VAE

May 1, 2025

1 VAE

Variational autoencoder [1] models inherit autoencoder architecture, but use variational approach for latent representation learning. In this homework, we will implement VAE and quantitatively measure the quality of the generated samples via Inception score [2,3].

[1] Auto-Encoding Variational Bayes, Diederik P Kingma, Max Welling 2013 https://arxiv.org/abs/1312.6114

[2] Improved techniques for training gans, Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Rad- ford, A., and Chen, X. 2016 In Advances in Neural Information Processing Systems

[3] A note on inception score, Shane Barratt, Rishi Sharma 2018 https://arxiv.org/abs/1801.01973

2 PART I. Train a good VAE model

2.1 Setup

[244]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import datasets, transforms

import numpy as np
import os

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

%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'

# A bunch of utility functions

def show_images(images):
    images = images.view(images.shape[0], -1).detach().cpu().numpy()
```

```
    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]))
    plt.show()

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):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

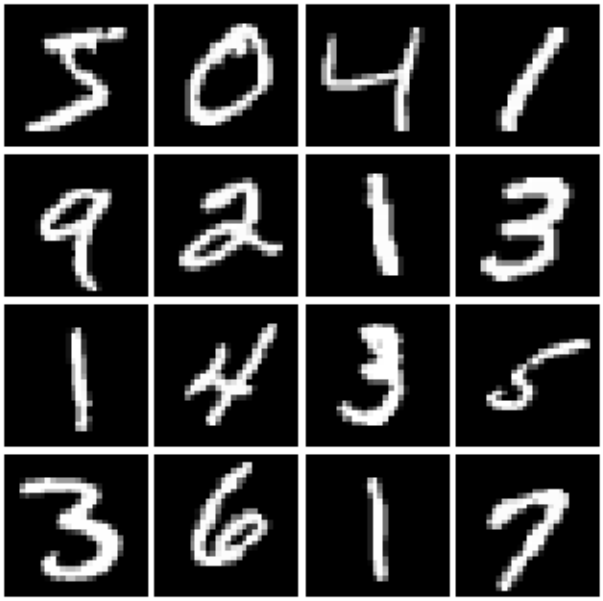
2.2 Dataset

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.

Heads-up: Our MNIST wrapper returns images as vectors. That is, they're size (batch, 784). If you want to treat them as images, we have to resize them to (batch,28,28) or (batch,28,28,1). They are also type np.float32 and bounded [0,1].

[245]:
```
transform = transforms.Compose([
    transforms.ToTensor(),                    # [0,1]
    # transforms.Lambda(lambda x: preprocess_img(x))  # [-1,1]
])

mnist_dataset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
batch_size = 16
mnist_loader = DataLoader(mnist_dataset, batch_size=batch_size, shuffle=False)
```

[246]:
```
# Show a batch
data_iter = iter(mnist_loader)
images, labels = next(data_iter)
show_images(images)
```



[247]:
```
X_DIM = images[0].numel()
num_samples = 100000
num_to_show = 100

# Hyperparamters. Your job is to find these.
# TODO:
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
num_epochs = 20
batch_size = 128
Z_DIM = 20
learning_rate = 1e-3
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

2.3 Encoder

Our first step is to build a variational encoder network $q_\phi(z \mid x)$.

Hint: Use four Linear layers.

The encoder should return two tensors of shape `[batch_size, z_dim]`, which corresponds to the mean $\mu(x_i)$ and diagonal log variance $\log \sigma(x_i)^2$ of each of the `batch_size` input images. Note, we want to make it return log of the variance for numerical stability.

WARNING: Do not apply any non-linearity to the last activation.

[248]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class Encoder(nn.Module):
    def __init__(self, z_dim=Z_DIM, x_dim=X_DIM):
        super(Encoder, self).__init__()
        self.z_dim = z_dim
        self.x_dim = x_dim
        # TODO: implement here
    def __init__(self, z_dim=Z_DIM, x_dim=X_DIM):
        super(Encoder, self).__init__()
        self.z_dim = z_dim
        self.x_dim = x_dim
        self.fc1 = nn.Linear(x_dim, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, z_dim * 2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        out = self.fc4(x)
        # TODO: implement here
        mu, log_var = out[:, :self.z_dim], out[:, self.z_dim:]
        return mu, log_var
```

[249]:
```python
# TODO: implement reparameterization trick
def sample_z(mu, log_var):
    # Your code here for the reparameterization trick.
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    std = torch.exp(0.5 * log_var)
    eps = torch.randn_like(std)
    samples = mu + eps * std

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

2.4 Decoder

Now to build a decoder network $p_\theta(x \mid z)$. Use four linear layers.

In this exercise, we will use continuous Bernoulli MLP decoder where $p_\theta(x \mid z)$ is modeled with multivariate continuous Bernoulli distribution, in contrast to the Gaussian distribution we discussed in the lecture, as following (see Appendix C.1 in the original paper and

https://arxiv.org/abs/1907.06845 for more details),

$\log p(x \mid z) = \sum_{i=1} x_i \log \lambda(z)_i + (1 - x_i) \log(1 - \lambda(z)_i) + \log C(\lambda(z)_i)$,

where $\lambda(z)_i$ is the parameter of continuous Bernoulli distribution corresponding to i-th pixel. (Note that $\lambda(z)$ is corresponding to sigmoid(x_logit) in this implementation, and it also can be seen as the decoded image for latent z.)

Note, the output of the decoder should have shape `[batch_size, x_dim]` and should output the unnormalized logits of x_i.

You can introduce a coefficient to the $\log C(\lambda(z)_i)$ term to achieve higher inception score.

WARNING: Do not apply any non-linearity to the last activation.

[250]:
```
class Decoder(nn.Module):
    def __init__(self, z_dim=Z_DIM, x_dim=X_DIM):
        super(Decoder, self).__init__()
        # TODO: implement here
    def __init__(self, z_dim=Z_DIM, x_dim=X_DIM):
        super(Decoder, self).__init__()
        self.fc1 = nn.Linear(z_dim, 400)
        self.fc2 = nn.Linear(400, 600)
        self.fc3 = nn.Linear(600, 800)
        self.fc4 = nn.Linear(800, x_dim)

    def forward(self, z):
        # TODO: implement here
        h = F.relu(self.fc1(z))
        h = F.relu(self.fc2(h))
        h = F.relu(self.fc3(h))
        x_logit = self.fc4(h)
        return x_logit
```

2.5 Loss definition

Compute the VAE loss. 1. For the reconstruction loss, you might find `F.binary_cross_entropy_with_logits` useful. 2. For the kl loss, we discussed the closed form kl divergence between two gaussians in the lecture.

[251]:
```
def vae_loss(x, x_logit, z_mu, z_logvar):
    recon_loss = None
    kl_loss = None

