gan

April 20, 2025

We would like to acknowledge Stanford University's CS231n on which we based the development of this project.

[1]:
```
USE_COLAB = False

import sys
import os

if USE_COLAB:
    # This mounts your Google Drive to the Colab VM.
    from google.colab import drive
    drive.mount('/content/drive')

    # TODO: Enter the foldername in your Drive where you have saved the unzipped
    # project folder, e.g. '239AS.2/project1/gan'
    FOLDERNAME = None
    assert FOLDERNAME is not None, "[!] Enter the foldername."

    # Now that we've mounted your Drive, this ensures that
    # the Python interpreter of the Colab VM can load
    # python files from within it.

    sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

    %cd /content/drive/My\ Drive/$FOLDERNAME


# You can use os.listdir() to find the folder
# Google drive is...weird.
print('cwd', os.getcwd())
```

```
cwd /Users/paultalma/Documents/UCLA/Work/Classes/2024-
2025/ee_239/projects/project_1/gan
```

TO FIND YOUR FILES IN COLAB:

Click the folder on the left sidebar (content->)drive->MyDrive->WhereverYouPutTheseFiles

Double clicking gan.py will open an editor on the right.

0.1 Enabling GPU:

Runtime > Change runtime type > Hardware accelerator: select T4 GPU or v2-8 TPU. You may run out of compute allowance at some point. **When you will be away for a long period of time, select Runtime->Disconnect and delete runtime! Make sure to preserve your compute allowance!**

[2]:
```
import torch
if torch.backends.mps.is_available():
    mps_device = torch.device("mps")
    x = torch.ones(1, device=mps_device)
    print (x)
else:
    print ("MPS device not found.")
```

```
tensor([1.], device='mps:0')
```

1 Generative Adversarial Networks (GANs)

In C147/C247, all the applications of neural networks that we have explored have been **discriminative models** that take an input and are trained to produce a labeled output. In this notebook, we will expand our repetoire, and build **generative models** using neural networks. Specifically, we will learn how to build models which generate novel images that resemble a set of training images.

1.0.1 What is a GAN?

In 2014, Goodfellow et al. presented a method for training generative models called Generative Adversarial Networks (GANs for short). In a GAN, we build two different neural networks. Our first network is a traditional classification network, called the **discriminator**. We will train the discriminator to take images and classify them as being real (belonging to the training set) or fake (not present in the training set). Our other network, called the **generator**, will take random noise as input and transform it using a neural network to produce images. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

We can think of this back and forth process of the generator (G) trying to fool the discriminator (D) and the discriminator trying to correctly classify real vs. fake as a minimax game:

$$\underset{G}{\text{minimize}}\; \underset{D}{\text{maximize}}\; \mathbb{E}_{x \sim p_\text{data}} \left[\log D(x)\right] + \mathbb{E}_{z \sim p(z)} \left[\log \left(1 - D(G(z))\right)\right]$$

where $z \sim p(z)$ are the random noise samples, $G(z)$ are the generated images using the neural network generator G, and D is the output of the discriminator, specifying the probability of an input being real. In Goodfellow et al., they analyze this minimax game and show how it relates to minimizing the Jensen-Shannon divergence between the training data distribution and the generated samples from G.

To optimize this minimax game, we will alternate between taking gradient *descent* steps on the objective for G and gradient *ascent* steps on the objective for D: 1. update the **generator** (G) to minimize the probability of the **discriminator making the correct choice**. 2. update the **discriminator** (D) to maximize the probability of the **discriminator making the correct choice**.

While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the probability of the **discriminator making the incorrect choice**. This small change helps to allevaiate problems with the generator gradient vanishing when the discriminator is confident. This is the standard update used in most GAN papers and was used in the original paper from Goodfellow et al..

In this assignment, we will alternate the following updates: 1. Update the generator (G) to maximize the probability of the discriminator making the incorrect choice on generated data:

$$\underset{G}{\text{maximize}}\; \mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$

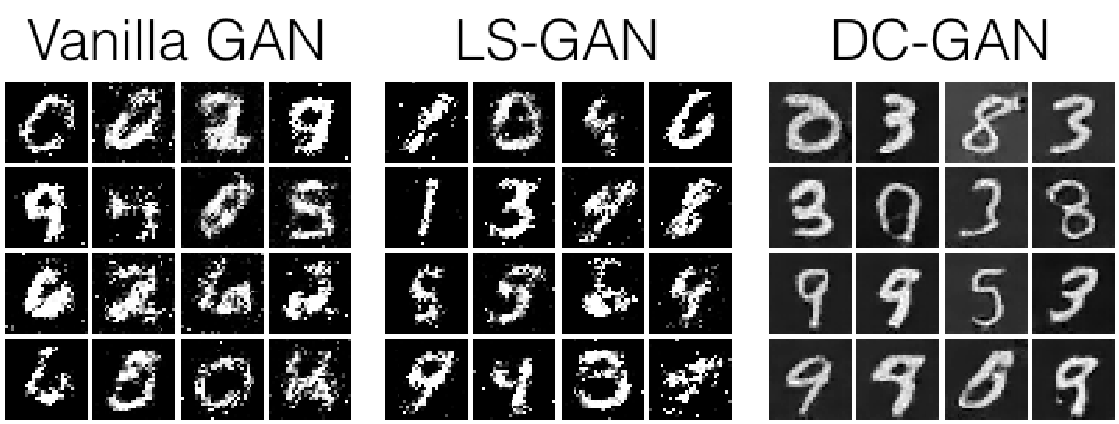
2. Update the discriminator (D), to maximize the probability of the discriminator making the correct choice on real and generated data:

$$\underset{D}{\text{maximize}}\; \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

Here's an example of what your outputs from the 3 different models you're going to train should look like. Note that GANs are sometimes finicky, so your outputs might not look exactly like this. This is just meant to be a *rough* guideline of the kind of quality you can expect:

[3]:
```python
# Run this cell to see sample outputs.
from IPython.display import Image
Image('nndl2/gan_outputs.png')
```

[3]:



[4]:
```python
# Setup cell.
import numpy as np
import torch
import torch.nn as nn
from torch.nn import init
import torchvision
import torchvision.transforms as T
```

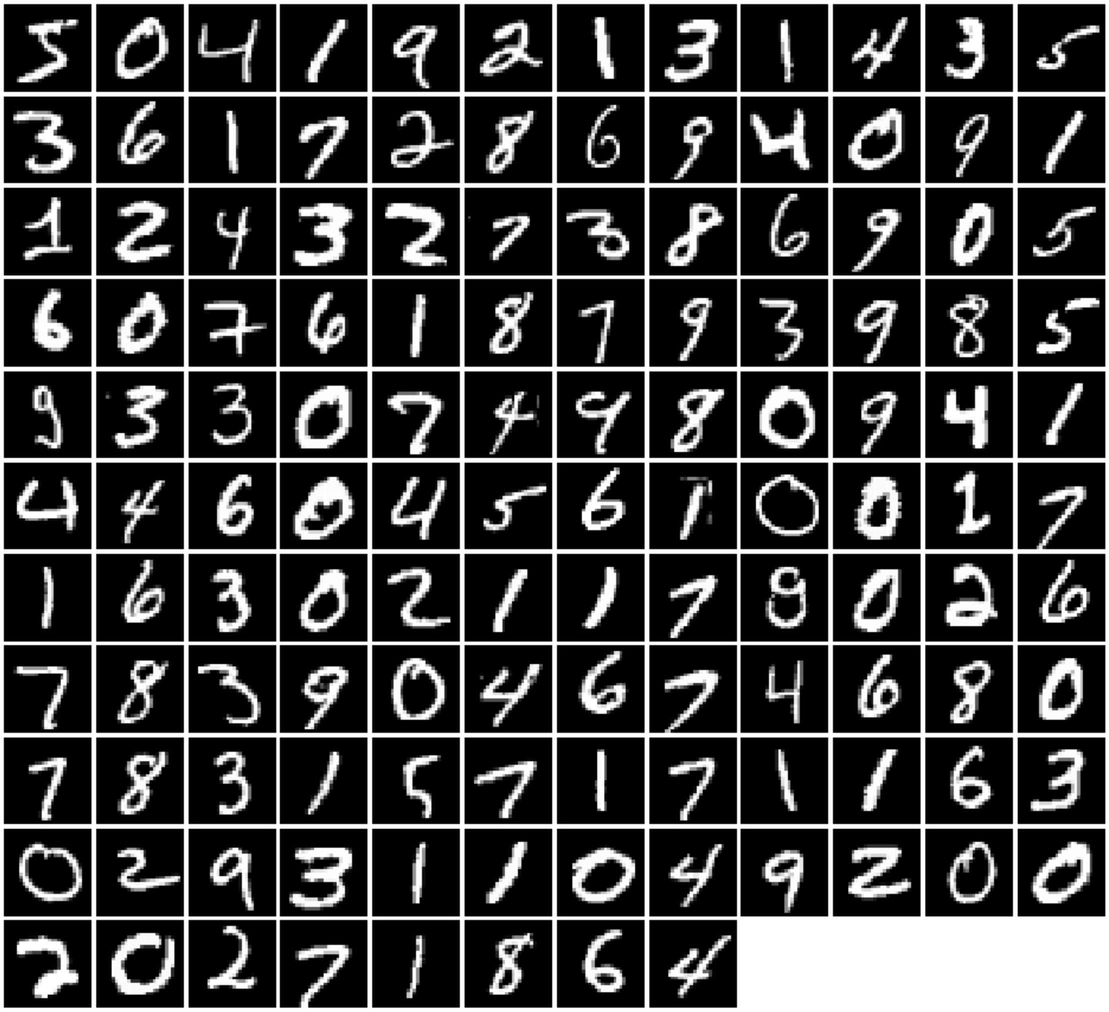
```python
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
import torchvision.datasets as dset
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from gan import preprocess_img, deprocess_img, rel_error, count_params,␣
↪ChunkSampler

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # Set default size of plots.
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

def show_images(images):
    images = np.reshape(images, [images.shape[0], -1]) # Images reshape to␣
↪(batch_size, D).
    sqrtn = int(np.ceil(np.sqrt(images.shape[0])))
    sqrtimg = int(np.ceil(np.sqrt(images.shape[1])))

    fig = plt.figure(figsize=(sqrtn, sqrtn))
    gs = gridspec.GridSpec(sqrtn, sqrtn)
    gs.update(wspace=0.05, hspace=0.05)

    for i, img in enumerate(images):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(img.reshape([sqrtimg,sqrtimg]))
    return

answers = dict(np.load('nndl2/gan-checks.npz'))
dtype = torch.cuda.FloatTensor if torch.cuda.is_available() else torch.
↪FloatTensor
```

1.1 Dataset

GANs 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`.

[5]:
```
NUM_TRAIN = 50000
NUM_VAL = 5000

NOISE_DIM = 96
batch_size = 128

mnist_train = dset.MNIST(
    './nndl2',
    train=True,
    download=True,
    transform=T.ToTensor()
)
loader_train = DataLoader(
    mnist_train,
    batch_size=batch_size,
    sampler=ChunkSampler(NUM_TRAIN, 0)
)

mnist_val = dset.MNIST(
    './nndl2',
    train=True,
    download=True,
    transform=T.ToTensor()
)
loader_val = DataLoader(
    mnist_val,
    batch_size=batch_size,
    sampler=ChunkSampler(NUM_VAL, NUM_TRAIN)
)

iterator = iter(loader_train)
imgs, labels = next(iterator)
imgs = imgs.view(batch_size, 784).numpy().squeeze()
show_images(imgs)
```



1.2 Random Noise (1 point)

Generate uniform noise from -1 to 1 with shape `[batch_size, dim]`.

