vae_ipynb

April 21, 2025

We would like to acknowledge University of Michigan's EECS 498-007/598-005 on which we based the development of this project.

1 Variational Autoencoder

In this notebook, you will implement a variational autoencoder and a conditional variational autoencoder with slightly different architectures and apply them to the popular MNIST handwritten dataset. Recall from C147/C247, an autoencoder seeks to learn a latent representation of our training images by using unlabeled data and learning to reconstruct its inputs. The *variational autoencoder* extends this model by adding a probabilistic spin to the encoder and decoder, allowing us to sample from the learned distribution of the latent space to generate new images at inference time.

1.1 Setup Code

Before getting started, we need to run some boilerplate code to set up our environment. You'll need to rerun this setup code each time you start the notebook.

First, run this cell that loads the autoreload extension. This allows us to edit .py source files and re-import them into the notebook for a seamless editing and debugging experience.

```
[12]: %load_ext autoreload
%autoreload 2
USE_COLAB = False
```

1.1.1 Google Colab Setup

Next we need to run a few commands to set up our environment on Google Colab. If you are running this notebook on a local machine you can skip this section.

Run the following cell to mount your Google Drive. Follow the link and sign in to your Google account (the same account you used to store this notebook!).

```
[13]: if USE_COLAB:
    from google.colab import drive
    drive.mount('/content/drive')
```

Now recall the path in your Google Drive where you uploaded this notebook and fill it in below. If everything is working correctly then running the following cell should print the filenames from the assignment:

```
['vae.ipynb', 'nndl2', 'vae.py']
```

```
if USE_COLAB:
    # TODO: Fill in the Google Drive path where you uploaded the assignment
    # Example: '239AS.2/project1/vae'
    GOOGLE_DRIVE_PATH_AFTER_MYDRIVE = '239AS.2/project1/vae'
    GOOGLE_DRIVE_PATH = os.path.join('drive', 'My Drive', \_
    GOOGLE_DRIVE_PATH_AFTER_MYDRIVE)
    print(os.listdir(GOOGLE_DRIVE_PATH))
```

Once you have successfully mounted your Google Drive and located the path to this assignment, run the following cell to allow us to import from the .py files of this assignment. If it works correctly, it should print the message:

```
Hello from vae.py!
Hello from helper.py!
```

```
import sys
if USE_COLAB:
    sys.path.append(GOOGLE_DRIVE_PATH)

from vae import hello_vae
hello_vae()

from nnd12.helper import hello_helper
hello_helper()
```

```
Hello from vae.py!
Hello from helper.py!
```

Load several useful packages that are used in this notebook:

```
[16]: from nndl2.grad import rel_error
from nndl2.utils import reset_seed
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import init
import torchvision
import torchvision.transforms as T
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
```

```
import torchvision.datasets as dset
import matplotlib.pyplot as plt
%matplotlib inline

# for plotting
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['font.size'] = 16
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

We will use GPUs to accelerate our computation in this notebook. Run the following to make sure GPUs are enabled:

```
[17]: if torch.cuda.is_available():
    print('Good to go!')
    else:
        print('Please set GPU via the downward triangle in the top right corner.')
```

Good to go!

1.2 Load MNIST Dataset

VAEs are notoriously finicky with hyperparameters, and also require many training epochs. In order to make this assignment approachable, we will be working on the MNIST dataset, which is 60,000 training and 10,000 test images. Each picture contains a centered image of white digit on black background (0 through 9). This was one of the first datasets used to train convolutional neural networks and it is fairly easy – a standard CNN model can easily exceed 99% accuracy.

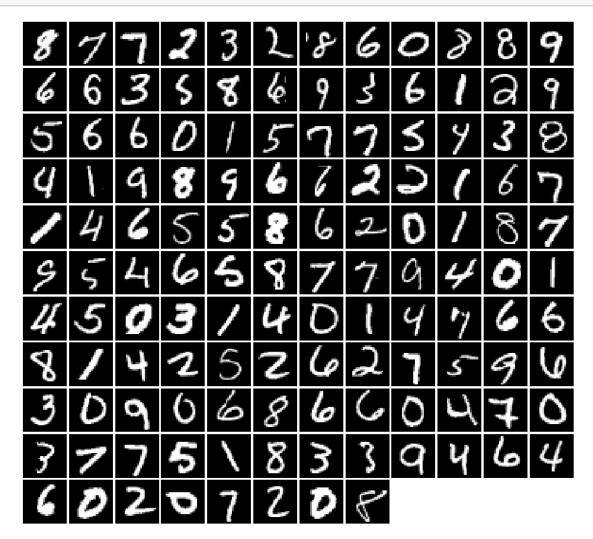
To simplify our code here, we will use the PyTorch MNIST wrapper, which downloads and loads the MNIST dataset. See the documentation for more information about the interface. The default parameters will take 5,000 of the training examples and place them into a validation dataset. The data will be saved into a folder called MNIST.

1.3 Visualize dataset

It is always a good idea to look at examples from the dataset before working with it. Let's visualize the digits in the MNIST dataset. We have defined the function <code>show_images</code> in <code>helper.py</code> that we call to visualize the images.