    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    recon_loss = F.binary_cross_entropy_with_logits(x_logit, x,␣
 ↪reduction='none')
    recon_loss = recon_loss.sum(dim=1)
```

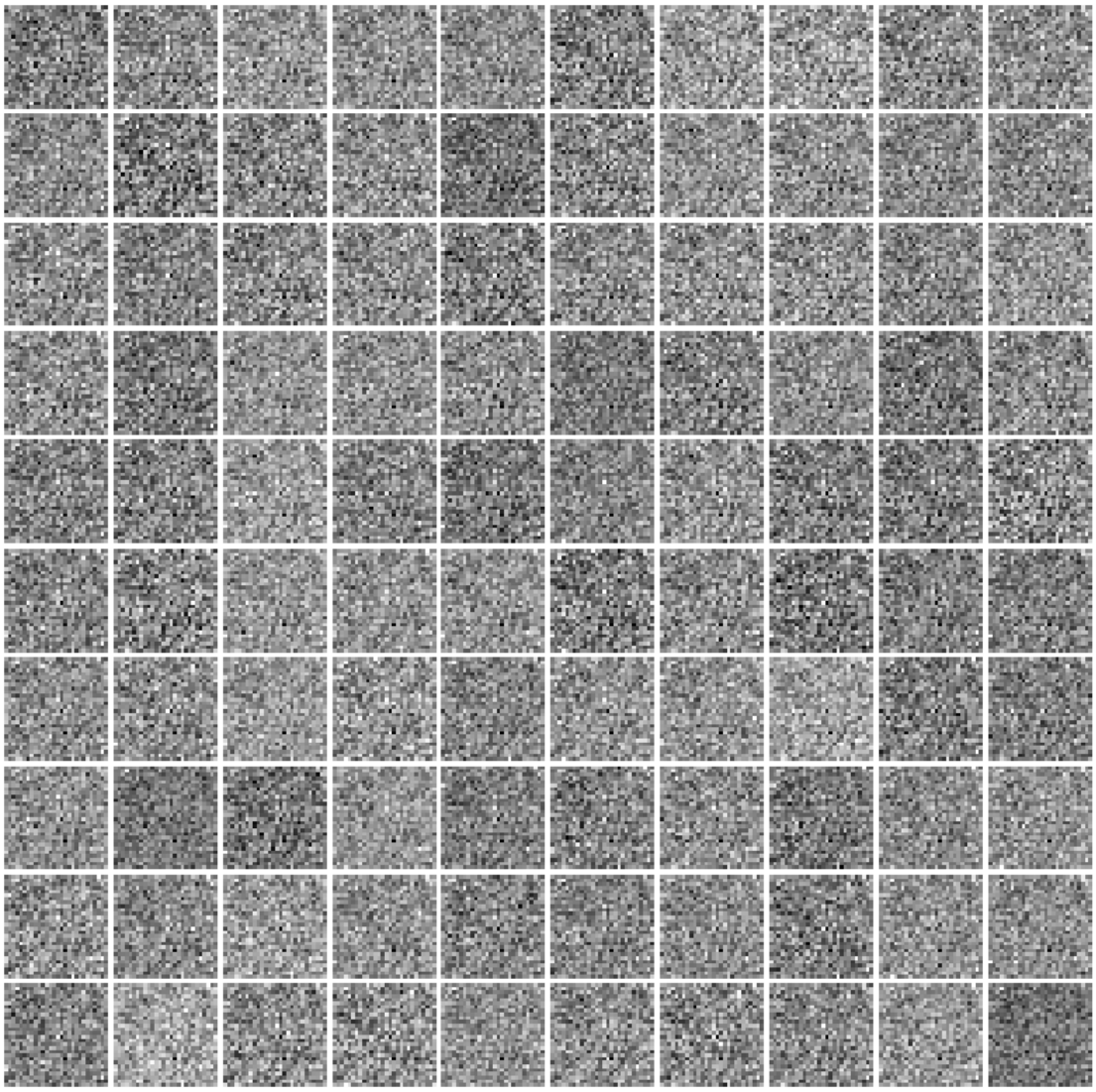
```
    kl_loss = -0.5 * torch.sum(1 + z_logvar - z_mu.pow(2) - z_logvar.exp(),␣
 ↪dim=1)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    vae_loss = torch.mean(recon_loss + kl_loss)
    return vae_loss, torch.mean(recon_loss)
```

2.6 Optimizing our loss

[252]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

Q = Encoder().to(device)
P = Decoder().to(device)

optimizer = torch.optim.Adam(list(Q.parameters()) + list(P.parameters()),␣
 ↪lr=learning_rate)
# MNIST DataLoader (shuffle=True)
transform = transforms.Compose([
    transforms.ToTensor(),
])
mnist_dataset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
mnist_loader = DataLoader(mnist_dataset, batch_size=batch_size, shuffle=True)
```

Visualize generated samples before training

[253]:
```
z_gen = torch.randn(num_to_show, Z_DIM).to(device)
x_gen = P(z_gen)
imgs_numpy = torch.sigmoid(x_gen).detach().cpu()
show_images(imgs_numpy)
plt.show()
```



2.7 Training a VAE!

If everything works, your batch average reconstruction loss should drop below 100.

[254]:
```
iter_count = 0
show_every = 200

# ----- Training Loop -----
for epoch in range(num_epochs):
    for x_i, _ in mnist_loader:
        x_i = x_i.view(x_i.size(0), -1).to(device)    # Flatten and move to␣
 ↪device
```

