.12236968427896 5,   Rec Loss: 0.12236968427896 5,     KL
It 1700: Total Loss: 0.10991916805505753,    Rec Loss: 0.10991916805505753,
It 1800: Total Loss: 0.10447939485311508,    Rec Loss: 0.10447939485311508,
Run Epoch 2
It 1900: Total Loss: 0.10880673676729202,    Rec Loss: 0.10880673676729202,
It 2000: Total Loss: 0.09521147608757019,    Rec Loss: 0.09521147608757019,
It 2100: Total Loss: 0.10607104003429413,    Rec Loss: 0.10607104003429413,
It 2200: Total Loss: 0.10478875786066055,    Rec Loss: 0.10478875786066055,
It 2300: Total Loss: 0.10391804575920105,    Rec Loss: 0.10391804575920105,
It 2400: Total Loss: 0.11021503806114197,    Rec Loss: 0.11021503806114197,
It 2500: Total Loss: 0.10728661715984344,    Rec Loss: 0.10728661715984344,
It 2600: Total Loss: 0.10223999619483948,    Rec Loss: 0.10223999619483948,
It 2700: Total Loss: 0.10572300851345062,    Rec Loss: 0.10572300851345062,
It 2800: Total Loss: 0.10877729952335358,    Rec Loss: 0.10877729952335358,
Run Epoch 3
It 2900: Total Loss: 0.10373447835445404,    Rec Loss: 0.10373447835445404,
It 3000: Total Loss: 0.10294432938098907,    Rec Loss: 0.10294432938098907,
It 3100: Total Loss: 0.1062920019030571,     Rec Loss: 0.1062920019030571,
It 3200: Total Loss: 0.10777934640645981,    Rec Loss: 0.10777934640645981,
It 3300: Total Loss: 0.10343660414218903,    Rec Loss: 0.10343660414218903,
It 3400: Total Loss: 0.10070425271987915,    Rec Loss: 0.10070425271987915,
It 3500: Total Loss: 0.10065937787294388,    Rec Loss: 0.10065937787294388,
It 3600: Total Loss: 0.09586238861083984,    Rec Loss: 0.09586238861083984,
It 3700: Total Loss: 0.09731833636760712,    Rec Loss: 0.09731833636760712,
Run Epoch 4
It 3800: Total Loss: 0.0975518524646759,     Rec Loss: 0.0975518524646759,
It 3900: Total Loss: 0.1040029376745224,     Rec Loss: 0.1040029376745224,
It 4000: Total Loss: 0.09543130546808243,    Rec Loss: 0.09543130546808243,
It 4100: Total Loss: 0.09476841241121292,    Rec Loss: 0.09476841241121292,
It 4200: Total Loss: 0.09623172134160995,    Rec Loss: 0.09623172134160995,
It 4300: Total Loss: 0.09614274650812149,    Rec Loss: 0.09614274650812149,
It 4400: Total Loss: 0.09131784737110138,    Rec Loss: 0.09131784737110138,
It 4500: Total Loss: 0.0943884626030922,     Rec Loss: 0.0943884626030922,
It 4600: Total Loss: 0.09198595583438873,    Rec Loss: 0.09198595583438873,
Done!
```





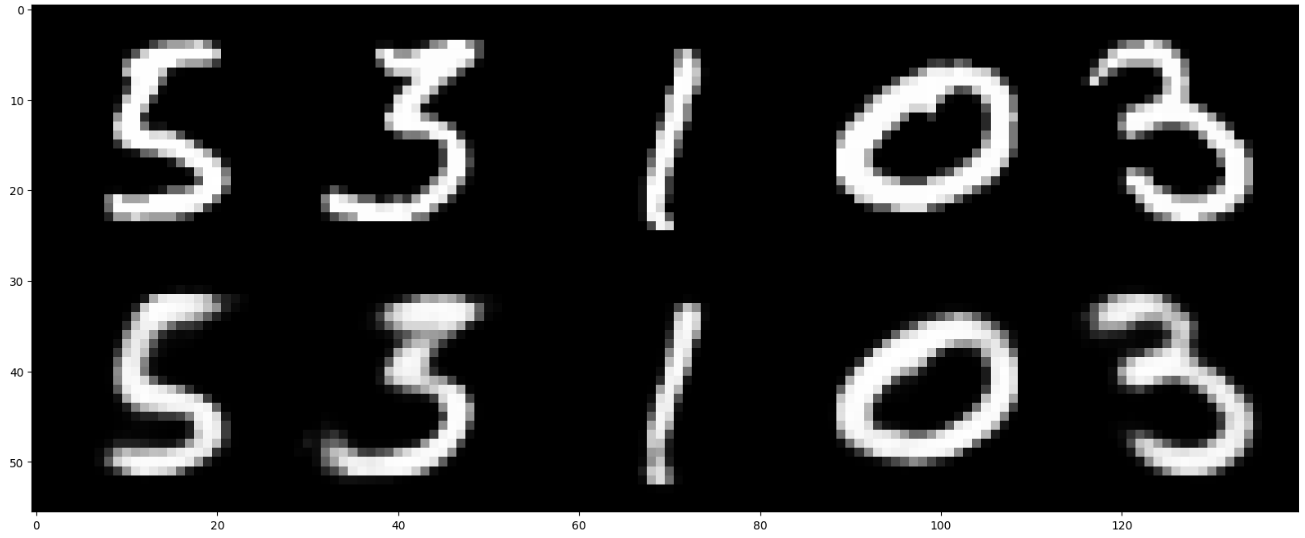


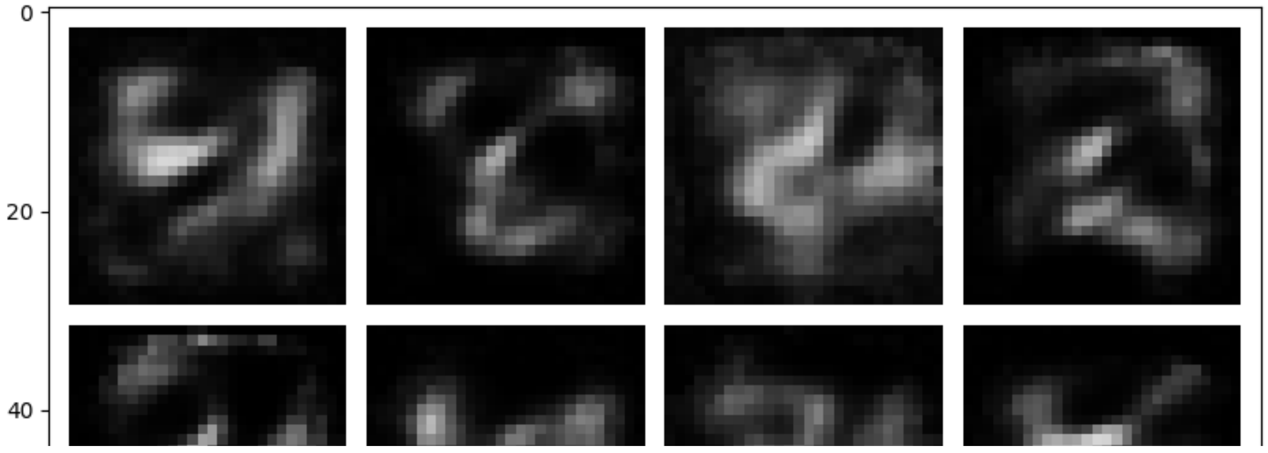


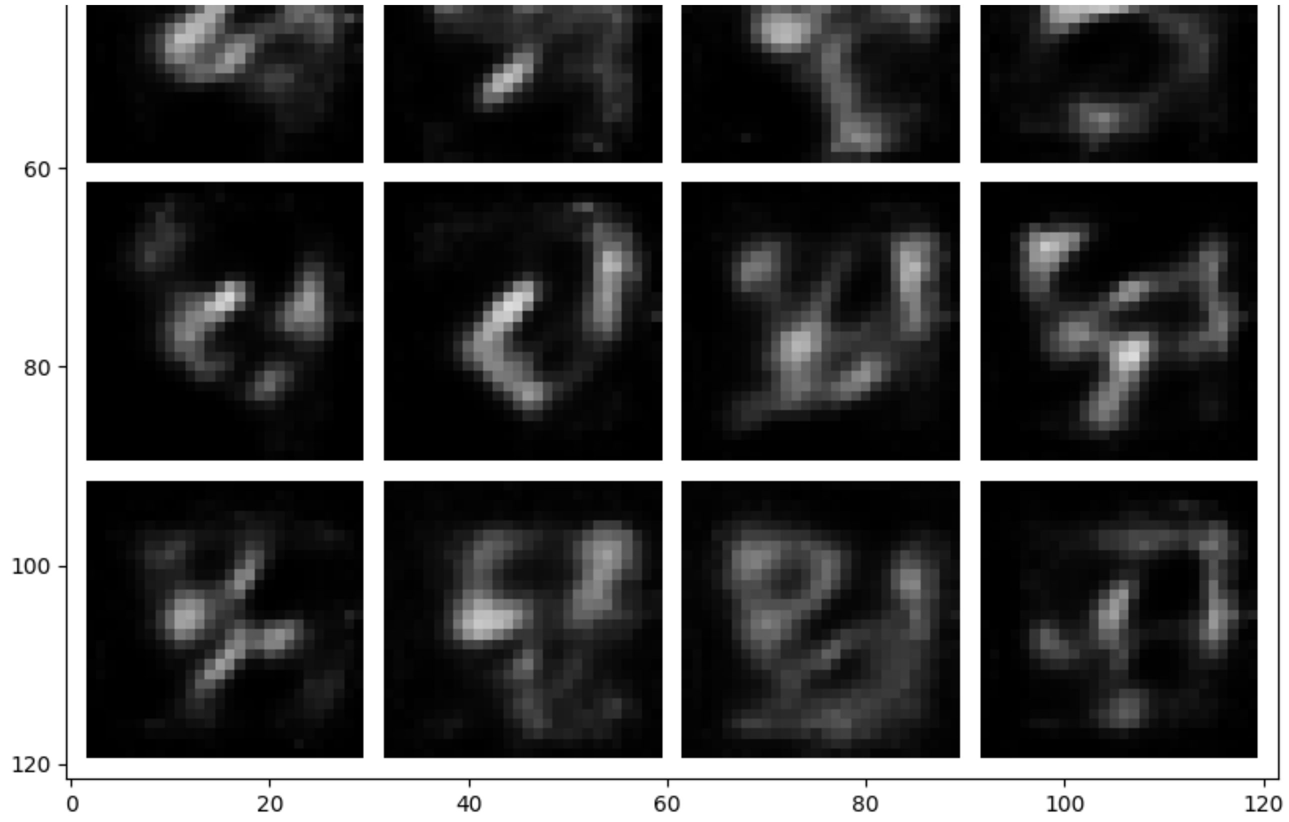
Let's look at some reconstructions and decoded embedding samples!

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