```
[19]: from nndl2.helper import show_images

imgs = next(iter(loader_train))[0].view(batch_size, 784)
    show_images(imgs)
```



2 Fully Connected VAE

Our first VAE implementation will consist solely of fully connected layers. We'll take the $1 \times 28 \times 28$ shape of our input and flatten the features to create an input dimension size of 784. In this section you'll define the Encoder and Decoder models in the VAE class of vae.py and implement the reparametrization trick, forward pass, and loss function to train your first VAE.

2.1 FC-VAE Encoder (4 points)

Now lets start building our fully-connected VAE network. We'll start with the encoder, which will take our images as input (after flattening C,H,W to D shape) and pass them through a three

Linear+ReLU layers. We'll use this hidden dimension representation to predict both the posterior mu and posterior log-variance using two separate linear layers (both shape (N,Z)).

Note that we are calling this the 'logvar' layer because we'll use the log-variance (instead of variance or standard deviation) to stabilize training. This will specifically matter more when you compute reparametrization and the loss function later.

Define hidden_dim=400, encoder, mu_layer, and logvar_layer in the initialization of the VAE class in vae.py. Use nn.Sequential to define the encoder, and separate Linear layers for the mu and logvar layers. Architecture for the encoder is described below:

- Flatten (Hint: nn.Flatten)
- Fully connected layer with input size input_size and output size hidden_dim
- ReI.II
- Fully connected layer with input size hidden_dim and output size hidden_dim
- ReLU
- Fully connected layer with input size hidden_dim and output size hidden_dim
- ReLU

```
[20]: from vae import VAE
      def count_params(model):
          return sum(p.numel() for p in model.parameters())
      def test encoder(model, input size, hidden_dim, n_encoder_lin_layers):
          model: Model defined as above
          input size: dimensionality of input
          hidden_dim: dimensionality of hidden state
          n_layers: number of Linear layers
          expected_n_params = (input_size+1)*hidden_dim + \
              (n_encoder_lin_layers-1)*(hidden_dim+1)*hidden_dim
          actual_n_params = count_params(model.encoder)
          if actual_n_params == expected_n_params:
              print('Correct number of parameters in model.encoder.')
              return True
          else:
              print('Incorrect number of parameters in model.encoder.' \
                ' model.encoder does not include mu_layer and the logvar_layer.' \
                ' Check your achitecture.')
              return False
          return
      def test_mu_logvar(model, hidden_dim, latent_size):
          model: Model defined as above
          input_size: dimensionality of input
          hidden_dim: dimensionality of hidden state
          n layers: number of Linear layers
          if count_params(model.mu_layer) == (hidden_dim+1)*latent_size:
```

```
print('Correct number of parameters in model.mu_layer.')
else:
    print('Incorrect number of parameters in model.mu_layer.')
if count_params(model.logvar_layer) == (hidden_dim+1)*latent_size:
    print('Correct number of parameters in model.logvar_layer.')
else:
    print('Incorrect number of parameters in model.logvar_layer.')
return
test_encoder(VAE(345, 17), 345, 400, 3)
test_mu_logvar(VAE(345, 17), 400, 17)
```

```
Correct number of parameters in model.encoder.