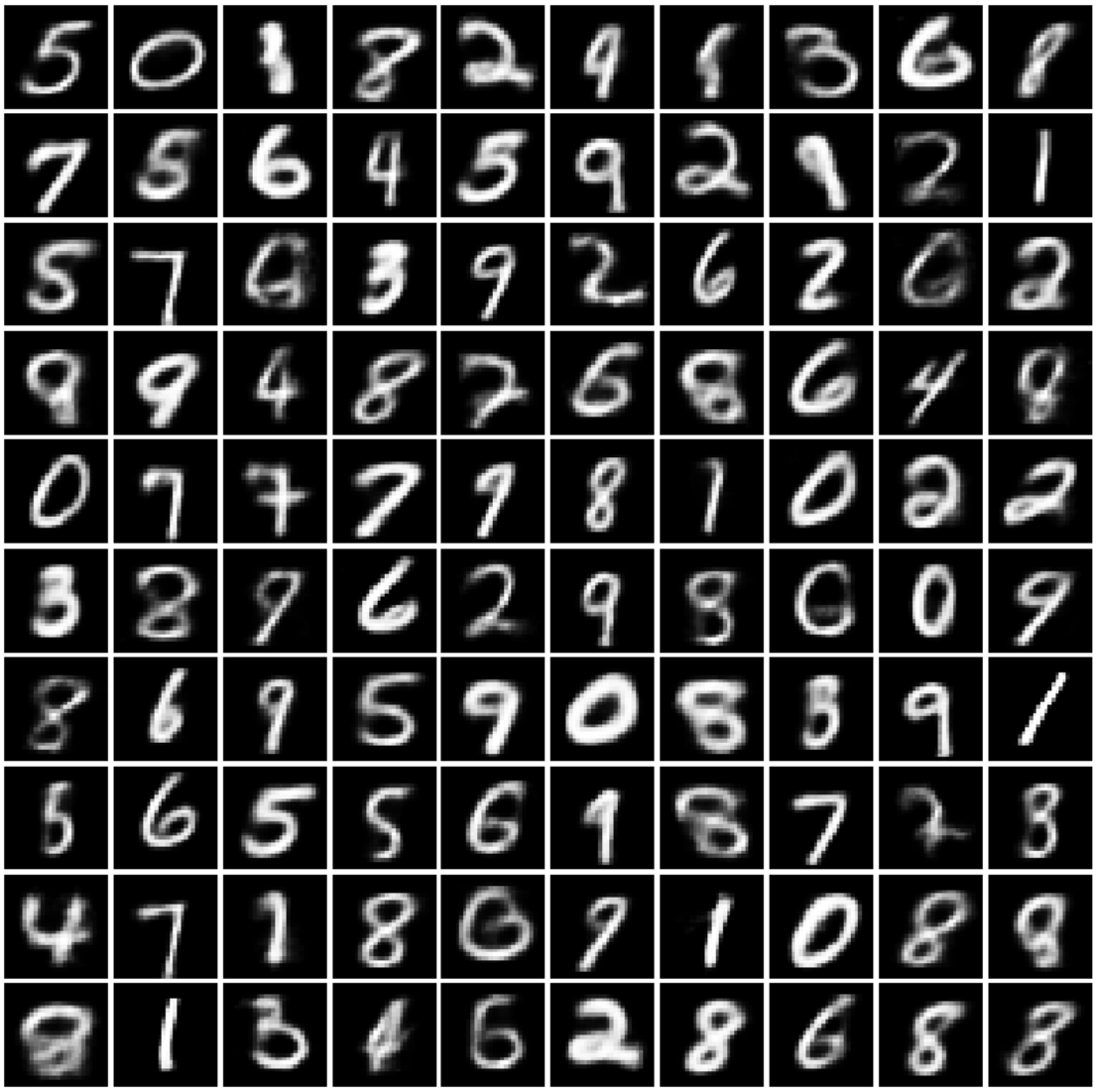
```
        z_mu, z_logvar = Q(preprocess_img(x_i))
        z_i = sample_z(z_mu, z_logvar)
        x_logit = P(z_i)

        loss, recon_loss = vae_loss(x_i, x_logit, z_mu, z_logvar)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if iter_count % show_every == 0:
            print(f'Epoch: {epoch}, Iter: {iter_count}, Loss: {loss.item():.
↪4f}, Recon: {recon_loss.item():.4f}')
            # imgs_numpy = torch.sigmoid(x_logit).detach().cpu()
            # show_images(imgs_numpy[:16])
            # plt.show()
        iter_count += 1
```

```
Epoch: 0, Iter: 0, Loss: 543.9945, Recon: 543.9343
Epoch: 0, Iter: 200, Loss: 195.1558, Recon: 192.0795
Epoch: 0, Iter: 400, Loss: 171.0682, Recon: 165.1683
Epoch: 1, Iter: 600, Loss: 156.2621, Recon: 147.8811
Epoch: 1, Iter: 800, Loss: 144.2961, Recon: 135.0350
Epoch: 2, Iter: 1000, Loss: 136.7245, Recon: 126.4303
Epoch: 2, Iter: 1200, Loss: 135.0218, Recon: 124.3832
Epoch: 2, Iter: 1400, Loss: 133.7334, Recon: 122.3903
Epoch: 3, Iter: 1600, Loss: 125.6340, Recon: 114.9075
Epoch: 3, Iter: 1800, Loss: 126.0188, Recon: 114.8924
Epoch: 4, Iter: 2000, Loss: 125.1432, Recon: 113.8018
Epoch: 4, Iter: 2200, Loss: 128.0200, Recon: 116.3927
Epoch: 5, Iter: 2400, Loss: 125.9601, Recon: 114.8956
Epoch: 5, Iter: 2600, Loss: 127.2093, Recon: 116.2531
Epoch: 5, Iter: 2800, Loss: 120.6557, Recon: 109.4584
Epoch: 6, Iter: 3000, Loss: 121.6137, Recon: 110.5558
Epoch: 6, Iter: 3200, Loss: 117.3924, Recon: 106.0040
Epoch: 7, Iter: 3400, Loss: 115.7692, Recon: 104.2017
Epoch: 7, Iter: 3600, Loss: 122.3400, Recon: 111.1993
Epoch: 8, Iter: 3800, Loss: 122.1366, Recon: 110.8441
Epoch: 8, Iter: 4000, Loss: 121.1614, Recon: 109.5243
Epoch: 8, Iter: 4200, Loss: 126.4925, Recon: 114.6616
Epoch: 9, Iter: 4400, Loss: 112.1536, Recon: 100.8407
Epoch: 9, Iter: 4600, Loss: 113.0489, Recon: 101.1688
Epoch: 10, Iter: 4800, Loss: 121.6966, Recon: 109.6233
Epoch: 10, Iter: 5000, Loss: 116.2486, Recon: 104.1490
Epoch: 11, Iter: 5200, Loss: 119.7371, Recon: 107.8240
Epoch: 11, Iter: 5400, Loss: 119.0359, Recon: 106.5928
```

```
Epoch: 11, Iter: 5600, Loss: 114.0579, Recon: 101.3157
Epoch: 12, Iter: 5800, Loss: 114.0075, Recon: 101.2991
Epoch: 12, Iter: 6000, Loss: 112.4443, Recon: 99.5273
Epoch: 13, Iter: 6200, Loss: 115.3136, Recon: 102.0778
Epoch: 13, Iter: 6400, Loss: 110.4615, Recon: 97.6006
Epoch: 14, Iter: 6600, Loss: 110.3705, Recon: 96.9587
Epoch: 14, Iter: 6800, Loss: 110.2188, Recon: 96.7641
Epoch: 14, Iter: 7000, Loss: 113.2832, Recon: 100.0941
Epoch: 15, Iter: 7200, Loss: 112.7359, Recon: 99.6122
Epoch: 15, Iter: 7400, Loss: 111.7380, Recon: 98.5635
Epoch: 16, Iter: 7600, Loss: 109.5828, Recon: 95.9408
Epoch: 16, Iter: 7800, Loss: 119.2860, Recon: 105.4755
Epoch: 17, Iter: 8000, Loss: 108.0109, Recon: 94.1297
Epoch: 17, Iter: 8200, Loss: 112.3596, Recon: 99.2380
Epoch: 17, Iter: 8400, Loss: 114.1174, Recon: 100.9255
Epoch: 18, Iter: 8600, Loss: 115.9904, Recon: 102.4055
Epoch: 18, Iter: 8800, Loss: 111.6362, Recon: 97.8279
Epoch: 19, Iter: 9000, Loss: 106.6584, Recon: 92.9972
Epoch: 19, Iter: 9200, Loss: 114.2803, Recon: 100.5176
```

Visualize generated samples after training

[255]:
```
z_gen = torch.randn(num_to_show, Z_DIM).to(device)
x_gen = P(z_gen)
imgs_numpy = torch.sigmoid(x_gen).detach().cpu()
show_images(imgs_numpy)
plt.show()
```



3 PART II. Compute the inception score for your trained VAE model

In this part, we will quantitavely measure how good your VAE model is.

3.0.1 Train a classifier

We first need to train a classifier.

[256]:
```
# ----- Hyperparameters -----
batch_size = 128
num_classes = 10
epochs = 20
```

```
# ----- Data Preparation -----
train_dataset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=transform)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size)

print(f'{len(train_dataset)} train samples')
print(f'{len(test_dataset)} test samples')

# ----- Model Definition -----
class MLPClassifier(nn.Module):
    def __init__(self):
        super(MLPClassifier, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = torch.flatten(x, start_dim=1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, 0.2)
        x = F.relu(self.fc2(x))
        x = F.dropout(x, 0.2)
        x = self.fc3(x)
        return x

    def prob(self, x):
        x = self.forward(x)
        prob = F.softmax(x, dim=-1)
        return prob

model = MLPClassifier().to(device)
optimizer = optim.RMSprop(model.parameters(), lr=0.001, alpha=0.9)
criterion = nn.CrossEntropyLoss()

# ----- Training -----
for epoch in range(epochs):
    model.train()
    for batch_x, batch_y in train_loader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)
        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
```

