```
# %load vae.py
from __future__ import print_function
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import numpy as np
import torch
import torch.utils.data
from torch import nn, optim
from torch.autograd import Variable
from torch.nn import functional as F
from torchvision import datasets, transforms
from torchvision.utils import save_image


def hello_vae():
    print("Hello from vae.py!")


class VAE(nn.Module):
    def __init__(self, input_size, latent_size=15):
        super(VAE, self).__init__()
        self.input_size = input_size # 784 # H*W
        self.latent_size = latent_size # Z
        self.hidden_dim = 400  # H_d
        self.encoder = None
        self.mu_layer = None
        self.logvar_layer = None
        self.decoder = None


###############################################################################################
        # TODO: Implement the fully-connected encoder architecture described in the notebook.
#
        # Specifically, self.encoder should be a network that inputs a batch of input images
of    #
        # shape (N, 1, H, W) into a batch of hidden features of shape (N, H_d). Set up
#
        # self.mu_layer and self.logvar_layer to be a pair of linear layers that map the hidden
#
        # features into estimates of the mean and log-variance of the posterior over the latent
#
        # vectors; the mean and log-variance estimates will both be tensors of shape (N, Z).
#

###############################################################################################
        # Replace "pass" statement with your code
        self.encoder = nn.Sequential(
                nn.Flatten(),
                nn.Linear(self.input_size, self.hidden_dim),
                nn.LeakyReLU(0.01),
                nn.Linear(self.hidden_dim, self.hidden_dim),
                nn.LeakyReLU(0.01),
                nn.Linear(self.hidden_dim, self.hidden_dim),
                nn.LeakyReLU(0.01),
                )
        self.mu_layer = nn.Linear(self.hidden_dim, self.latent_size)

        self.logvar_layer = nn.Linear(self.hidden_dim, self.latent_size)


###############################################################################################
        # TODO: Implement the fully-connected decoder architecture described in the notebook.
#
        # Specifically, self.decoder should be a network that inputs a batch of latent vectors
of  #
        # shape (N, Z) and outputs a tensor of estimated images of shape (N, 1, H, W).
#
```

```
##############################################################################################
        # Replace "pass" statement with your code
        self.decoder = nn.Sequential(
                nn.Linear(self.latent_size, self.hidden_dim),
                nn.LeakyReLU(0.01),
                nn.Linear(self.hidden_dim, self.hidden_dim),
                nn.LeakyReLU(0.01),
                nn.Linear(self.hidden_dim, self.hidden_dim),
                nn.LeakyReLU(0.01),
                nn.Linear(self.hidden_dim, self.input_size),
                nn.Sigmoid(),
                nn.Unflatten(dim =1 ,unflattened_size=(1, 28, 28) )
                )

##############################################################################################
        #                                      END OF YOUR CODE
#

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


    def forward(self, x):
        """
        Performs forward pass through FC-VAE model by passing image through
        encoder, reparametrize trick, and decoder models

        Inputs:
        - x: Batch of input images of shape (N, 1, H, W)

        Returns:
        - x_hat: Reconstruced input data of shape (N,1,H,W)
        - mu: Matrix representing estimated posterior mu (N, Z), with Z latent space dimension
        - logvar: Matrix representing estimataed variance in log-space (N, Z), with Z latent
space dimension
        """
        x_hat = None
        mu = None
        logvar = None

##############################################################################################
        # TODO: Implement the forward pass by following these steps
#
        # (1) Pass the input batch through the encoder model to get posterior mu and
logvariance   #
        # (2) Reparametrize to compute  the latent vector z
#
        # (3) Pass z through the decoder to resconstruct x
#

##############################################################################################
        # Replace "pass" statement with your code
        result = self.encoder(x) #torch.from_numpy(input)
        #result = torch.flatten(result, start_dim =1)

        mu = self.mu_layer(result)
        logvar = self.logvar_layer(result)
        z = reparametrize(mu, logvar)
        print(z.shape)  # Should be [N, latent_size] like [3, 17]
        out = self.decoder(z)
        print(out.shape)  # Should now be [N, 1, 28, 28]
        x_hat = self.decoder(z)

##############################################################################################
        #                                      END OF YOUR CODE
#
```

```
#########################################################################################
        return x_hat, mu, logvar


class CVAE(nn.Module):
    def __init__(self, input_size, num_classes=10, latent_size=15):
        super(CVAE, self).__init__()
        self.input_size = input_size # H*W
        self.latent_size = latent_size # Z
        self.num_classes = num_classes # K
        self.hidden_dim = 400 # H_d
        self.encoder = None
        self.mu_layer = None
        self.logvar_layer = None
        self.decoder = None


#########################################################################################
        # TODO: Define a FC encoder as described in the notebook that transforms the image--
after  #
        # flattening and now adding our one-hot class vector (N, H*W + K)--into a
hidden_dimension #               #
        # (N, H_d) feature space, and a final two layers that project that feature space
#
        # to posterior mu and posterior log-variance estimates of the latent space (N, Z)
#

#########################################################################################
        # Replace "pass" statement with your code
        self.encoder = nn.Sequential(
                nn.Linear(self.input_size + self.num_classes, self.hidden_dim),
                nn.LeakyReLU(0.01),
                nn.Linear(self.hidden_dim, self.hidden_dim),
                nn.LeakyReLU(0.01),
                nn.Linear(self.hidden_dim, self.hidden_dim),
                nn.LeakyReLU(0.01),
                )
        self.mu_layer = nn.Linear(self.hidden_dim, self.latent_size)

        self.logvar_layer = nn.Linear(self.hidden_dim, self.latent_size)


#########################################################################################