Correct number of parameters in model.mu_layer.

Correct number of parameters in model.logvar_layer.
```

2.2 FC-VAE Decoder (1 point)

We'll now define the decoder, which will take the latent space representation and generate a reconstructed image. The architecture is as follows:

- Fully connected layer with input size latent_size and output size hidden_dim
- ReLU
- Fully connected layer with input size hidden_dim and output size hidden_dim
- ReLU
- Fully connected layer with input size hidden_dim and output size hidden_dim
- ReLU
- Fully connected layer with input size hidden_dim and output size input_size
- Sigmoid
- Unflatten (nn.Unflatten)

Define a decoder in the initialization of the VAE class in vae.py. Like the encoding step, use nn.Sequential

```
if actual_n_params == expected_n_params:
    print('Correct number of parameters in model.decoder.')
else:
    print('Incorrect number of parameters in model.decoder.')
return
test_decoder(VAE(345, 17), 345, 400, 17, 4)
```

Correct number of parameters in model.decoder.

2.3 Reparametrization (2 points)

Now we'll apply a reparametrization trick in order to estimate the posterior z during our forward pass, given the μ and σ^2 estimated by the encoder. A simple way to do this could be to simply generate a normal distribution centered at our μ and having a std corresponding to our σ^2 . However, we would have to backpropogate through this random sampling that is not differentiable. Instead, we sample initial random data ϵ from a fixed distribution, and compute z as a function of $(\epsilon, \sigma^2, \mu)$. Specifically:

```
z = \mu + \sigma \epsilon
```

We can easily find the partial derivatives w.r.t μ and σ^2 and backpropagate through z. If $\epsilon = \mathcal{N}(0,1)$, then its easy to verify that the result of our forward pass calculation will be a distribution centered at μ with variance σ^2 .

Implement reparametrization in vae.py and verify your mean and std error are at or less than 1e-4.

```
[22]: reset_seed(0)
    from vae import reparametrize
    latent_size = 15
    size = (1, latent_size)
    mu = torch.zeros(size)
    logvar = torch.ones(size)

z = reparametrize(mu, logvar)

expected_mean = torch.FloatTensor([-0.4363])
    expected_std = torch.FloatTensor([1.6860])
    z_mean = torch.mean(z, dim=-1)
    z_std = torch.std(z, dim=-1)
    assert z.size() == size

print('Mean Error', rel_error(z_mean, expected_mean))
    print('Std Error', rel_error(z_std, expected_std))
```

Mean Error 5.639056398351415e-05 Std Error 7.1412955526273885e-06

2.4 FC-VAE Forward (1 point)

Complete the VAE class by writing the forward pass. The forward pass should pass the input image through the encoder to calculate the estimation of mu and logvar, reparametrize to estimate the latent space z, and finally pass z into the decoder to generate an image.

```
[23]: from vae import VAE
      def test_VAE_shapes():
          all shapes correct = True
          with torch.no_grad():
              batch_size = 3
              latent_size = 17
              x_hat, mu, logvar = VAE(28*28, latent_size)(torch.ones(batch_size, 1,_
       428, 28)
              if x_hat.shape != (batch_size, 1, 28, 28):
                  print(f'x_hat has incorrect shape. Expected (batch_size, 1, 28, 28)
       ←= ({batch_size}, 1, 28, 28).'
                      f' Got {tuple(x_hat.shape)}.')
                  all_shapes_correct = False
              if mu.shape != (batch_size, latent_size):
                  print(f'mu has incorrect shape. Expected (batch_size, latent_size) ∪
       ←= ({batch_size}, {latent_size}).'
                      f' Got {tuple(mu.shape)}.')
                  all_shapes_correct = False
              if logvar.shape != (batch_size, latent_size):
                  print(f'logvar has incorrect shape. Expected (batch_size, __
       Glatent_size) = ({batch_size}, {latent_size}).'
                      f' Got {tuple(logvar.shape)}.')
                  all shapes correct = False
              if all_shapes_correct:
                  print('Shapes of x_hat, mu, and logvar are correct.')
              if batch_size > 1 and x_hat.std(0).mean() == 0:
                  print('x_hat has no randomness.')
          return
      test_VAE_shapes()
```

```
torch.Size([3, 17])
torch.Size([3, 1, 28, 28])
Shapes of x_hat, mu, and logvar are correct.
```

2.5 Loss Function (1 point)

Before we're able to train our final model, we'll need to define our loss function. As seen below, the loss function for VAEs contains two terms: A reconstruction loss term (left) and KL divergence term (right).

