projects/project_1/vae/vae.py

```
from __future__ import print_function
from torch import nn
import torch

device = torch.device("cpu")


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  # 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.ReLU(),
            nn.Linear(self.hidden_dim, self.hidden_dim),
            nn.ReLU(),
            nn.Linear(self.hidden_dim, self.hidden_dim),
            nn.ReLU(),
        )

        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.ReLU(),
            nn.Linear(self.hidden_dim, self.hidden_dim),
            nn.ReLU(),
            nn.Linear(self.hidden_dim, self.hidden_dim),
            nn.ReLU(),
            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
        output = self.encoder(x)
        mu = self.mu_layer(output)
        logvar = self.logvar_layer(output)

        reparametrized_output = reparametrize(mu=mu, logvar=logvar)
        x_hat = self.decoder(reparametrized_output)

###############################################################################################
        #                                       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 = None  # 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.hidden_dim = 400

        self.encoder = nn.Sequential(
            nn.Linear(self.input_size + self.num_classes, self.hidden_dim),
            nn.ReLU(),
            nn.Linear(self.hidden_dim, self.hidden_dim),
            nn.ReLU(),
            nn.Linear(self.hidden_dim, self.hidden_dim),
            nn.ReLU(),
        )
        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.ReLU(),
            nn.Linear(self.hidden_dim, self.hidden_dim),
            nn.ReLU(),
            nn.Linear(self.hidden_dim, self.hidden_dim),
            nn.ReLU(),
            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, 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
        flattened = torch.flatten(x, start_dim=1)
        inputs = torch.cat((flattened, labels), dim=1)
        output = self.encoder(inputs)
        mu = self.mu_layer(output)
        logvar = self.logvar_layer(output)

        z = reparametrize(mu=mu, logvar=logvar)
        z = torch.cat((z, labels), dim=1)
        x_hat = self.decoder(z)

############################################################################################
        #                                      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
    epsilon = torch.randn(logvar.shape).to(device)
```

```
    sigma = torch.exp(0.5 * logvar).to(device)
    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
    reconstruction_loss = (
        nn.functional.binary_cross_entropy(x_hat, x, reduction="sum") / x_hat.shape[0]
    )

    kl_loss = -0.5 * torch.mean(
        torch.sum(1 + logvar - torch.square(mu) - torch.exp(logvar), dim=1)
    )

    loss = reconstruction_loss + kl_loss


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

################################################################################################
    return loss
```