Implement `sample_noise` in `gan.py`.

Hint: use `torch.rand`.

Make sure noise is the correct shape and type:

[6]:
```
from gan import sample_noise

def test_sample_noise():
    batch_size = 3
    dim = 4
    torch.manual_seed(231)
    z = sample_noise(batch_size, dim)
```

```
    np_z = z.cpu().numpy()
    assert np_z.shape == (batch_size, dim)
    assert torch.is_tensor(z)
    assert np.all(np_z >= -1.0) and np.all(np_z <= 1.0)
    assert np.any(np_z < 0.0) and np.any(np_z > 0.0)
    print('All tests passed!')

test_sample_noise()
```

```
All tests passed!
```

1.3 Flatten

We provide an Unflatten, which you might want to use when implementing the convolutional generator. We also provide a weight initializer (and call it for you) that uses Xavier initialization instead of PyTorch's uniform default.

[7]:
```
from gan import Flatten, Unflatten, initialize_weights
```

2 Discriminator (1 point)

Our first step is to build a discriminator. Fill in the architecture as part of the `nn.Sequential` constructor in the function below. All fully connected layers should include bias terms. The architecture is: * Fully connected layer with input size 784 and output size 256 * LeakyReLU with alpha 0.01 * Fully connected layer with input_size 256 and output size 256 * LeakyReLU with alpha 0.01 * Fully connected layer with input size 256 and output size 1

Recall that the Leaky ReLU nonlinearity computes $f(x) = \max(\alpha x, x)$ for some fixed constant α; for the LeakyReLU nonlinearities in the architecture above we set $\alpha = 0.01$.

The output of the discriminator should have shape `[batch_size, 1]`, and contain real numbers corresponding to the scores that each of the `batch_size` inputs is a real image.

Implement `discriminator` in `gan.py`

Test to make sure the number of parameters in the discriminator is correct:

[8]:
```
from gan import discriminator

def test_discriminator(true_count=267009):
    model = discriminator()
    cur_count = count_params(model)
    if cur_count != true_count:
        print('Incorrect number of parameters in discriminator. Check your␣
 ↪achitecture.')
    else:
        print('Correct number of parameters in discriminator.')
```

```
test_discriminator()
```

```
Correct number of parameters in discriminator.
```

3 Generator (1 point)

Now to build the generator network: * Fully connected layer from noise_dim to 1024 * `ReLU` * Fully connected layer with size 1024 * `ReLU` * Fully connected layer with size 784 * `TanH` (to clip the image to be in the range of [-1,1])

All fully connected layers should include bias terms. Implement `generator` in `gan.py`

Test to make sure the number of parameters in the generator is correct:

[9]:
```
from gan import generator

def test_generator(true_count=1858320):
    model = generator(4)
    cur_count = count_params(model)
    if cur_count != true_count:
        print('Incorrect number of parameters in generator. Check your␣
 ↪achitecture.')
    else:
        print('Correct number of parameters in generator.')

test_generator()
```

```
Correct number of parameters in generator.
```

4 GAN Loss (2 points)

Compute the generator and discriminator loss. The generator loss is:

$$\ell_G = -\mathbb{E}_{z \sim p(z)} [\log D(G(z))]$$

and the discriminator loss is:

$$\ell_D = -\mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] - \mathbb{E}_{z \sim p(z)} [\log (1 - D(G(z)))]$$

Note that these are negated from the equations presented earlier as we will be *minimizing* these losses.

HINTS: You should use the `bce_loss` function defined below to compute the binary cross entropy loss which is needed to compute the log probability of the true label given the logits output from the discriminator. Given a score $s \in \mathbb{R}$ and a label $y \in \{0, 1\}$, the binary cross entropy loss is

$$bce(s, y) = -y * \log(s) - (1 - y) * \log(1 - s)$$

A naive implementation of this formula can be numerically unstable, so we have provided a numerically stable implementation that relies on PyTorch's `nn.BCEWithLogitsLoss`.

You will also need to compute labels corresponding to real or fake and use the logit arguments to determine their size. Make sure you cast these labels to the correct data type using the global `dtype` variable for compatibility with BCEWithLogitsLoss, which uses floating point data types.

```
true_labels = torch.ones(size).type(dtype)
```

Instead of computing the expectation of $\log D(G(z))$, $\log D(x)$ and $\log(1 - D(G(z)))$, we will be averaging over elements of the minibatch. This is taken care of in `bce_loss` which combines the loss by averaging. **Hint**: Do NOT concatenate real and fake losses. Sum the two (empirical) expectations instead.

Implement `discriminator_loss` and `generator_loss` in `gan.py`

Test your generator and discriminator loss. You should see errors < 1e-7.

[10]:
```
from gan import bce_loss, discriminator_loss, generator_loss

def test_discriminator_loss(logits_real, logits_fake, d_loss_true):
    d_loss = discriminator_loss(torch.Tensor(logits_real).type(dtype),
                                torch.Tensor(logits_fake).type(dtype)).cpu().
 ↪numpy()
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))

test_discriminator_loss(
    answers['logits_real'],
    answers['logits_fake'],
    answers['d_loss_true']
)
```

```
Maximum error in d_loss: 3.97058e-09
```

[11]:
```
def test_generator_loss(logits_fake, g_loss_true):
    g_loss = generator_loss(torch.Tensor(logits_fake).type(dtype)).cpu().numpy()
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

test_generator_loss(
    answers['logits_fake'],
    answers['g_loss_true']
)
```

```
Maximum error in g_loss: 4.4518e-09
```

5 Optimizing Our Loss (1 point)

Make a function that returns an `optim.Adam` optimizer for the given model with a 1e-3 learning rate, beta1=0.5, beta2=0.999. You'll use this to construct optimizers for the generators and discriminators for the rest of the notebook.

Implement `get_optimizer` in `gan.py`

6 Training a GAN!

We provide you the main training loop. You won't need to change `run_a_gan` in `gan.py`, but we encourage you to read through it for your own understanding. If you train with the CPU, it takes about 7 minutes. If you train with the T4 GPU, it takes about 1 minute and 30 seconds.

If it doesn't work the first time, check the discriminator: Did you use Flatten()?

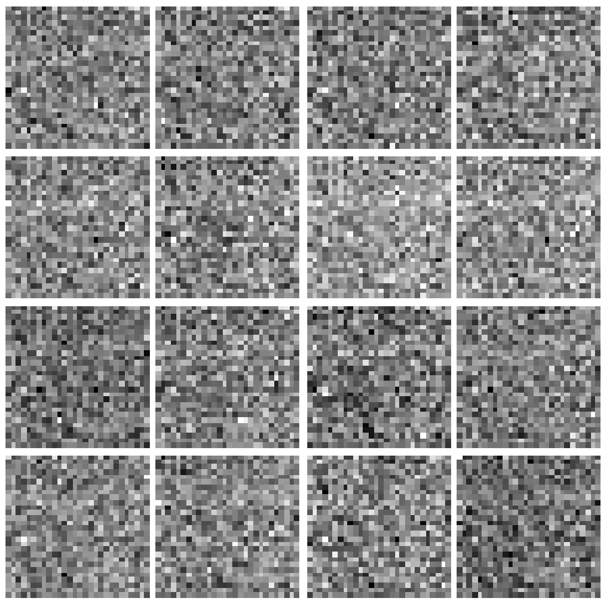
[12]:
```python
from gan import get_optimizer, run_a_gan

# Make the discriminator
D = discriminator().type(dtype)

# Make the generator
G = generator().type(dtype)

# Use the function you wrote earlier to get optimizers for the Discriminator
 ↪and the Generator
D_solver = get_optimizer(D)
G_solver = get_optimizer(G)

# Run it!
images = run_a_gan(
    D,
    G,
    D_solver,
    G_solver,
    discriminator_loss,
    generator_loss,
    loader_train
)
```

```
Iter: 0, D: 1.328, G:0.7202
Iter: 250, D: 1.408, G:0.7591
Iter: 500, D: 1.012, G:1.619
Iter: 750, D: 1.196, G:1.07
Iter: 1000, D: 1.197, G:1.186
Iter: 1250, D: 1.332, G:0.9702
Iter: 1500, D: 1.001, G:1.138
Iter: 1750, D: 1.183, G:0.9221
Iter: 2000, D: 1.281, G:0.9347
Iter: 2250, D: 1.259, G:0.9923
Iter: 2500, D: 1.311, G:0.7731
Iter: 2750, D: 1.257, G:0.7311
Iter: 3000, D: 1.298, G:0.6164
Iter: 3250, D: 1.343, G:0.8261
Iter: 3500, D: 1.283, G:0.8249
Iter: 3750, D: 1.276, G:0.801
```

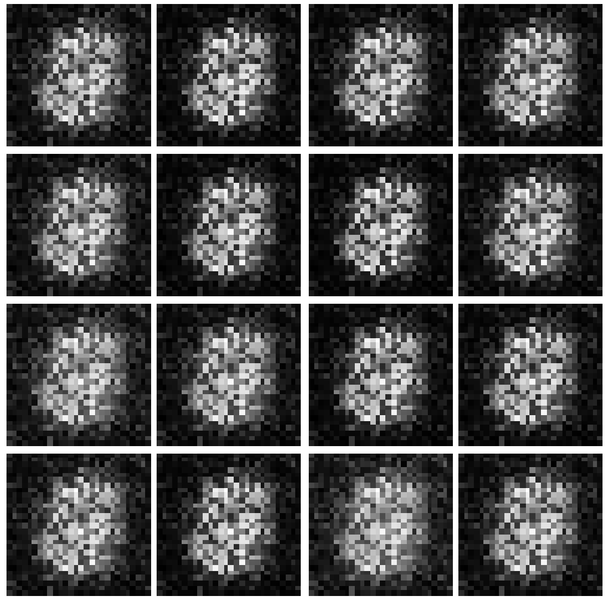
Run the cell below to show the generated images.

