Programming Exercise 2

In this exercise, we will show you an example about how to train a VAE model for generating handwritten digits.

1. Prepare environment

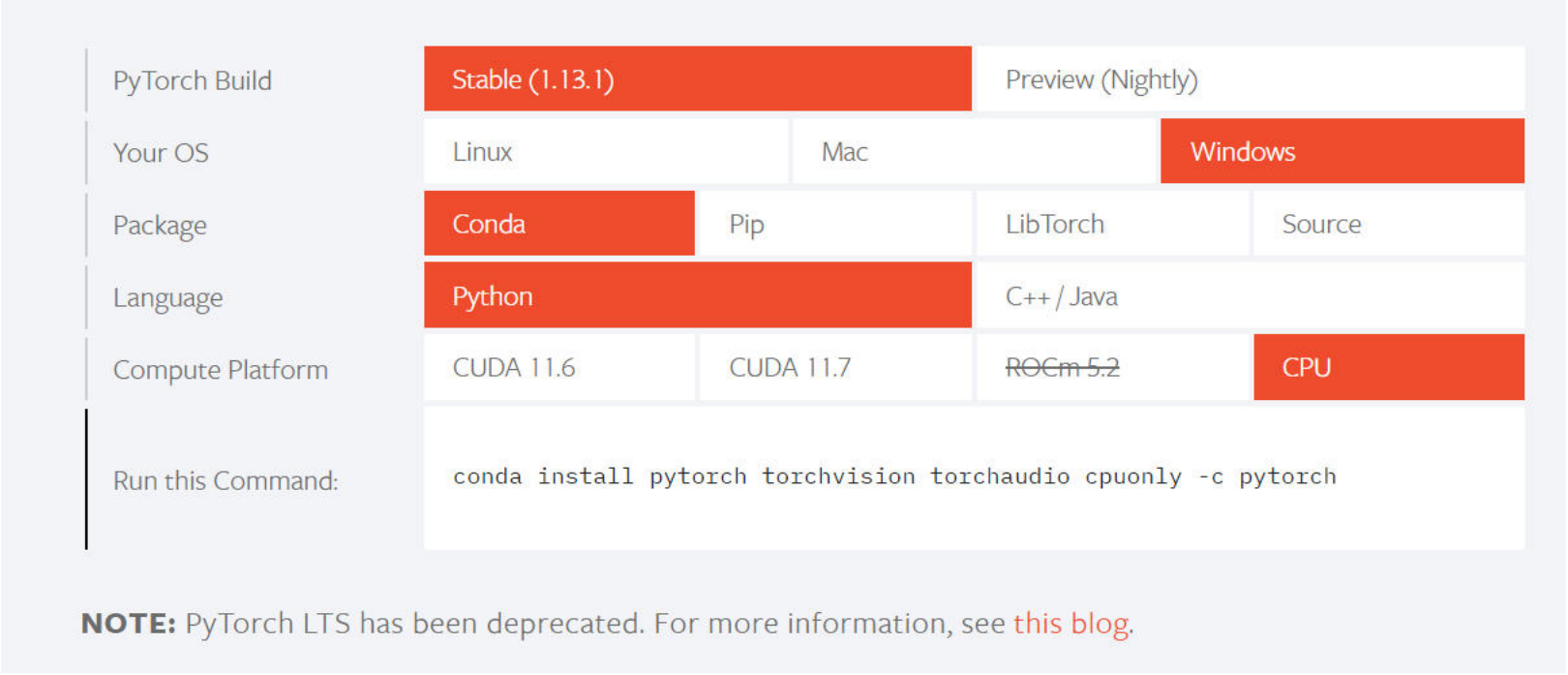
If you have installed following libraries in your python environment, you can skip this step.

```
pytorch
torchvision
numpy
matplotlib
tqdm
```

Otherwise, you can create a new Anaconda environment:

```
conda create -n vae python=3.6.5
conda activate vae
```

And follow the instruction of installing pytorch according to your device.