```
beta =  0
```






60
80
100
120
0
20
40
60
80
100
120

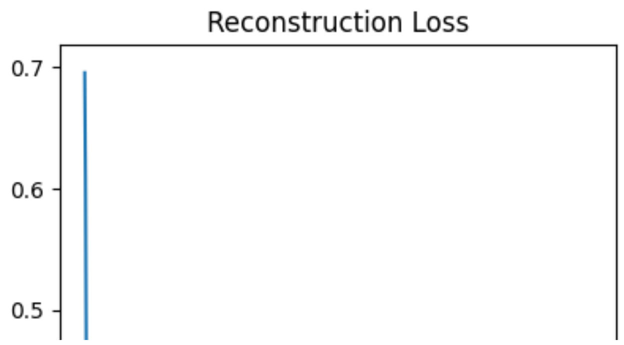
Tweaking the loss function β [2pt]

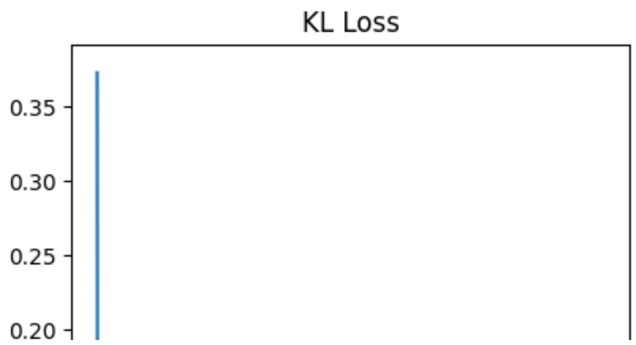
Prob1-6: Let's repeat the same experiment for $\beta = 10$, a very high value for the coefficient.

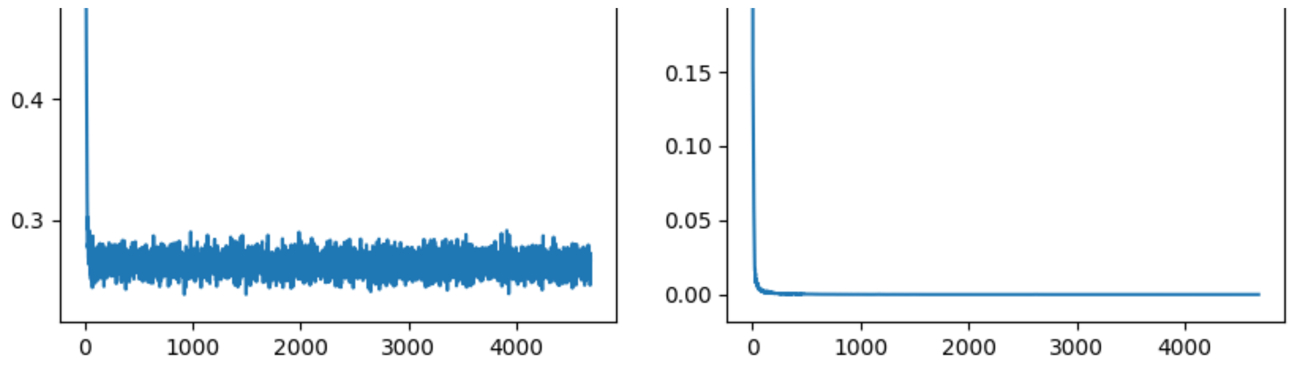
```
# VAE training loop
learning_rate = 1e-3
nz = 32
beta = 10

