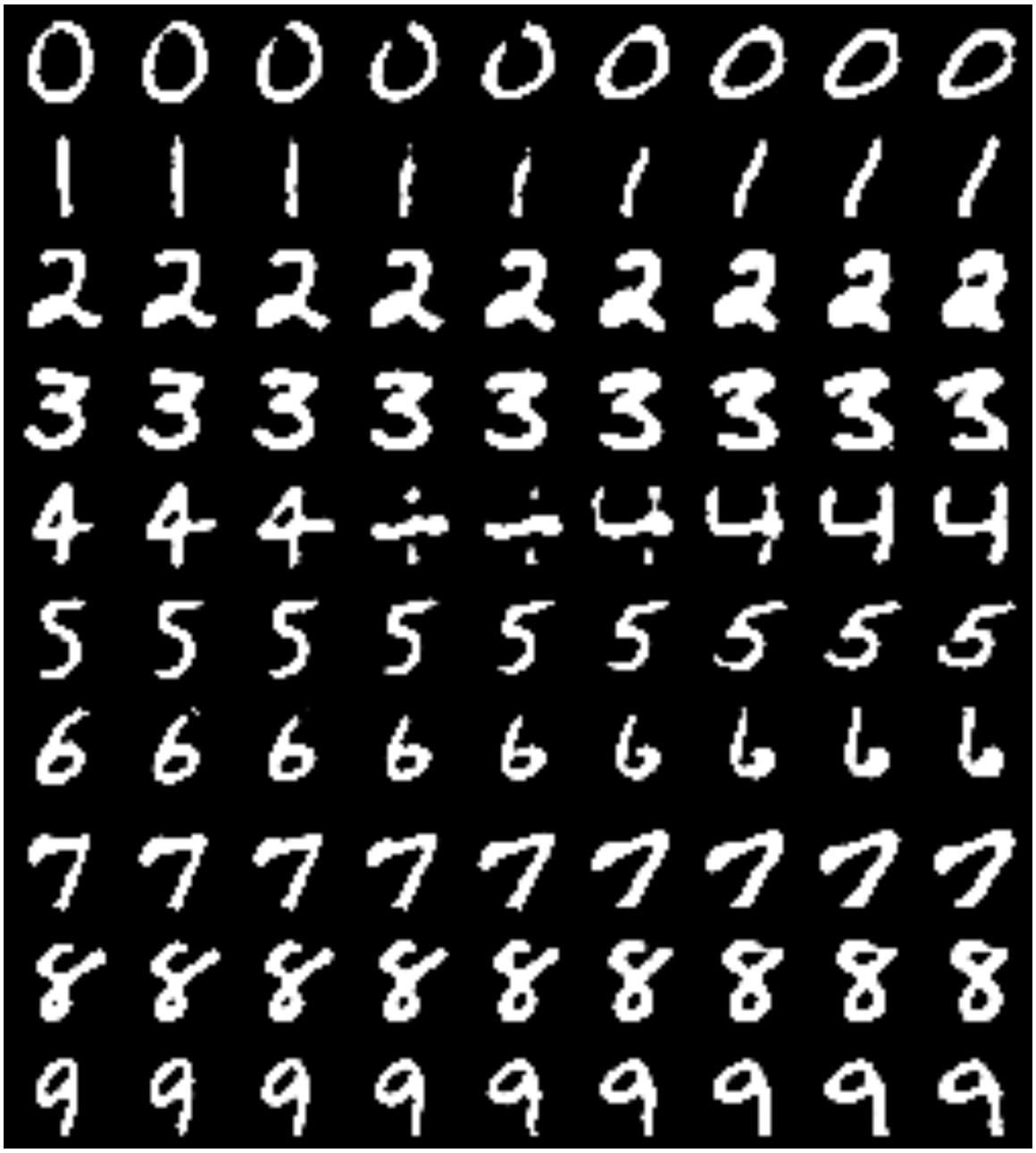