[13]:
```
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

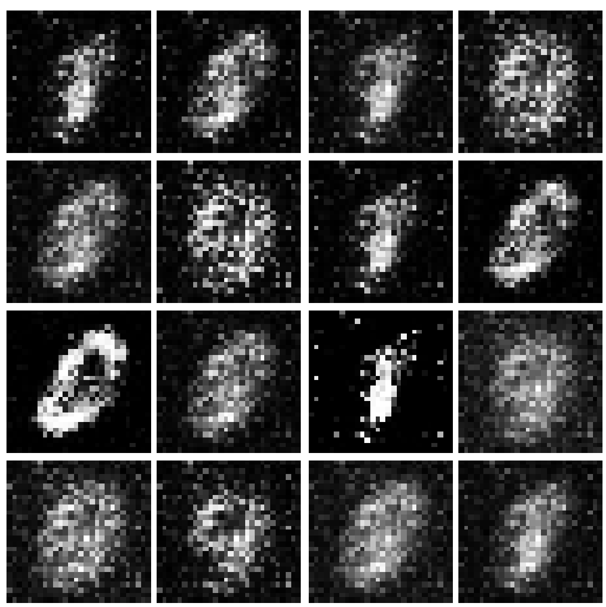
```
Iter: 0
```



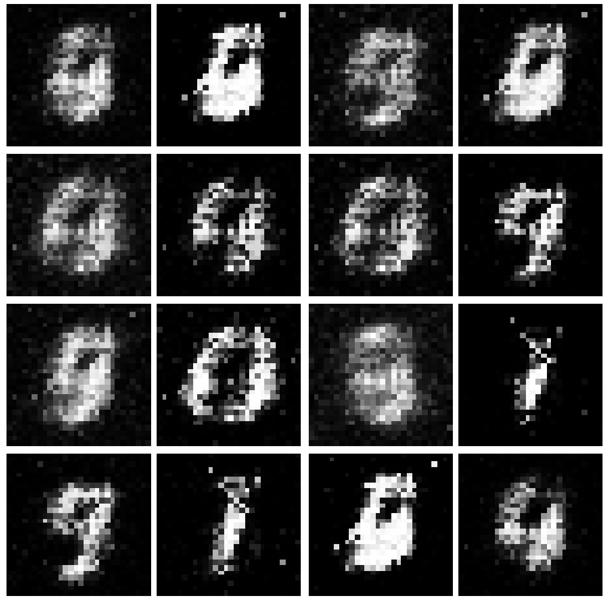
```
Iter: 250
```



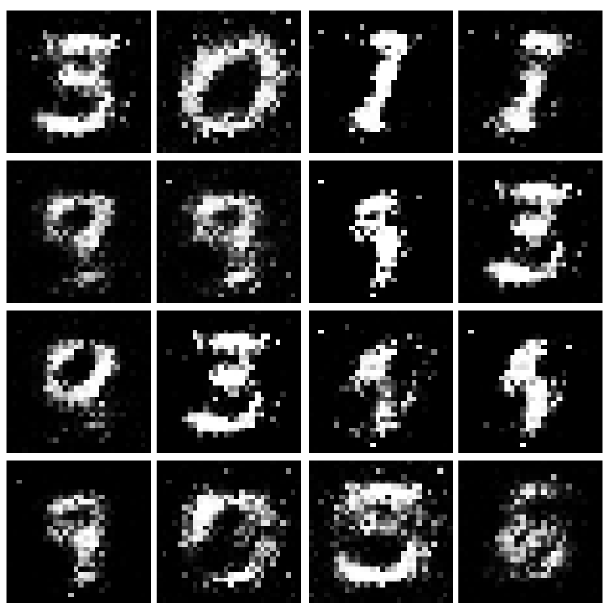
Iter: 500



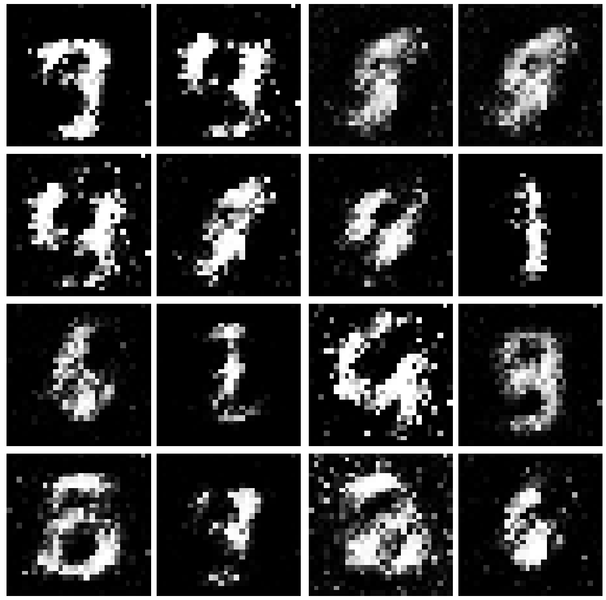
Iter: 750



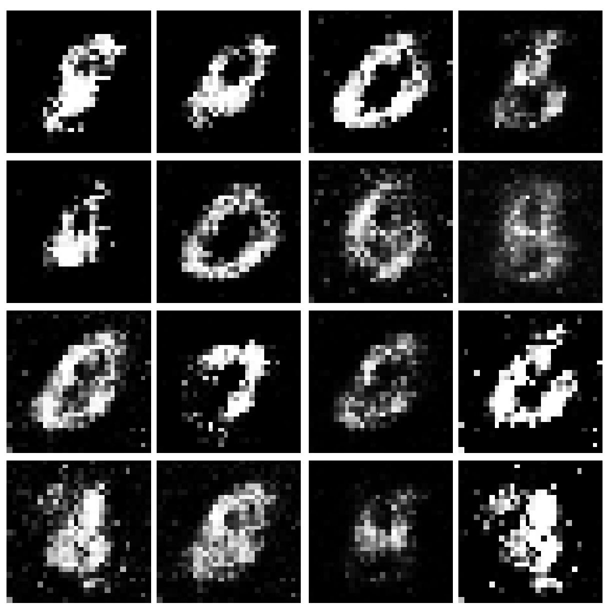
Iter: 1000



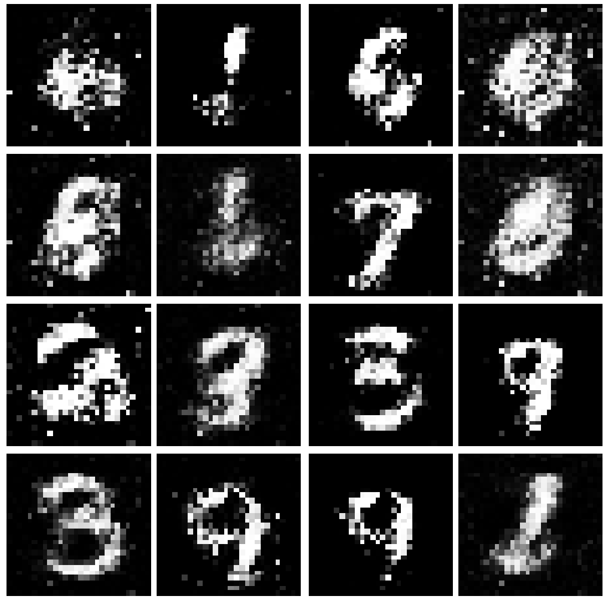
Iter: 1250



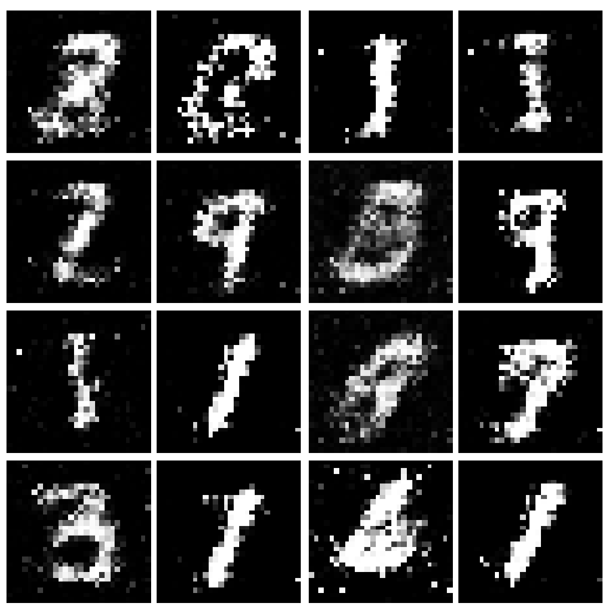
Iter: 1500



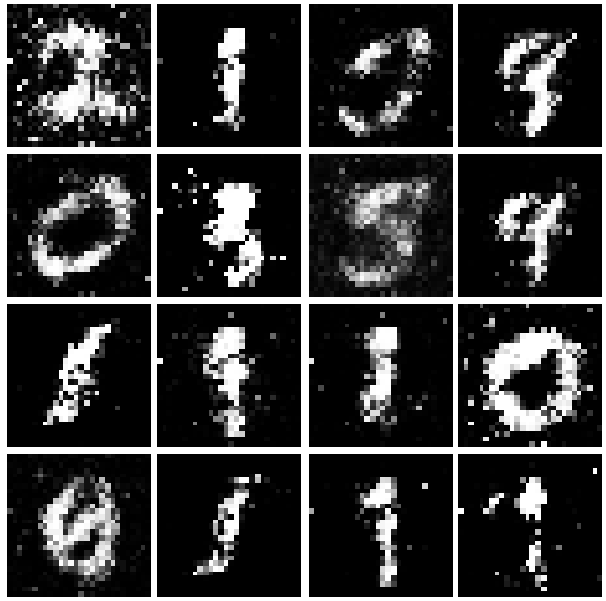
Iter: 1750



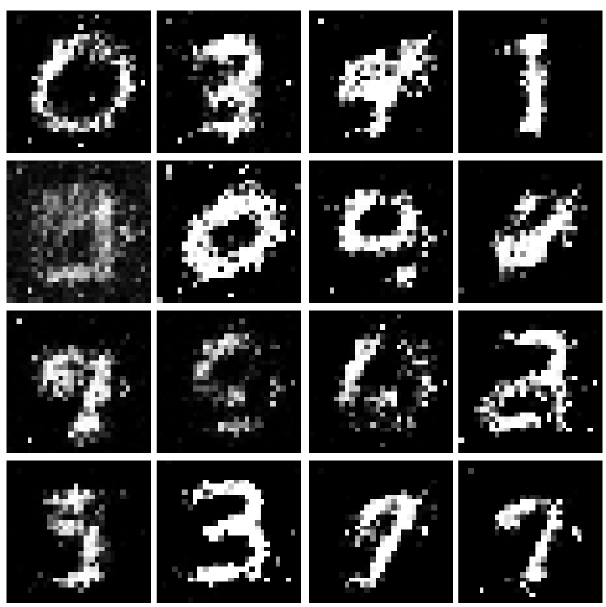
Iter: 2000



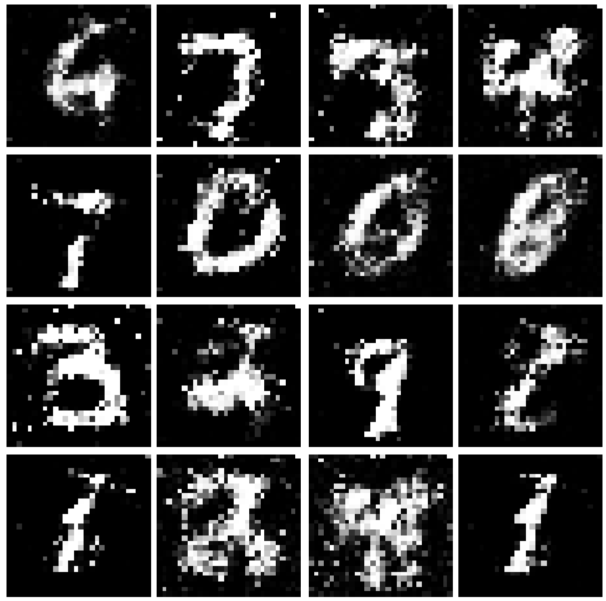
Iter: 2250



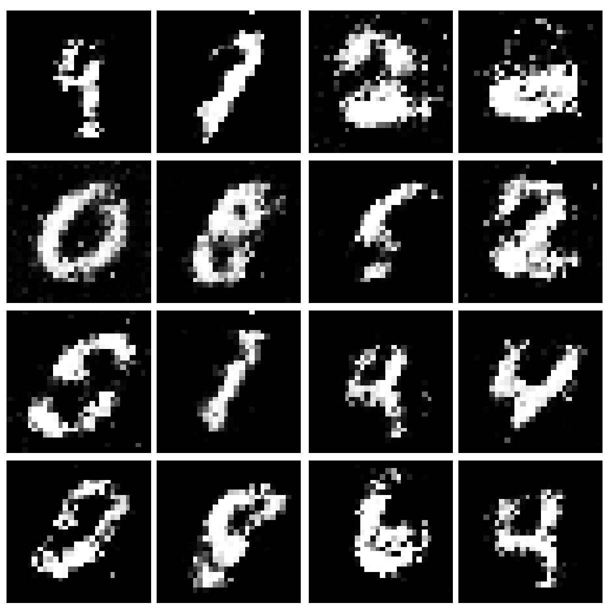
Iter: 2500



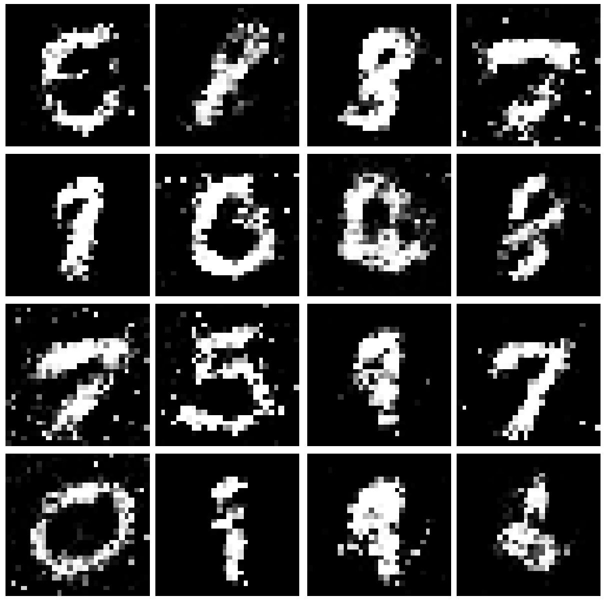
Iter: 2750



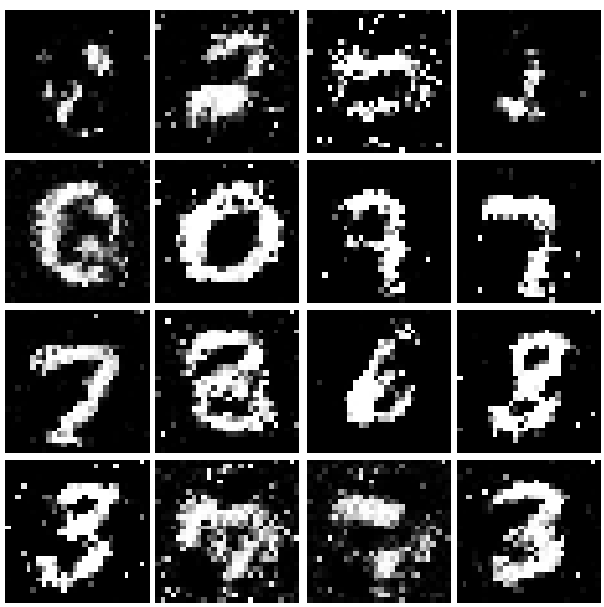
Iter: 3000



Iter: 3250



Iter: 3500



Iter: 3750

6.1 Inline Question 1

What does your final vanilla GAN image look like?

[14]:
```
# This output is your answer.
print("Vanilla GAN final image:")
show_images(images[-1])
plt.show()
```