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

# 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

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

#################################### END TODO ######################################

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

    input = sample_img.reshape([batch_size, in_size])
    optimizer.zero_grad()
```

```
      input_cuda = input.to(device)
      prediction = vae_model.forward(input_cuda)
      q, rec = prediction['q'], prediction['rec']
      #prediction = prediction.reshape([batch_size,in_size])
      rec = rec.reshape([batch_size,in_size])

      total_loss,losses = vae_model.loss(input_cuda,rec)
      #print(total_loss)
      #print(total_loss.shape)
      #print(total_loss)
      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: 4.4247965812683105,    Rec Loss: 0.6956602334976196,    KL
It 100: Total Loss: 0.28082987666130066,     Rec Loss: 0.2626760005950928,
It 200: Total Loss: 0.266466349363327,   Rec Loss: 0.25904056429862976,   KL
It 300: Total Loss: 0.2691638767719269,  Rec Loss: 0.26327207684516907,   KL
It 400: Total Loss: 0.2785989046096802,  Rec Loss: 0.2750103175640106,    KL
It 500: Total Loss: 0.2668313980102539,  Rec Loss: 0.2647014260292053,    KL
It 600: Total Loss: 0.26306915283203125,     Rec Loss: 0.26170384883880615,
It 700: Total Loss: 0.26954880356788635,     Rec Loss: 0.26836153864860535,
It 800: Total Loss: 0.2698362171649933,  Rec Loss: 0.26878902316093445,   KL
It 900: Total Loss: 0.2547992467880249,  Rec Loss: 0.25418442487716675,   KL
Run Epoch 1
It 1000: Total Loss: 0.26306724548339844,    Rec Loss: 0.26254919171333313,
It 1100: Total Loss: 0.26944202184677124,    Rec Loss: 0.26901593804359436,
It 1200: Total Loss: 0.27351197600364685,    Rec Loss: 0.2731682062149048,
It 1300: Total Loss: 0.25392913818359375,    Rec Loss: 0.2535867989063263,
It 1400: Total Loss: 0.2696192264556885,     Rec Loss: 0.26942721009254456,
It 1500: Total Loss: 0.2597530782222748,     Rec Loss: 0.2594548463821411,
It 1600: Total Loss: 0.26407137513160706,    Rec Loss: 0.2639058232307434,
It 1700: Total Loss: 0.253385066986084,  Rec Loss: 0.2532432973384857,    KL
It 1800: Total Loss: 0.2669208347797394,     Rec Loss: 0.26676928997039795,
Run Epoch 2
It 1900: Total Loss: 0.26098576188087463,    Rec Loss: 0.2609044313430786,
It 2000: Total Loss: 0.26232025027275085,    Rec Loss: 0.2622188925743103,
It 2100: Total Loss: 0.2645381689071655,     Rec Loss: 0.26445022225379944,
It 2200: Total Loss: 0.2750239968299866,     Rec Loss: 0.27493226528167725,
It 2300: Total Loss: 0.2627781331539154,     Rec Loss: 0.2627277970314026,
It 2400: Total Loss: 0.25959062576293945,    Rec Loss: 0.2595142424106598,
It 2500: Total Loss: 0.2620534300804138,     Rec Loss: 0.26201075315475464,
It 2600: Total Loss: 0.26726752519607544,    Rec Loss: 0.267061710357666,
It 2700: Total Loss: 0.2651333510875702,     Rec Loss: 0.26505419611930847,
It 2800: Total Loss: 0.2624427378177643,     Rec Loss: 0.26239344477653503,
Run Epoch 3
It 2900: Total Loss: 0.26194238662719727,    Rec Loss: 0.2619076073169708,
It 3000: Total Loss: 0.2626136243343353,     Rec Loss: 0.2625880241394043,
It 3100: Total Loss: 0.26531991362571716,    Rec Loss: 0.2652973234653473,
It 3200: Total Loss: 0.26363036036491394,    Rec Loss: 0.263607919216156,
It 3300: Total Loss: 0.2592599391937256,     Rec Loss: 0.2591421604156494,
It 3400: Total Loss: 0.27803346514701843,    Rec Loss: 0.27794745564460754,
It 3500: Total Loss: 0.28258126974105835,    Rec Loss: 0.2824898660182953,
It 3600: Total Loss: 0.26536351442337036,    Rec Loss: 0.26530563831329346,
It 3700: Total Loss: 0.26158303022384644,    Rec Loss: 0.26144808530807495,
Run Epoch 4
It 3800: Total Loss: 0.26244184374809265,    Rec Loss: 0.26240965723991394,
It 3900: Total Loss: 0.25731995701789856,    Rec Loss: 0.2572704553604126,
It 4000: Total Loss: 0.2734385132789612,     Rec Loss: 0.27340149879455566,
It 4100: Total Loss: 0.2578706443309784,     Rec Loss: 0.25780370831489563,
It 4200: Total Loss: 0.27357977628707886,    Rec Loss: 0.2734690308570862,
It 4300: Total Loss: 0.2574811577796936,     Rec Loss: 0.25745460391044617,
It 4400: Total Loss: 0.26318755745887756,    Rec Loss: 0.26314932107925415,
It 4500: Total Loss: 0.26878437399864197,    Rec Loss: 0.2687363922595978,
It 4600: Total Loss: 0.2700561285018921,     Rec Loss: 0.27001404762268066,
Done!
```







> **Inline Question: What can you observe when setting $\beta = 0$ and $\beta = 10$? Explain your observations! [2pt]** (max 200 words)
>
> **Answer**: When $\beta = 0$ KL-loss has no effect on the total reconstruction loss since total_loss = rec_loss + beta(kl_loss) this makes it so that the only thing VAE does is add a little bit of gaussian regularization noise to the encoder. During training, KL-loss increase to a very high value as the encoder distribution strays very far away from the posterior gaussian distribution. When $\beta = 10$, KL-loss becomes a dominant term and forces encodings to be gaussian. This causes the reconstruction loss to remain relatively constant and not have as much of an effect on the gradient.

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 good result should be able to reconstruct dataset images relatively well (like in the original autoencoder) but also intermediate encodings generated from out of dataset examples should also be useful features for reproducing numbers. The β factors should be tuned to allow the autoencoder to make more generalized encodings instead of overfitting to images seen in the MNIST dataset while preserving reconstructions. Essentially both outputs from vis_reconstruction

> preserving reconstructions. Essentially both outputs from vis_reconstruction (reconstruct MNIST dataset) and vis_sample (generate encodings from out of dataset samples) should vaguely look like usable numbers.

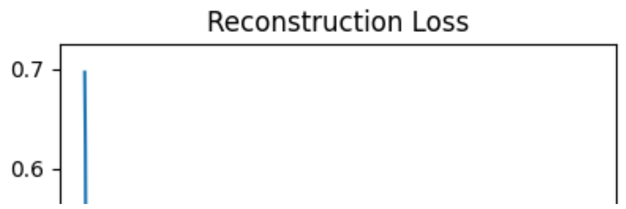
Now that you know what outcome we would like to obtain, try to tune β to achieve this result. Logarithmic search in steps of 10x will be helpful, good results can be achieved after ~20 epochs of training. Training reconstructions should be high quality, test samples should be diverse, distinguishable numbers, most samples recognizable as numbers.

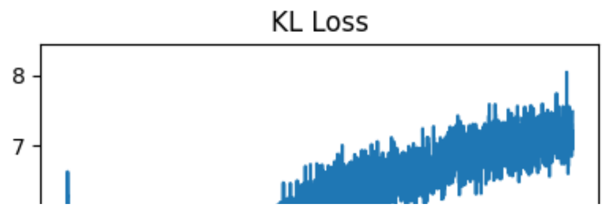
Answer: Tuned beta value *5e-3*_ [2pt]

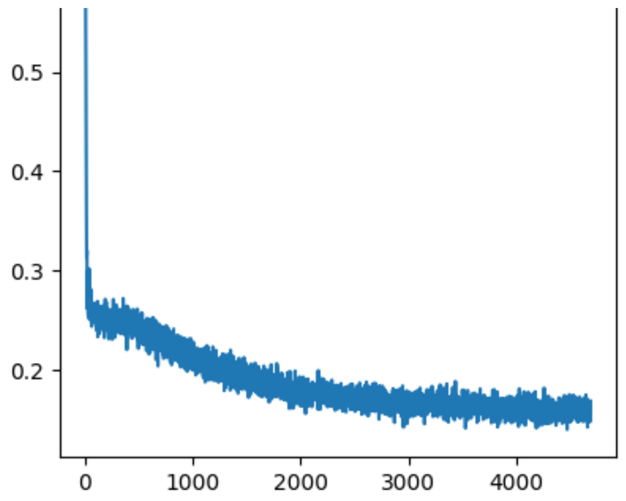
```
# Tuning for best beta
learning_rate = 1e-3
nz = 32