```
-E_{Z~q_{\phi}(z|x)}[logp_{\theta}(x|z)] + D_{KL}(q_{\phi}(z|x),p(z)))
```

Note that this is the negative of the variational lowerbound shown in lecture—this ensures that when we are minimizing this loss term, we're maximizing the variational lowerbound. The reconstruction

loss term can be computed by simply using the binary cross entropy loss between the original input pixels and the output pixels of our decoder (Hint: nn.functional.binary_cross_entropy). The KL divergence term works to force the latent space distribution to be close to a prior distribution (we're using a standard normal gaussian as our prior).

To help you out, we've derived an unvectorized form of the KL divergence term for you. Suppose that $q_{\phi}(z|x)$ is a Z-dimensional diagonal Gaussian with mean $\mu_{z|x}$ of shape (Z,) and standard deviation $\sigma_{z|x}$ of shape (Z,), and that p(z) is a Z-dimensional Gaussian with zero mean and unit variance. Then we can write the KL divergence term as:

$$D_{KL}(q_{\phi}(z|x),p(z))) = -\frac{1}{2} \sum_{j=1}^{J} (1 + log(\sigma_{z|x}^2)_j - (\mu_{z|x})_j^2 - (\sigma_{z|x})_j^2)$$

It's up to you to implement a vectorized version of this loss that also operates on minibatches. You should average the loss across samples in the minibatch.

Implement loss_function in vae.py and verify your implementation below. Your relative error should be less than or equal to 1e-5

```
[24]: from vae import loss_function
    size = (1,15)

image_hat = torch.sigmoid(torch.FloatTensor([[2,5], [6,7]]).unsqueeze(0).
    unsqueeze(0))

image = torch.sigmoid(torch.FloatTensor([[1,10], [9,3]]).unsqueeze(0).
    unsqueeze(0))

expected_out = torch.tensor(8.5079)
    mu, logvar = torch.ones(size), torch.zeros(size)
    out = loss_function(image_hat, image, mu, logvar)
    print('Loss error', rel_error(expected_out,out))
```

Loss error 2.1297676389877955e-06

2.6 Train a model

Now that we have our VAE defined and loss function ready, lets train our model! Our training script is provided in nndl2/helper.py, and we have pre-defined an Adam optimizer, learning rate, and # of epochs for you to use.

Training for 10 epochs should take ~2 minutes and your loss should be less than 120.

```
[25]: num_epochs = 10
    latent_size = 15
    from vae import VAE
    from nndl2.helper import train_vae
    input_size = 28*28
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    if device == 'cpu':
        print(f'Warning: using device {device} may take longer.')
    vae_model = VAE(input_size, latent_size=latent_size)
    vae_model.to(device)
```

```
for epoch in range(0, num_epochs):
    train_vae(epoch, vae_model, loader_train)
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
Train Epoch: 0 Loss: 150.360550
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
Train Epoch: 1 Loss: 128.647781
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
Train Epoch: 2 Loss: 123.949516
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
Train Epoch: 3 Loss: 115.536644
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
Train Epoch: 4 Loss: 121.911789
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
Train Epoch: 5 Loss: 114.624428
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
Train Epoch: 6 Loss: 110.758972
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
Train Epoch: 7 Loss: 109.777832
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
Train Epoch: 8 Loss: 116.603508
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
```

```
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
torch.Size([128, 15])
torch.Size([128, 1, 28, 28])
Train Epoch: 9 Loss: 114.974007
```

2.7 Visualize results

After training our VAE network, we're able to take advantage of its power to generate new training examples. This process simply involves the decoder: we intialize some random distribution for our latent spaces z, and generate new examples by passing these latent space into the decoder.

Run the cell below to generate new images! You should be able to visually recognize many of the digits, although some may be a bit blurry or badly formed. Our next model will see improvement in these results.