```
Vanilla GAN final image:
```

Well that wasn't so hard, was it? In the iterations in the low 100s you should see black backgrounds, fuzzy shapes as you approach iteration 1000, and decent shapes, about half of which will be sharp and clearly recognizable as we pass 3000.

7 Least Squares GAN (2 points)

We'll now look at Least Squares GAN, a newer, more stable alernative to the original GAN loss function. For this part, all we have to do is change the loss function and retrain the model. We'll implement equation (9) in the paper, with the generator loss:

$$\ell_G = \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[\left(D(G(z))-1\right)^2\right]$$

and the discriminator loss:

$$\ell_D = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[\left(D(x)-1\right)^2\right] + \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[\left(D(G(z))\right)^2\right]$$

HINTS: Instead of computing the expectation, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing. When plugging in for $D(x)$ and $D(G(z))$ use the direct output from the discriminator (`scores_real` and `scores_fake`). **If** you get errors of 0.333…, you probably forgot to include the 1/2 term.

Implement `ls_discriminator_loss`, `ls_generator_loss` in `gan.py`

Before running a GAN with our new loss function, let's check it:

[15]:
```
from gan import ls_discriminator_loss, ls_generator_loss

def test_lsgan_loss(score_real, score_fake, d_loss_true, g_loss_true):
```

```
    score_real = torch.Tensor(score_real).type(dtype)
    score_fake = torch.Tensor(score_fake).type(dtype)
    d_loss = ls_discriminator_loss(score_real, score_fake).cpu().numpy()
    g_loss = ls_generator_loss(score_fake).cpu().numpy()
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

test_lsgan_loss(
    answers['logits_real'],
    answers['logits_fake'],
    answers['d_loss_lsgan_true'],
    answers['g_loss_lsgan_true']
)
```

```
Maximum error in d_loss: 1.64377e-08
Maximum error in g_loss: 3.36961e-08
```

Run the following cell to train your model! If you train with the CPU, it takes about 7 minutes. If you train with the T4 GPU, it takes about 1 minute and 30 seconds.

[16]:
```
D_LS = discriminator().type(dtype)
G_LS = generator().type(dtype)

D_LS_solver = get_optimizer(D_LS)
G_LS_solver = get_optimizer(G_LS)

images = run_a_gan(
    D_LS,
    G_LS,
    D_LS_solver,
    G_LS_solver,
    ls_discriminator_loss,
    ls_generator_loss,
    loader_train
)
```

```
Iter: 0, D: 0.5689, G:0.51
Iter: 250, D: 0.2989, G:0.09619
Iter: 500, D: 0.1829, G:0.3198
Iter: 750, D: 0.2, G:0.2018
Iter: 1000, D: 0.1175, G:0.3948
Iter: 1250, D: 0.1435, G:0.261
Iter: 1500, D: 0.1948, G:0.2359
Iter: 1750, D: 0.204, G:0.1978
Iter: 2000, D: 0.1945, G:0.2219
Iter: 2250, D: 0.2283, G:0.1662
Iter: 2500, D: 0.226, G:0.2242
Iter: 2750, D: 0.2324, G:0.1521
Iter: 3000, D: 0.2256, G:0.1741
```

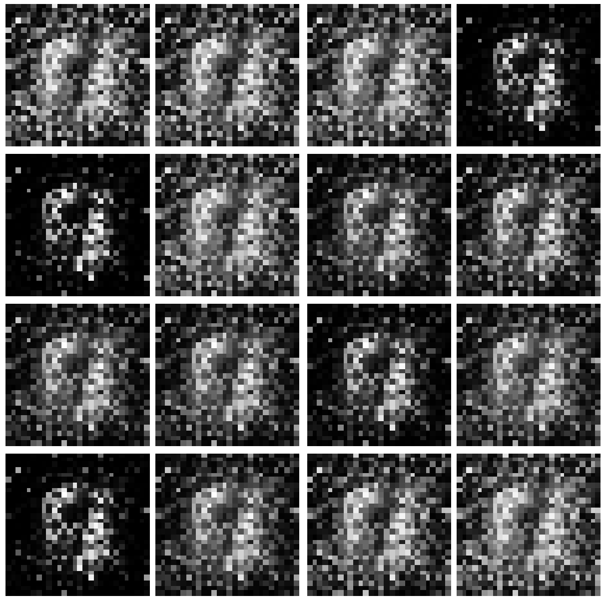
```
Iter: 3250, D: 0.2295, G:0.1645
Iter: 3500, D: 0.2241, G:0.1763
Iter: 3750, D: 0.2408, G:0.1605
```

Run the cell below to show generated images.

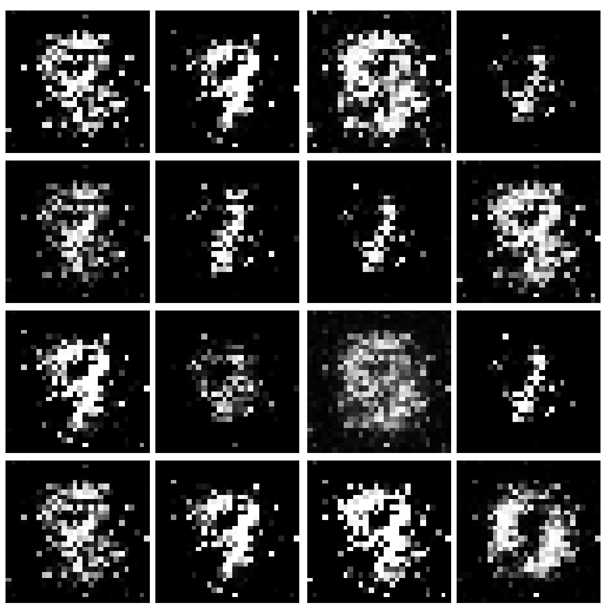
[17]:
```python
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

```
Iter: 0
```