######################################## TODO ########################################
epochs = 5      # recommended 5-20 epochs
beta = 5e-3 # 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

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

##################################### END TODO ########################################

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

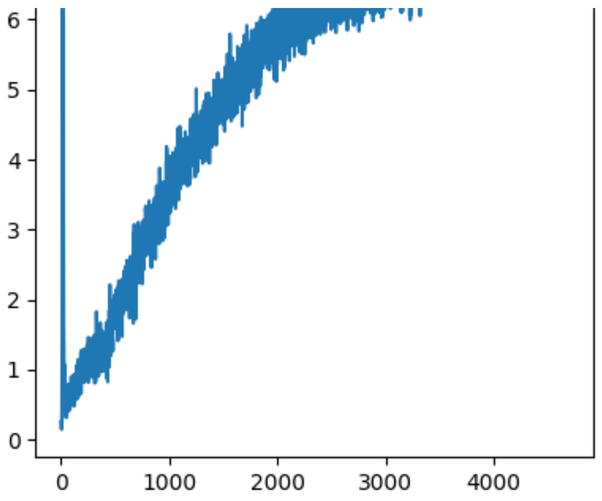
```
    """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
    for sample_img,sample_label in mnist_data_loader:

      input = sample_img.reshape([batch_size, in_size])
      optimizer.zero_grad()

      input_cuda = input.to(device)
      prediction = vae_model.forward(input_cuda)
      q, rec = prediction['q'], prediction['rec']
      #prediction = prediction.reshape([batch_size,in_size])
      rec = rec.reshape([batch_size,in_size])

      total_loss,losses = vae_model.loss(input_cuda,rec)
      #print(total_loss)
      #print(total_loss.shape)
      #print(total_loss)
      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: 0.6982954740524292,    Rec Loss: 0.6970106363296509,    KL
It 100: Total Loss: 0.25563496351242065,     Rec Loss: 0.252066969871521,
It 200: Total Loss: 0.24893948435783386,     Rec Loss: 0.2441517859697342,
It 300: Total Loss: 0.2644747197628021,   Rec Loss: 0.2591792047023773,    KL
It 400: Total Loss: 0.260764032602310 2,   Rec Loss: 0.255128413438797,     KL
It 500: Total Loss: 0.25507012009620667      Rec Loss: 0.2466602325439453
It 500: Total Loss: 0.25507012009620667,     Rec Loss: 0.24066025254594355,
It 600: Total Loss: 0.24842539429664612,     Rec Loss: 0.2388802319765091,
It 700: Total Loss: 0.24728378653526306,     Rec Loss: 0.23247388005256653,
It 800: Total Loss: 0.2310614436864853,   Rec Loss: 0.217431902885437,     KL
It 900: Total Loss: 0.23153185844421387,     Rec Loss: 0.21720196306705475,
Run Epoch 1
It 1000: Total Loss: 0.22887808084487915,    Rec Loss: 0.21137666702270508,
It 1100: Total Loss: 0.22808942198753357,    Rec Loss: 0.20864422619342804,
It 1200: Total Loss: 0.2252519428730011,     Rec Loss: 0.20435300469398499,
It 1300: Total Loss: 0.22695665061473846,    Rec Loss: 0.2053939402103424,
It 1400: Total Loss: 0.21708542108535767,    Rec Loss: 0.19338960945606232,
It 1500: Total Loss: 0.21928146481513977,    Rec Loss: 0.19473285973072052,
It 1600: Total Loss: 0.21335023641586304,    Rec Loss: 0.19003570079803467,
It 1700: Total Loss: 0.2157924473285675,     Rec Loss: 0.18892616033554077,
It 1800: Total Loss: 0.21635428071022034,    Rec Loss: 0.18854662775993347,
Run Epoch 2
It 1900: Total Loss: 0.22681750357151031,    Rec Loss: 0.19855822622776031,
It 2000: Total Loss: 0.20731064677238464,    Rec Loss: 0.17842997610569,   KL
It 2100: Total Loss: 0.20179685950279236,    Rec Loss: 0.17322294414043427,
It 2200: Total Loss: 0.1958099901676178,     Rec Loss: 0.16664959490299225,
It 2300: Total Loss: 0.21504303812980652,    Rec Loss: 0.18382816016674042,
It 2400: Total Loss: 0.20334254205226898,    Rec Loss: 0.17113031446933746,
It 2500: Total Loss: 0.20795683562755585,    Rec Loss: 0.17743059992790222,
It 2600: Total Loss: 0.20805290341377258,    Rec Loss: 0.17685078084468842,
It 2700: Total Loss: 0.2069604992866516,     Rec Loss: 0.17420990765094757,
It 2800: Total Loss: 0.20569074153900146,    Rec Loss: 0.17396973073482513,
Run Epoch 3
It 2900: Total Loss: 0.19109103083610535,    Rec Loss: 0.16009995341300964,
It 3000: Total Loss: 0.19384905695915222,    Rec Loss: 0.16046179831027985,
It 3100: Total Loss: 0.19403672218322754,    Rec Loss: 0.1608966588973999,
It 3200: Total Loss: 0.2036001980304718,     Rec Loss: 0.17045339941978455,
It 3300: Total Loss: 0.2086380273103714,     Rec Loss: 0.17467322945594788,
It 3400: Total Loss: 0.1963382363319397,     Rec Loss: 0.161451295018196l,
It 3500: Total Loss: 0.199398204684257 5,    Rec Loss: 0.16615310311317444,
It 3600: Total Loss: 0.19897247850894928,    Rec Loss: 0.16574493050575256,
It 3700: Total Loss: 0.18761609494686127,    Rec Loss: 0.15470589697360992,
Run Epoch 4
It 3800: Total Loss: 0.18445999920368195,    Rec Loss: 0.14897800981998444,
It 3900: Total Loss: 0.20037783682346344,    Rec Loss: 0.16586564481258392,
It 4000: Total Loss: 0.19738073647022247,    Rec Loss: 0.16355939209461212,
It 4100: Total Loss: 0.20442984998226166,    Rec Loss: 0.17005440592765808,
It 4200: Total Loss: 0.189599871635437,   Rec Loss: 0.15667970478534698,   KL
It 4300: Total Loss: 0.19928795099258423,    Rec Loss: 0.16319359838962555,
It 4400: Total Loss: 0.20433777570724487,    Rec Loss: 0.16874925792217255,
It 4500: Total Loss: 0.19931171834468842,    Rec Loss: 0.16438885033130646,
It 4600: Total Loss: 0.20197032392024994,    Rec Loss: 0.16591955721378326,
Done!
```