```
[26]: device = next(vae_model.parameters()).device
    z = torch.randn(10, latent_size).to(device=device)
    import matplotlib.gridspec as gridspec
    vae_model.eval()
    samples = vae_model.decoder(z).data.cpu().numpy()

fig = plt.figure(figsize=(10, 1))
    gspec = gridspec.GridSpec(1, 10)
    gspec.update(wspace=0.05, hspace=0.05)
    for i, sample in enumerate(samples):
        ax = plt.subplot(gspec[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(sample.reshape(28,28), cmap='Greys_r')
```

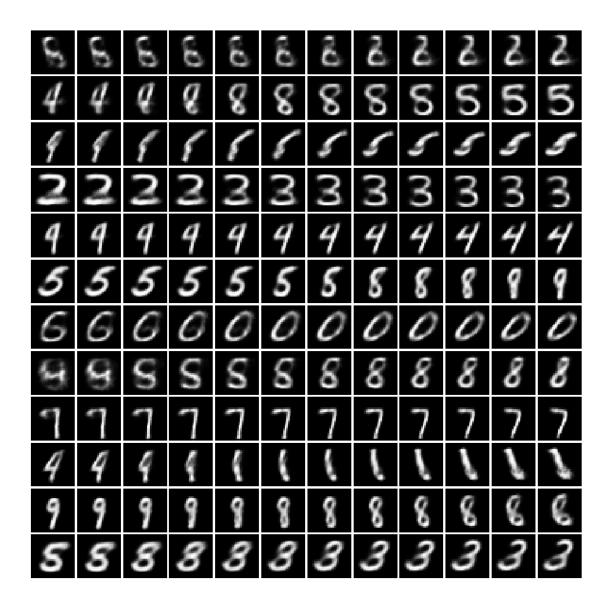


2.8 Latent Space Interpolation

As a final visual test of our trained VAE model, we can perform interpolation in latent space. We generate random latent vectors z_0 and z_1 , and linearly interplate between them; we run each interpolated vector through the trained generator to produce an image.

Each row of the figure below interpolates between two random vectors. For the most part the model should exhibit smooth transitions along each row, demonstrating that the model has learned something nontrivial about the underlying spatial structure of the digits it is modeling.

```
[27]: S = 12
latent_size = 15
device = next(vae_model.parameters()).device
z0 = torch.randn(S, latent_size, device=device)
z1 = torch.randn(S, latent_size, device=device)
w = torch.linspace(0, 1, S, device=device).view(S, 1, 1)
z = (w * z0 + (1 - w) * z1).transpose(0, 1).reshape(S * S, latent_size)
x = vae_model.decoder(z)
show_images(x.data.cpu())
```



3 Conditional FC-VAE

The second model you'll develop will be very similar to the FC-VAE, but with a slight conditional twist to it. We'll use what we know about the labels of each MNIST image, and condition our latent space and image generation on the specific class. Instead of $q_{\phi}(z|x)$ and $p_{\phi}(x|z)$ we have $q_{\phi}(z|x,c)$ and $p_{\phi}(x|z,c)$

This will allow us to do some powerful conditional generation at inference time. We can specifically choose to generate more 1s, 2s, 9s, etc. instead of simply generating new digits randomly.

3.1 Define Network with class input (3 points)

Our CVAE architecture will be the same as our FC-VAE architecture, except we'll now add a one-hot label vector to both the x input (in our case, the flattened image dimensions) and the z

latent space.

If our one-hot vector is called c, then c[label] = 1 and c = 0 elsewhere.

For the CVAE class in vae.py use the same FC-VAE architecture implemented in the last network with the following modifications:

- 1. Modify the first linear layer of your encoder to take in not only the flattened input image, but also the one-hot label vector c. The CVAE encoder should not have a Flatten layer.
- 2. Modify the first layer of your decoder to project the latent space + one-hot vector to the hidden_dim
- 3. Lastly, implement the forward pass to combine the flattened input image with the one-hot vectors (torch.cat) before passing them to the encoder and combine the latent space with the one-hot vectors (torch.cat) before passing them to the decoder. You should flatten the image before concatenation (e.g. with torch.flatten or torch.reshape).