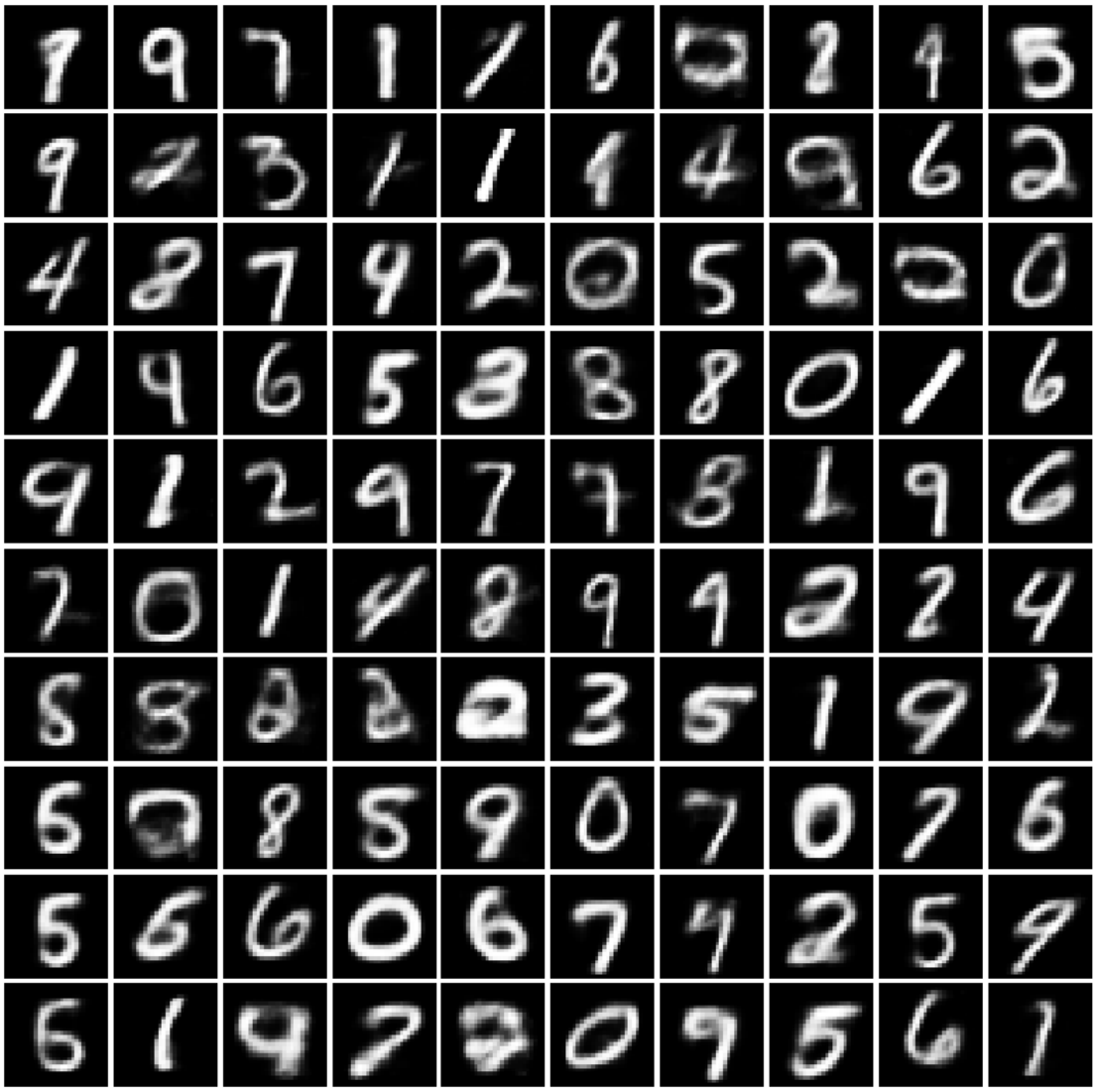
```
        loss.backward()
        optimizer.step()
    print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}')

# ----- Evaluation -----
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for batch_x, batch_y in test_loader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)
        outputs = model(batch_x)
        predicted = torch.argmax(outputs, dim=1)
        total += batch_y.size(0)
        correct += (predicted == batch_y).sum().item()

print('Test accuracy:', correct / total)
```

```
60000 train samples
10000 test samples
Epoch 1/20, Loss: 0.0894
Epoch 2/20, Loss: 0.0263
Epoch 3/20, Loss: 0.1862
Epoch 4/20, Loss: 0.0313
Epoch 5/20, Loss: 0.1074
Epoch 6/20, Loss: 0.0152
Epoch 7/20, Loss: 0.0517
Epoch 8/20, Loss: 0.0519
Epoch 9/20, Loss: 0.0351
Epoch 10/20, Loss: 0.0055
Epoch 11/20, Loss: 0.0137
Epoch 12/20, Loss: 0.0260
Epoch 13/20, Loss: 0.0280
Epoch 14/20, Loss: 0.0449
Epoch 15/20, Loss: 0.0017
Epoch 16/20, Loss: 0.0448
Epoch 17/20, Loss: 0.0001
Epoch 18/20, Loss: 0.0010
Epoch 19/20, Loss: 0.0196
Epoch 20/20, Loss: 0.0001
Test accuracy: 0.9794
```

3.0.2 Verify the trained classifier on the generated samples

Generate samples and visually inspect if the predicted labels on the samples match the actual digits in generated images.

[257]:
```
z_gen = torch.randn(num_samples, Z_DIM).to(device)
x_gen = P(z_gen)
imgs_numpy = torch.sigmoid(x_gen[:num_to_show]).detach().cpu()
show_images(imgs_numpy)
plt.show()
```



[ ]:
```
preds = torch.argmax(model(torch.sigmoid(x_gen[:20])), dim=1)
print(preds.cpu().numpy())
```

3.0.3 Implement the inception score

Implement Equation 1 in the reference [3]. Replace expectation in the equation with empirical average of `num_samples` samples. Don't forget the exponentiation at the end. You should get

Inception score of at least 9.0.

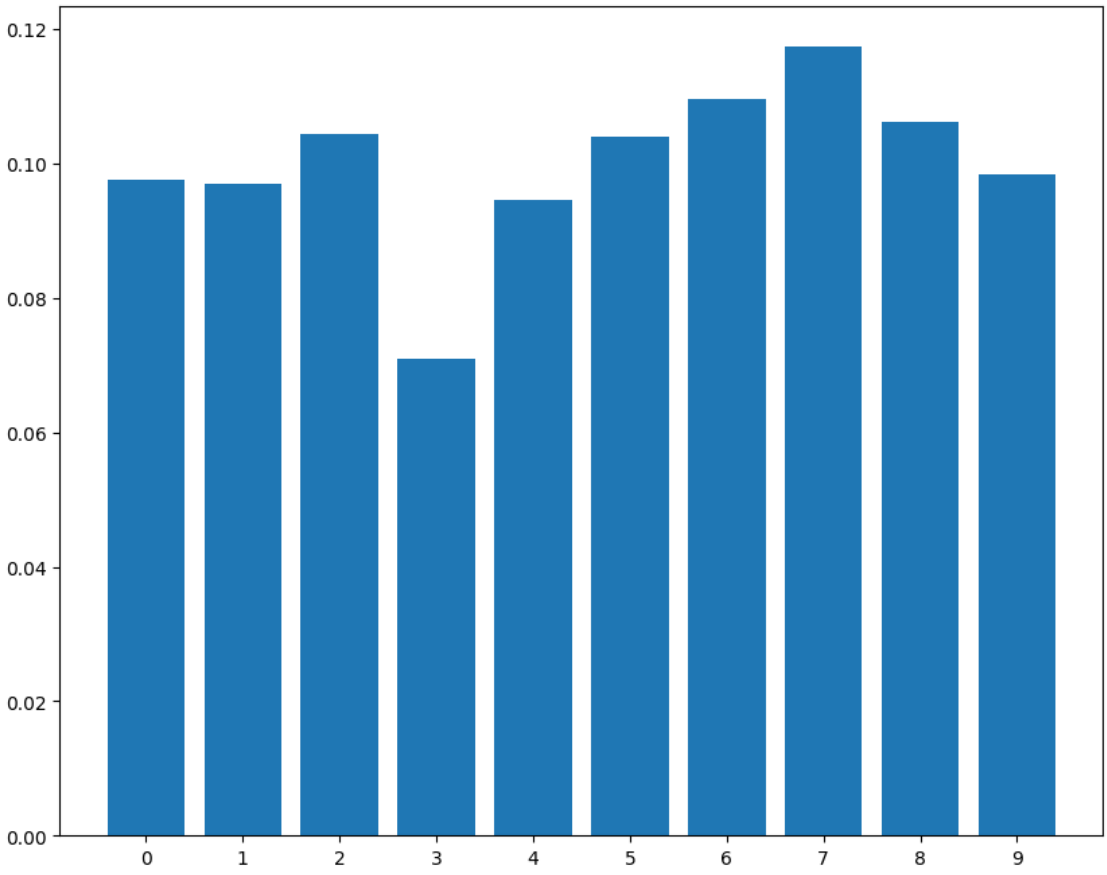
[261]:
```
with torch.no_grad():
    # TODO: implement here
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    z_gen = torch.randn(num_samples, Z_DIM).to(device)
    x_gen = P(z_gen)
    x_gen_sigmoid = torch.sigmoid(x_gen)

    p_y_given_x = F.softmax(model(x_gen_sigmoid), dim=1)

    p_y = torch.mean(p_y_given_x, dim=0)

    eps = 1e-10
    kl_div = p_y_given_x * (torch.log(p_y_given_x + eps) - torch.log(p_y.
 ↪unsqueeze(0) + eps))
    kl_div = torch.sum(kl_div, dim=1)

    mean_kl_div = torch.mean(kl_div)

    inception_score = torch.exp(mean_kl_div).item()
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

print(f'Inception score: {inception_score:.4f}')
```

```
Inception score: 9.0851
```

3.0.4 Plot the histogram of predicted labels

Let's additionally inspect the class diversity of the generated samples.