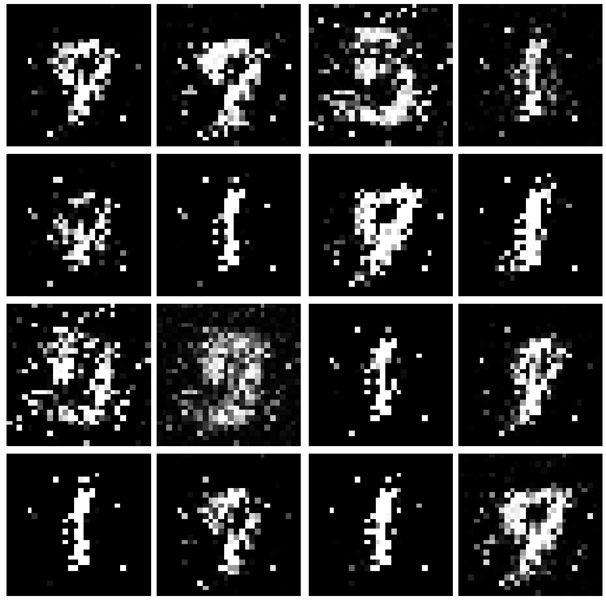
```
Iter: 250
```



Iter: 500

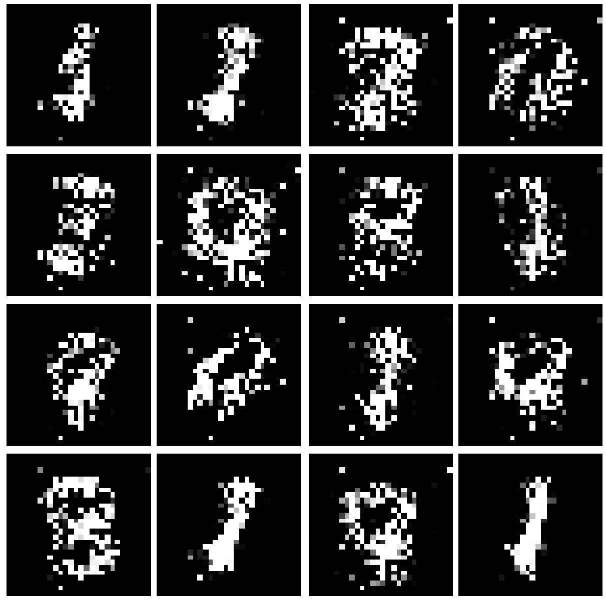


Iter: 750

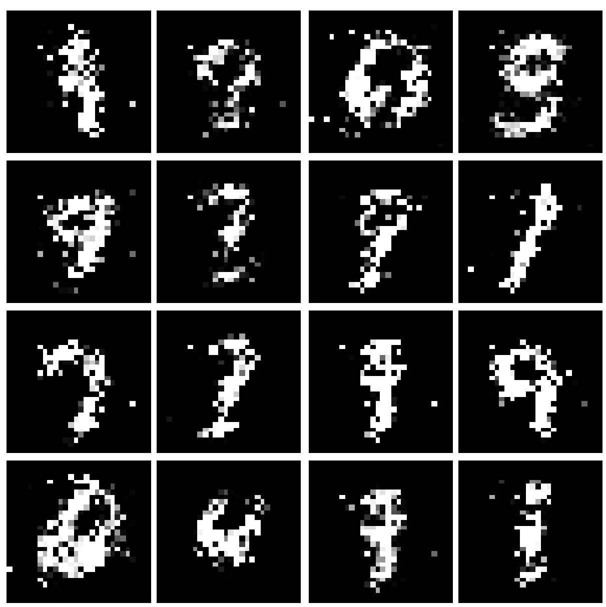


Iter: 1000

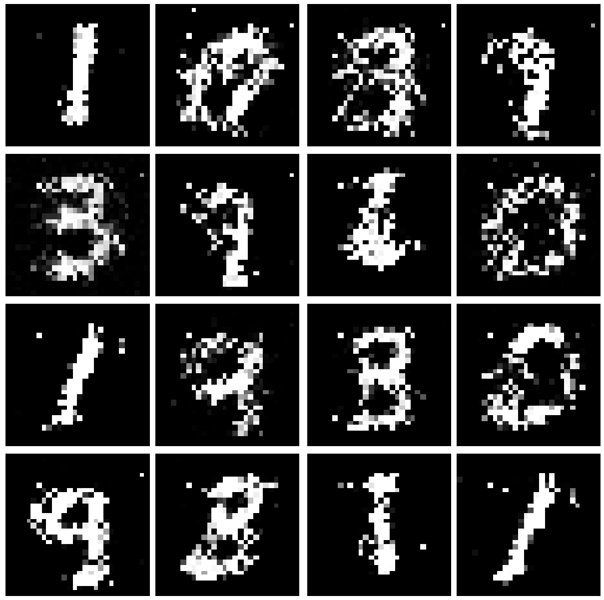
Iter: 1250



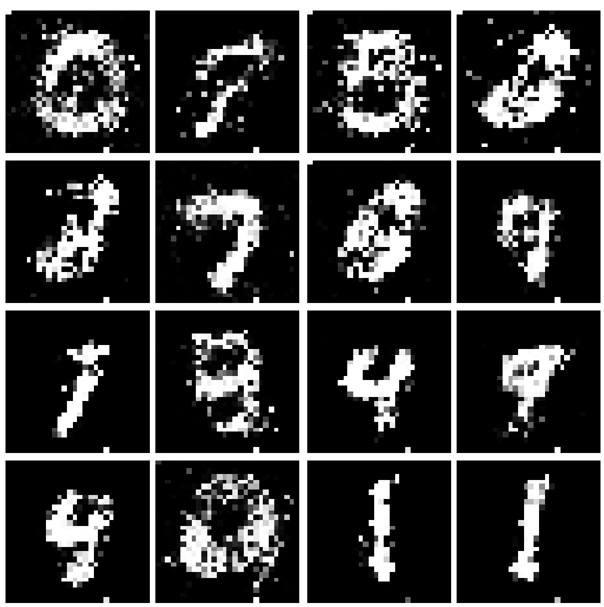
Iter: 1500



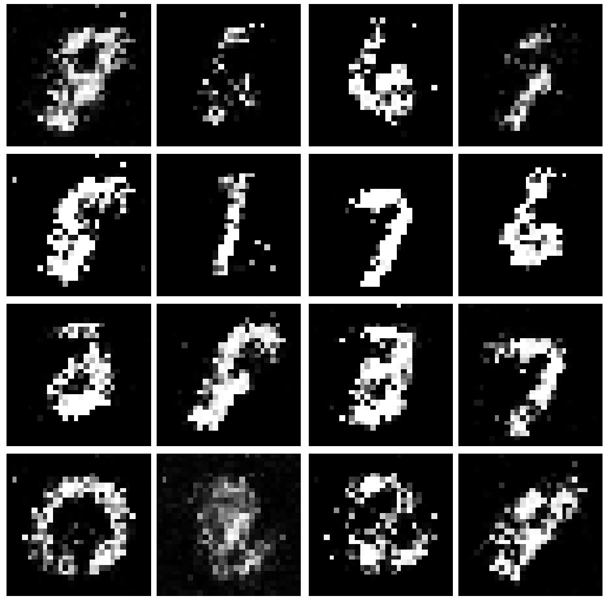
Iter: 1750



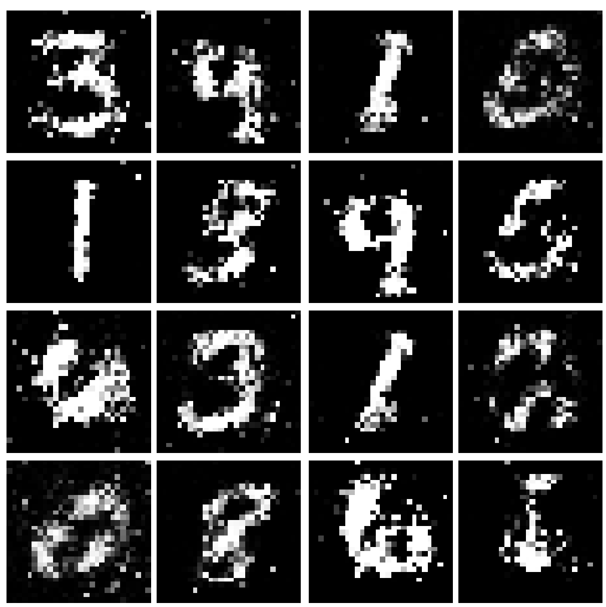
Iter: 2000



Iter: 2250



Iter: 2500



Iter: 2750

Iter: 3000

Iter: 3250

Iter: 3500

Iter: 3750

7.1 Inline Question 2

What does your final LSGAN image look like?

[18]:
```
# This output is your answer.
print("LSGAN final image:")
show_images(images[-1])
plt.show()
```

```
LSGAN final image:
```

8 Deep Convolutional GAN (2 points)

In the first part of the notebook, we implemented an almost direct copy of the original GAN network from Ian Goodfellow. However, this network architecture allows no real spatial reasoning. It is unable to reason about things like "sharp edges" in general because it lacks any convolutional layers. Thus, in this section, we will implement some of the ideas from DCGAN, where we use convolutional networks

Discriminator We will use a discriminator inspired by the TensorFlow MNIST classification tutorial, which is able to get above 99% accuracy on the MNIST dataset fairly quickly. **Hint**: There is no need to specify padding. **Warning**: If you use LazyLinear I will take off points. * Conv2D: 32 Filters, 5x5, Stride 1 * Leaky ReLU(alpha=0.01) * Max Pool 2x2, Stride 2 * Conv2D: 64 Filters, 5x5, Stride 1 * Leaky ReLU(alpha=0.01) * Max Pool 2x2, Stride 2 * Flatten * Fully Connected with output size 4 x 4 x 64 * Leaky ReLU(alpha=0.01) * Fully Connected with output size 1

Implement `build_dc_classifier` in `gan.py`

[19]:
```
from gan import build_dc_classifier

data = next(enumerate(loader_train))[-1][0].type(dtype)
b = build_dc_classifier().type(dtype)
out = b(data)
print(out.size())
```

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

Check the number of parameters in your classifier as a sanity check:

[20]:
```
def test_dc_classifer(true_count=1102721):
    model = build_dc_classifier()
    cur_count = count_params(model)
    if cur_count != true_count:
        print('Incorrect number of parameters in classifier. Check your␣
 ↪achitecture.')
    else:
        print('Correct number of parameters in classifier.')

test_dc_classifer()
```

```
Correct number of parameters in classifier.
```

Generator For the generator, we will copy the architecture exactly from the InfoGAN paper. See Appendix C.1 MNIST. See the documentation for nn.ConvTranspose2d. We are always "training" in GAN mode. * Fully connected with output size 1024 * `ReLU` * BatchNorm * Fully connected with output size 7 x 7 x 128 * ReLU * BatchNorm * Use `Unflatten()` to reshape into Image Tensor of shape 7, 7, 128 * `ConvTranspose2d`: 64 filters of 4x4, stride 2, 'same' padding (use `padding=1`) * `ReLU` * BatchNorm * `ConvTranspose2d`: 1 filter of 4x4, stride 2, 'same' padding (use `padding=1`) * `TanH` * Should have a 28x28x1 image, reshape back into 784 vector (using `Flatten()`)

Implement `build_dc_generator` in `gan.py`

[21]:
```
from gan import build_dc_generator

test_g_gan = build_dc_generator().type(dtype)
test_g_gan.apply(initialize_weights)

fake_seed = torch.randn(batch_size, NOISE_DIM).type(dtype)
fake_images = test_g_gan.forward(fake_seed)
fake_images.size()
```

[21]:
```
torch.Size([128, 784])
```

Check the number of parameters in your generator as a sanity check:

[22]:
```
def test_dc_generator(true_count=6580801):
    model = build_dc_generator(4)
    cur_count = count_params(model)
    if cur_count != true_count:
        print('Incorrect number of parameters in generator. Check your␣
 ↪achitecture.')
    else:
        print('Correct number of parameters in generator.')

test_dc_generator()
```

```
Correct number of parameters in generator.
```

Run the following cell to train your DCGAN. If you train with the CPU, it takes about 35 minutes. If you train with the T4 GPU, it takes about 1 minute.

[23]:
```
D_DC = build_dc_classifier().type(dtype)
D_DC.apply(initialize_weights)
G_DC = build_dc_generator().type(dtype)
G_DC.apply(initialize_weights)

D_DC_solver = get_optimizer(D_DC)
G_DC_solver = get_optimizer(G_DC)

images = run_a_gan(
    D_DC,
    G_DC,
    D_DC_solver,
    G_DC_solver,
    discriminator_loss,
    generator_loss,
    loader_train,
    num_epochs=5
)
```

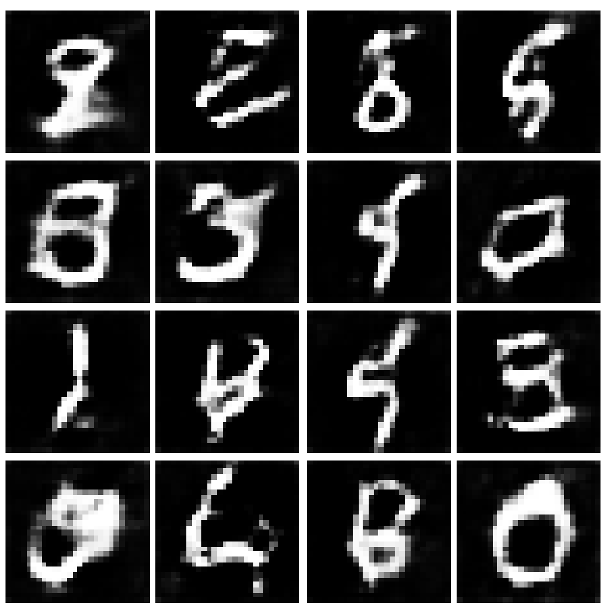
```
Iter: 0, D: 1.4, G:2.369
Iter: 250, D: 1.373, G:0.8342
Iter: 500, D: 1.242, G:0.9312
Iter: 750, D: 1.399, G:0.6719
Iter: 1000, D: 1.284, G:0.9758
Iter: 1250, D: 1.313, G:0.9711
Iter: 1500, D: 1.067, G:1.111
Iter: 1750, D: 1.08, G:1.144
```

Run the cell below to show generated images.

