CS 5814: Homework 3, Part 3

Conditional Variational Autoencoder

In this final part of the homework, you'll implement a conditional variational autoencoder. As we discussed, an autoencoder is a model that learns a representation from which the original image can be regenerated. One of the challenges of using an autoencoder directly for generating samples is that the representation space tends not to be as smooth. If you try to sample a random vector in the space and feed it into the decoder, the results may be undesirable. The problem is the standard autoencoder may have sharp discontinuities within the space, where the region between the two classes either produces noise outputs or suddely changes (i.e. one vector looks like class X, but a very close vector suddenly looks like class Y - with a large jump). This is because a standard autoencoder learns a deterministic mapping from the input to the latent representation and back. Variational autoencoders addresses this issue by introducing randomness in the latent representation.

In a VAE, the encoder predicts the parameters of a probability distribution in the latent space, rather than a single representation (aka an embedding). To generate a sample, a latent vector is drawn randomly from this predicted distribution, and then passed through the decoder to generate the output. The key idea is that the randomness baked into the latent space allows the VAE to better generate diverse samples that were not present in the training data because the space is smoother.

In addition, the VAE objective function includes a regularization term which encourages the learned latent distribution to be close to a predefined prior distribution. This regularization helps to smooth the latent space, making it more continuous and well-behaved, which in turn leads to better generated samples.

In this notebook, you'll be implementing a conditional variational autoencoder (i.e. one that is also conditioned on the class label). You'll be training your autoencoder on the famous MNIST dataset of handwritten digits.

Starter code

In [1]:
```
#References :
# https://medium.com/@rekalantar/variational-auto-encoder-vae-pytorch-tutorial-dce2
# https://towardsdatascience.com/understanding-conditional-variational-autoencoders
# https://www.youtube.com/watch?v=9zKuYvjFFS8&t=513s
```

In [2]:
```
import math
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
from torch.nn import init
import time, os, sys
import cs5814_vae
import numpy as np
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 datasets
import matplotlib.pyplot as plt
import numpy
%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'
```

MNIST Dataset

Generative models can be difficult to train, as we'll discuss at length when we discuss GANs. To simplify issues, we'll use the MNIST dataset that is a famous dataset for handwritten digit recognition.

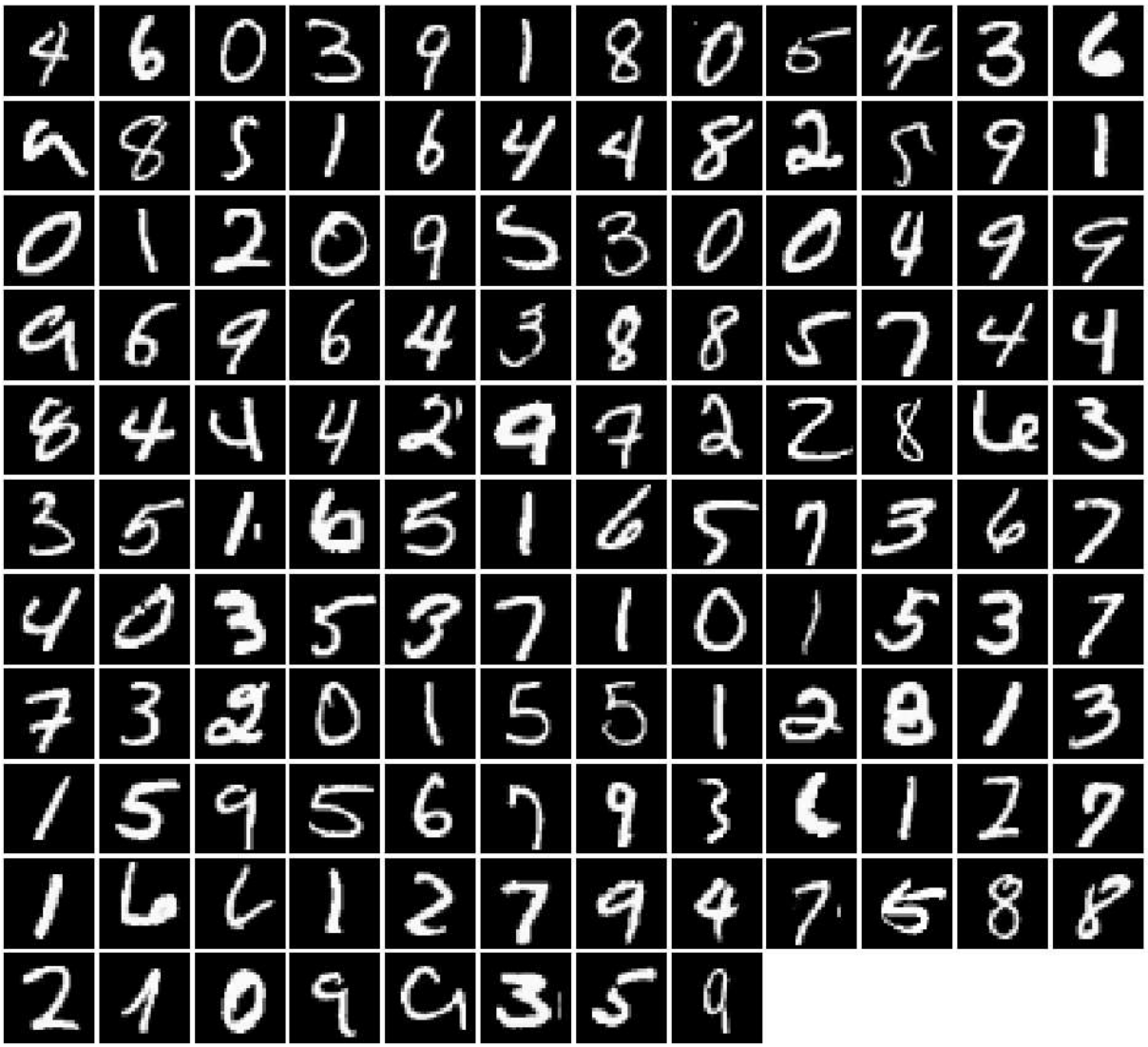
Once again, fortunately for us PyTorch has the MNIST dataset already implemented.