[262]:
```
hist_preds = torch.argmax(model(torch.sigmoid(x_gen)), dim=1).cpu().numpy()
plt.hist(hist_preds, bins=np.arange(11)-0.5, rwidth=0.8, density=True)
plt.xticks(range(10))
plt.show()
```


0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
1
2
3
4
5
6
7
8
9

GAN

May 1, 2025

[2]:
```
from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# 'cs231n' folder containing the '.py', 'classifiers' and 'datasets'
# folders.
FOLDERNAME = 'hw5/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../..
%cd 'hw5/'
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/hw5
```

0.0.1 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 $x \sim p_{\text{data}}$ are samples from the input data, $z \sim p(z)$ 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 aternate 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:

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

0.0.2 What else is there?

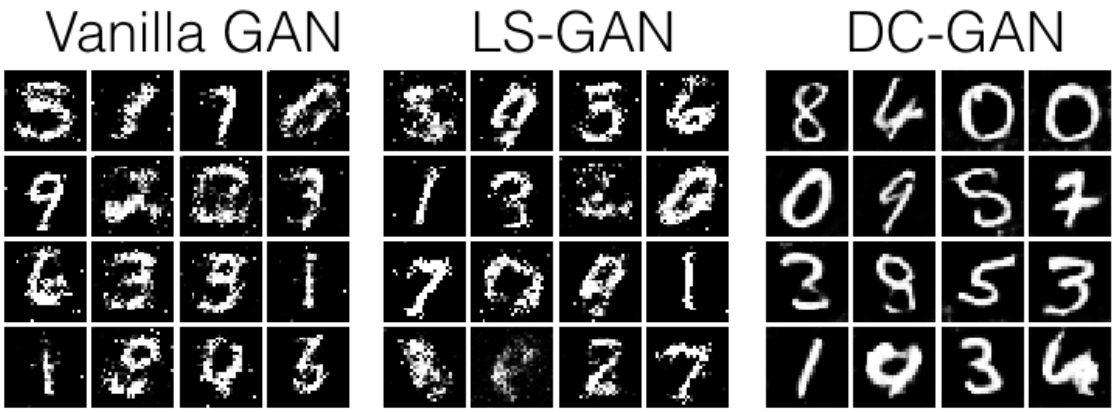
Since 2014, GANs have exploded into a huge research area, with massive workshops, and hundreds of new papers. Compared to other approaches for generative models, they often produce the highest quality samples but are some of the most difficult and finicky models to train (see this github repo that contains a set of 17 hacks that are useful for getting models working). Improving the stabiilty and robustness of GAN training is an open research question, with new papers coming out every day! For a more recent tutorial on GANs, see here. There is also some even more recent exciting work that changes the objective function to Wasserstein distance and yields much more stable results across model architectures: WGAN, WGAN-GP.

GANs are not the only way to train a generative model! For other approaches to generative modeling check out the deep generative model chapter of the Deep Learning book.

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]:
```
from IPython.display import Image
Image(filename="gan_outputs_tf.png")
```

[3]:



0.1 Setup

[4]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import datasets, transforms

import numpy as np
import os

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

%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'

# A bunch of utility functions

def show_images(images):
    images = images.view(images.shape[0], -1).detach().cpu().numpy()
    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]))
    plt.show()

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):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

answers = {}

for k, v in np.load('gan-checks-tf.npz').items():
    answers[k] = torch.tensor(v)

NOISE_DIM = 10
NUM_SAMPLES = 10000
```

0.2 Dataset

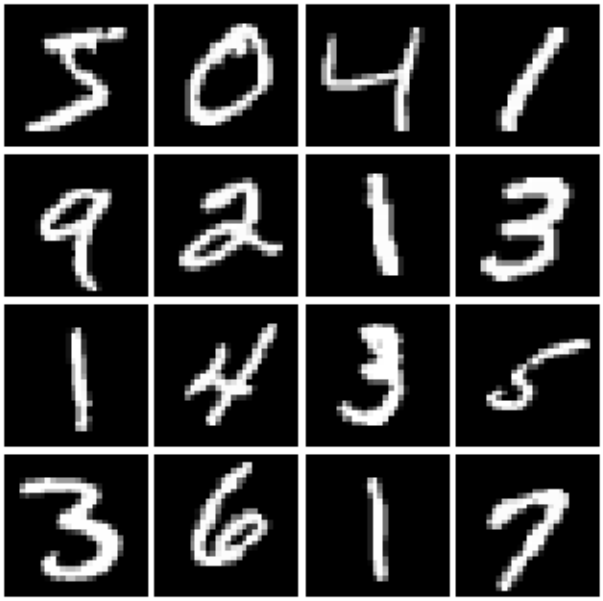
GANs are notoriously finicky with hyperparameters, and also require many training epochs. In order to make this assignment approachable without a GPU, we will be working on the MNIST dataset, which is 60,000 training and 10,000 test images. Each picture contains a centered image of white digit on black 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.

Heads-up: Our MNIST wrapper returns images as vectors. That is, they're size (batch, 784). If you want to treat them as images, we have to resize them to (batch,28,28) or (batch,28,28,1). They are also type np.float32 and bounded [0,1].

[5]:
```
transform = transforms.Compose([
    transforms.ToTensor(), # [0,1]
])

mnist_dataset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
batch_size = 16
mnist_loader = DataLoader(mnist_dataset, batch_size=batch_size, shuffle=False)
```

```
100%|      | 9.91M/9.91M [00:02<00:00, 4.59MB/s]
100%|      | 28.9k/28.9k [00:00<00:00, 57.9kB/s]
100%|      | 1.65M/1.65M [00:06<00:00, 242kB/s]
100%|      | 4.54k/4.54k [00:00<00:00, 8.96MB/s]
```

[6]:
```
# Show a batch
data_iter = iter(mnist_loader)
images, labels = next(data_iter)
show_images(images)
```



0.3 Random Noise

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

[7]:
```
def sample_noise(batch_size, dim):
    """Generate random uniform noise from -1 to 1.

    Inputs:
    - batch_size: integer giving the batch size of noise to generate
    - dim: integer giving the dimension of the noise to generate

    Returns:
    TensorFlow Tensor containing uniform noise in [-1, 1] with shape␣
 ↪[batch_size, dim]
    """
    # TODO: sample and return noise
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
    return torch.rand(batch_size, dim) * 2 - 1

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

Make sure noise is the correct shape and type:

[8]:
```
def test_sample_noise():
    batch_size = 3
    dim = 4
    z = sample_noise(batch_size, dim)
    # Check z has the correct shape
    assert list(z.shape) == [batch_size, dim]
    # Make sure z is a Tensor and not a numpy array
    assert isinstance(z, torch.Tensor)
    # Check that we get different noise for different evaluations
    z1 = sample_noise(batch_size, dim)
    z2 = sample_noise(batch_size, dim)
    assert not np.array_equal(z1.numpy(), z2.numpy())
    # Check that we get the correct range
    assert np.all(z1.numpy() >= -1.0) and np.all(z1.numpy() <= 1.0)
    print("All tests passed!")

test_sample_noise()
```

```
All tests passed!
```

0.4 Discriminator

Our first step is to build a discriminator. **Hint:** You should use the layers in `torch.nn` to build the model.