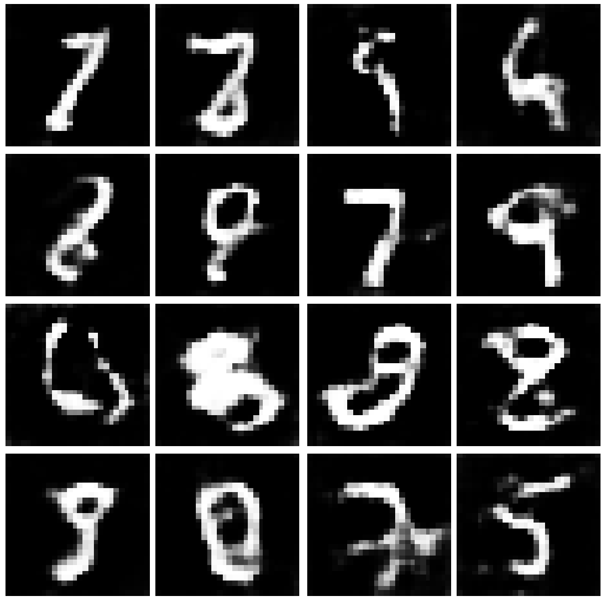
[24]:
```
numIter = 0
for img in images:
    print("Iter: {}".format(numIter))
    show_images(img)
    plt.show()
    numIter += 250
    print()
```

```
Iter: 0
```

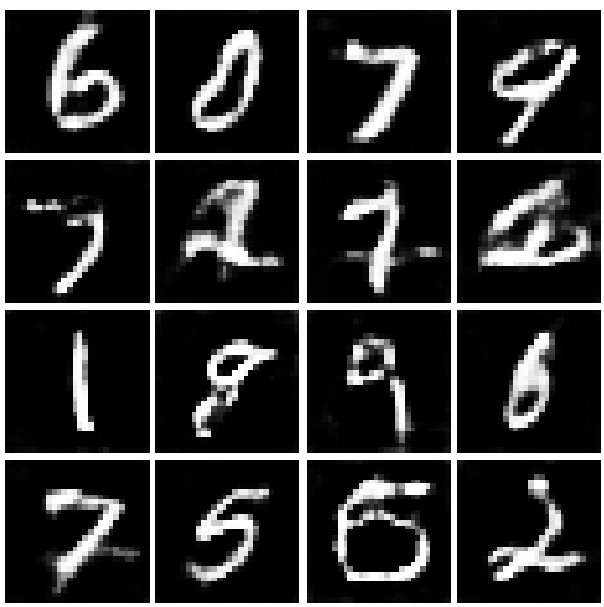
Iter: 250



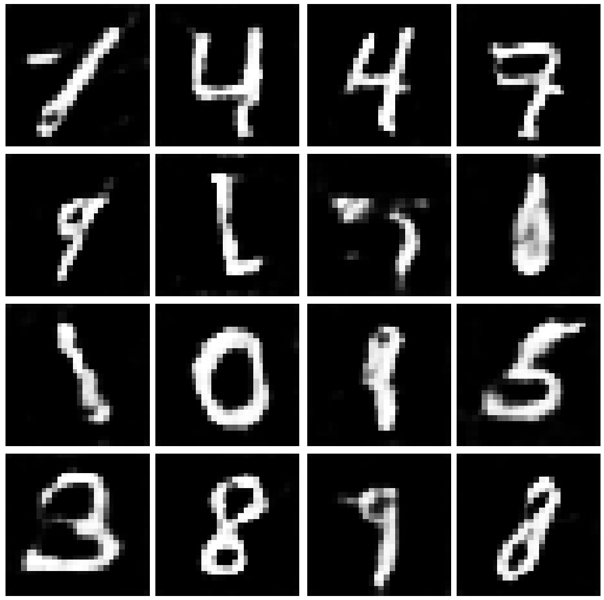
Iter: 500



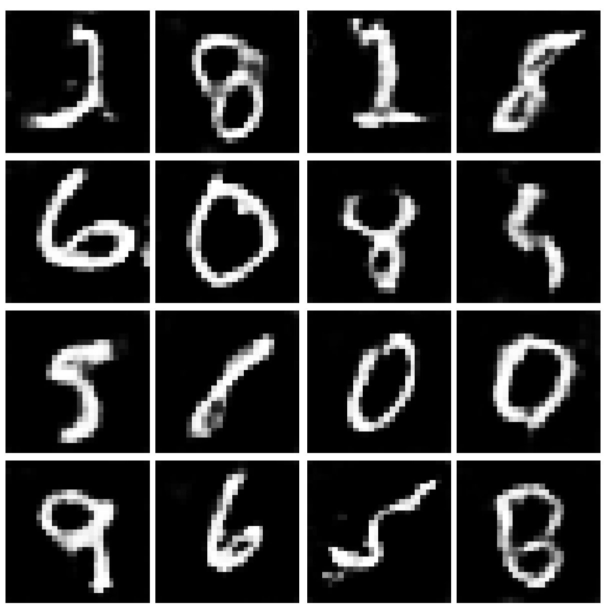
Iter: 750



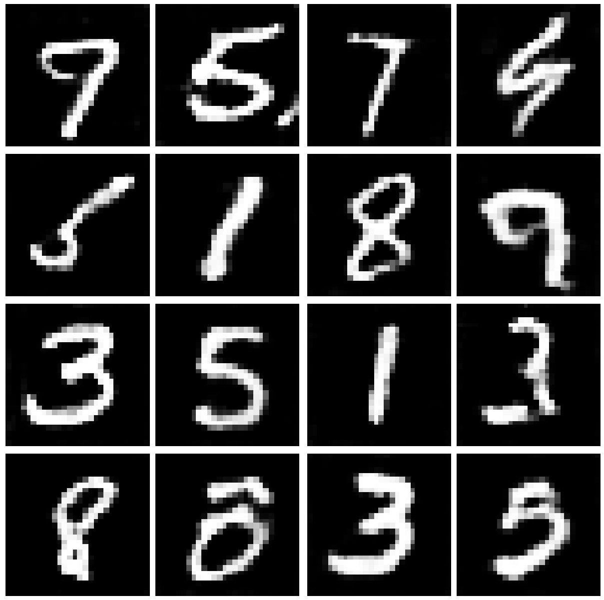
Iter: 1000



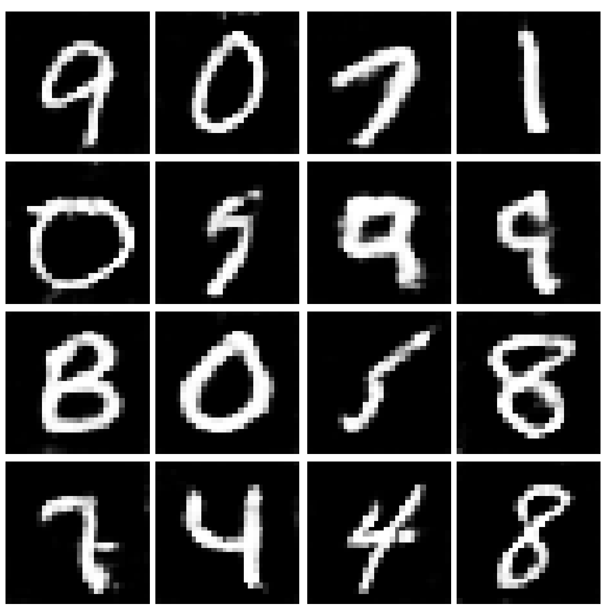
Iter: 1250



Iter: 1500



Iter: 1750

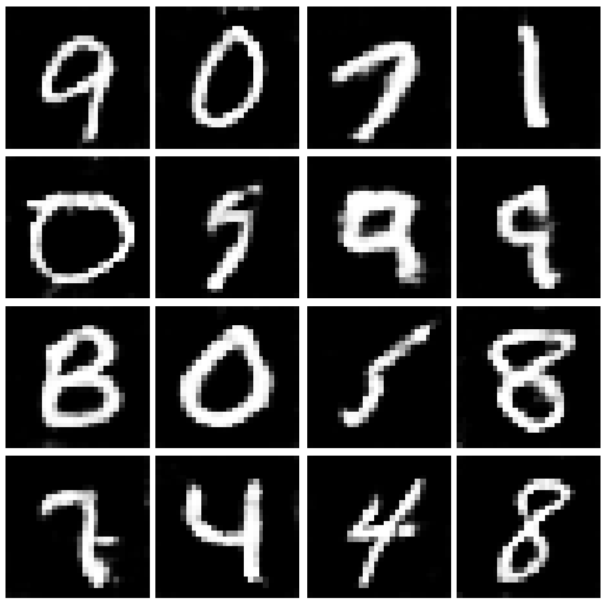


8.1 Inline Question 3

What does your final DCGAN image look like?

[25]:
```
# This output is your answer.
print("DCGAN final image:")
show_images(images[-1])
plt.show()
```

DCGAN final image:



8.2 Inline Question 4 (1 point)

We will look at an example to see why alternating minimization of the same objective (like in a GAN) can be tricky business.

Consider $f(x, y) = xy$. What does $\min_x \max_y f(x, y)$ evaluate to? (Hint: minmax tries to minimize the maximum value achievable.)

Now try to evaluate this function numerically for 6 steps, starting at the point $(1, 1)$, by using alternating gradient (first updating y, then updating x using that updated y) with step size 1. **Here step size is the learning_rate, and steps will be learning_rate * gradient.** You'll find that writing out the update step in terms of $x_t, y_t, x_{t+1}, y_{t+1}$ will be useful.

Record the six pairs of explicit values for (x_t, y_t) in the table below and briefly explain what $f(x, y)$ evaluates to.

8.2.1 Your answer:

We have $\min_x \max_y f(x, y) = 0$. Note that for any nonzero x, the max attainable value of f is unbounded.

y_0	y_1	y_2	y_3	y_4	y_5	y_6
1	2	1	-1	-2	-1	1
x_0	x_1	x_2	x_3	x_4	x_5	x_6
1	-1	-2	-1	1	2	1
$f(x_0, y_0)$	$f(x_1, y_1)$	$f(x_2, y_2)$	$f(x_3, y_3)$	$f(x_4, y_4)$	$f(x_5, y_5)$	$f(x_6, y_6)$
1	-2	-2	1	-2	-2	1

8.3 Inline Question 5 (1 point)

Using this method, will we ever reach the optimal value? Why or why not?