In [3]:
```
batch_size = 128

mnist_train = datasets.MNIST('./MNIST_data_directory', train=True, download=True,
                             transform=T.ToTensor())
data_loader_train = DataLoader(mnist_train, batch_size=batch_size,
                               shuffle=True, drop_last=True, num_workers=2)
```

In [4]:
```
import torch
import math
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

def display_data(data_tensors):
    data_tensors = torch.reshape(data_tensors, [data_tensors.shape[0], -1])

    shape = int(math.ceil(math.sqrt(data_tensors.shape[0])))
    data_dim = int(math.ceil(math.sqrt(data_tensors.shape[1])))

    fig = plt.figure(figsize=(shape, shape))
    grid_spec = gridspec.GridSpec(shape, shape)
    grid_spec.update(wspace=0.05, hspace=0.05)

    for i, sample in enumerate(data_tensors):
        ax = plt.subplot(grid_spec[i])
```

```
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(sample.reshape([data_dim, data_dim]))
    return
```

In [5]:
```
for batch in data_loader_train:
    imgs = batch[0].view(batch_size, 784)  # Assuming the images are in the first e
    display_data(imgs)
    break  # Break after visualizing the first batch if needed
```



Model architecture

Our latent variable model will consist solely of linear (dense) layers. We'll take the input tensor shape and flatten it to create a single input vector dimension. We'll concatenate the image with a one hot vector representing the class to allow us to do generation of a specific digit.

In this section, you'll define the compression (encoder) and decompression (decoder) networks in the ConditionalVariationalAutoencoder class and implement the functions needed for your model.

Compression network (encoder) architecture

Now let's start constructing our latent model architecture. We'll use this transformed representation to predict the parameters of the latent distribution using two linear layers (both with output size equal to the latent dimensionality).

You might be wondering why we don't just predict the variance directly. Why predict the log variance? Isn't this more work? Using the log-variance instead of the variance itself provides improved numerical stability and easier optimization during training. By definition, the variance parameter must be positive. One approach to enforce this constraint would be to apply a ReLU activation function, but the gradient becomes an issue around zero. Additionally, the typical range of desirable variance values is often very small, lying between 0 and a small positive number close to 1. Optimizing over such a narrow range of small numbers can lead to numerical instabilities due to limitations in floating-point arithmetic and poorly-behaved gradients.

By taking the logarithm of the variance, we map the numerically unstable small positive numbers in the range $[0,1]$ to the interval $[-inf, log(1)]$, which provides a much larger space for the optimization variable to explore. Calculating logarithms and exponentials are numerically stable operations, effectively expanding the optimization landscape and improving convergence.

Remember the log-variance is not used directly as the variance parameter itself. Instead, the optimization is performed in the log-space, and the resulting value is transformed back to the original space by taking the exponential before using it as the variance parameter.

