Generative Adversarial Networks (GANs)

So far in CS231N, 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. This has ranged from straightforward classification of image categories to sentence generation (which was still phrased as a classification problem, our labels were in vocabulary space and we'd learned a recurrence to capture multi-word labels). 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.

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:

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

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 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:

$$\underset{G}{\text{maximize}}\ \mathrm{E}_{z\sim p(z)}[\log D(G(z))]$$

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

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

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. Another popular way of training neural networks as generative models is Variational Autoencoders (co-discovered here and here). Variatonal autoencoders combine neural networks with variationl inference to train deep generative models. These models tend to be far more stable and easier to train but currently don't produce samples that are

as pretty as GANs.

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:

Random Noise

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

Hint: use `torch.rand`.

In [4]:

```
def sample_noise(batch_size, dim):
    """
    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).
    """
    # *****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:

In [5]:

```
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!
```

Discriminator

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 $f(x) = \max$ for some fixed constant \alpha; 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.

In [8]:

```
def discriminator():
    """
    Build and return a PyTorch model implementing the architecture above.
    """
    model = nn.Sequential(
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
            Flatten(),
            nn.Linear(784, 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)*****
    )
    return model
```

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

In [9]:

```
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.
```

Generator

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])

In [10]:

```
def generator(noise_dim=NOISE_DIM):
    """
    Build and return a PyTorch model implementing the architecture above.
    """
    model = nn.Sequential(
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            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)*****
    )
    return model
```

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

In [11]:

```
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.
```

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: 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 for you below.

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 example:

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

Instead of computing the expectation of $\log D(G(z))$, $\log D(x)$ and $\log \left(1-D(G(z))\right)$, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing.

In [26]:

```
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.
    """
    N = logits_real.size()
    true_labels = torch.ones(N).type(dtype)
    loss_true = bce_loss(logits_real, true_labels)

    fake_labels = 1 - true_labels
    loss_fake = bce_loss(logits_fake, fake_labels)

    loss = loss_true + loss_fake
    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.
    """
    N = logits_fake.size()
    true_labels = torch.ones(N).type(dtype)
    loss = bce_loss(logits_fake, true_labels)
    return loss
```

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

In [27]:

```
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: 2.83811e-08
```

In [28]:

```
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
```

Optimizing our loss

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.

In [20]:

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

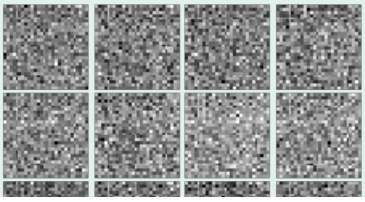
Training a GAN!

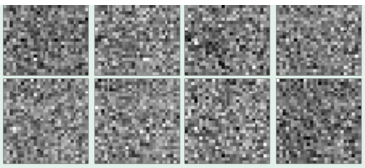
We provide you the main training loop... you won't need to change this function, but we encourage you to read through and understand it.

In [87]:

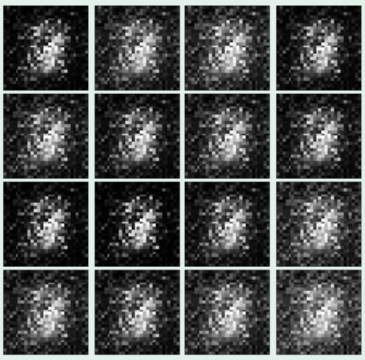
```
# Make the discriminator
D = discriminator().type(dtype)

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

```
Iter: 0, D: 1.328, G:0.7202
```