Then install other dependencies:

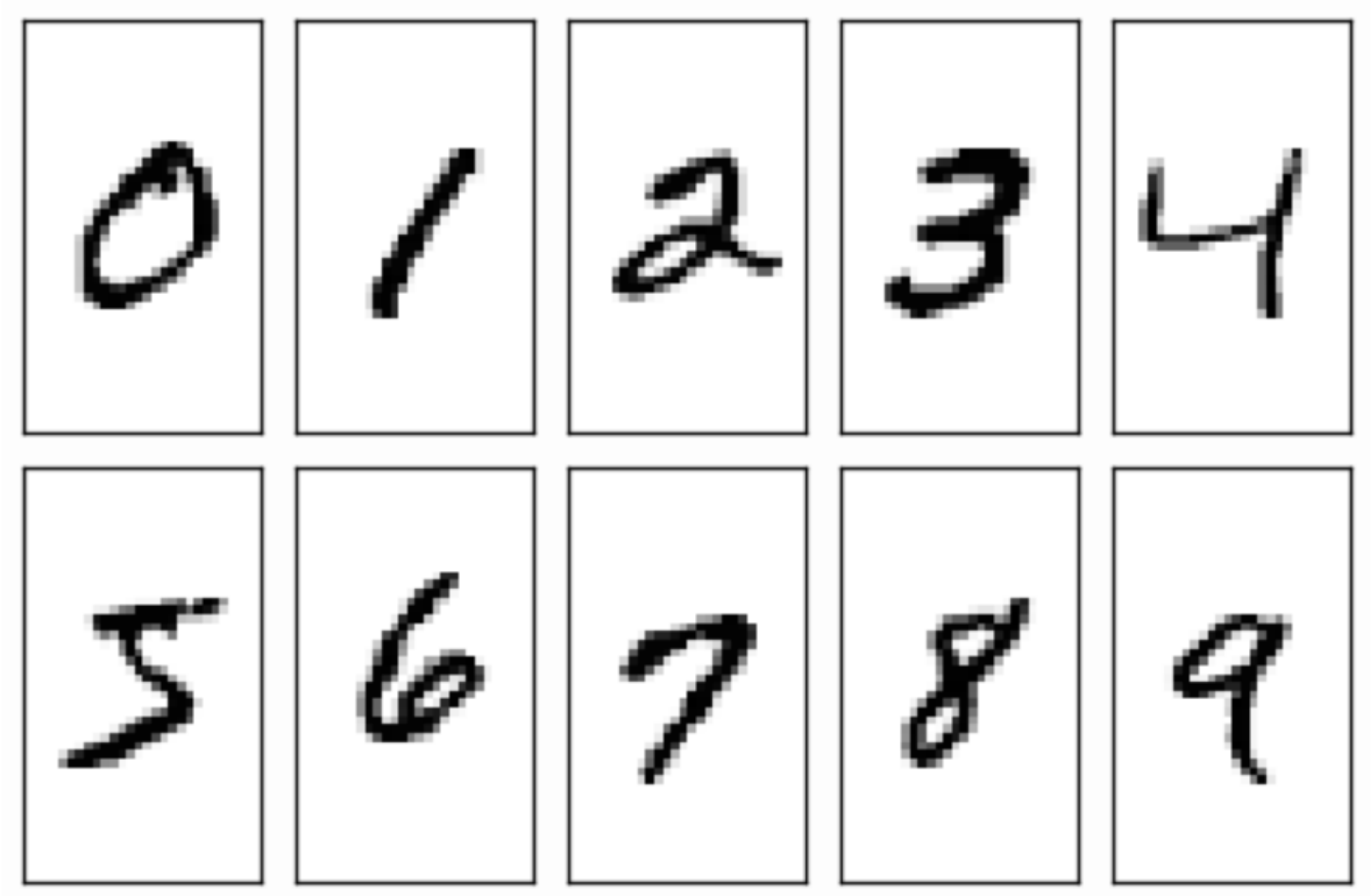
```
conda install numpy
conda install matplotlib
conda install tqdm
```