Your encoder itself will have three linear layers (not including the two distribution prediction layers). The first layer takes the input (which is an image and a one hot vector of the digit class) and projects it to the hidden state shape. The next two layers do not change the hidden state shape (i.e. they take in vectors of dimension of hidden state and output a vector of the same dimensionality). We will use `ReLU` after all three fully connected layers. We don't use `ReLU` on our predicted distributions.

You should now go into the code and implement the compression network.

Decompression network (decoder) architecture

The decompression network will take the latent representation concatenated with the one hot vector of the class and generate a reconstructed version of the original data. The reason we are adding this concatenation is so that we can generate images of a particular class by

controlling the conditioning. The decompression network will have four dense (linear) layers. The first dense layer takes in the sampled latent coming from the `sample_latent` function and projects to the hidden state size. The next two dense layers take in the previous input and also project to the hidden state size (i.e. no change in dimensionality). The final dense layer projects the hidden state dimensionality back to the input dimensionality (note this is a flattened version of the input, you can just rearrange it to an image shape). The first three dense layers should be followed by `ReLU` activation. In order to clamp outputs between 0 and 1, we'll perform sigmoid activation on the predictions from the last dense layer (i.e. the output). You can then reshape it to be like the input shape.

You should now go into the code and implement the decompression network.

Reparameterization trick

We'll now apply a sampling technique to estimate the latent representation during the forward pass, given the mean and variance parameters predicted by the compression network. Unlike a regular autoencoder, your model predicts a mean and variance for each dimension of the latent, which you then actually get by sampling from the distribution.

You might think you can just directly use the predicted mean and variance to parameterize a Gaussian distribution which you then probablistically sample from. This random sampling process is the problem for us, however, because it is not differentiable, preventing backpropagation through this step. If you tried to directly do this, you would find no gradient reached the encoder. The essential problem is that

Instead, we'll use a mathematical trick. We'll first sample random noise using a Gaussian distribution $\epsilon = \mathcal{N} (0,1)$. Once you have your random noise, we can compute our latent vector like so: $z = \mu + \sigma\epsilon$. Notice now we can compute gradients through μ and σ back to the encoder.

If you want a deeper mathematical understanding of why we actually need this trick, I encourage you to read this page for a detailed overview.

In [6]:
```
# def reparameterize(mu, logvar):
#     std = torch.exp(0.5*logvar)
#     eps = torch.randn_like(std)
#     return eps.mul(std).add_(mu)
```

In [7]:
```
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
```

```
class ConditionalVariationalAutoencoder(nn.Module):
    def __init__(self, input_dim, num_categories=10, latent_dimensionality=15):
        super(ConditionalVariationalAutoencoder, self).__init__()

        self.input_dim = input_dim  # Flattened input dimensionality
        self.latent_dimensionality = latent_dimensionality  # Dimensionality of lat
        self.num_categories = num_categories  # Number of output categories

        self.hidden_size = 64  # Hidden layer size
        self.encoder = None  # Encoder network
        self.mean_layer = None  # Layer to compute mean of latent distribution
        self.logvariance_layer = None  # Layer to compute log variance of latent di
        self.decoder = None  # Decoder network

        self.encoder = nn.Sequential(
            nn.Linear(input_dim + num_categories, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, self.hidden_size),
            nn.ReLU()
        )

        # Layers to compute mean and log variance of latent distribution
        self.mean_layer = nn.Linear(self.hidden_size, latent_dimensionality)
        self.logvariance_layer = nn.Linear(self.hidden_size, latent_dimensionality)

        # Decoder network
        self.decoder = nn.Sequential(
            nn.Linear(latent_dimensionality + num_categories, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, input_dim),
            nn.Sigmoid()  # Assuming input data is in the range [0, 1]
        )
def sample_latent(means, logvariances):
            std = torch.exp(0.5 * logvariances)
            epsilon = torch.randn_like(std)
            latent_samples = means + epsilon*std
            return latent_samples

def loss_function(reconstructions, originals, means, logvariances):
             recon = F.binary_cross_entropy(reconstructions, originals, reduction="
             dkl =  -0.5 * torch.sum(1 + logvariances - means.pow(2) - logvariances
             loss = recon + dkl
             return loss
def forward(self, data_input, condition):
             data_recon = None
             mean_params = None
             logvariance_params = None
             new_input = torch.cat([data_input, condition], dim=1)
```

```
        encoded = self.encoder(new_input)
        mean_params = self.mean_layer(encoded)
        logvariance_params = self.logvariance_layer(encoded)
        latent_sample = sample_latent(mean_params, logvariance_params)
        new_latent = torch.cat([latent_sample, condition], dim=1)
        data_recon = self.decoder(new_latent)
        return data_recon, mean_params, logvariance_params
```