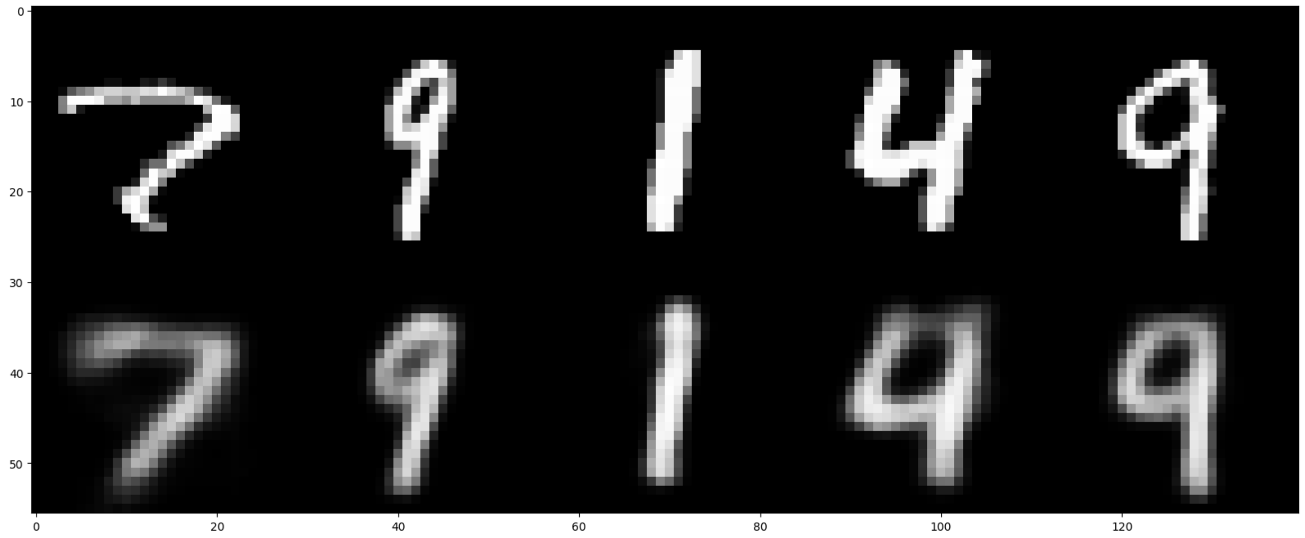


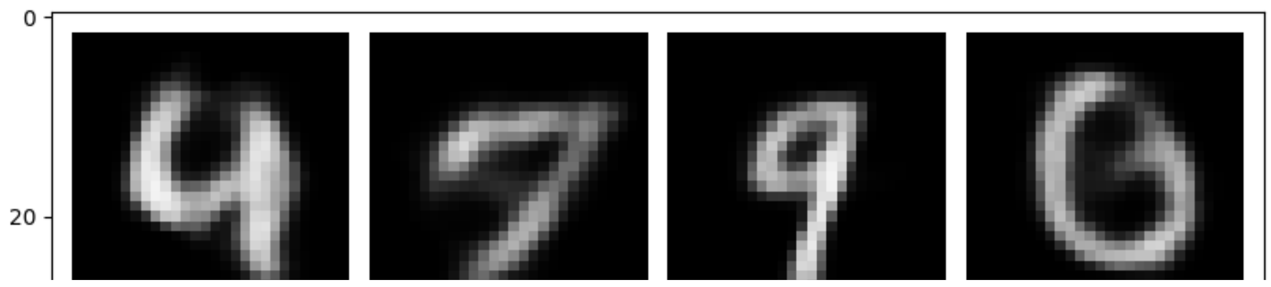


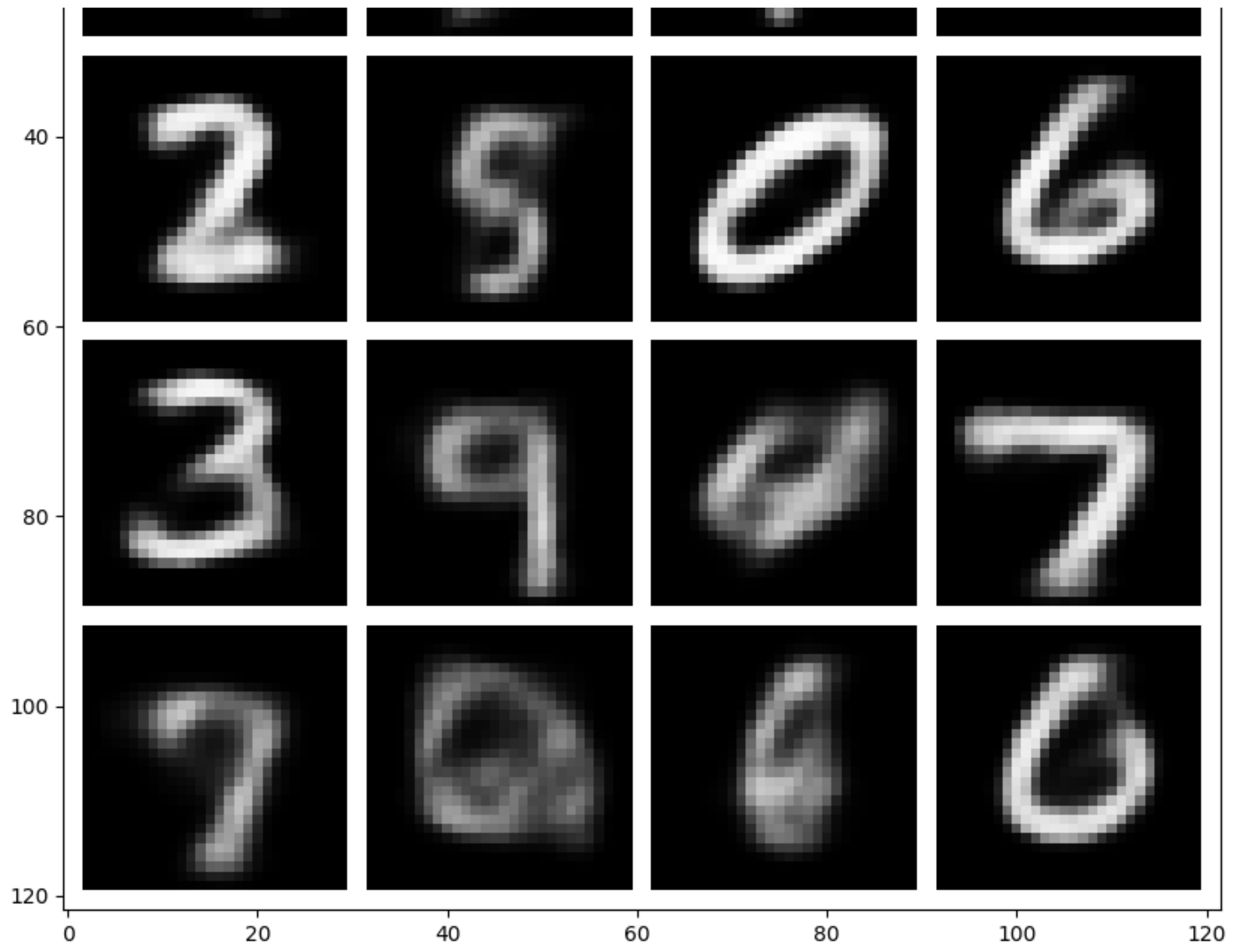
Let's look at some reconstructions and decoded embedding samples for this beta!

```
# [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
```