2. VAE for generating handwritten digits

First, create a new python script and import libraries below:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

import numpy as np
import os

from tqdm import tqdm
from torchvision.utils import save_image, make_grid
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

from torch.optim import Adam

import matplotlib.pyplot as plt
```

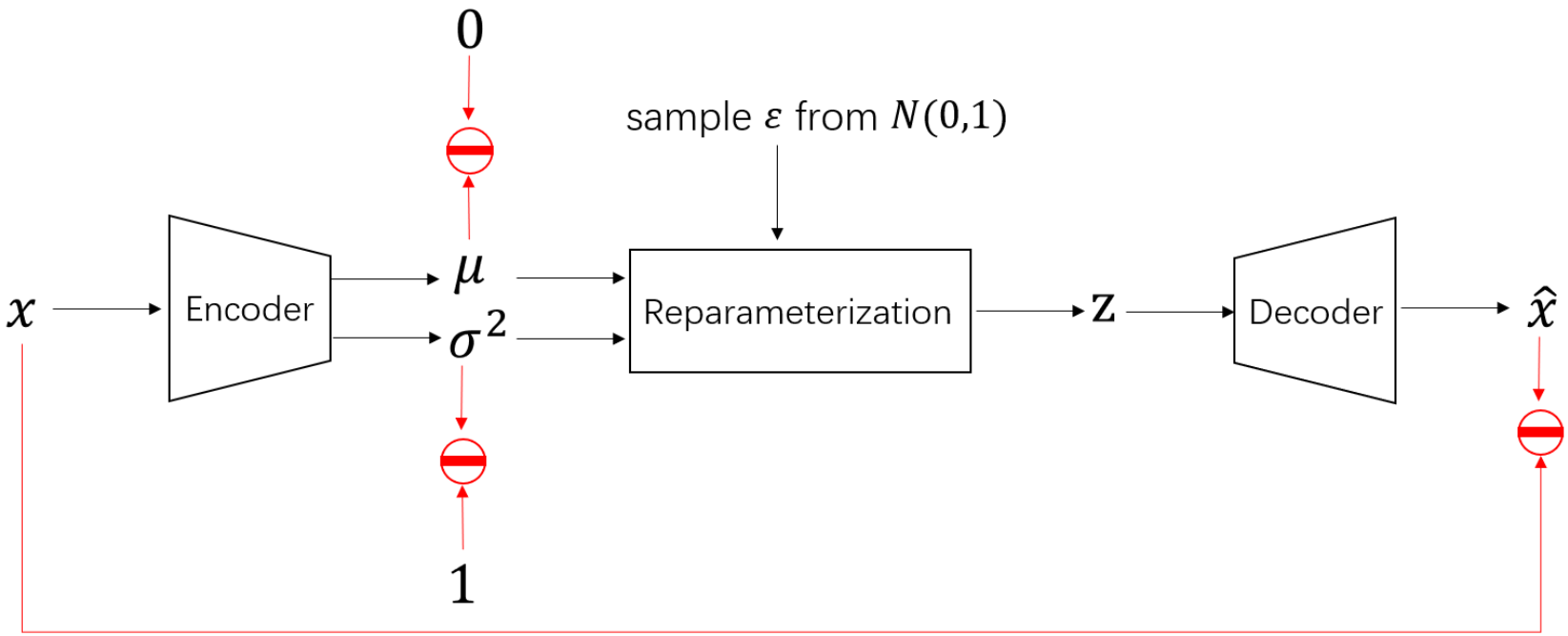
This experiment adopts the handwritten digits data from MNIST. The figure below illustrates some instances from MNIST dataset.



Then we define the data loader function.

```
def load_data(dataset_path, batch_size):

    mnist_transform = transforms.Compose([transforms.ToTensor()])

    kwargs = {'num_workers': 1, 'pin_memory': True}

    train_dataset = MNIST(dataset_path, transform=mnist_transform, train=True, download=True)
    test_dataset  = MNIST(dataset_path, transform=mnist_transform, train=False, download=True)
```

```
    train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True, **kwargs)
    test_loader  = DataLoader(dataset=test_dataset,  batch_size=batch_size, shuffle=False, **kwargs)

    return train_loader, test_loader
```



The VAE model contains an encoder and a decoder. We first define the encoder which is composed of four fully connected layers. Note that for the ease of computation, we let the encoder produces **log of variance**.

```
class Encoder(nn.Module):

    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(Encoder, self).__init__()

        self.FC_input = nn.Linear(input_dim, hidden_dim)
        self.FC_input2 = nn.Linear(hidden_dim, hidden_dim)
        self.FC_mean  = nn.Linear(hidden_dim, latent_dim)
        self.FC_var   = nn.Linear (hidden_dim, latent_dim)

        self.LeakyReLU = nn.LeakyReLU(0.2)
```

```
        self.training = True

    def forward(self, x):
        h_       = self.LeakyReLU(self.FC_input(x))
        h_       = self.LeakyReLU(self.FC_input2(h_))
        mean     = self.FC_mean(h_)
        log_var  = self.FC_var(h_)   #log of variance

        return mean, log_var
```

The decoder is also a fully connected network with 3 layers.