In [8]:
```
# from cs5814_vae import sample_latent
import random
random.seed(5814)
torch.manual_seed(5814)
latent_dimension = 15
size = (1, latent_dimension)
means = torch.zeros(size)
logvars = torch.ones(size)
z = sample_latent(means, logvars)
# print(z)
print(f'Sampled latent: {z}')
print(f'Average latent: {torch.mean(z, dim=-1)}')
print(f'Std of latent: {torch.std(z, dim=-1)}')
```

```
Sampled latent: tensor([[ 0.4761,  1.3325, -0.7178,  0.3497, -0.2346, -0.0667, -0.44
26, -0.6182,
         -0.5145, -2.1463, -1.4753, -0.9416,  0.6395,  1.2382,  0.7332]])
Average latent: tensor([-0.1592])
Std of latent: tensor([0.9751])
```

Loss implementation

The objective for a variational autoencoder typically consists of two terms: A reconstruction term, that measures the degree to which the generated image looks like the original, and a regularization term.

$-E_{Z \sim q_{\phi}(z|x,c)}[log p_{\theta}(x|z,c)] + D_{KL}(q_{\phi}(z|x,c), p(z)))$

The reconstruction term shown here is just the cross entropy between the original image and your generated image (i.e. taking the pixelwise cross-entropy).

The second term is a regularization term. It esssentially forces the learned distribution to not deviate too far from some prior distribution (we could choose any distribution we wanted for the prior, but we'll use the a standard normal ($\mathcal{N} (0,1)$). Essentially we're penalizing the model for how far each parameter's distribution moves away from this prior. This is important to have because otherwise the model could essentially just go back to being a regular autoencoder by making the means arbitrarily far apart. It turns out that you can simplify the KL term if you assume this type of prior:

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

See here for the complete derivation.

You must implement this loss function over a minibatch without any for loops! This is to get you used to working with vectorized calculations in PyTorch. Using for loops is inefficient because PyTorch has specialized code to work quickly over vectors. Take the average loss over all the samples in a minibatch.

In [9]:
```
# def loss_function(reconstructions, originals, means, logvariances):
#     reconstruction_loss = F.binary_cross_entropy(reconstructions, originals, redu
#     kl_divergence = -0.5 * torch.sum(1 + logvariances - means.pow(2) - logvarianc
#     total_loss = reconstruction_loss + kl_divergence
#     return total_loss
```

In [10]:
```
# from cs5814_vae import loss_function
size = (1,15)
image = torch.sigmoid(torch.FloatTensor([[7,3], [2,6]]).unsqueeze(0).unsqueeze(0))
image_recon = torch.sigmoid(torch.FloatTensor([[5,2], [4,2]]).unsqueeze(0).unsqueez
out = loss_function(image, image_recon, torch.ones(size), torch.zeros(size))
print(f'Output of loss function: {out}')
```

```
Output of loss function: 8.834553718566895
```

Training

In [11]:
```
def train_model(epoch, model, train_loader):
    model.train()
    curr_train_loss = 0
    num_classes = 10
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    for batch_idx, (data, labels) in enumerate(train_loader):
        optimizer.zero_grad()
        one_hot_labels = torch.eye(num_classes)[labels]
        data = data.view(-1,784)
        data_recon, mean_params, logvariance_params = model(data, one_hot_labels)
        loss = loss_function(data_recon,data,mean_params,logvariance_params)
        optimizer.zero_grad()
        loss.backward()
        curr_train_loss += loss.item() # Get it as a scalar
        optimizer.step()  # This applies your optimizer
    print('====> Epoch: {} Average loss: {:.4f}'.format(epoch, curr_train_loss / le
```