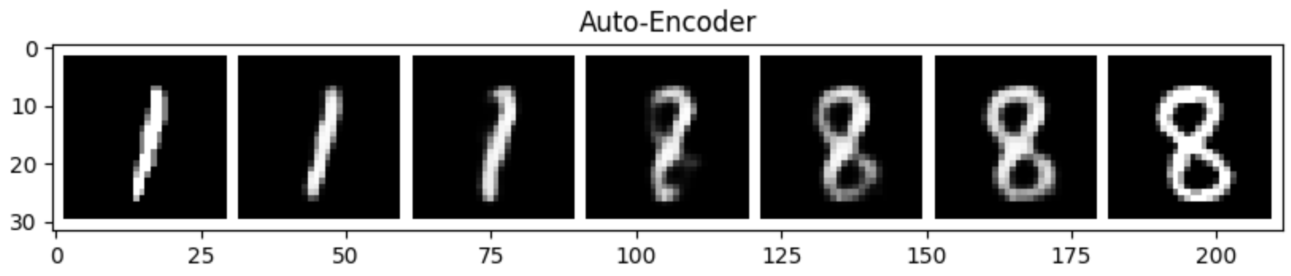


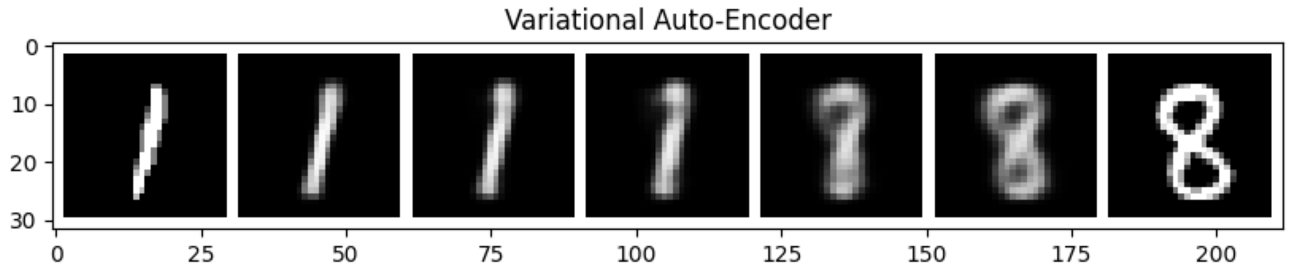

40
60
80
100
120
0
20
40
60
80
100
120

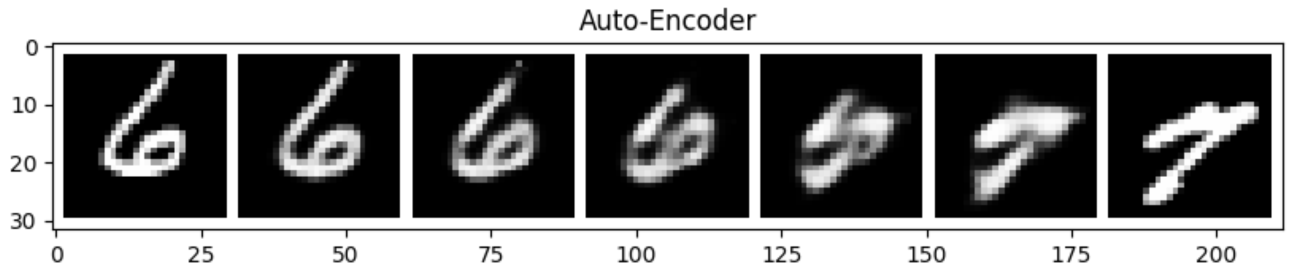
4. Embedding Space Interpolation [3pt]

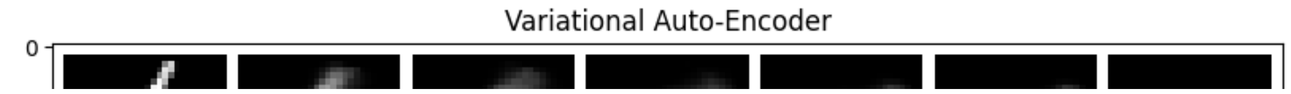
As mentioned in the introduction, AEs and VAEs cannot only be used to generate images, but also to learn low-dimensional representations of their inputs. In this final section we will investigate the representations we learned with both models by **interpolating in embedding space** between different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