        # TODO: Define a fully-connected decoder as described in the notebook that transforms
the  #
        # latent space (N, Z + K) to the estimated images of shape (N, 1, H, W).
#

#########################################################################################
        # Replace "pass" statement with your code
        self.decoder = nn.Sequential(
                nn.Linear(self.latent_size + self.num_classes, self.hidden_dim),
                nn.LeakyReLU(0.01),
                nn.Linear(self.hidden_dim, self.hidden_dim),
                nn.LeakyReLU(0.01),
                nn.Linear(self.hidden_dim, self.hidden_dim),
                nn.LeakyReLU(0.01),
                nn.Linear(self.hidden_dim, self.input_size),
                nn.Sigmoid(),
                nn.Unflatten(dim =1 ,unflattened_size=(1, 28, 28) )
                )

#########################################################################################
        #                                       END OF YOUR CODE
#

#########################################################################################
```

```python
    def forward(self, x, labels):
        """
        Performs forward pass through FC-CVAE model by passing image through
        encoder, reparametrize trick, and decoder models

        Inputs:
        - x: Input data for this timestep of shape (N, 1, H, W)
        - labels: One hot vector representing the input class (0-9) (N, K)

        Returns:
        - x_hat: Reconstruced input data of shape (N, 1, H, W)
        - mu: Matrix representing estimated posterior mu (N, Z), with Z latent space dimension
        - logvar: Matrix representing estimated variance in log-space (N, Z),  with Z latent
space dimension
        """
        x_hat = None
        mu = None
        logvar = None
##############################################################################################
        # TODO: Implement the forward pass by following these steps
#
        # (1) Pass the concatenation of input batch and one hot vectors through the encoder
model  #
        # to get posterior mu and logvariance
#
        # (2) Reparametrize to compute the latent vector z
#
        # (3) Pass concatenation of z and one hot vectors through the decoder to resconstruct x
#
##############################################################################################
        # Replace "pass" statement with your code
        # step 1:
        N = x.size(0)

        x_flat   = x.view(N, -1)                                  # (N, H*W)
        enc_input = torch.cat([x_flat, labels], dim=1)            # (N, H*W + K)
        h         = self.encoder(enc_input)                       # (N, hidden_dim)
        mu        = self.mu_layer(h)                              # (N, Z)
        logvar    = self.logvar_layer(h)                          # (N, Z)

        # step 2:
        z = reparametrize(mu, logvar)                             # (N, Z)

        # step 3:
        dec_input = torch.cat([z, labels], dim=1)                 # (N, Z + K)
        x_hat     = self.decoder(dec_input)                       # (N,1,H,W)

##############################################################################################
        #                                       END OF YOUR CODE
#
##############################################################################################
        return x_hat, mu, logvar


def reparametrize(mu, logvar):
    """
    Differentiably sample random Gaussian data with specified mean and variance using the
    reparameterization trick.

    Suppose we want to sample a random number z from a Gaussian distribution with mean mu and
    standard deviation sigma, such that we can backpropagate from the z back to mu and sigma.
```

```
    We can achieve this by first sampling a random value epsilon from a standard Gaussian
    distribution with zero mean and unit variance, then setting z = sigma * epsilon + mu.

    For more stable training when integrating this function into a neural network, it helps to
    pass this function the log of the variance of the distribution from which to sample, rather
    than specifying the standard deviation directly.

    Inputs:
    - mu: Tensor of shape (N, Z) giving means
    - logvar: Tensor of shape (N, Z) giving log-variances

    Returns:
    - z: Estimated latent vectors, where z[i, j] is a random value sampled from a Gaussian with
         mean mu[i, j] and log-variance logvar[i, j].
    """
    z = None

##################################################################################################
    # TODO: Reparametrize by initializing epsilon as a normal distribution and scaling by
#
    # posterior mu and sigma to estimate z
#
##################################################################################################
    # Replace "pass" statement with your code
    sigma = torch.exp(0.5 * logvar)
    epsilon =  torch.randn_like(sigma)
    z = sigma*epsilon + mu

##################################################################################################
    #                               END OF YOUR CODE
#
##################################################################################################
    return z


def loss_function(x_hat, x, mu, logvar):
    """
    Computes the negative variational lower bound loss term of the VAE (refer to formulation in
notebook).

    Inputs:
    - x_hat: Reconstruced input data of shape (N, 1, H, W)
    - x: Input data for this timestep of shape (N, 1, H, W)
    - mu: Matrix representing estimated posterior mu (N, Z), with Z latent space dimension
    - logvar: Matrix representing estimated variance in log-space (N, Z), with Z latent space
dimension

    Returns:
    - loss: Tensor containing the scalar loss for the negative variational lowerbound
    """
    loss = None

##################################################################################################
    # TODO: Compute negative variational lowerbound loss as described in the notebook
#
##################################################################################################
    # Replace "pass" statement with your code
    N = x.size(0)
```

```
    # flatten images to (N, D) for BCE
    x_hat_flat = x_hat.view(N, -1)
    x_flat     = x.view(N, -1)

    # Reconstruction term:
    recon_loss = F.binary_cross_entropy(
        x_hat_flat, x_flat,
        reduction='sum'
    ) / N

    # KL divergence term:
    kl_per_sample = -0.5 * torch.sum(
        1 + logvar - mu.pow(2) - logvar.exp(),
        dim=1
    )
    kl_loss = kl_per_sample.mean()

    loss = recon_loss + kl_loss

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

    #                             END OF YOUR CODE
#

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

    return loss
```