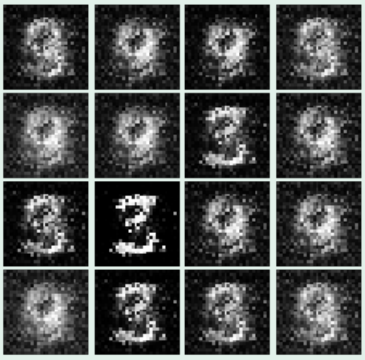




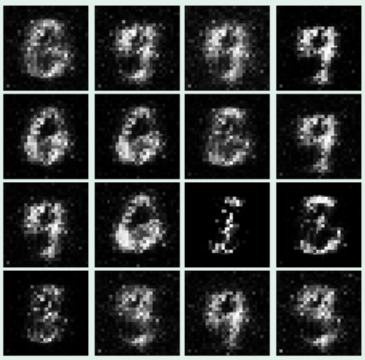
Iter: 250, D: 1.939, G:0.8623



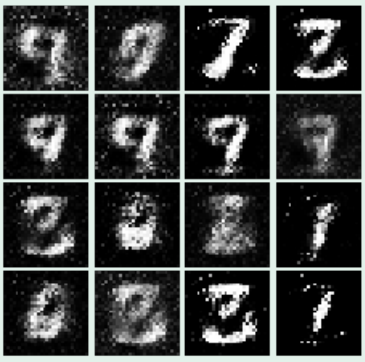
Iter: 500, D: 1.212, G:1.01



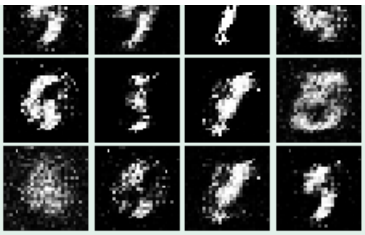
Iter: 750, D: 1.407, G:0.7779



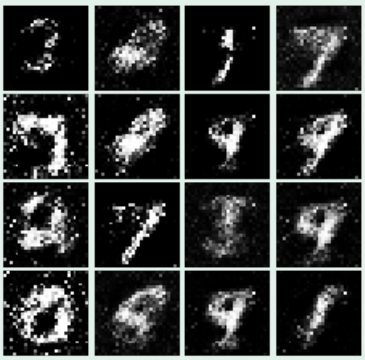
Iter: 1000, D: 1.138, G:1.107



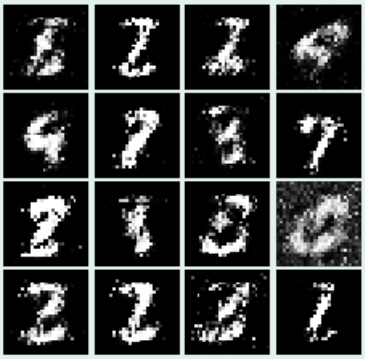
Iter: 1250, D: 1.201, G:0.9872



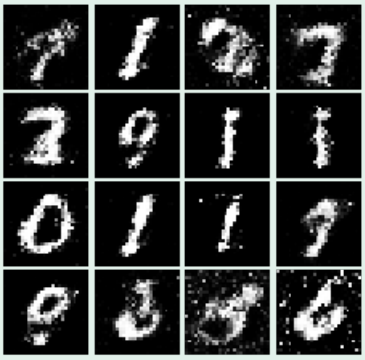
Iter: 1500, D: 1.204, G:0.9861



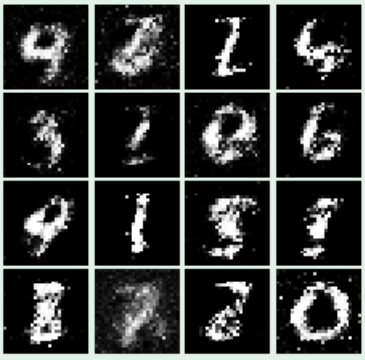
Iter: 1750, D: 1.228, G:0.911



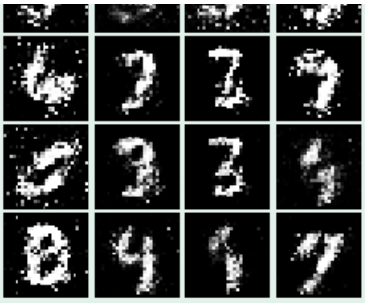
Iter: 2000, D: 1.188, G:0.7323



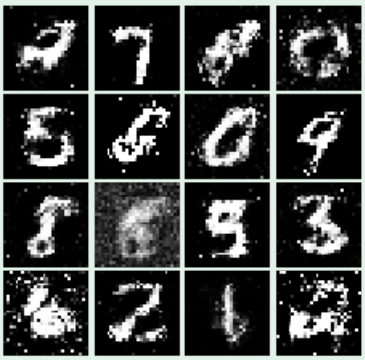
Iter: 2250, D: 1.267, G:0.9026



Iter: 2500, D: 1.444, G:0.7764

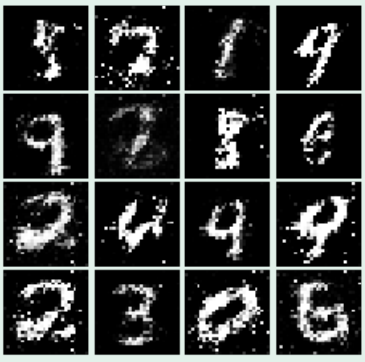


Iter: 2750, D: 1.282, G:0.8262



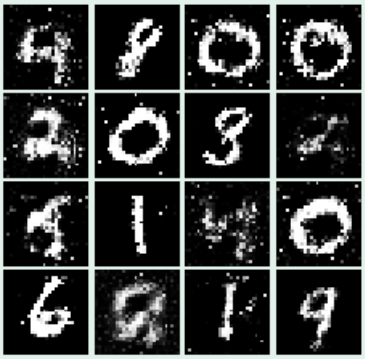
Iter: 3000, D: 1.287, G:0.9628

Iter: 3250, D: 1.312, G:0.8871



Iter: 3500, D: 1.273, G:0.8278

Iter: 3750, D: 1.273, G:0.8697



Least Squares GAN

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`).

In [17]:

```