```
[30]: from vae import CVAE
     def test_CVAE_shapes():
         all_shapes_correct = True
         with torch.no_grad():
             batch_size = 3
             num_classes = 10
             latent_size = 17
             cls = nn.functional.one_hot(torch.tensor([3]*batch_size, dtype=torch.
       →long), num_classes=num_classes)
             x_hat, mu, logvar = CVAE(28*28, __
       torch.ones(batch_size, 1, 28, 28), cls)
             if x_hat.shape != (batch_size, 1, 28, 28):
                 print(f'x_hat has incorrect shape. Expected (batch_size, 1, 28, 28)
       ←= ({batch_size}, 1, 28, 28).'
                     f' Got {tuple(x_hat.shape)}.')
                 all_shapes_correct = False
             if mu.shape != (batch_size, latent_size):
                 print(f'mu has incorrect shape. Expected (batch_size, latent_size) ∪
       ←= ({batch_size}, {latent_size}).'
                     f' Got {tuple(mu.shape)}.')
                 all_shapes_correct = False
             if logvar.shape != (batch_size, latent_size):
                 print(f'logvar has incorrect shape. Expected (batch_size, __
       Glatent_size) = ({batch_size}, {latent_size}).'
                     f' Got {tuple(logvar.shape)}.')
                 all_shapes_correct = False
             if all_shapes_correct:
                 print('Shapes of x_hat, mu, and logvar are correct.')
             if batch_size > 1 and x_hat.std(0).mean() == 0:
                 print('x hat has no randomness.')
         return
```

```
test_CVAE_shapes()
```

Shapes of x_hat, mu, and logvar are correct.

3.2 Train model

Using the same training script, let's now train our CVAE!

Training for 10 epochs should take ~2 minutes and your loss should be less than 120.

```
[31]: from vae import CVAE
  num_epochs = 10
  latent_size = 15
  from nndl2.helper import train_vae
  input_size = 28*28
  device = 'cuda' if torch.cuda.is_available() else 'cpu'
  if device == 'cpu':
       print(f'Warning: using device {device} may take longer.')

  cvae = CVAE(input_size, latent_size=latent_size)
  cvae.to(device)
  for epoch in range(0, num_epochs):
       train_vae(epoch, cvae, loader_train, cond=True)
```

```
Train Epoch: 0 Loss: 135.927734
Train Epoch: 1 Loss: 122.679947
Train Epoch: 2 Loss: 112.725868
Train Epoch: 3 Loss: 109.824448
Train Epoch: 4 Loss: 107.520432
Train Epoch: 5 Loss: 108.468788
Train Epoch: 6 Loss: 105.107819
Train Epoch: 7 Loss: 108.116425
Train Epoch: 8 Loss: 104.715103
Train Epoch: 9 Loss: 101.960243
```

3.3 Visualize Results

We've trained our CVAE, now lets conditionally generate some new data! This time, we can specify the class we want to generate by adding our one hot matrix of class labels. We use torch.eye to create an identity matrix, gives effectively gives us one label for each digit. When you run the cell below, you should get one example per digit. Each digit should be reasonably distinguishable (it is ok to run this cell a few times to save your best results).

```
[33]: device = next(cvae.parameters()).device
z = torch.randn(10, latent_size)
c = torch.eye(10, 10) # [one hot labels for 0-9]
import matplotlib.gridspec as gridspec
z = torch.cat((z,c), dim=-1).to(device=device)
cvae.eval()
```

```
samples = cvae.decoder(z).data.cpu().numpy()

fig = plt.figure(figsize=(10, 1))
gspec = gridspec.GridSpec(1, 10)
gspec.update(wspace=0.05, hspace=0.05)

for i, sample in enumerate(samples):
    ax = plt.subplot(gspec[i])
    plt.axis('off')
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.set_aspect('equal')
    plt.imshow(sample.reshape(28, 28), cmap='Greys_r')
```

0 1 2 3 4 5 6 7 8 9