```
class Decoder(nn.Module):
    def __init__(self, latent_dim, hidden_dim, output_dim):
        super(Decoder, self).__init__()
        self.FC_hidden = nn.Linear(latent_dim, hidden_dim)
        self.FC_hidden2 = nn.Linear(hidden_dim, hidden_dim)
        self.FC_output = nn.Linear(hidden_dim, output_dim)

        self.LeakyReLU = nn.LeakyReLU(0.2)

    def forward(self, x):
        h     = self.LeakyReLU(self.FC_hidden(x))
        h     = self.LeakyReLU(self.FC_hidden2(h))

        x_hat = torch.sigmoid(self.FC_output(h))
        return x_hat
```

Then the complete VAE model is defined as:

```
class Model(nn.Module):
    def __init__(self, Encoder, Decoder, device):
        super(Model, self).__init__()
        self.Encoder = Encoder
        self.Decoder = Decoder
        self.device = device

    def reparameterization(self, mean, var):
        epsilon = torch.randn_like(var).to(self.device)        # sampling epsilon
        z = mean + var*epsilon                          # reparameterization trick
```

```
        return z


    def forward(self, x ):
        mean, log_var = self.Encoder(x)
        z = self.reparameterization(mean, torch.exp(0.5 * log_var))
        x_hat            = self.Decoder(z)

        return x_hat, mean, log_var
```

In addition, we need the loss function to train the model. The reproduction loss (binary cross entropy) aims to minimize the difference between the input image and the recovered image. The KL-Divergence loss `KLD` minimizes the KL divergence between the posterior distribution of embedded images and the normal distribution. In this implementation, we assume the posterior distribution has diagonal covariance structure: $p(\mathbf{z}|\mathbf{x}^{(i)}) = \mathcal{N}(\mathbf{z}; \mu^{(i)}, (\sigma^{(\mathbf{i})})^{\mathbf{2}}\mathbf{I})$. Suppose the latent embedding $\mathbf{z}$ has k dimensions, the KL divergence loss for sample $\mathbf{x}^{(i)}$ can be written as:

$$D_{KL}(\mathcal{N}(\mu^{(i)}, (\sigma^{(i)})^2 I)||\mathcal{N}(0, I)) = -\frac{1}{2}\sum_{j=1}^{k}\left(1 + \log\left((\sigma_j^{(i)})^2\right) - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2\right).$$

```
def loss_function(x, x_hat, mean, log_var):

    reproduction_loss = nn.functional.binary_cross_entropy(x_hat, x, reduction='sum')
    KLD      = - 0.5 * torch.sum(1+ log_var - mean.pow(2) - log_var.exp())

    return reproduction_loss + KLD
```

Then we define the training function.

```
def train(model, train_loader, batch_size, lr, epochs, device):

    model.train()
    optimizer = Adam(model.parameters(), lr=lr)
    print("Start training VAE...")
    model.train()

    for epoch in range(epochs):
        overall_loss = 0
```

```
        for batch_idx, (x, y) in enumerate(train_loader):
            x = x.view(batch_size, x_dim)
            x = x.to(device)

            optimizer.zero_grad()

            x_hat, mean, log_var = model(x)
            loss = loss_function(x, x_hat, mean, log_var)

            overall_loss += loss.item()

            loss.backward()
            optimizer.step()

        print("\tEpoch", epoch + 1, "complete!", "\tAverage Loss: ", overall_loss / (batch_idx*batch_size))

    print("Finish!!")

    checkpoint_save_dir = "./checkpoints/"
    if not os.path.exists(checkpoint_save_dir):
        os.makedirs(checkpoint_save_dir)

    torch.save(model.state_dict(), os.path.join(checkpoint_save_dir,'VAE_CP%d.pth'%(epoch+1)))
    print('Checkpoint saved !')