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)*****

    N = scores_real.size()
    loss = (0.5 * torch.mean((scores_real - 1)**2)) + (0.5 * torch.mean(scores_fake**2))

    # *****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 = 0.5 * torch.mean((scores_fake - 1) ** 2)
    return loss
```

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

In [18]:

```
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'])
```

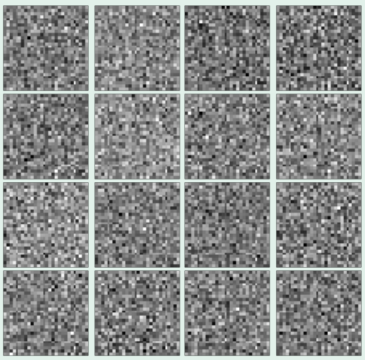
```
Maximum error in d_loss: 1.64377e-08
Maximum error in g_loss: 3.36961e-08
```

Run the following cell to train your model!

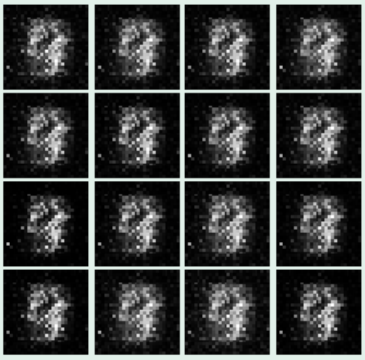
In [22]:

```
D_LS = discriminator().type(dtype)
G_LS = generator().type(dtype)

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

run_a_gan(D_LS, G_LS, D_LS_solver, G_LS_solver, ls_discriminator_loss, ls_generator_loss)
```

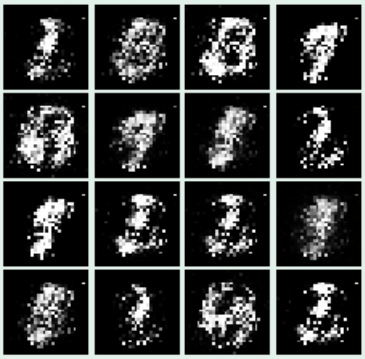
```
Iter: 0, D: 0.476, G:0.4826
```



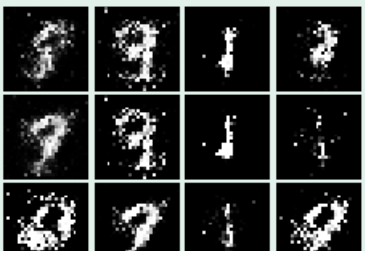
```
Iter: 250, D: 0.1693, G:0.3271
```

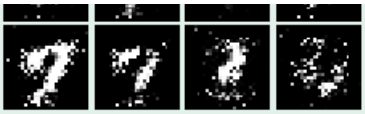


```
Iter: 500, D: 0.1771, G:0.8023
```