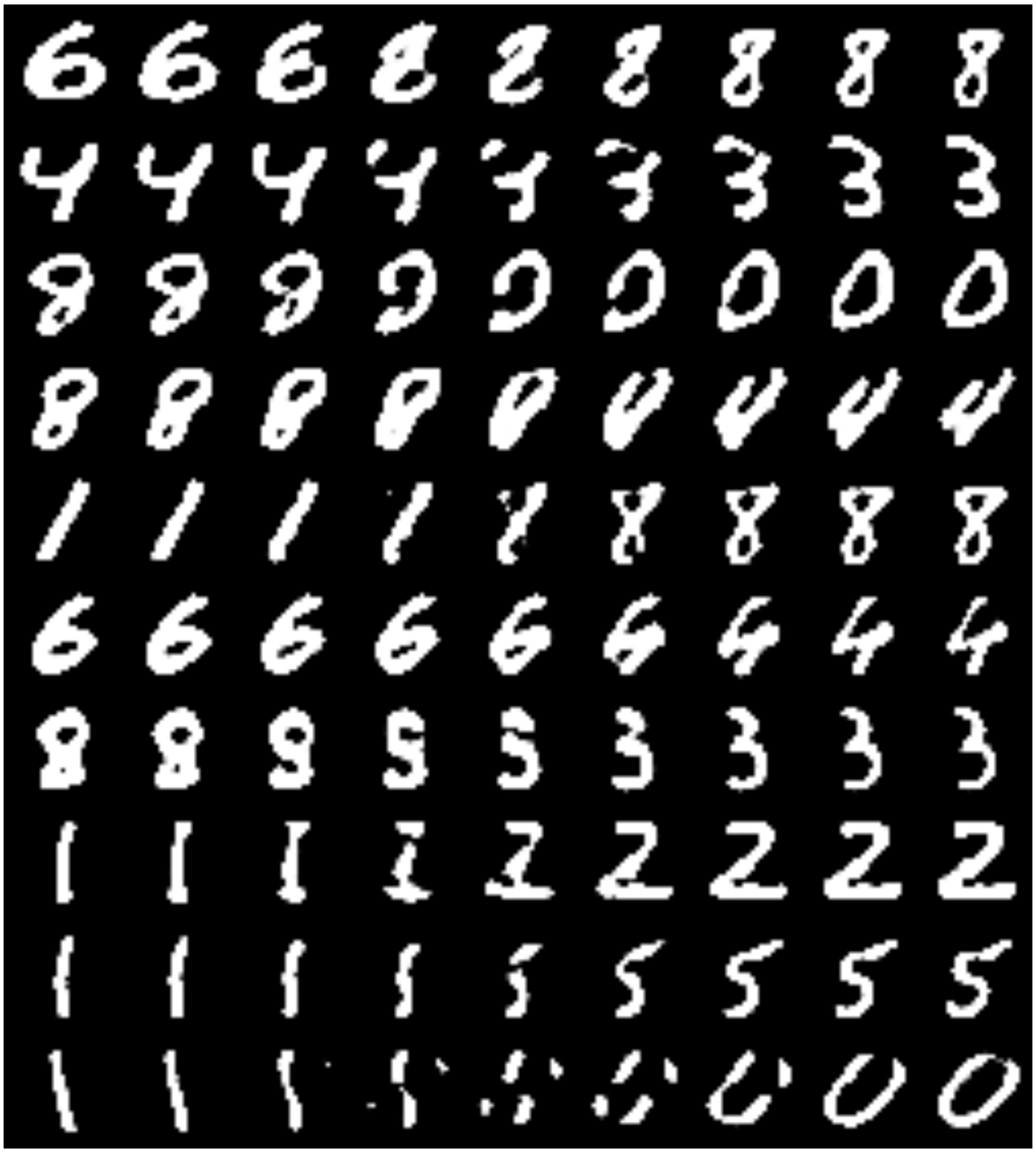
Homework 9: Variational Autoencoders