In [12]:
```
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

class ConditionalVariationalAutoencoder(nn.Module):
    def __init__(self, input_dim, num_categories=10, latent_dimensionality=15):
        super(ConditionalVariationalAutoencoder, self).__init__()
```

```
        self.input_dim = input_dim  # Flattened input dimensionality
        self.latent_dimensionality = latent_dimensionality  # Dimensionality of lat
        self.num_categories = num_categories  # Number of output categories

        self.hidden_size = 64  # Hidden layer size
        self.encoder = None  # Encoder network
        self.mean_layer = None  # Layer to compute mean of latent distribution
        self.logvariance_layer = None  # Layer to compute log variance of latent di
        self.decoder = None  # Decoder network
        self.encoder = nn.Sequential(
            nn.Linear(input_dim + num_categories, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, self.hidden_size),
            nn.ReLU()
        )

        # Layers to compute mean and log variance of latent distribution
        self.mean_layer = nn.Linear(self.hidden_size, latent_dimensionality)
        self.logvariance_layer = nn.Linear(self.hidden_size, latent_dimensionality)

        # Decoder network
        self.decoder = nn.Sequential(
            nn.Linear(latent_dimensionality + num_categories, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, input_dim),
            nn.Sigmoid()
        )

    def forward(self, data_input, condition):
        data_recon = None
        mean_params = None
        logvariance_params = None
        new_input = torch.cat([data_input, condition], dim=1)
        encoded = self.encoder(new_input)
        mean_params = self.mean_layer(encoded)
        logvariance_params = self.logvariance_layer(encoded)
        latent_sample = sample_latent(mean_params, logvariance_params)
        new_latent = torch.cat([latent_sample, condition], dim=1)
        data_recon = self.decoder(new_latent)

        return data_recon, mean_params, logvariance_params

def sample_latent(means, logvariances):
    std = torch.exp(0.5 * logvariances)
    epsilon = torch.randn_like(std)
    latent_samples = means + epsilon*std

    return latent_samples

def loss_function(reconstructions, originals, means, logvariances):
    recon = F.binary_cross_entropy(reconstructions, originals, reduction="sum")
    dkl =  -0.5 * torch.sum(1 + logvariances - means.pow(2) - logvariances.exp())
```

```
    loss = recon + dkl
    return loss
```

In [13]:
```
# from cs5814_vae import ConditionalVariationalAutoencoder
# from cs5814_vae import sample_latent
num_epochs = 20 # feel free to explore with this - 10 may work ell as well def: 50,
latent_dimensionality = 100 # feel free to explore
# from cs5814_vae import ConditionalVariationalAutoencoder
image_size = 28*28 # size of the images
cvae_model = ConditionalVariationalAutoencoder(image_size, latent_dimensionality=la
# cvae_model.to(device)
for epoch in range(0, num_epochs):
  train_model(epoch, cvae_model, data_loader_train)
```

```
====> Epoch: 0 Average loss: 215.8724
====> Epoch: 1 Average loss: 172.2223
====> Epoch: 2 Average loss: 149.1253
====> Epoch: 3 Average loss: 138.4145
====> Epoch: 4 Average loss: 132.1710
====> Epoch: 5 Average loss: 128.4033
====> Epoch: 6 Average loss: 125.4964
====> Epoch: 7 Average loss: 123.2576
====> Epoch: 8 Average loss: 121.4628
====> Epoch: 9 Average loss: 120.0902
====> Epoch: 10 Average loss: 119.0961
====> Epoch: 11 Average loss: 118.2118
====> Epoch: 12 Average loss: 117.5042
====> Epoch: 13 Average loss: 116.8375
====> Epoch: 14 Average loss: 116.2659
====> Epoch: 15 Average loss: 115.7765
====> Epoch: 16 Average loss: 115.3006
====> Epoch: 17 Average loss: 114.9543
====> Epoch: 18 Average loss: 114.5805
====> Epoch: 19 Average loss: 114.2753
```

Create visualizations

Because we conditioned our model on the class label, we can now control which class we are generating. If we didn't have this conditioning, sampling would just produce images of a random class. Because the model was conditioned on the digit, the class information is encoded by the conditioning and the latent space just captures the variation in appearance.

In [15]:
```
import matplotlib.gridspec as gridspec
latents = torch.randn(10, latent_dimensionality)
one_hots = torch.eye(10, 10)
latents_and_one_hots = torch.cat((latents, one_hots), dim=-1)
cvae_model.eval()
samples = cvae_model.decoder(latents_and_one_hots).data.cpu().numpy()

fig = plt.figure(figsize=(10, 1)) # setting the figure size of 10
gspec = gridspec.GridSpec(1, 10) # this sets each row
gspec.update(wspace=0.06, hspace=0.06)
```

```
for i, sample in enumerate(samples):
  ax = plt.subplot(gspec[i])
  plt.axis('off')
  ax.set_xticklabels([]) # remove ticks
  ax.set_yticklabels([]) # remove ticks
  ax.set_aspect('equal')
  plt.imshow(sample.reshape(28, 28), cmap='Greys_r')
  plt.savefig(f'your_result_{i}.jpg')
```