8.3.1 Your answer:

We will not. We can see that the sequence hits a fixed point every 6 steps, and so will fail to converge to the optimal value (at which $x = 0$).

8.4 Inline Question 6 (1 point)

If the generator loss decreases during training while the discriminator loss stays at a constant high value from the start, is this a good sign? Why or why not? A qualitative answer is sufficient.

8.4.1 Your answer:

This is not a good sign. If the discriminator loss is high from the beginning, then it is not providing a high-quality feedback signal to the generator, and so the generator may be able to decrease its loss without learning to generate realistic images.

projects/project_1/gan/gan.py

```python
import numpy as np

import torch
import torch.nn as nn
from torch.utils.data import sampler


NOISE_DIM = 96

dtype = torch.cuda.FloatTensor if torch.cuda.is_available() else torch.FloatTensor


def sample_noise(batch_size, dim, seed=None):
    """
    Generate a PyTorch Tensor of uniform random noise.
    Input:
    - batch_size: Integer giving the batch size of noise to generate.
    - dim: Integer giving the dimension of noise to generate.
    Output:
    - A PyTorch Tensor of shape (batch_size, dim) containing uniform
      random noise in the range (-1, 1).
    """
    if seed is not None:
        torch.manual_seed(seed)

    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return torch.FloatTensor(batch_size, dim).uniform_(-1, 1)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****


def discriminator(seed=None):
    """
    Build and return a PyTorch model implementing the architecture above.
    """

    if seed is not None:
        torch.manual_seed(seed)

    model = None

    ##############################################################################
    # TODO: Implement architecture                                               #
    #                                                                            #
    # HINT: nn.Sequential might be helpful. You'll start by calling Flatten().   #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```python
    model = nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, 256),
        nn.LeakyReLU(negative_slope=0.01),
        nn.Linear(256, 256),
        nn.LeakyReLU(negative_slope=0.01),
        nn.Linear(256, 1),
    )
    pass

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return model


def generator(noise_dim=NOISE_DIM, seed=None):
    """
    Build and return a PyTorch model implementing the architecture above.
    """

    if seed is not None:
        torch.manual_seed(seed)

    model = None

    ##############################################################################
    # TODO: Implement architecture                                               #
    #                                                                            #
    # HINT: nn.Sequential might be helpful.                                      #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    model = nn.Sequential(
        nn.Linear(noise_dim, 1024),
        nn.ReLU(),
        nn.Linear(1024, 1024),
        nn.ReLU(),
        nn.Linear(1024, 784),
        nn.Tanh(),
    )

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return model
```

```
def bce_loss(input, target):
    """
    Numerically stable version of the binary cross-entropy loss function in PyTorch.
    Inputs:
    - input: PyTorch Tensor of shape (N, ) giving scores.
    - target: PyTorch Tensor of shape (N,) containing 0 and 1 giving targets.
          dtype is float! (a global dtype is defined above).
    Returns:
    - A PyTorch Tensor containing the mean BCE loss over the minibatch of input data.
    """
    bce = nn.BCEWithLogitsLoss()
    return bce(input, target)


def discriminator_loss(logits_real, logits_fake):
    """
    Computes the discriminator loss described above.
    Inputs:
    - logits_real: PyTorch Tensor of shape (N,) giving scores for the real data.
    - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.
    Returns:
    - loss: PyTorch Tensor containing (scalar) the loss for the discriminator.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    true_labels = torch.ones_like(logits_real)  # .type(dtype)
    real_loss = bce_loss(logits_real, true_labels)

    fake_labels = torch.zeros_like(logits_fake)  # .type(dtype)
    fake_loss = bce_loss(logits_fake, fake_labels)

    loss = real_loss + fake_loss

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss


def generator_loss(logits_fake):
    """
    Computes the generator loss described above.
    Inputs:
    - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.
    Returns:
    - loss: PyTorch Tensor containing the (scalar) loss for the generator.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
    targets = torch.ones_like(logits_fake).type(dtype)
    loss = bce_loss(logits_fake, targets)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss


def get_optimizer(model):
    """
    Construct and return an Adam optimizer for the model with learning rate 1e-3,
    beta1=0.5, and beta2=0.999.
    Input:
    - model: A PyTorch model that we want to optimize.
    Returns:
    - An Adam optimizer for the model with the desired hyperparameters.
    """
    optimizer = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    optimizer = torch.optim.Adam(params=model.parameters(), lr=1e-3, betas=(0.5, 0.999))

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return optimizer


def ls_discriminator_loss(scores_real, scores_fake):
    """
    Compute the Least-Squares GAN loss for the discriminator.
    Inputs:
    - scores_real: PyTorch Tensor of shape (N,) giving scores for the real data.
    - scores_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.
    Outputs:
    - loss: A PyTorch Tensor containing the loss.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    real_loss = 0.5 * torch.mean(torch.square(scores_real - 1))
    fake_loss = 0.5 * torch.mean(torch.square(scores_fake))
    loss = real_loss + fake_loss

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss


def ls_generator_loss(scores_fake):
    """
    Computes the Least-Squares GAN loss for the generator.
    Inputs:
    - scores_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.
```

```
    Outputs:
    - loss: A PyTorch Tensor containing the loss.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    loss = 0.5 * torch.mean(torch.square(scores_fake - 1))

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss


def build_dc_classifier():
    """
    Build and return a PyTorch model for the DCGAN discriminator implementing
    the architecture above.
    """

    ##############################################################################
    # TODO: Implement architecture                                               #
    #                                                                            #
    # HINT: nn.Sequential might be helpful.                                      #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    model = nn.Sequential(
        nn.Conv2d(in_channels=1, out_channels=32, kernel_size=5, stride=1),
        nn.LeakyReLU(negative_slope=0.01),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, stride=1),
        nn.LeakyReLU(negative_slope=0.01),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.Linear(1024, 1024),
        nn.LeakyReLU(negative_slope=0.01),
        nn.Linear(1024, 1),
    )

    return model

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################


def build_dc_generator(noise_dim=NOISE_DIM):
    """
    Build and return a PyTorch model implementing the DCGAN generator using
    the architecture described above.
```

```
    """

    ##############################################################################
    # TODO: Implement architecture                                               #
    #                                                                            #
    # HINT: nn.Sequential might be helpful.                                      #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    model = nn.Sequential(
        nn.Linear(noise_dim, 1024),
        nn.ReLU(),
        nn.BatchNorm1d(num_features=1024),
        nn.Linear(1024, 6272),
        nn.ReLU(),
        nn.BatchNorm1d(num_features=6272),
        nn.Unflatten(1, (128, 7, 7)),
        nn.ConvTranspose2d(
            in_channels=128,
            out_channels=64,
            kernel_size=4,
            stride=2,
            padding=1,
        ),
        nn.ReLU(),
        nn.BatchNorm2d(num_features=64),
        nn.ConvTranspose2d(
            in_channels=64,
            out_channels=1,
            kernel_size=4,
            stride=2,
            padding=1,
        ),
        nn.Tanh(),
        nn.Flatten(),
    )

    return model

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################


def run_a_gan(
    D,
    G,
    D_solver,
    G_solver,
```

```
        discriminator_loss,
        generator_loss,
        loader_train,
        show_every=250,
        batch_size=128,
        noise_size=96,
        num_epochs=10,
    ):
        """
        Train a GAN!
        Inputs:
        - D, G: PyTorch models for the discriminator and generator
        - D_solver, G_solver: torch.optim Optimizers to use for training the
          discriminator and generator.
        - discriminator_loss, generator_loss: Functions to use for computing the generator
    and
          discriminator loss, respectively.
        - show_every: Show samples after every show_every iterations.
        - batch_size: Batch size to use for training.
        - noise_size: Dimension of the noise to use as input to the generator.
        - num_epochs: Number of epochs over the training dataset to use for training.
        """
        images = []
        iter_count = 0
        for epoch in range(num_epochs):
            for x, _ in loader_train:
                if len(x) != batch_size:
                    continue
                D_solver.zero_grad()
                real_data = x.type(dtype)
                logits_real = D(2 * (real_data - 0.5)).type(dtype)

                g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
                fake_images = G(g_fake_seed).detach()
                logits_fake = D(fake_images.view(batch_size, 1, 28, 28))

                d_total_error = discriminator_loss(logits_real, logits_fake)
                d_total_error.backward()
                D_solver.step()

                G_solver.zero_grad()
                g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
                fake_images = G(g_fake_seed)

                gen_logits_fake = D(fake_images.view(batch_size, 1, 28, 28))
                g_error = generator_loss(gen_logits_fake)
                g_error.backward()
                G_solver.step()

                if iter_count % show_every == 0:
```

```
                print(
                    "Iter: {}, D: {:.4}, G:{:.4}".format(
                        iter_count, d_total_error.item(), g_error.item()
                    )
                )
                imgs_numpy = fake_images.data.cpu().numpy()
                images.append(imgs_numpy[0:16])

            iter_count += 1

    return images


class ChunkSampler(sampler.Sampler):
    """Samples elements sequentially from some offset.
    Arguments:
        num_samples: # of desired datapoints
        start: offset where we should start selecting from
    """

    def __init__(self, num_samples, start=0):
        self.num_samples = num_samples
        self.start = start

    def __iter__(self):
        return iter(range(self.start, self.start + self.num_samples))

    def __len__(self):
        return self.num_samples


class Flatten(nn.Module):
    def forward(self, x):
        N, C, H, W = x.size()  # read in N, C, H, W
        return x.view(
            N, -1
        )  # "flatten" the C * H * W values into a single vector per image


class Unflatten(nn.Module):
    """
    An Unflatten module receives an input of shape (N, C*H*W) and reshapes it
    to produce an output of shape (N, C, H, W).
    """

    def __init__(self, N=-1, C=128, H=7, W=7):
        super(Unflatten, self).__init__()
        self.N = N
        self.C = C
        self.H = H
```

```
        self.W = W

    def forward(self, x):
        return x.view(self.N, self.C, self.H, self.W)


def initialize_weights(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.ConvTranspose2d):
        nn.init.xavier_uniform_(m.weight.data)


def preprocess_img(x):
    return 2 * x - 1.0


def deprocess_img(x):
    return (x + 1.0) / 2.0


def rel_error(x, y):
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))


def count_params(model):
    """Count the number of parameters in the model."""
    param_count = np.sum([np.prod(p.size()) for p in model.parameters()])
    return param_count
```

vae

April 20, 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.