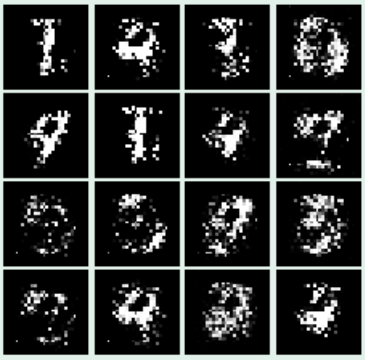


```
Iter: 750, D: 0.1909, G:0.1692
```

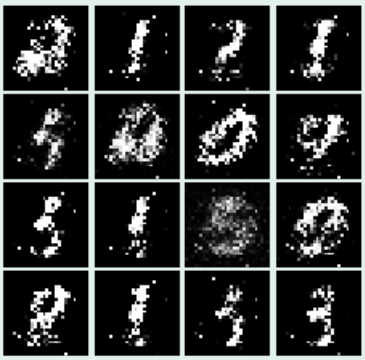




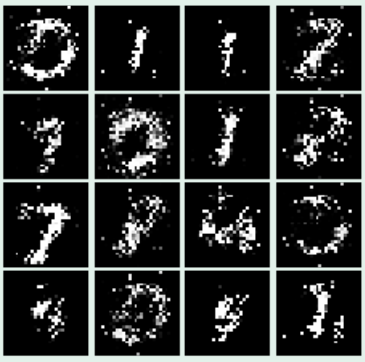
Iter: 1000, D: 0.1256, G:0.3286



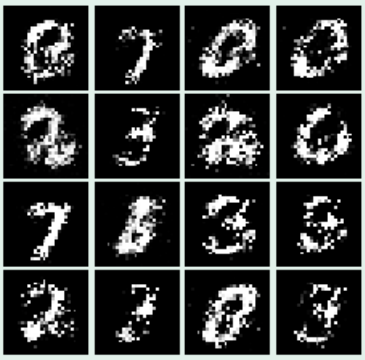
Iter: 1250, D: 0.1562, G:0.2851



Iter: 1500, D: 0.2746, G:0.12

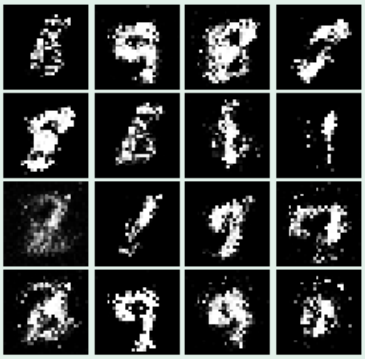


Iter: 1750, D: 0.221, G:0.1799

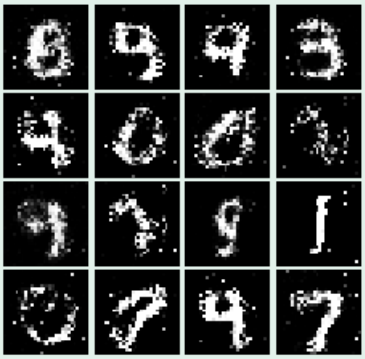


Iter: 2000, D: 0.2323, G:0.1965

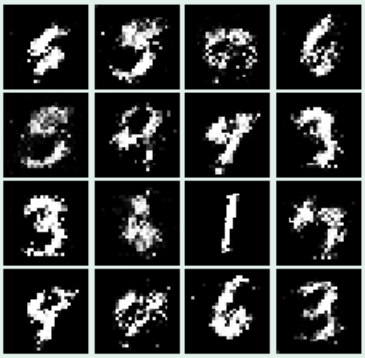
Iter: 2250, D: 0.2279, G:0.1575



Iter: 2500, D: 0.2109, G:0.1929



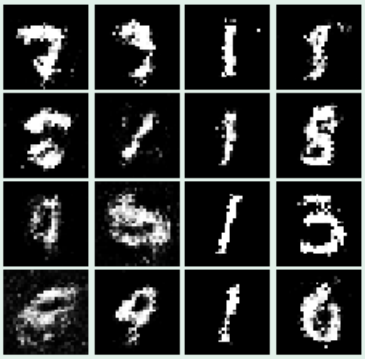
Iter: 2750, D: 0.2293, G:0.158



Iter: 3000, D: 0.2388, G:0.1622

Iter: 3250, D: 0.2377, G:0.1619

```
Iter: 3500, D: 0.2396, G:0.1667
```



```
Iter: 3750, D: 0.218, G:0.1694
```