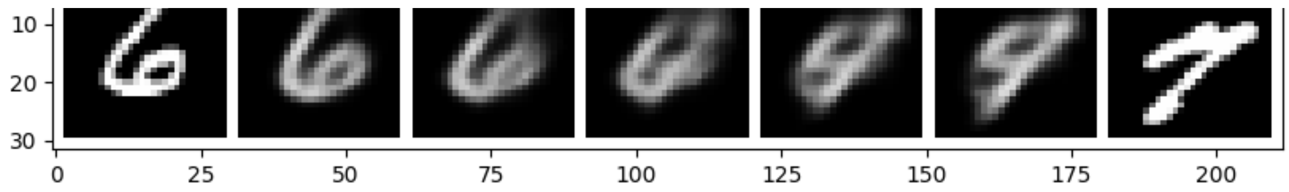
```
# Prob1-7
nz=32

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_S

  # 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=
                .data.cpu().numpy().transpose(1, 2, 0), cmap='gray')
  plt.title(tag)
  plt.show()


### Interpolation 1
START_LABEL = 1
END_LABEL = 8
# 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 = 6
END_LABEL = 7
# 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. The interpolations in VAE appear to be more gradual than AE since the change between immediate pictures are relatively abrupt in the autoencoder. In some blurry labels, the number gets de-blurred during its interpolation
> 2. The VAE encodings are probably more useful since they wont overfit as easily to specific labels and can provide better "in-between" values for images that are not clearly one number or the other.

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.

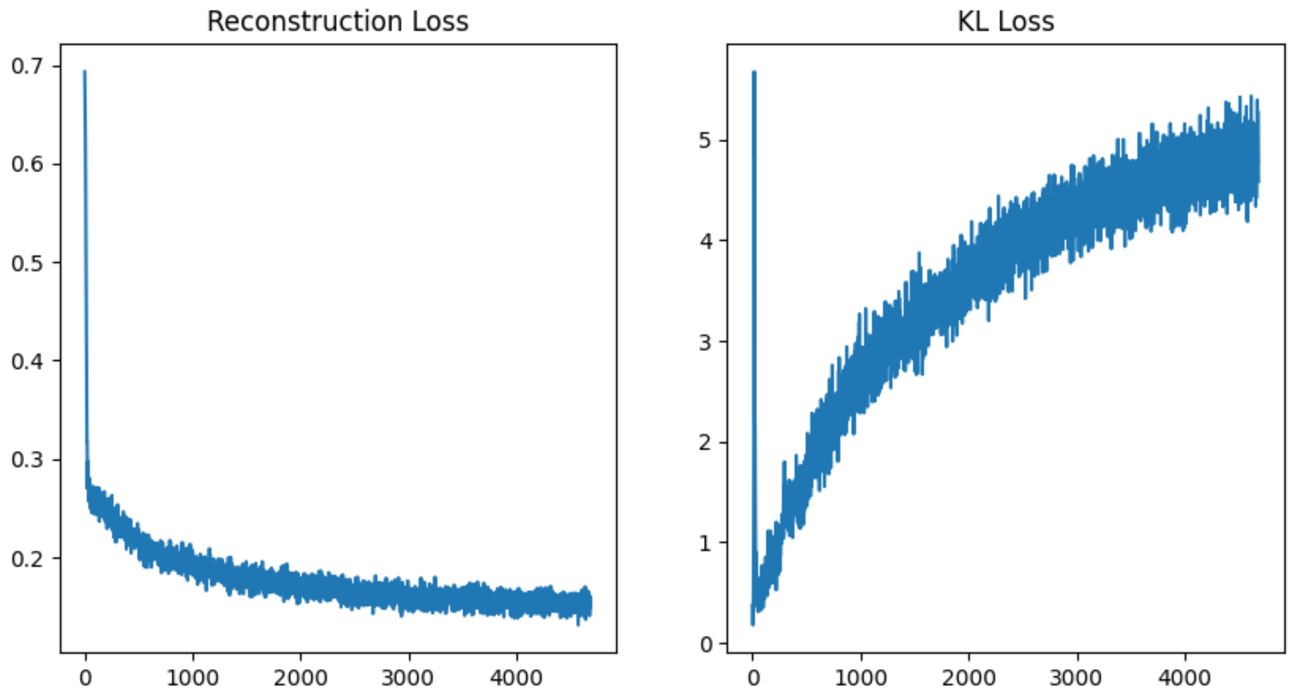
```
def idx2onehot(idx, n):
    """Converts a batch of indices to a one-hot representation."""
    assert torch.max(idx).item() < n
    if idx.dim() == 1:
        idx = idx.unsqueeze(1)
    onehot = torch.zeros(idx.size(0), n).to(idx.device)
    onehot.scatter_(1, idx, 1)

    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.
    ############################################################################
    onehot = idx2onehot(c,self.num_labels).to(device)
    x = torch.cat([x,onehot],-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.
    ##########################################################################
    onehot = idx2onehot(c,self.num_labels).to(device)
    z = torch.cat([z,onehot],-1)
    ################################ END TODO ################################

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


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


    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 abo
        #          2. compute the KL divergence loss between the inferred posterior
        #             distribution and a unit Gaussian prior; you can use the provi
        #             function above for computing the KL divergence between two Ga
        #             parametrized by mean and log_sigma
        # HINT: Make sure to compute the KL divergence in the correct order since i
        #       not symmetric!!  ie. KL(p, q) != KL(q, p)
        ###########################################################################
        q, rec, c = outputs['q'], outputs['rec'], outputs['c']

        rec = rec.reshape([batch_size,in_size])

        rec_loss = self.bce_loss(rec,x) #should be bce(output,input)

        q_mu, q_logsig = torch.chunk(q,2,dim=-1)

        # In p = N(0,1) p_mu=0, p_logsig=log(1)=0
        p_mu, p_logsig = torch.zeros(q_mu.shape).to(device),torch.zeros(q_logsig.sh
        # KL(q,p) as specified by ELBO loss
        kl_loss = kl_divergence(q_mu,q_logsig,p_mu,p_logsig)
        kl_loss = torch.sum(kl_loss)/batch_size

        #################################### 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. T
        # obtain the maximum likelihood estimate we bypass the sampling procedure o
        # inferred latent and instead directly use the mean of the inferred posteri
        # HINT: encode the input image and then decode the mean of the posterior to
        #       the reconstruction.
        #################################################################################
        q = self.encoder(x)
        q_mu, q_logsig = torch.chunk(q,2,dim=-1)
        flattened = self.decoder(q_mu)
        image = flattened.reshape(-1, 28, 28)
        ###################################### 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 = 5e-3
###################################### END TODO ####################################

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

# 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 ar
  #       perform one gradient step with the optimizer.
  # HINT: Don't forget to erase old gradients before performing the backward pass.
  # HINT: As before, we will use the loss() function of our model for computing the
  #       training loss. It outputs the total training loss and a dict containing
  #       the breakdown of reconstruction and KL loss.
  ################################################################################
  for sample_img,sample_label in mnist_data_loader:

    input = sample_img.reshape([batch_size, in_size])
    optimizer.zero_grad()

    input_cuda = input.to(device)
    prediction = cvae_model.forward(input_cuda,sample_label)


    total_loss,losses = cvae_model.loss(input_cuda,prediction)