1. **(45 points)** Page 1: Same digit interpolates (Make your figure large!)



2. **(45 points)** Page 2: Different digit interpolates (Make your figure large!)



3. **(10 points)** Page 3+: Your code.

```
[2]  !mkdir hw9_data
```

```
[4]  from torchvision import datasets, transforms

     ## YOUR CODE HERE ##
     transformations = transforms.Compose([
         transforms.ToTensor(),
         transforms.Normalize((0.1307,), (0.3081,))
     ])
     mnist_train = datasets.MNIST(root = 'hw9_data', transform = transformations, train = True, download = True)
     mnist_test = datasets.MNIST(root = 'hw9_data', transform = transformations, train = False, download = True)
```

```
[5]  from torch.utils.data import DataLoader

     ## YOUR CODE HERE ##
     train_loader = DataLoader(dataset = mnist_train, batch_size = 32, shuffle = True)
     test_loader = DataLoader(dataset = mnist_test, batch_size = 32, shuffle = True)
```

```
[14] import os
     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

     # changed configuration to this instead of argparse for easier interaction
     CUDA = False
     SEED = 1
     BATCH_SIZE = 128
     LOG_INTERVAL = 10
     EPOCHS = 10
```

```
[14] class VAE(nn.Module):
         def __init__(self):
             super(VAE, self).__init__()

             # ENCODER
             # 28 x 28 pixels = 784 input pixels, 400 outputs
             self.fc1 = nn.Linear(784, 400)
             # rectified linear unit layer from 400 to 400
             # max(0, x)
             self.relu = nn.ReLU()
             self.fc21 = nn.Linear(400, ZDIMS)  # mu layer
             self.fc22 = nn.Linear(400, ZDIMS)  # logvariance layer
             # this last layer bottlenecks through ZDIMS connections

             # DECODER
             # from bottleneck to hidden 400
             self.fc3 = nn.Linear(ZDIMS, 400)
             # from hidden 400 to 784 outputs
             self.fc4 = nn.Linear(400, 784)
             self.sigmoid = nn.Sigmoid()

         def encode(self, x: Variable) -> (Variable, Variable):
             """Input vector x -> fully connected 1 -> ReLU -> (fully connected
             21, fully connected 22)

             Parameters
             ----------
             x : [128, 784] matrix; 128 digits of 28x28 pixels each

             Returns
             -------

             (mu, logvar) : ZDIMS mean units one for each latent dimension, ZDIMS
                 variance units one for each latent dimension

             """
```

[14]
```
        # h1 is [128, 400]
        h1 = self.relu(self.fc1(x))  # type: Variable
        return self.fc21(h1), self.fc22(h1)

    def reparameterize(self, mu: Variable, logvar: Variable) -> Variable:
        """THE REPARAMETERIZATION IDEA:

        For each training sample (we get 128 batched at a time)

        - take the current learned mu, stddev for each of the ZDIMS
          dimensions and draw a random sample from that distribution
        - the whole network is trained so that these randomly drawn
          samples decode to output that looks like the input
        - which will mean that the std, mu will be learned
          *distributions* that correctly encode the inputs
        - due to the additional KLD term (see loss_function() below)
          the distribution will tend to unit Gaussians

        Parameters
        ----------
        mu : [128, ZDIMS] mean matrix
        logvar : [128, ZDIMS] variance matrix

        Returns
        -------

        During training random sample from the learned ZDIMS-dimensional
        normal distribution; during inference its mean.

        """

        if self.training:
            # multiply log variance with 0.5, then in-place exponent
            # yielding the standard deviation
            std = logvar.mul(0.5).exp_()  # type: Variable
            # - std.data is the [128,ZDIMS] tensor that is wrapped by std
            # - so eps is [128,ZDIMS] with all elements drawn from a mean 0
            #   and stddev 1 normal distribution that is 128 samples
            #   of random ZDIMS-float vectors
```