Architecture: * Fully connected layer with input size 784 and output size 256 * LeakyReLU with alpha 0.01 * Fully connected layer with output size 256 * LeakyReLU with alpha 0.01 * Fully connected layer with output size 1

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

[9]:
```
import torch.nn as nn

class Discriminator(nn.Module):
    def __init__(self, input_dim=784):
        super(Discriminator, self).__init__()
        # TODO: implement architecture
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        self.model = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.LeakyReLU(0.01),
            nn.Linear(256, 256),
```

```
            nn.LeakyReLU(0.01),
            nn.Linear(256, 1)
        )
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    def forward(self, x):
        # TODO: forward function
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        if len(x.shape) == 4:
            x = x.view(x.size(0), -1)
        return self.model(x)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

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

[10]:
```
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. {0} instead of␣
 ↪{1}. Check your achitecture.'.format(cur_count,true_count))
    else:
        print('Correct number of parameters in discriminator.')

test_discriminator()
```

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

0.5 Generator

Now to build a generator. You should use the layers in `torch.nn` to construct the model. All fully connected layers should include bias terms. Note that you can use the tf.nn module to access activation functions. Once again, use the default initializers for parameters.

Architecture: * Fully connected layer with input size z.shape[1] (the number of noise dimensions) and output size 1024 * `ReLU` * Fully connected layer with output size 1024 * `ReLU` * Fully connected layer with output size 784 * `Tanh` (To restrict every element of the output to be in the range [-1,1])

[11]:
```
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self, noise_dim=NOISE_DIM):
        super(Generator, self).__init__()
        # TODO: implement architecture
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        self.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)*****

    def forward(self, z):
        # TODO: implement forward function
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return self.model(z)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

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

[12]:
```
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. {0} instead of {1}.␣
 ↪Check your achitecture.'.format(cur_count,true_count))
    else:
        print('Correct number of parameters in generator.')

test_generator()
```

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

1 GAN Loss

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

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

and the discriminator loss is:

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

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

HINTS: Use `torch.ones_like` and `torch.zeros_like` to generate labels for your discriminator. Use `torch.nn.BCEWithLogitsLoss` to help compute your loss function.

[13]:
```
import torch
import torch.nn as nn


def discriminator_loss(logits_real, logits_fake):
```

```
    """
    Computes the discriminator loss described above.

    Inputs:
    - logits_real: Tensor of shape (N, 1) giving scores for the real data.
    - logits_fake: Tensor of shape (N, 1) giving scores for the fake data.

    Returns:
    - loss: Tensor containing (scalar) the loss for the discriminator.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    real_labels = torch.ones_like(logits_real)
    fake_labels = torch.zeros_like(logits_fake)

    criterion = nn.BCEWithLogitsLoss()

    loss_real = criterion(logits_real, real_labels)

    loss_fake = criterion(logits_fake, fake_labels)

    loss = loss_real + loss_fake
    # *****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)*****
    fake_labels = torch.ones_like(logits_fake)

    criterion = nn.BCEWithLogitsLoss()

    loss = criterion(logits_fake, fake_labels)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return loss
```

Test your GAN loss. Make sure both the generator and discriminator loss are correct. You should see errors less than 1e-8.

[14]:
```
def test_discriminator_loss(logits_real, logits_fake, d_loss_true):
    d_loss = discriminator_loss(logits_real,
                                logits_fake)
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true.numpy(), d_loss.
 ↪numpy()))

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

```
Maximum error in d_loss: 0
```

[15]:
```
def test_generator_loss(logits_fake, g_loss_true):
    g_loss = generator_loss(logits_fake)
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true.numpy(), g_loss.
 ↪numpy()))

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

```
Maximum error in g_loss: 7.19722e-17
```

2 Optimizing our loss

Make an `Adam` optimizer with a 1e-3 learning rate, beta1=0.5 to mininize G_loss and D_loss separately. The trick of decreasing beta was shown to be effective in helping GANs converge in the Improved Techniques for Training GANs paper. In fact, with our current hyperparameters, if you set beta1 to the Tensorflow default of 0.9, there's a good chance your discriminator loss will go to zero and the generator will fail to learn entirely. In fact, this is a common failure mode in GANs; if your D(x) learns too fast (e.g. loss goes near zero), your G(z) is never able to learn. Often D(x) is trained with SGD with Momentum or RMSProp instead of Adam, but here we'll use Adam for both D(x) and G(z).

[16]:
```
import torch.optim as optim

def get_solvers(D, G, learning_rate=1e-3, beta1=0.5):
    """
    Create Adam optimizers for GAN training in PyTorch.

    Inputs:
    - D: Discriminator (nn.Module)
    - G: Generator (nn.Module)
    - learning_rate: learning rate for both solvers
    - beta1: beta1 value for Adam (1st moment decay)

    Returns:
    - D_solver: optimizer for the discriminator
```

```
    - G_solver: optimizer for the generator
    """
    D_solver = None
    G_solver = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    D_solver = optim.Adam(D.parameters(), lr=learning_rate, betas=(beta1, 0.
↪999))
    G_solver = optim.Adam(G.parameters(), lr=learning_rate, betas=(beta1, 0.
↪999))
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return D_solver, G_solver
```

3 Training a GAN!

Well that wasn't so hard, was it? After the first epoch, you should see fuzzy outlines, clear shapes as you approach epoch 3, and decent shapes, about half of which will be sharp and clearly recognizable as we pass epoch 5. In our case, we'll simply train D(x) and G(z) with one batch each every iteration. However, papers often experiment with different schedules of training D(x) and G(z), sometimes doing one for more steps than the other, or even training each one until the loss gets "good enough" and then switching to training the other.

[17]:
```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

def run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss,
              show_every=20, print_every=20, batch_size=128, num_epochs=10,
              noise_size=NOISE_DIM):
    """
    Train a GAN in PyTorch.

    Inputs:
    - D: Discriminator model (nn.Module)
    - G: Generator model (nn.Module)
    - D_solver: optimizer for Discriminator
    - G_solver: optimizer for Generator
    - discriminator_loss: function to compute D loss
    - generator_loss: function to compute G loss
    """
    D = D.to(device)
    G = G.to(device)
    transform = transforms.Compose([transforms.ToTensor()])
    dataloader = DataLoader(
```

```
        datasets.MNIST(root='./data', train=True, download=True,␣
↪transform=transform),
        batch_size=batch_size, shuffle=True
    )

    iter_count = 0
    for epoch in range(num_epochs):
        for x, _ in dataloader:
            real_data = x.view(x.size(0), -1).to(device)

            # ---------------------
            # 1. Update Discriminator
            # ---------------------
            D_solver.zero_grad()
            logits_real = D(preprocess_img(real_data))

            g_fake_seed = sample_noise(batch_size, noise_size).to(device)
            fake_images = G(g_fake_seed)
            logits_fake = D(fake_images)

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

            # ---------------------
            # 2. Update Generator
            # ---------------------
            G_solver.zero_grad()
            g_fake_seed = sample_noise(batch_size, noise_size).to(device)
            fake_images = G(g_fake_seed)
            gen_logits_fake = D(fake_images)

            g_error = generator_loss(gen_logits_fake)
            g_error.backward()
            G_solver.step()

            # ---------------------
            # 3. Logging & Visualization
            # ---------------------
            if iter_count % show_every == 0:
                print(f'Epoch: {epoch}, Iter: {iter_count}, D: {d_total_error.
↪item():.4f}, G: {g_error.item():.4f}')
                imgs_numpy = fake_images[:16].detach().cpu()
                show_images(imgs_numpy)
                plt.show()

            iter_count += 1
```

```
    # ----- Final Visualization -----
    z = sample_noise(batch_size, noise_size).to(device)
    G_sample = G(z).detach().cpu()
    print('Final images')
    show_images(G_sample[:16])
    plt.show()
```

Train your GAN! This should take about 10 minutes on a CPU, or about 2 minutes on GPU.

[18]:
```
# Make the discriminator
D = Discriminator()

# Make the generator
G = Generator()

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

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

```
Output hidden; open in https://colab.research.google.com to view.
```

4 Least Squares GAN

We'll now look at Least Squares GAN, a newer, more stable alternative 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:

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

and the discriminator loss:

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

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 (`score_real` and `score_fake`).

[19]:
```
import torch

def ls_discriminator_loss(scores_real, scores_fake):
    """
    Compute the Least-Squares GAN loss for the discriminator.
```

```
    Inputs:
    - scores_real: Tensor of shape (N, 1) giving scores for the real data.
    - scores_fake: Tensor of shape (N, 1) giving scores for the fake data.

    Outputs:
    - loss: A Tensor containing the loss.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    loss_real = 0.5 * torch.mean((scores_real - 1) ** 2)
    loss_fake = 0.5 * torch.mean(scores_fake ** 2)
    loss = loss_real + loss_fake
    # *****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: Tensor of shape (N, 1) giving scores for the fake data.