    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: 0.6952542662620544,    Rec Loss: 0.6933870911598206,    KL
    It 100: Total Loss: 0.2522275447845459,  Rec Loss: 0.24970632791519165,   KL
    It 200: Total Loss: 0.2576017677783966,  Rec Loss: 0.2537820339202881,    KL
    It 300: Total Loss: 0.2389606535434723,  Rec Loss: 0.23318414390087128,   KL
    It 400: Total Loss: 0.2350231409072876,  Rec Loss: 0.22742484509944916,   KL
    It 500: Total Loss: 0.23685193061828613,     Rec Loss: 0.22806739807128906,
    It 600: Total Loss: 0.21701402962207794,     Rec Loss: 0.20569159090518951,
    It 700: Total Loss: 0.21746103465557098,     Rec Loss: 0.20686334371566772,
    It 800: Total Loss: 0.20803217589855194,     Rec Loss: 0.19659939408302307,
    It 900: Total Loss: 0.20884250104427338,     Rec Loss: 0.19539423286914825,
    Run Epoch 1
    It 1000: Total Loss: 0.20615154504776,   Rec Loss: 0.19315668940544128,   KL
    It 1100: Total Loss: 0.1878625899553299,     Rec Loss: 0.17484988272190094,
    It 1200: Total Loss: 0.20012341439723969,    Rec Loss: 0.18620148301124573,
    It 1300: Total Loss: 0.204044908285141,  Rec Loss: 0.19020876288414001,   KL
    It 1400: Total Loss: 0.2015991359949112,     Rec Loss: 0.18657350540161133,
    It 1500: Total Loss: 0.1882256418466568,     Rec Loss: 0.17056243121623993,
    It 1600: Total Loss: 0.19344797730445862,    Rec Loss: 0.17676684260368347,
    It 1700: Total Loss: 0.1983565390110016,     Rec Loss: 0.1827373057603836,
    It 1800: Total Loss: 0.1922939419746399,     Rec Loss: 0.17606376111507416,
    Run Epoch 2
    It 1900: Total Loss: 0.18830160796642303,    Rec Loss: 0.17125354707241058,
    It 2000: Total Loss: 0.18949031829833984,    Rec Loss: 0.1716623455286026,
    It 2100: Total Loss: 0.1781170815229416,     Rec Loss: 0.16078729927539825,
    It 2200: Total Loss: 0.17748747766017914,    Rec Loss: 0.159993514418602,
    It 2300: Total Loss: 0.19229847192764282,    Rec Loss: 0.17308880388736725,
    It 2400: Total Loss: 0.17546427249908447,    Rec Loss: 0.15583889186382294,
    It 2500: Total Loss: 0.1771366149187088,     Rec Loss: 0.1579066962003708,
    It 2600: Total Loss: 0.1904798448085785,     Rec Loss: 0.16941356658935547,
    It 2700: Total Loss: 0.1771659106016159,     Rec Loss: 0.15522131323814392,
    It 2800: Total Loss: 0.19718176126480103,    Rec Loss: 0.1745477020740509,
    Run Epoch 3
    It 2900: Total Loss: 0.18245911598205566,    Rec Loss: 0.16142591834068298,
    It 3000: Total Loss: 0.17998453974723816,    Rec Loss: 0.1574341058731079,
    It 3100: Total Loss: 0.18420955538749695,    Rec Loss: 0.16196145117282867,
    It 3200: Total Loss: 0.1833973228931427,     Rec Loss: 0.16048841178417206,
    It 3300: Total Loss: 0.16899164021015167,    Rec Loss: 0.14727789163589478,
    It 3400: Total Loss: 0.17065085470676422,    Rec Loss: 0.14882682263851166,
    It 3500: Total Loss: 0.1732618510723114,     Rec Loss: 0.15163007378578186,
    It 3600: Total Loss: 0.17250166833400726,    Rec Loss: 0.15032121539115906,
    It 3700: Total Loss: 0.1900259256362915,     Rec Loss: 0.16423606872558594,
    Run Epoch 4
    It 3800: Total Loss: 0.17977219820022583,    Rec Loss: 0.1558186113834381,
    It 3900: Total Loss: 0.17609700560569763,    Rec Loss: 0.1519889384508133,
    It 4000: Total Loss: 0.18475675582885742,    Rec Loss: 0.15897135436534882,
    It 4100: Total Loss: 0.17418712377548218,    Rec Loss: 0.15073177218437195,
    It 4200: Total Loss: 0.16669651865959167,    Rec Loss: 0.14271005988121033,
    It 4300: Total Loss: 0.1889031082391739,     Rec Loss: 0.16556809842586517,
    It 4300: Total Loss: 0.1889051082391739,     Rec Loss: 0.1655680984258651 7,
    It 4400: Total Loss: 0.1766437143087387,     Rec Loss: 0.15228933095932007,
    It 4500: Total Loss: 0.1814947873353958,     Rec Loss: 0.15706801414489746,
    It 4600: Total Loss: 0.1740051507949829,     Rec Loss: 0.1508335918188095,
    Done!
```

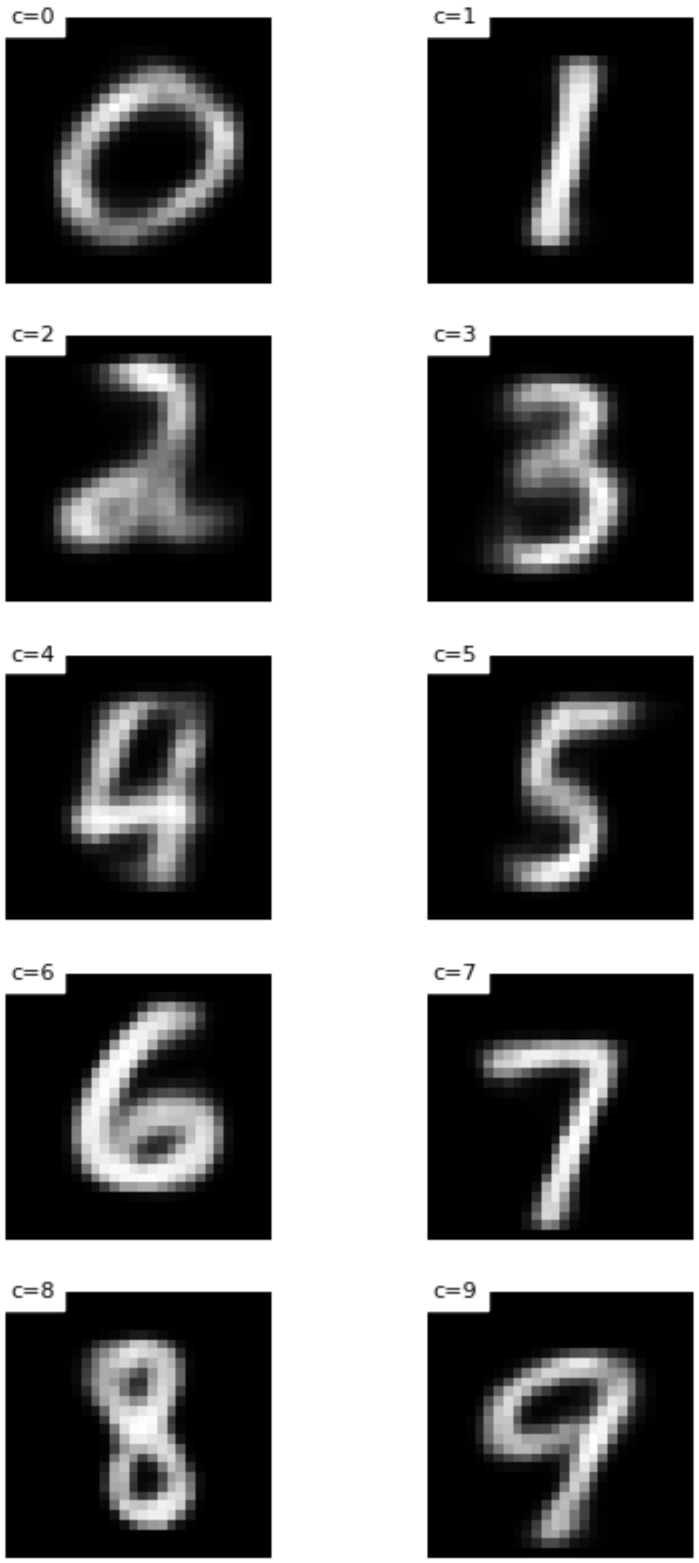


Verifying conditional samples from CVAE [6 pt]

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'
            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

Submission instructions

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