Deeply 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

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.

- Reshape into image tensor (Use Unflatten!)
- 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

In [91]:

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

```
    the architecture above.
    """
    return nn.Sequential(
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        Unflatten(batch_size, 1, 28, 28),
        nn.Conv2d(1, 32, kernel_size = 5, stride = 1),
        nn.LeakyReLU(0.01),
        nn.MaxPool2d(kernel_size = 2, stride = 2),
        nn.Conv2d(32, 64, kernel_size = 5, stride = 1),
        nn.LeakyReLU(0.01),
        nn.MaxPool2d(kernel_size = 2, stride = 2),
        Flatten(),
        nn.Linear(4*4*64, 4*4*64),
        nn.LeakyReLU(0.01),
        nn.Linear(4*4*64, 1)

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

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:

In [92]:

```
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 generator. Check your achitecture.')
    else:
        print('Correct number of parameters in generator.')

test_dc_classifer()
```

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

Generator

For the generator, we will copy the architecture exactly from the InfoGAN paper. See Appendix C.1 MNIST. See the documentation for tf.nn.conv2d_transpose. 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
- Reshape into Image Tensor of shape 7, 7, 128
- Conv2D^T (Transpose): 64 filters of 4x4, stride 2, 'same' padding (use `padding=1`)
- `ReLU`
- BatchNorm
- Conv2D^T (Transpose): 1 filter of 4x4, stride 2, 'same' padding (use `padding=1`)
- `TanH`
- Should have a 28x28x1 image, reshape back into 784 vector

In [93]:

```
def build_dc_generator(noise_dim=NOISE_DIM):
    """
    Build and return a PyTorch model implementing the DCGAN generator using
    the architecture described above.
    """
    return nn.Sequential(
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
        nn.Linear(noise_dim, 1024),
        nn.ReLU(),
        nn.BatchNorm1d(1024),
        nn.Linear(1024, 7*7*128),
        nn.ReLU(),
        nn.BatchNorm1d(7*7*128),
        Unflatten(batch_size, 128, 7, 7),
        nn.ConvTranspose2d(128, 64, kernel_size = 4, stride = 2, padding = 1),
        nn.ReLU(),
        nn.BatchNorm2d(64),
        nn.ConvTranspose2d(64, 1, kernel_size = 4, stride = 2, padding = 1),
        nn.Tanh(),
        Flatten()

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

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()
```

Out[93]:

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

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

In [94]:

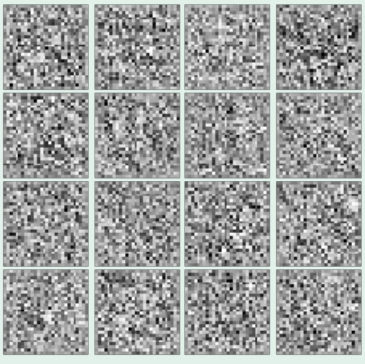
```
def test_dc_generator(true_count=6580801):
    model = build_dc_generator(4)
    cur_count = count_params(model)
    if cur_count != true_count:
        print('Incorrect number of parameters in generator. Check your achitecture.')
    else:
        print('Correct number of parameters in generator.')

test_dc_generator()
```

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

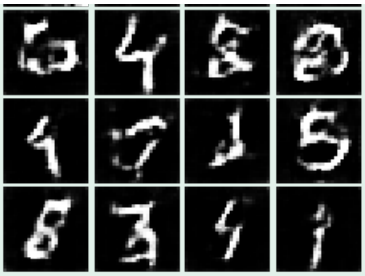
In [95]:

```
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)

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

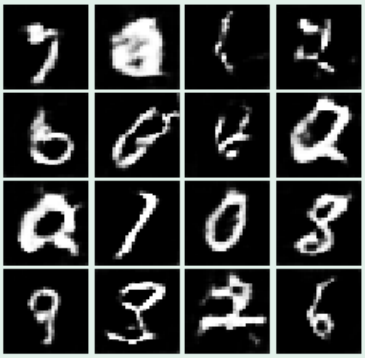
```
Iter: 0, D: 1.541, G:0.5609
```



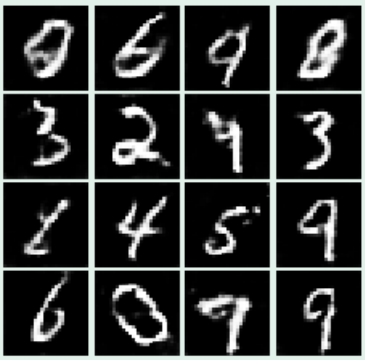
```
Iter: 250, D: 1.448, G:0.2154
```



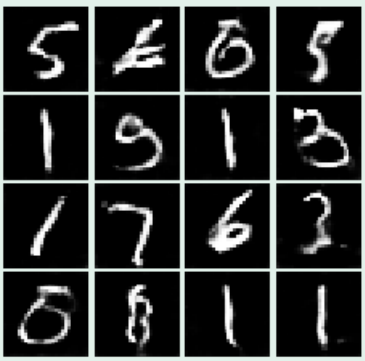
Iter: 500, D: 1.349, G:1.197



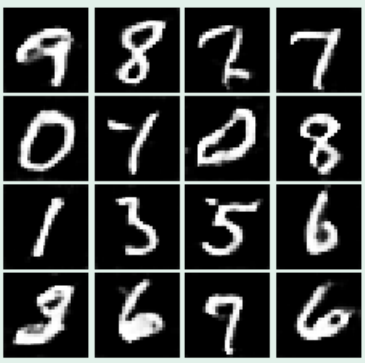
Iter: 750, D: 1.273, G:1.437



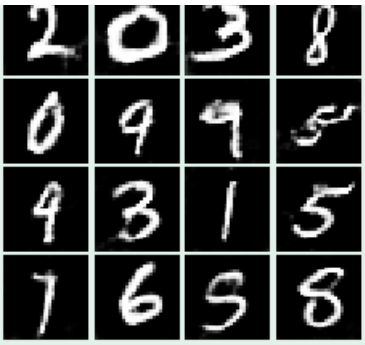
Iter: 1000, D: 1.214, G:1.03



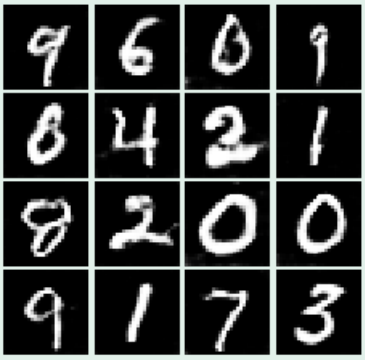
Iter: 1250, D: 1.208, G:1.206



Iter: 1500, D: 1.102, G:0.9956



```
Iter: 1750, D: 1.223, G:1.101
```



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.

Your answer:

y_0	y_1	y_2	y_3	y_4	y_5	y_6
1	2	0	0	0	0	0
x_0	x_1	x_2	x_3	x_4	x_5	x_6
1	-2	0	0	0	0	0

$\min_x\max_y f(x, y)$ evaluates to 0. As we can see from the table above, within 6 steps we get the answer.

INLINE QUESTION 2

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

Your answer:

Yes. We can see we got the same answers after y2, after two steps we reach the optimal value and remain unchanged.

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.

Your answer:

It is not a good sign. If generator loss decreases and discriminator loss stays at a constant high value, it means discriminator can't catch up with generator. In this case, discriminator is decreasing the probability of real data being considered to be real while generator is generating data. But we actually want them to reach an equilibrium. So this is not a good sign.

In [ ]:

```
```