    return model
```

And the evaluation function.

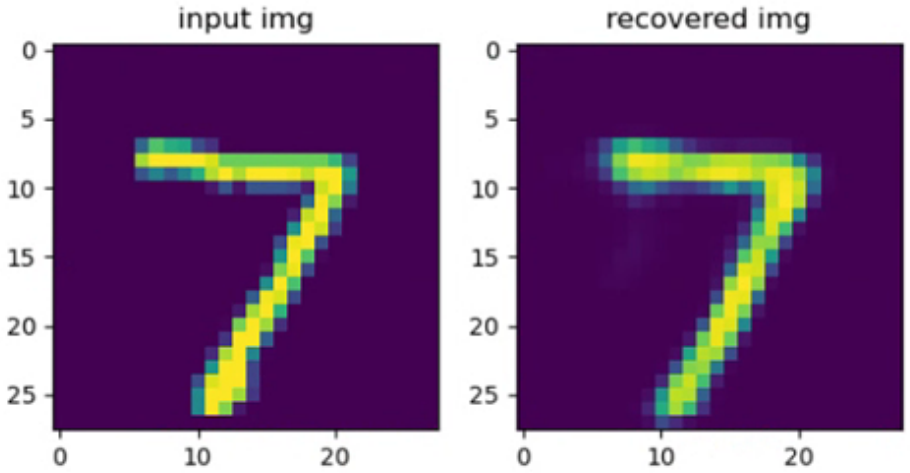
```
def eval(model, test_loader, device):

    model.eval()
    with torch.no_grad():
        for batch_idx, (x, y) in enumerate(tqdm(test_loader)):
            x = x.view(batch_size, x_dim)
            x = x.to(device)
            x_hat, _, _ = model(x)

            break

    return x, x_hat
```

Finally, define the main function to enable the model training and evaluation.

```
if __name__ == '__main__':

    # Model Hyperparameters
    dataset_path = './dataset/'
    is_train = True
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    batch_size = 100
    x_dim  = 784
    hidden_dim = 400
    latent_dim = 200
    lr = 1e-3
    epochs = 30

    # initialize model
    encoder = Encoder(input_dim=x_dim, hidden_dim=hidden_dim, latent_dim=latent_dim)
    decoder = Decoder(latent_dim=latent_dim, hidden_dim = hidden_dim, output_dim = x_dim)
    model = Model(encoder, decoder, device).to(device)

    # load data
    train_loader, test_loader = load_data(dataset_path, batch_size)

    if is_train:
        model = train(model,train_loader, batch_size, lr, epochs, device)
    else:
        model.load_state_dict(torch.load('./checkpoints/VAE_CP30.pth'))
```

```
    x, x_hat = eval(model, test_loader, device)

    # ------1. compare the input image with the recovered image by VAE--------
    input_img = x.view(batch_size, 28, 28)
    recovered_img = x_hat.view(batch_size, 28, 28)

    plt.subplot(1,2,1)
    plt.title("input img")
    if device == "cpu":
        input_img_show = input_img[0].numpy()
    else:
        input_img_show = input_img[0].cpu().numpy()
    plt.imshow(input_img_show)

    plt.subplot(1,2,2)
    plt.title("recovered img")
    if device == "cpu":
        recovered_img_show = recovered_img[0].numpy()
    else:
        recovered_img_show = recovered_img[0].cpu().numpy()
    plt.imshow(recovered_img_show)
    plt.savefig("comparison.png")


    # -----2. generate image from random noise------
    with torch.no_grad():
        noise = torch.randn(batch_size, latent_dim).to(device)
        generated_images = decoder(noise)
        save_image(generated_images.view(batch_size, 1, 28, 28), 'generated_sample.png')
```

Given the trained VAE model, we can visually compare the input image with the recovered image.

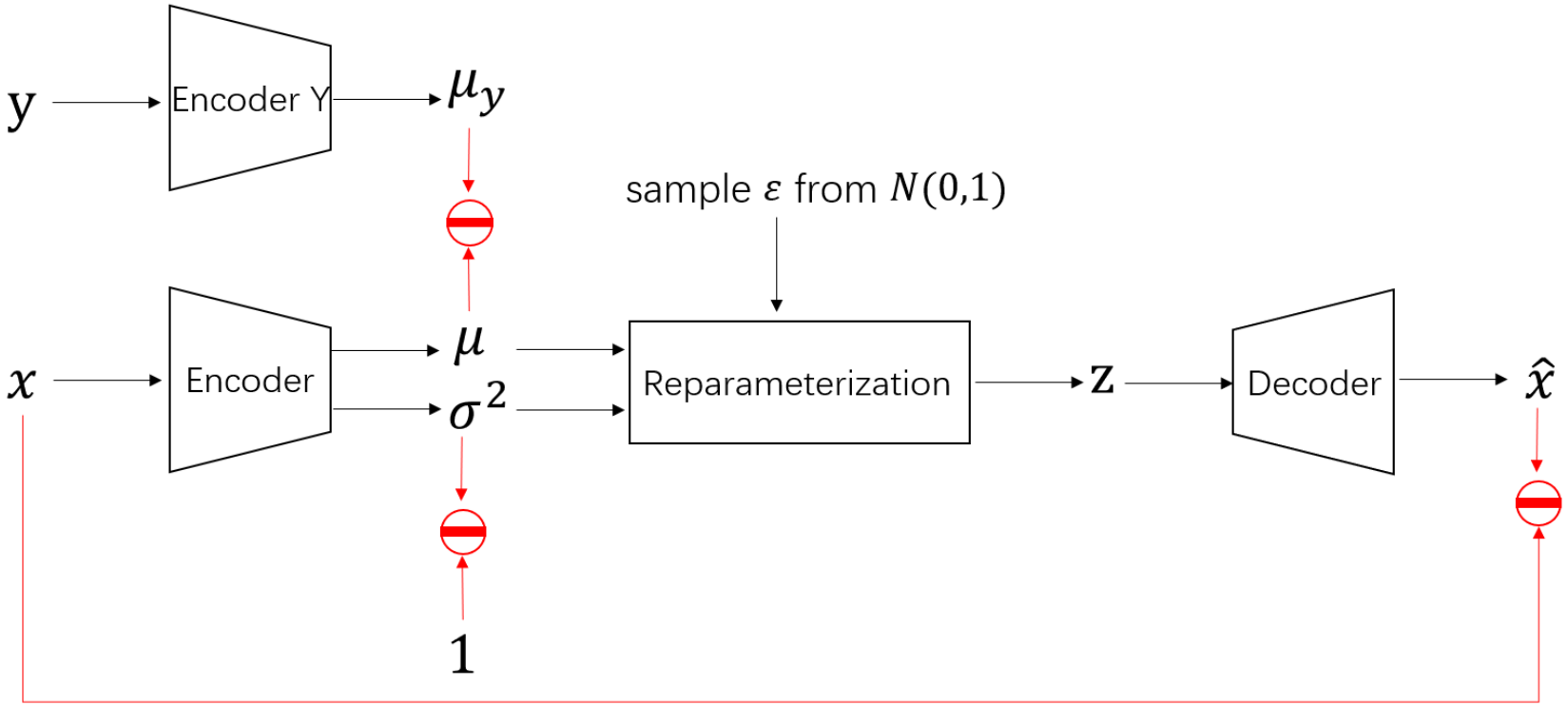


Besides, we can generate image from random gaussian noise.

However, the VAE model is trained in an unsupervised fashion, and it can not generate images for a specified class. The Conditional VAE (CVAE) is able to address the problem above.

3. Group exercise: Implement CVAE

CVAE involves label information to make the gaussian distribution of each category has a unique mean value, rather than equals to zero.



Hint: You need to define an encoder for label y, which takes the one-hot label as input and outputs a latent feature exhibiting the same dimension as the latent feature of x. Below is an example of the label encoder.

```
class Encoder_y(nn.Module):

    def __init__(self, input_dim, latent_dim):
        super(Encoder_y, self).__init__()

        self.FC_input = nn.Linear(input_dim, latent_dim)
        self.training = True

    def forward(self, x):

        class_mean = self.FC_input(x)

        return class_mean
```

During the training phase, instead of optimizing the mean value to be zero, here we requires the mean value equals to the class mean. Thus the KLD loss can be modified as:

```
    KLD      = - 0.5 * torch.sum(1+ log_var - (mean - class_mean).pow(2) - log_var.exp())
```

Below is an example of generating images for digit 6 by CVAE model.

References

This exercise is adapted from Pytorch-VAE-Tutorial.

For more information, see the Variational AutoEncoder paper (D.P. Kingma et. al., 2013)