    Outputs:
    - loss: A Tensor containing the loss.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    loss = 0.5 * torch.mean((scores_fake - 1) ** 2)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss
```

Test your LSGAN loss. You should see errors less than 1e-8.

[20]:
```
def test_lsgan_loss(score_real, score_fake, d_loss_true, g_loss_true):

    d_loss = ls_discriminator_loss(score_real, score_fake)
    g_loss = ls_generator_loss(score_fake)
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true.numpy(), d_loss.
 ↪numpy()))
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true.numpy(), g_loss.
 ↪numpy()))

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

```
Maximum error in d_loss: 0
```

```
Maximum error in g_loss: 0
```

Create new training steps so we instead minimize the LSGAN loss:

[21]:
```
# Make the discriminator
D = Discriminator()

# Make the generator
G = Generator()

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

# Run it!
run_a_gan(D, G, D_solver, G_solver, ls_discriminator_loss, ls_generator_loss)
```

```
Output hidden; open in https://colab.research.google.com to view.
```

5 Deep Convolutional GANs

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 as our discriminators and generators.

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. *Be sure to check the dimensions of x and reshape when needed*, fully connected blocks expect [N,D] Tensors while conv2d blocks expect [N,H,W,C] Tensors. Please use `tf.keras.layers` to define the following architecture:

Architecture: * Conv2D: 32 Filters, 5x5, Stride 1, padding 0 * Leaky ReLU(alpha=0.01) * Max Pool 2x2, Stride 2 * Conv2D: 64 Filters, 5x5, Stride 1, padding 0 * 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

Once again, please use biases for all convolutional and fully connected layers, and use the default parameter initializers. Note that a padding of 0 can be accomplished with the 'VALID' padding option.

[22]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
```

```
        # TODO: implement architecture
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=0)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=0)
        self.fc1 = nn.Linear(4 * 4 * 64, 4 * 4 * 64)
        self.fc2 = nn.Linear(4 * 4 * 64, 1)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    def forward(self, x):
        # TODO: implement forward function
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        if len(x.shape) == 2:
            x = x.view(-1, 1, 28, 28)

        x = self.conv1(x)
        x = F.leaky_relu(x, negative_slope=0.01)
        x = self.pool(x)

        x = self.conv2(x)
        x = F.leaky_relu(x, negative_slope=0.01)
        x = self.pool(x)

        x = x.view(-1, 4 * 4 * 64)

        x = self.fc1(x)
        x = F.leaky_relu(x, negative_slope=0.01)

        x = self.fc2(x)

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

model = Discriminator()
test_discriminator(1102721)
```

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

Generator For the generator, we will copy the architecture exactly from the InfoGAN paper. See Appendix C.1 MNIST. Please use `tf.keras.layers` for your implementation. You might find the documentation for tf.keras.layers.Conv2DTranspose useful. The architecture is as follows.

Architecture: * Fully connected with output size 1024 * `ReLU` * BatchNorm * Fully connected with output size 7 x 7 x 128 * `ReLU` * BatchNorm * Resize into Image Tensor of size 7, 7, 128 * Conv2D^T (transpose): 64 filters of 4x4, stride 2 * `ReLU` * BatchNorm * Conv2d^T (transpose): 1 filter of 4x4, stride 2 * `TanH`

Once again, use biases for the fully connected and transpose convolutional layers. Please use

the default initializers for your parameters. For padding, choose the 'same' option for transpose convolutions. For Batch Normalization, assume we are always in 'training' mode.

[23]:
```
import torch
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self, noise_dim=NOISE_DIM):
        super(Generator, self).__init__()
        # TODO: implement architecture
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        self.fc1 = nn.Linear(noise_dim, 1024)
        self.bn1 = nn.BatchNorm1d(1024)

        self.fc2 = nn.Linear(1024, 7 * 7 * 128)
        self.bn2 = nn.BatchNorm1d(7 * 7 * 128)

        self.conv_t1 = nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2,␣
 ↪padding=1)
        self.bn3 = nn.BatchNorm2d(64)

        self.conv_t2 = nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2,␣
 ↪padding=1)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    def forward(self, z):
        # TODO: implement forward function
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        x = self.fc1(z)
        x = nn.functional.relu(x)
        x = self.bn1(x)

        x = self.fc2(x)
        x = nn.functional.relu(x)
        x = self.bn2(x)

        x = x.view(-1, 128, 7, 7)

        x = self.conv_t1(x)
        x = nn.functional.relu(x)
        x = self.bn3(x)

        x = self.conv_t2(x)
        x = torch.tanh(x)

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

```
test_generator(6580801)
```

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

We have to recreate our network since we've changed our functions.

5.0.1 Train and evaluate a DCGAN

This is the one part of A3 that significantly benefits from using a GPU. It takes 3 minutes on a GPU for the requested five epochs. Or about 50 minutes on a dual core laptop on CPU (feel free to use 3 epochs if you do it on CPU).

[24]:
```
# Make the discriminator
D = Discriminator()

# Make the generator
G = Generator()

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

# Run it!
run_a_gan(D, G, D_solver, G_solver, ls_discriminator_loss, ls_generator_loss,␣
 ↪num_epochs=5)
```

```
Output hidden; open in https://colab.research.google.com to view.
```

5.0.2 Inception score

[25]:
```
# ----- Hyperparameters -----
batch_size = 128
num_classes = 10
epochs = 20
# ----- Data Preparation -----
# (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
train_dataset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=transform)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size)

print(f'{len(train_dataset)} train samples')
print(f'{len(test_dataset)} test samples')

# ----- Model Definition -----
```

```
class MLPClassifier(nn.Module):
    def __init__(self):
        super(MLPClassifier, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = torch.flatten(x, start_dim=1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, 0.2)
        x = F.relu(self.fc2(x))
        x = F.dropout(x, 0.2)
        x = self.fc3(x)
        return x

    def prob(self, x):
        x = self.forward(x)
        prob = F.softmax(x, dim=-1)
        return prob

model = MLPClassifier().to(device)
optimizer = optim.RMSprop(model.parameters(), lr=0.001, alpha=0.9)
criterion = nn.CrossEntropyLoss()

# ----- Training -----
for epoch in range(epochs):
    model.train()
    for batch_x, batch_y in train_loader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)
        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
    print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}')

# ----- Evaluation -----
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for batch_x, batch_y in test_loader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)
        outputs = model(batch_x)
```

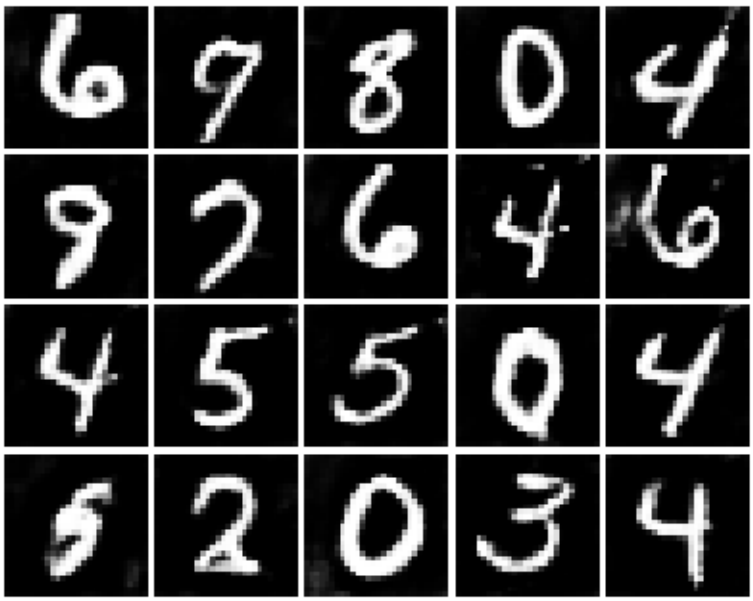
```
        predicted = torch.argmax(outputs, dim=1)
        total += batch_y.size(0)
        correct += (predicted == batch_y).sum().item()

print('Test accuracy:', correct / total)
```

```
60000 train samples
10000 test samples
Epoch 1/20, Loss: 0.2663
Epoch 2/20, Loss: 0.1784
Epoch 3/20, Loss: 0.1099
Epoch 4/20, Loss: 0.0786
Epoch 5/20, Loss: 0.0603
Epoch 6/20, Loss: 0.0029
Epoch 7/20, Loss: 0.0452
Epoch 8/20, Loss: 0.0165
Epoch 9/20, Loss: 0.0210
Epoch 10/20, Loss: 0.1004
Epoch 11/20, Loss: 0.0405
Epoch 12/20, Loss: 0.0880
Epoch 13/20, Loss: 0.0153
Epoch 14/20, Loss: 0.0073
Epoch 15/20, Loss: 0.0012
Epoch 16/20, Loss: 0.0009
Epoch 17/20, Loss: 0.0294
Epoch 18/20, Loss: 0.0000
Epoch 19/20, Loss: 0.1306
Epoch 20/20, Loss: 0.0002
Test accuracy: 0.9798
```

5.0.3 Verify the trained classifier on the generated samples

Generate samples and visually inspect if the predicted labels on the samples match the actual digits in generated images.