[1]:
```
%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!).

[2]:
```
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 folowing cell should print the filenames from the assignment:

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

[ ]:
```python
import os

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!
```

[4]:
```python
import sys
if USE_COLAB:
    sys.path.append(GOOGLE_DRIVE_PATH)

from vae import hello_vae
hello_vae()

from nndl2.helper import hello_helper
hello_helper()
```

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

Load several useful packages that are used in this notebook:

[5]:
```python
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
```

[6]:
```
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:

[6]:
```
if torch.cuda.is_available():
    print('Good to go!')
else:
    print('Please set GPU via the downward triangle in the top right corner.')
```

```
Please set GPU via the downward triangle in the top right corner.
```

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.

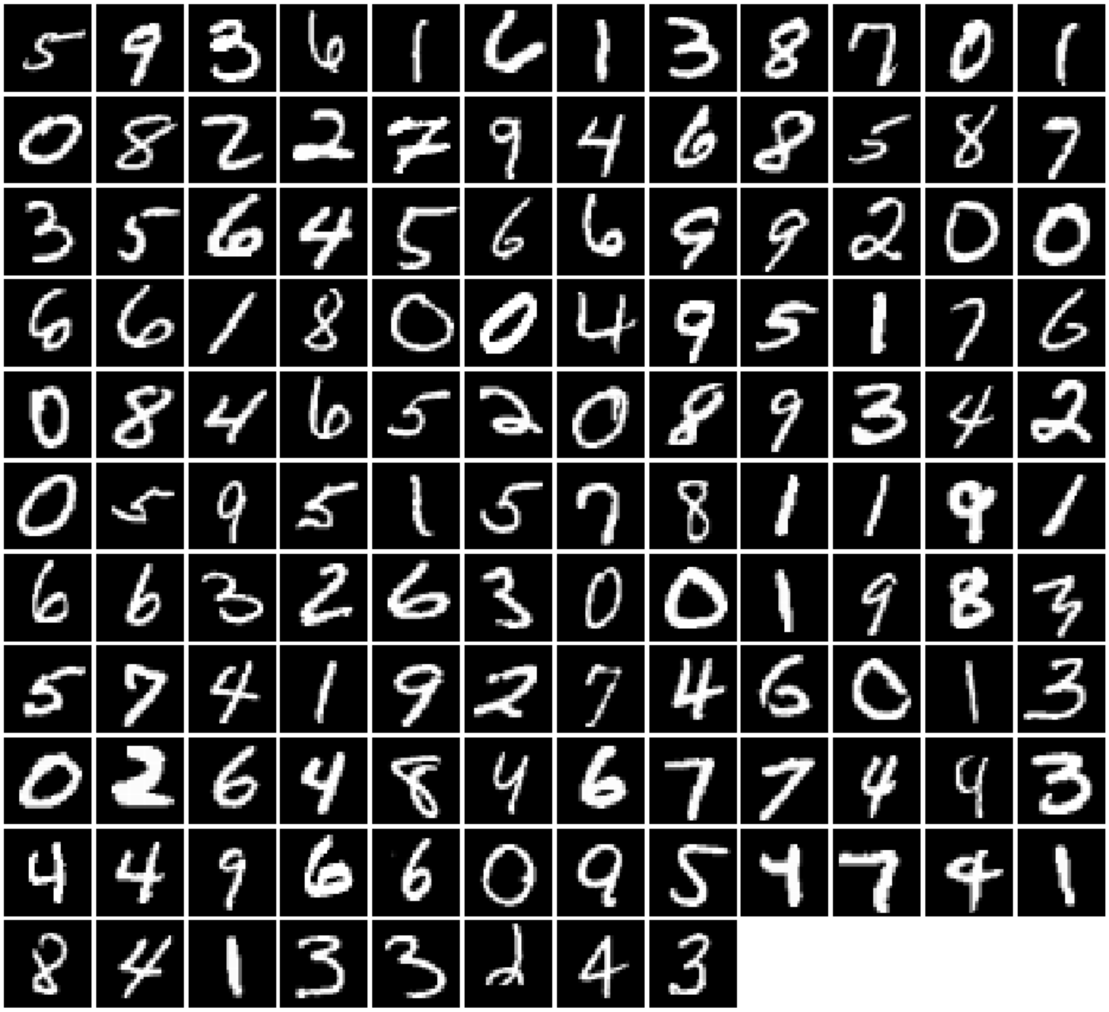
To simplify our code here, we will use the PyTorch MNIST wrapper, which downloads and loads the MNIST dataset. See the documentation for more information about the interface. The default parameters will take 5,000 of the training examples and place them into a validation dataset. The data will be saved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

[7]:
```
if USE_COLAB:
    %cd /content/drive/My\ Drive/$GOOGLE_DRIVE_PATH_AFTER_MYDRIVE

batch_size = 128

mnist_train = dset.MNIST('./nndl2', train=True, download=True,
                           transform=T.ToTensor())
loader_train = DataLoader(mnist_train, batch_size=batch_size,
                          shuffle=True, drop_last=True, num_workers=2)
```

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 `show_images` in `helper.py` that we call to visualize the images.

[8]:
```python
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 x 28 x 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`
- `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`

[9]:
```
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`***

[10]:
```
from vae import VAE
def count_params(model):
    return sum(p.numel() for p in model.parameters())
def test_decoder(model, input_size, hidden_dim, latent_size,␣
 ↪n_decoder_lin_layers):
    '''
    model: Model defined as above
    input_size: dimensionality of input
    hidden_dim: dimensionality of hidden state
    latent_size: dimensionality of latent space
    n_layers: number of Linear layers in model.decoder
    '''
    expected_n_params = (latent_size+1)*hidden_dim + \
        (n_decoder_lin_layers-2)*(hidden_dim+1)*hidden_dim + \
        (hidden_dim+1)*input_size
    actual_n_params = count_params(model.decoder)
```

```python
    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 distrubtion, and compute z as a function of (ϵ, σ^2, μ). 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`.

[11]:
```python
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 5.621977930696792e-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.

[12]:

```python
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,␣
 ↪28, 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,␣
 ↪latent_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()
```

```
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).

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

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`

[13]:
```
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.

[15]:
```
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)
```

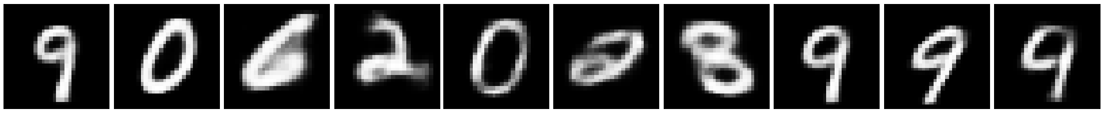
```
Warning: using device cpu may take longer.
Train Epoch: 0  Loss: 163.394287
Train Epoch: 1  Loss: 134.598206
Train Epoch: 2  Loss: 130.691803
Train Epoch: 3  Loss: 129.569839
Train Epoch: 4  Loss: 118.755501
Train Epoch: 5  Loss: 120.766190
Train Epoch: 6  Loss: 119.026901
Train Epoch: 7  Loss: 118.184242
Train Epoch: 8  Loss: 117.046165
Train Epoch: 9  Loss: 114.382652
```

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.

[16]:
```
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')
```

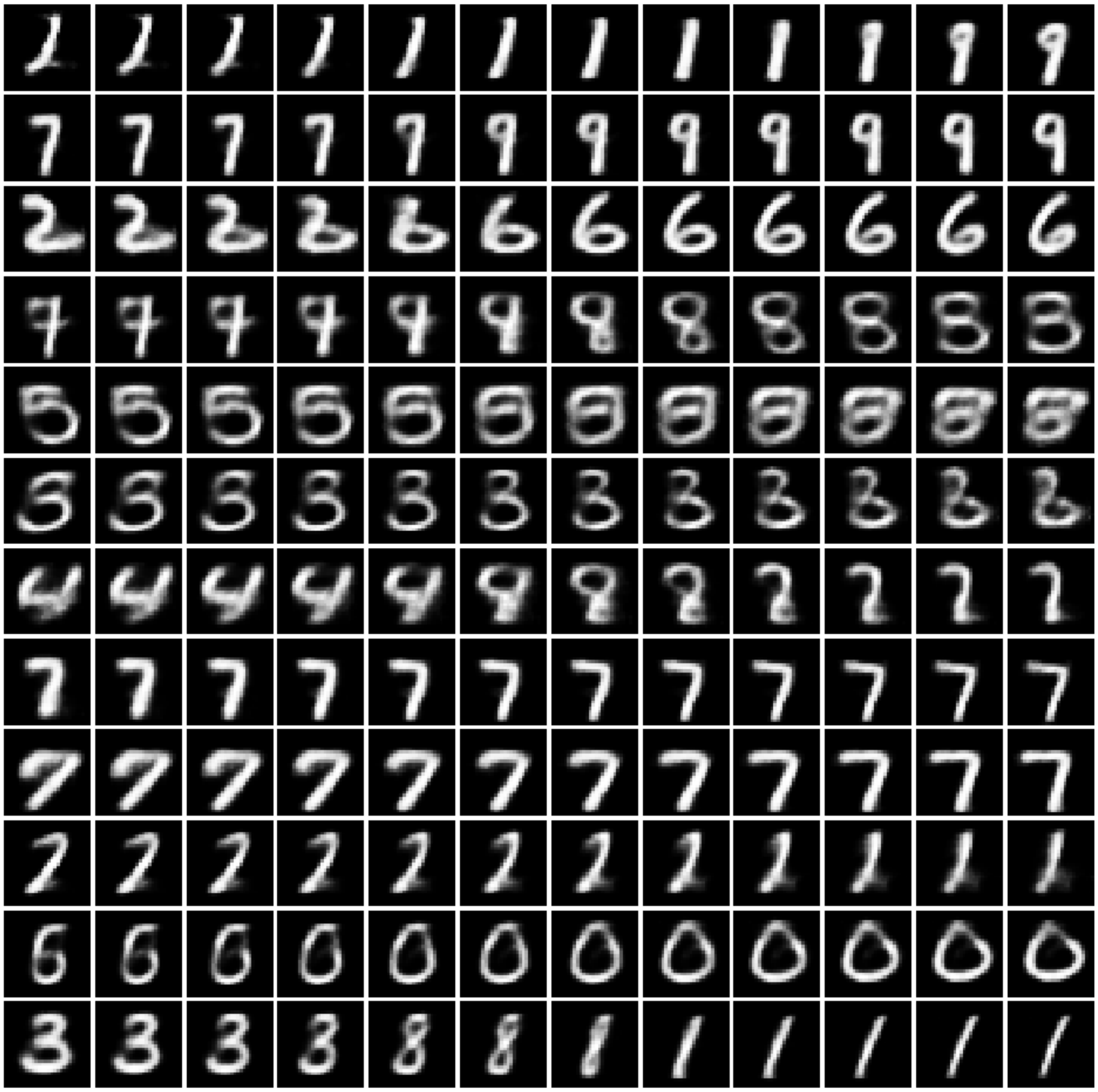


2.8 Latent Space Interpolation

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.

[17]:
```
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`).

[18]:
```
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,␣
 ↪num_classes=num_classes,latent_size=latent_size)(
            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,␣
 ↪latent_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.

[19]:
```
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)
```

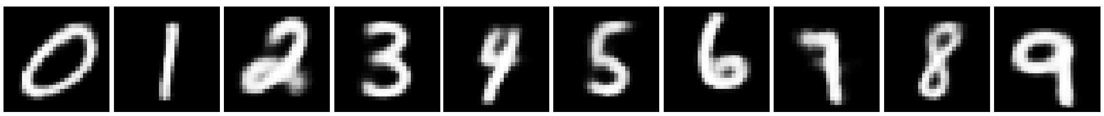
```
Warning: using device cpu may take longer.
Train Epoch: 0  Loss: 136.998932
Train Epoch: 1  Loss: 127.882439
Train Epoch: 2  Loss: 122.547318
Train Epoch: 3  Loss: 118.689209
Train Epoch: 4  Loss: 115.164726
Train Epoch: 5  Loss: 118.835289
Train Epoch: 6  Loss: 110.265244
Train Epoch: 7  Loss: 114.702408
Train Epoch: 8  Loss: 114.393059
Train Epoch: 9  Loss: 108.250839
```

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).

[21]:
```
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)
```

```python
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')
```



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
```