[14]
```
            eps = Variable(std.data.new(std.size()).normal_())
            # - sample from a normal distribution with standard
            #   deviation = std and mean = mu by multiplying mean 0
            #   stddev 1 sample with desired std and mu, see
            #   https://stats.stackexchange.com/a/16338
            # - so we have 128 sets (the batch) of random ZDIMS-float
            #   vectors sampled from normal distribution with learned
            #   std and mu for the current input
            return eps.mul(std).add_(mu)

        else:
            # During inference, we simply spit out the mean of the
            # learned distribution for the current input.  We could
            # use a random sample from the distribution, but mu of
            # course has the highest probability.
            return mu

    def decode(self, z: Variable) -> Variable:
        h3 = self.relu(self.fc3(z))
        return self.sigmoid(self.fc4(h3))

    def forward(self, x: Variable) -> (Variable, Variable, Variable):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), z, mu, logvar


model = VAE()
if CUDA:
    model.cuda()
```

[14]
```
def loss_function(recon_x, x, mu, logvar) -> Variable:
    # how well do input x and output recon_x agree?
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784))

    # KLD is Kullback–Leibler divergence -- how much does one learned
    # distribution deviate from another, in this specific case the
    # learned distribution from the unit Gaussian

    # see Appendix B from VAE paper:
    # Kingma and Welling. Auto-Encoding Variational Bayes. ICLR, 2014
    # https://arxiv.org/abs/1312.6114
    # - D_{KL} = 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    # note the negative D_{KL} in appendix B of the paper
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    # Normalise by same number of elements as in reconstruction
    KLD /= BATCH_SIZE * 784

    # BCE tries to make our reconstruction as accurate as possible
    # KLD tries to push the distributions as close as possible to unit Gaussian
    return BCE + KLD

# Dr Diederik Kingma: as if VAEs weren't enough, he also gave us Adam!
optimizer = optim.Adam(model.parameters(), lr=1e-3)


def train(epoch):
    # toggle model to train mode
    model.train()
    train_loss = 0
    # in the case of MNIST, len(train_loader.dataset) is 60000
    # each `data` is of BATCH_SIZE samples and has shape [128, 1, 28, 28]
    for batch_idx, (data, _) in enumerate(train_loader):
        data = Variable(data)
        if CUDA:
            data = data.cuda()
        optimizer.zero_grad()
```

[14]
```
        # push whole batch of data through VAE.forward() to get recon_loss
        recon_batch, code, mu, logvar = model(data)
        # calculate scalar loss
        loss = loss_function(recon_batch, data, mu, logvar)
        # calculate the gradient of the loss w.r.t. the graph leaves
        # i.e. input variables -- by the power of pytorch!
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
        if batch_idx % LOG_INTERVAL == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader),
                loss.item() / len(data)))

    print('====> Epoch: {} Average loss: {:.4f}'.format(
          epoch, train_loss / len(train_loader.dataset)))


def test(epoch):
    # toggle model to test / inference mode
    model.eval()
    test_loss = 0

    # each data is of BATCH_SIZE (default 128) samples
    for i, (data, _) in enumerate(test_loader):
        if CUDA:
            # make sure this lives on the GPU
            data = data.cuda()

        # we're only going to infer, so no autograd at all required: volatile=True
        data = Variable(data, volatile=True)
        recon_batch, mu, logvar = model(data)
        test_loss += loss_function(recon_batch, data, mu, logvar).item()
        if i == 0:
          n = min(data.size(0), 8)
          # for the first 128 batch of the epoch, show the first 8 input digits
          # with right below them the reconstructed output digits
```

```
[14]            comparison = torch.cat([data[:n],
                                        recon_batch.view(BATCH_SIZE, 1, 28, 28)[:n]])
                save_image(comparison.data.cpu(),
                           'results/reconstruction_' + str(epoch) + '.png', nrow=n)

        test_loss /= len(test_loader.dataset)
        print('====> Test set loss: {:.4f}'.format(test_loss))


    for epoch in range(1, EPOCHS + 1):
        train(epoch)
    #     test(epoch)

        # 64 sets of random ZDIMS-float vectors, i.e. 64 locations / MNIST
        # digits in latent space
        sample = Variable(torch.randn(64, ZDIMS))
        if CUDA:
            sample = sample.cuda()
        sample = model.decode(sample).cpu()
```

Part 4: Visualizing the VAE output (90 points)

We will look at how well the codes produced by the VAE can be interpolated. **For this section we will only use the MNIST test set.**

To create an interpolation between two images A and B, we encode both these images and generate a sample code for each of them. We now consider 7 equally spaced points in between these two sample codes giving us a total of 9 points including the samples. We then decode these images to get interpolated images in between A and B.

Complete the interpolation function below that takes a pair of images A and B and returns 9 images. (You are free to use any data structure you want to return these images)

```
[18] import matplotlib.pyplot as plt
     from torchvision import utils
     %matplotlib inline
     import numpy as np

     def create_interpolates(A, B, vae):
       ## YOUR CODE HERE ##
       mu_a, logvar_a = model.encode(A)
       mu_b, logvar_b = model.encode(B)
       code_a = model.reparameterize(mu_a, logvar_a)
       code_b = model.reparameterize(mu_b, logvar_b)
       interval = (code_b - code_a)/8
       index = 1
       for i in range(8):
         code = code_a + interval * i
         image = model.decode(code).view(28, 28)
         if index:
           interpolates = image
           index = 0
         else:
           interpolates = torch.cat((interpolates, image))
       interpolates = torch.cat((interpolates, model.decode(code_b).view(28, 28)))
       interpolates = interpolates.view(9, 1, 28, 28)
       grid = utils.make_grid(interpolates, nrow = 1)
       show(grid)
```

```
def show(img):
    npimg = img.detach().numpy()
    plt.imshow(np.transpose(npimg, (1,2,0)), interpolation='nearest')

similar_pairs = {}
for _, (x, y) in enumerate(test_loader):
  for i in range(len(y)):
    if y[i].item() not in similar_pairs:
      similar_pairs[y[i].item()] = []
    if len(similar_pairs[y[i].item()])<2:
      similar_pairs[y[i].item()].append(x[i])

  done = True
  for i in range(10):
    if i not in similar_pairs or len(similar_pairs[i])<2:
      done = False

  if done:
    break

# similar_pairs[i] contains two images indexed at 0 and 1 that have images of the digit i

## YOUR CODE HERE ##
interpolates = []
index = 1
for i in range(10):
  digits = similar_pairs[i]
  digit1 = digits[0].view(-1, 28*28)
  digit2 = digits[1].view(-1, 28*28)
  interpolate = create_interpolates(digit1, digit2, model)
  if index:
    interpolates = interpolate
    index = 0
  else:
    interpolates = torch.cat((interpolates, interpolate))
```

```
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize(size=800),
    transforms.ToTensor()
])
interpolates = [transform(interpolates_) for interpolates_ in interpolates]
grid = utils.make_grid(interpolates, nrow = 9)
utils.save_image(grid, 'same_digit.png')
show(grid)
```

```
[34] random_pairs = {}
     for _, (x, y) in enumerate(test_loader):
       # Make sure the batch size is greater than 20
       for i in range(10):
         random_pairs[i] = []
         random_pairs[i].append(x[2*i])
         random_pairs[i].append(x[2*i+1])
       break

     # random_pairs[i] contains two images indexed at 0 and 1 that are chosen at random

     ## YOUR CODE HERE ##
     interpolates = []
     index = 1
     for i in range(10):
       digits = random_pairs[i]
       digit1 = digits[0].view(-1, 28*28)
       digit2 = digits[1].view(-1, 28*28)
       interpolate = create_interpolates(digit1, digit2, model)
       if index:
         interpolates = interpolate
         index = 0
       else:
         interpolates = torch.cat((interpolates, interpolate))
     transform = transforms.Compose([
         transforms.ToPILImage(),
         transforms.Resize(size=800),
         transforms.ToTensor()
     ])
     interpolates = [transform(interpolates_) for interpolates_ in interpolates]
     grid = utils.make_grid(interpolates, nrow = 9)
     utils.save_image(grid, 'different_digit.png')
     show(grid)
```