[26]:
```
with torch.no_grad():
    z = sample_noise(NUM_SAMPLES, NOISE_DIM).to(device)
    G_sample = G(z)
    G_sample = deprocess_img(G_sample)
show_images(G_sample[:20].cpu())
plt.show()
```



[27]:
```
with torch.no_grad():
    G_sample = G_sample.reshape(NUM_SAMPLES, 784)
    print(np.argmax(model(G_sample[:20].to(device)).cpu().numpy(), axis=-1))
```

```
[6 7 8 0 4 9 7 6 4 6 4 5 5 0 4 9 2 0 3 4]
```

5.0.4 Implement the inception score

Implement Equation 1 in the reference [3]. Replace expectation in the equation with empirical average of `num_samples` samples. Don't forget the exponentiation at the end. You should get Inception score of at least 8.5

[28]:
```
with torch.no_grad():
    # TODO: implement here
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    probs = F.softmax(model(G_sample.to(device)), dim=1).cpu().numpy()

    p_y = np.mean(probs, axis=0)

    kl_divs = []
    for i in range(probs.shape[0]):
        p = probs[i]
        kl_div = np.sum(p * np.log(p / p_y + 1e-10))
        kl_divs.append(kl_div)

    inception_score = np.exp(np.mean(kl_divs))
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

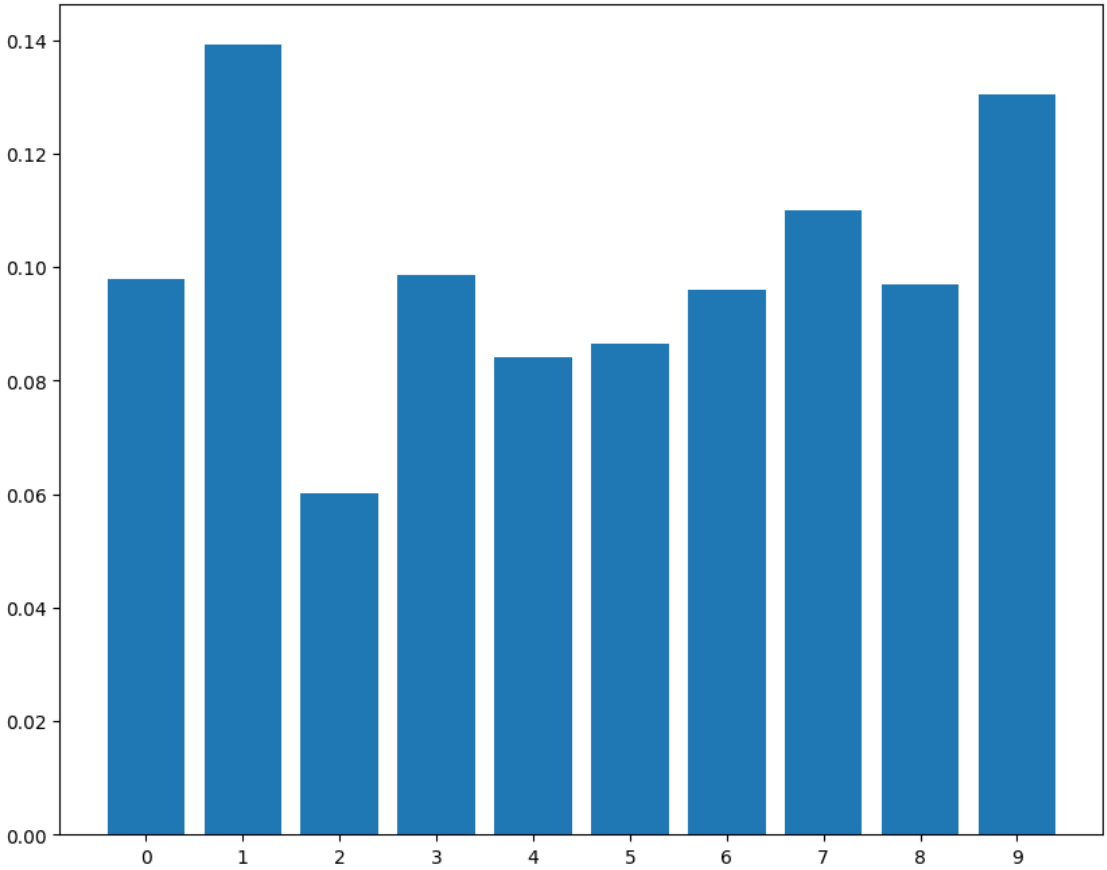
```
print(f'Inception score: {inception_score:.4f}')
```

```
Inception score: 9.3821
```

5.0.5 Plot the histogram of predicted labels

Let's additionally inspect the class diversity of the generated samples.

[29]:
```
with torch.no_grad():
    plt.hist(np.argmax(model(G_sample).cpu(), axis=-1),
             bins=np.arange(11)-0.5, rwidth=0.8, density=True)
plt.xticks(range(10))
plt.show()
```



5.1 INLINE QUESTION 1

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.

Breifly explain what $\min_x \max_y f(x, y)$ evaluates to and record the six pairs of explicit values for (x_t, y_t) in the table below.

5.1.1 Your answer:

For $\min_x \max_y f(x, y) = xy$, the $\max_y xy$ reaches its maximum value when y approaches infinity. However, considering $\min_x$, when $x = 0$, the result is always 0, so $\min_x \max_y xy = 0$.

Calculating with alternating gradient descent: $\nabla_y f(x, y) = x$, $\nabla_x f(x, y) = y$ $y_{t+1} = y_t + \text{step size} \cdot \nabla_y f(x_t, y_t) = y_t + x_t$ $x_{t+1} = x_t - \text{step size} \cdot \nabla_x f(x_t, y_{t+1}) = x_t - y_{t+1} = x_t - (y_t + x_t) = -y_t$

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	-1	1	1	-1	-1

5.2 INLINE QUESTION 2

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

5.2.1 Your answer:

Using this method, we cannot reach the optimal value. As seen in the calculations above, the values of (x_t, y_t) repeat periodically, forming patterns like $(1, 1)$, $(-1, 2)$, $(-1, 1)$, $(1, -1)$, $(1, -2)$, $(-1, -1)$, $(-1, 1)$. This is because the algorithm cycles through these values instead of converging to the optimal value $(0, y)$. This phenomenon illustrates the instability that can also occur in GAN training.

5.3 INLINE QUESTION 3

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.

5.3.1 Your answer:

This is not a good sign. When the generator loss decreases while the discriminator loss remains consistently high from the beginning, it indicates that the discriminator is not learning properly. In ideal GAN training, the generator and discriminator should compete with each other and maintain a balance. If the discriminator fails to learn effectively, it cannot provide useful feedback to the generator, and consequently, the generator may be under the illusion that it's improving while actually producing low-quality samples. This can lead to mode collapse or a situation where the generator only exploits the weaknesses of the discriminator.

[ ]:

[ ]: