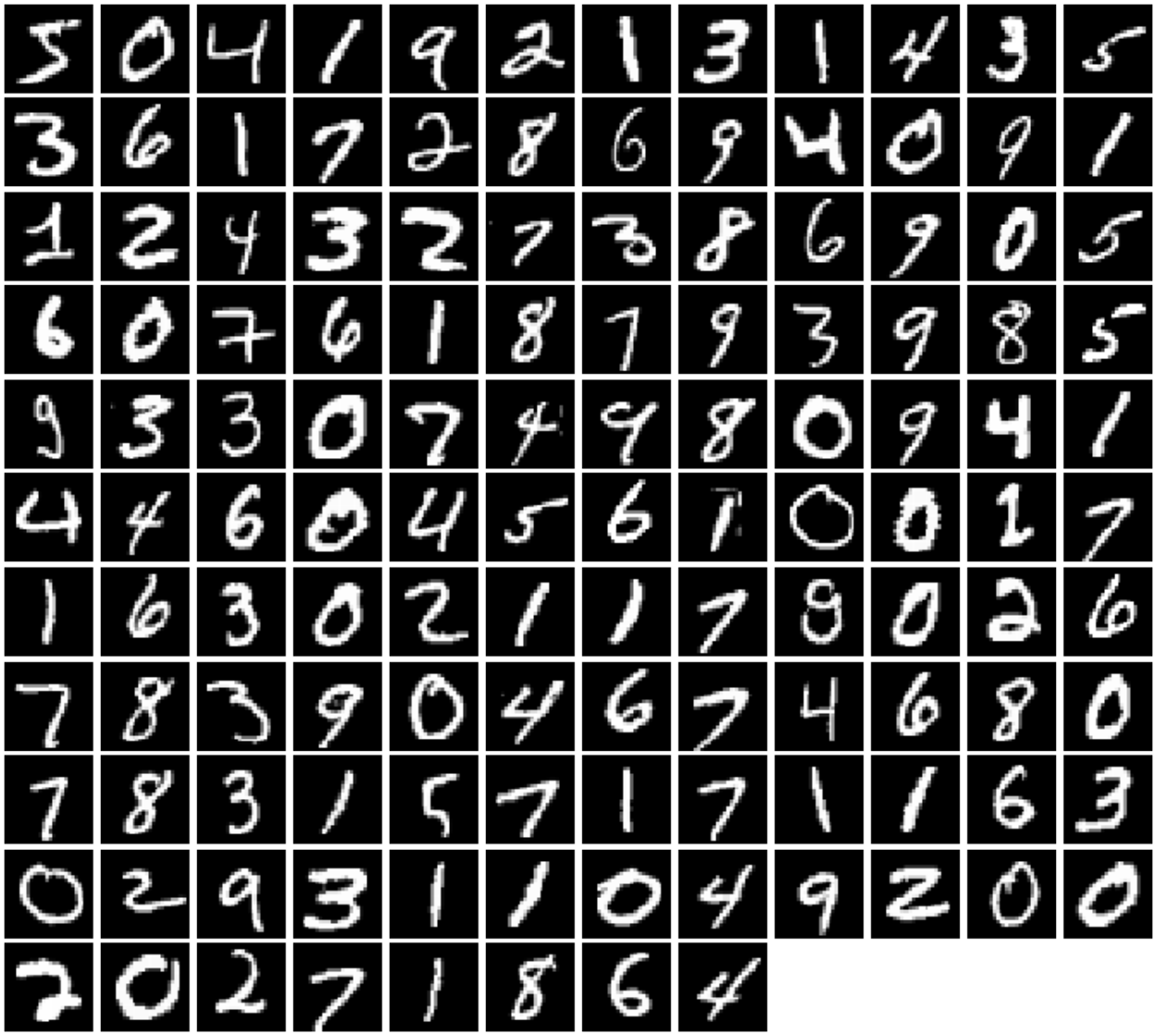
生成对抗网络

关于生成对抗网络，出现了很多变形，比如 WGAN，LS-GAN 等等，这一节我们只使用 mnist 举一些简单的例子来说明，更复杂的网络结构可以再 github 上找到相应的实现

```
import torch
from torch import nn
from torch.autograd import Variable

import torchvision.transforms as tfs
from torch.utils.data import DataLoader, sampler
from torchvision.datasets import MNIST

import numpy as np

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

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # 设置画图的尺寸
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

def show_images(images): # 定义画图工具
    images = np.reshape(images, [images.shape[0], -1])
    sqrtn = int(np.ceil(np.sqrt(images.shape[0])))
    sqrtimg = int(np.ceil(np.sqrt(images.shape[1])))

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

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

def preprocess_img(x):
    x = tfs.ToTensor()(x)
```

```
    return (x - 0.5) / 0.5

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

```
class ChunkSampler(sampler.Sampler): # 定义一个取样的函数
    """Samples elements sequentially from some offset.
    Arguments:
        num_samples: # of desired datapoints
        start: offset where we should start selecting from
    """
    def __init__(self, num_samples, start=0):
        self.num_samples = num_samples
        self.start = start

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

    def __len__(self):
        return self.num_samples

NUM_TRAIN = 50000
NUM_VAL = 5000

NOISE_DIM = 96
batch_size = 128

train_set = MNIST('./mnist', train=True, download=True, transform=preprocess_img)

train_data = DataLoader(train_set, batch_size=batch_size,
sampler=ChunkSampler(NUM_TRAIN, 0))

val_set = MNIST('./mnist', train=True, download=True, transform=preprocess_img)

val_data = DataLoader(val_set, batch_size=batch_size, sampler=ChunkSampler(NUM_VAL,
NUM_TRAIN))


imgs = deprocess_img(train_data.__iter__().next()[0].view(batch_size,
784)).numpy().squeeze() # 可视化图片效果
show_images(imgs)
```



简单版本的生成对抗网络

通过前面我们知道生成对抗网络有两个部分构成，一个是生成网络，一个是对抗网络，我们首先写一个简单版本的网络结构，生成网络和对抗网络都是简单的多层神经网络

判别网络

判别网络的结构非常简单，就是一个二分类器，结构如下:

- 全连接(784 -> 256)
- leakyrelu，α 是 0.2
- 全连接(256 -> 256)
- leakyrelu, α 是 0.2
- 全连接(256 -> 1)

其中 leakyrelu 是指 f(x) = max(α x, x)

```
def discriminator():
    net = nn.Sequential(
            nn.Linear(784, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1)
        )
    return net
```

生成网络

接下来我们看看生成网络，生成网络的结构也很简单，就是根据一个随机噪声生成一个和数据维度一样的张量，结构如下：

- 全连接(噪音维度 -> 1024)
- relu
- 全连接(1024 -> 1024)
- relu
- 全连接(1024 -> 784)
- tanh 将数据裁剪到 -1 ~ 1 之间

```
def generator(noise_dim=NOISE_DIM):
    net = nn.Sequential(
        nn.Linear(noise_dim, 1024),
        nn.ReLU(True),
        nn.Linear(1024, 1024),
        nn.ReLU(True),
        nn.Linear(1024, 784),
        nn.Tanh()
    )
    return net
```

接下来我们需要定义生成对抗网络的 loss，通过前面的讲解我们知道，对于对抗网络，相当于二分类问题，将真的判别为真的，假的判别为假的，作为辅助，可以参考一下论文中公式

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

而对于生成网络，需要去骗过对抗网络，也就是将假的也判断为真的，作为辅助，可以参考一下论文中公式

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

如果你还记得前面的二分类 loss，那么你就会发现上面这两个公式就是二分类 loss

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

如果我们把 D(x) 看成真实数据的分类得分，那么 D(G(z)) 就是假数据的分类得分，所以上面判别器的 loss 就是将真实数据的得分判断为 1，假的数据的得分判断为 0，而生成器的 loss 就是将假的数据判断为 1

下面我们来实现一下

```
bce_loss = nn.BCEWithLogitsLoss()

def discriminator_loss(logits_real, logits_fake): # 判别器的 loss
    size = logits_real.shape[0]
    true_labels = Variable(torch.ones(size, 1)).float().cuda()
    false_labels = Variable(torch.zeros(size, 1)).float().cuda()
    loss = bce_loss(logits_real, true_labels) + bce_loss(logits_fake, false_labels)
    return loss
```

```
def generator_loss(logits_fake): # 生成器的 loss
    size = logits_fake.shape[0]
    true_labels = Variable(torch.ones(size, 1)).float().cuda()
    loss = bce_loss(logits_fake, true_labels)
    return loss
```

```
# 使用 adam 来进行训练，学习率是 3e-4，beta1 是 0.5，beta2 是 0.999
def get_optimizer(net):
    optimizer = torch.optim.Adam(net.parameters(), lr=3e-4, betas=(0.5, 0.999))
    return optimizer
```

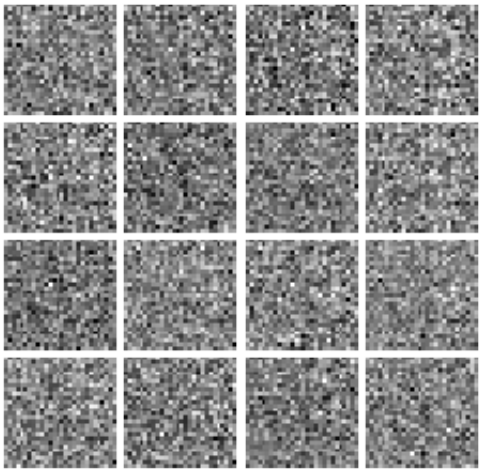
下面我们开始训练一个这个简单的生成对抗网络

```
def train_a_gan(D_net, G_net, D_optimizer, G_optimizer, discriminator_loss,
generator_loss, show_every=250,
                noise_size=96, num_epochs=10):
    iter_count = 0
    for epoch in range(num_epochs):
        for x, _ in train_data:
            bs = x.shape[0]
            # 判别网络
            real_data = Variable(x).view(bs, -1).cuda() # 真实数据
            logits_real = D_net(real_data) # 判别网络得分

            sample_noise = (torch.rand(bs, noise_size) - 0.5) / 0.5 # -1 ~ 1 的均匀分
布
            g_fake_seed = Variable(sample_noise).cuda()
            fake_images = G_net(g_fake_seed) # 生成的假的数据
            logits_fake = D_net(fake_images) # 判别网络得分
```

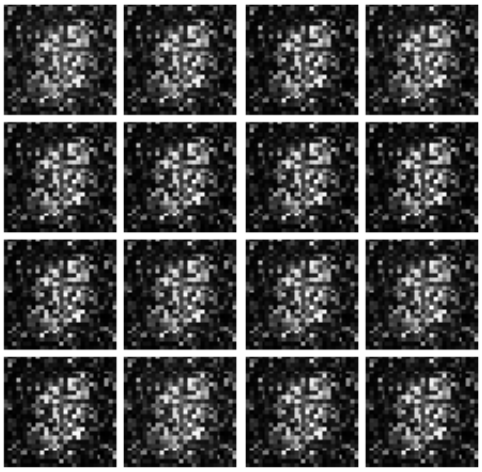
```
            d_total_error = discriminator_loss(logits_real, logits_fake) # 判别器的
loss
            D_optimizer.zero_grad()
            d_total_error.backward()
            D_optimizer.step() # 优化判别网络

            # 生成网络
            g_fake_seed = Variable(sample_noise).cuda()
            fake_images = G_net(g_fake_seed) # 生成的假的数据

            gen_logits_fake = D_net(fake_images)
            g_error = generator_loss(gen_logits_fake) # 生成网络的 loss
            G_optimizer.zero_grad()
            g_error.backward()
            G_optimizer.step() # 优化生成网络

            if (iter_count % show_every == 0):
                print('Iter: {}, D: {:.4}, G:{:.4}'.format(iter_count,
d_total_error.data[0], g_error.data[0]))
                imgs_numpy = deprocess_img(fake_images.data.cpu().numpy())
                show_images(imgs_numpy[0:16])
                plt.show()
                print()
            iter_count += 1
```

```
D = discriminator().cuda()
G = generator().cuda()

D_optim = get_optimizer(D)
G_optim = get_optimizer(G)

train_a_gan(D, G, D_optim, G_optim, discriminator_loss, generator_loss)
```

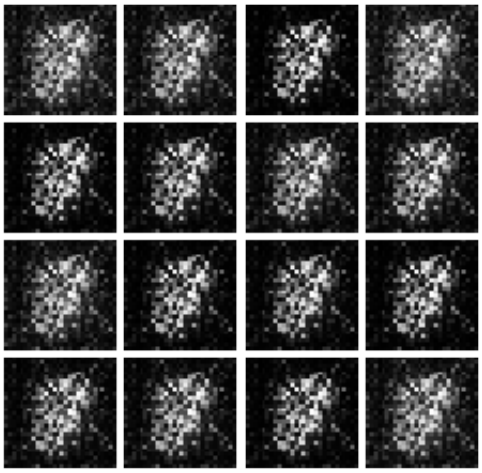
```
Iter: 0, D: 1.364, G:0.6648
```



```
Iter: 250, D: 1.362, G:0.8941
```

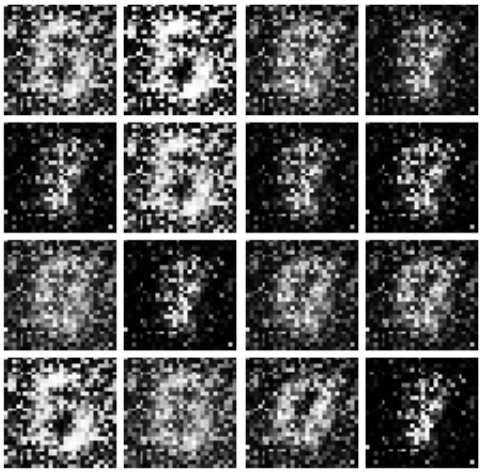


```
Iter: 500, D: 0.9882, G:1.22
```

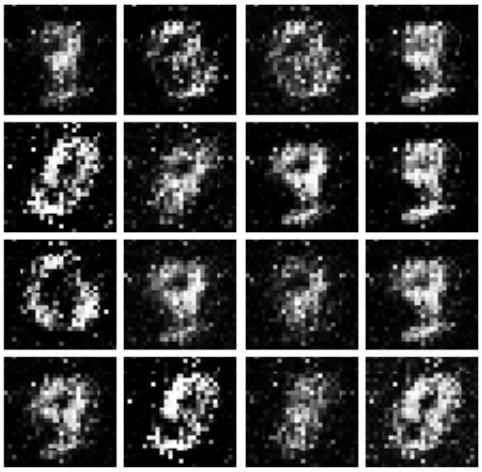


```
Iter: 750, D: 0.6571, G:1.987
```

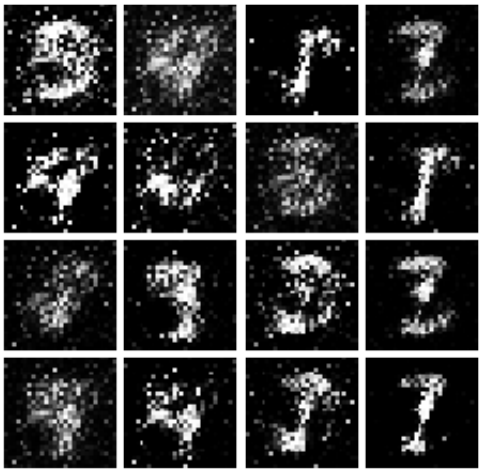
```
Iter: 1000, D: 1.359, G:1.354
```



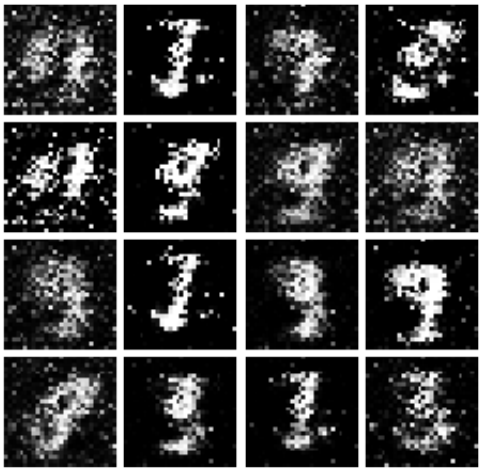
```
Iter: 1250, D: 0.826, G:1.92
```



```
Iter: 1500, D: 0.9988, G:2.19
```

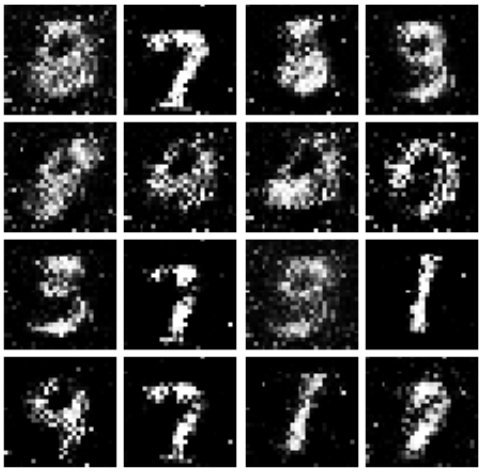


```
Iter: 1750, D: 0.8991, G:2.082
```

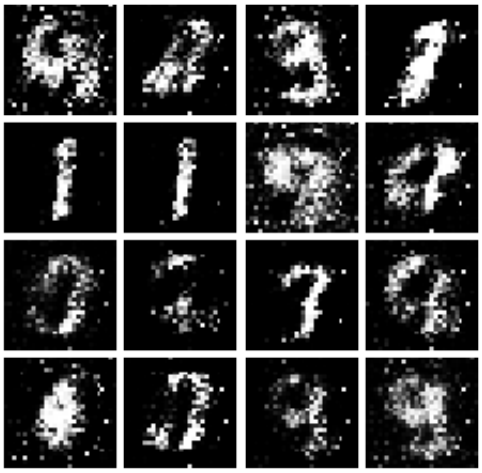


```
Iter: 2000, D: 0.8586, G:1.804
```

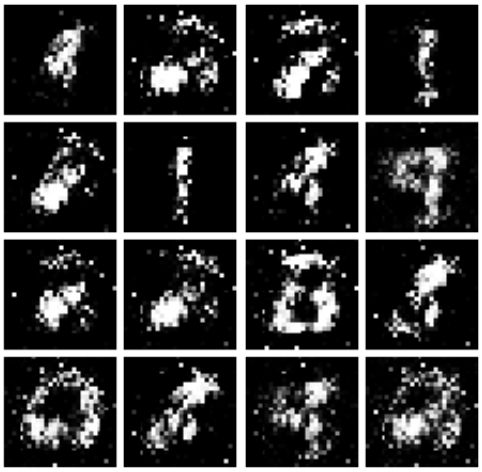
```
Iter: 2250, D: 0.749, G:1.984
```



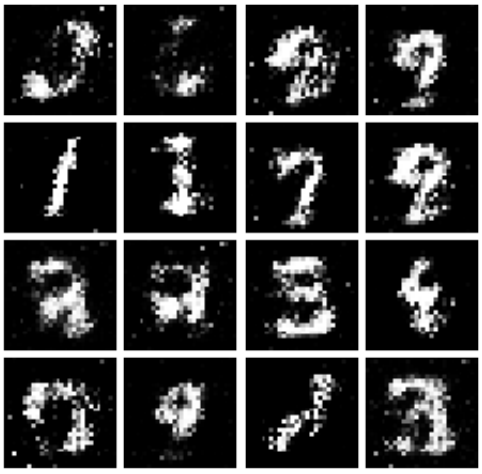
```
Iter: 2500, D: 0.9037, G:1.484
```



```
Iter: 2750, D: 1.115, G:1.097
```



```
Iter: 3000, D: 1.064, G:1.385
```



```
Iter: 3250, D: 1.177, G:1.115
```

```
Iter: 3500, D: 1.05, G:1.166
```

```
Iter: 3750, D: 1.23, G:0.9212
```

我们已经完成了一个简单的生成对抗网络，是不是非常容易呢。但是可以看到效果并不是特别好，生成的数字也不是特别完整，因为我们仅仅使用了简单的多层全连接网络。

除了这种最基本的生成对抗网络之外，还有很多生成对抗网络的变式，有结构上的变式，也有 loss 上的变式，我们先讲一讲其中一种在 loss 上的变式，Least Squares GAN

Least Squares GAN

Least Squares GAN 比最原始的 GANs 的 loss 更加稳定，通过名字我们也能够看出这种 GAN 是通过最小平方误差来进行估计，而不是通过二分类的损失函数，下面我们看看 loss 的计算公式

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

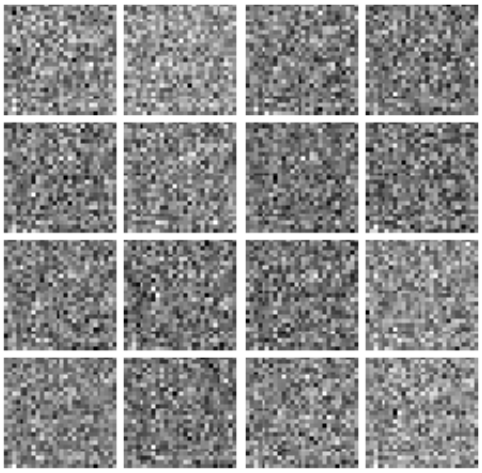
$$\ell_D = \frac{1}{2}\mathbb{E}_{x \sim p_{\text{data}}}\left[(D(x)-1)^2\right] + \frac{1}{2}\mathbb{E}_{z \sim p(z)}\left[(D(G(z)))^2\right]$$

可以看到 Least Squares GAN 通过最小二乘代替了二分类的 loss，下面我们定义一下 loss 函数

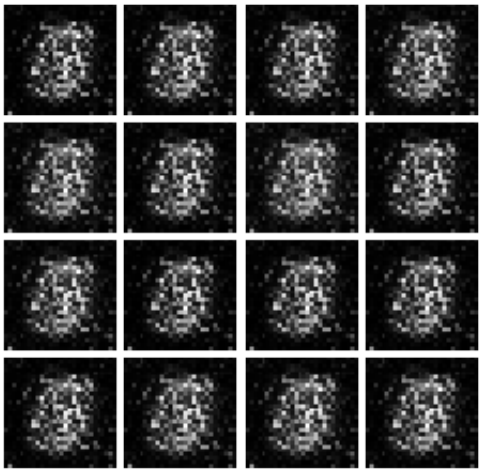
```
def ls_discriminator_loss(scores_real, scores_fake):
    loss = 0.5 * ((scores_real - 1) ** 2).mean() + 0.5 * (scores_fake ** 2).mean()
    return loss

def ls_generator_loss(scores_fake):
    loss = 0.5 * ((scores_fake - 1) ** 2).mean()
    return loss
```

```
D = discriminator().cuda()
G = generator().cuda()

D_optim = get_optimizer(D)
G_optim = get_optimizer(G)

train_a_gan(D, G, D_optim, G_optim, ls_discriminator_loss, ls_generator_loss)
```

```
Iter: 0, D: 0.5524, G:0.4728
```

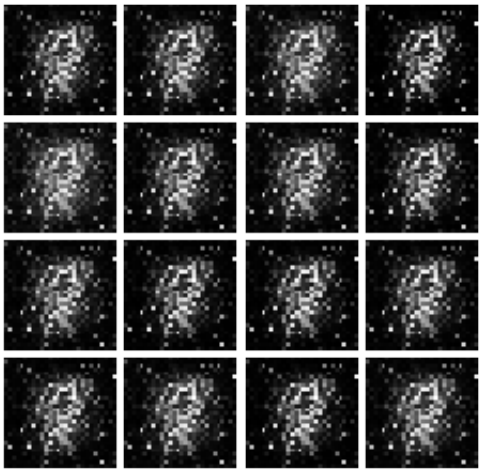


```
Iter: 250, D: 0.2155, G:0.1959
```

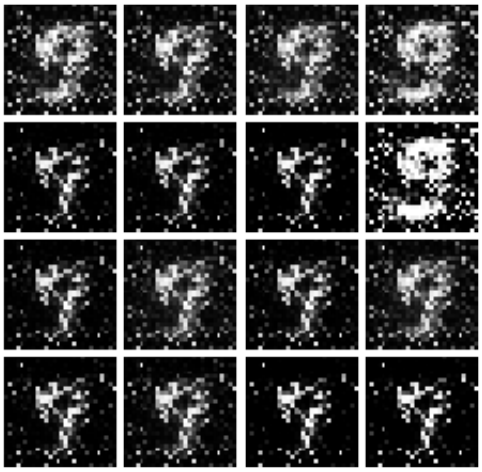


```
Iter: 500, D: 0.07639, G:0.6031
```

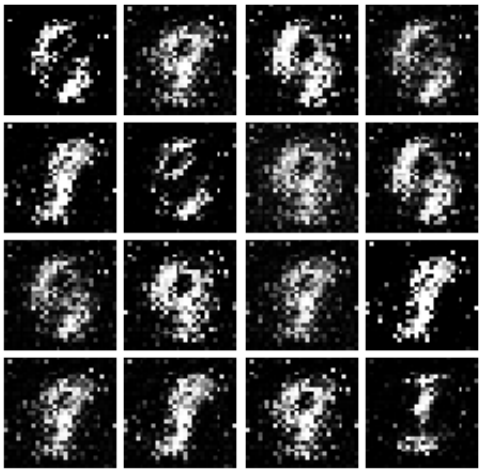
```
Iter: 750, D: 0.08781, G:0.5468
```



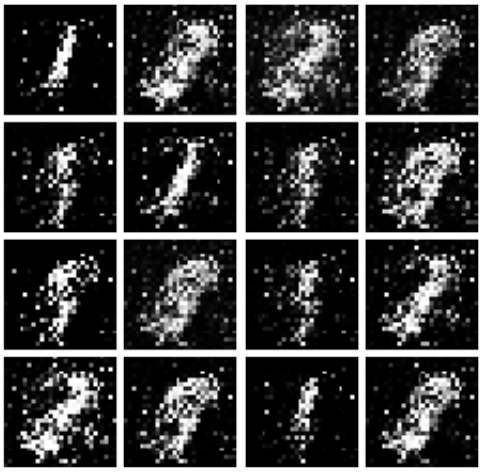
```
Iter: 1000, D: 0.621, G:0.2852
```



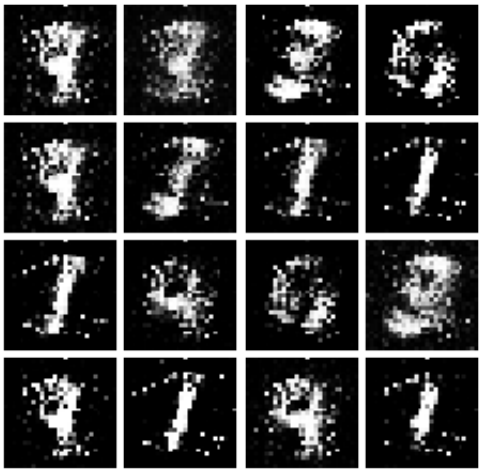
```
Iter: 1250, D: 0.08534, G:0.4106
```



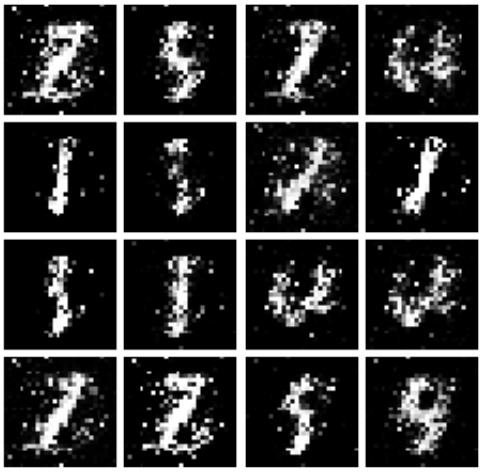
```
Iter: 1500, D: 0.08328, G:0.5062
```



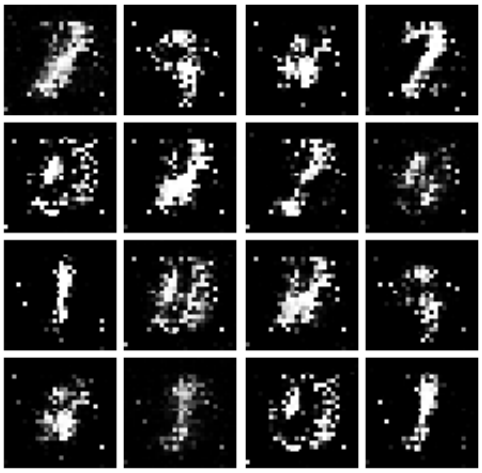
```
Iter: 1750, D: 0.1538, G:0.2972
```



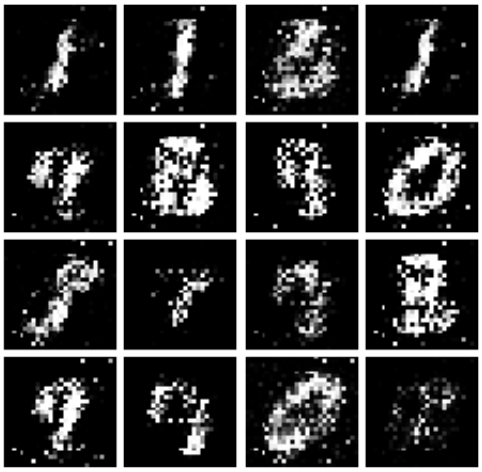
```
Iter: 2000, D: 0.07425, G:0.4241
```



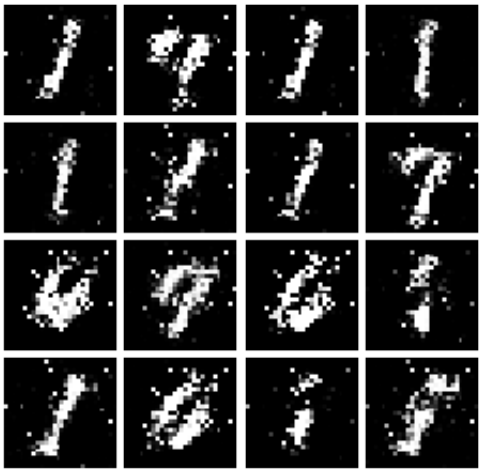
```
Iter: 2250, D: 0.05706, G:0.4568
```



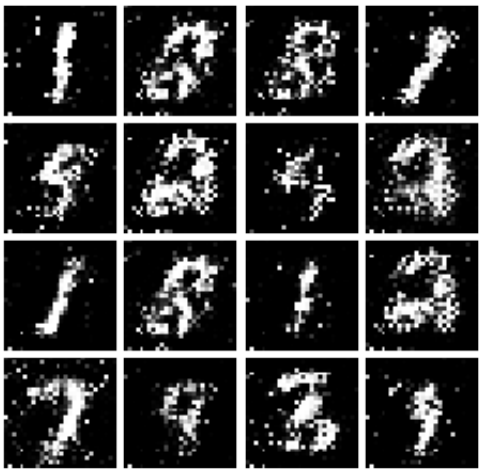
```
Iter: 2500, D: 0.0743, G:0.4576
```



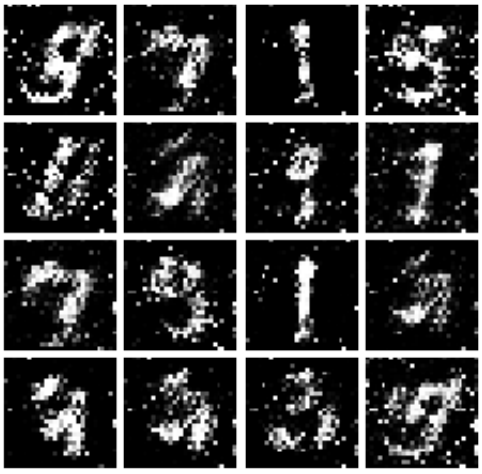
```
Iter: 2750, D: 0.1021, G:0.3624
```



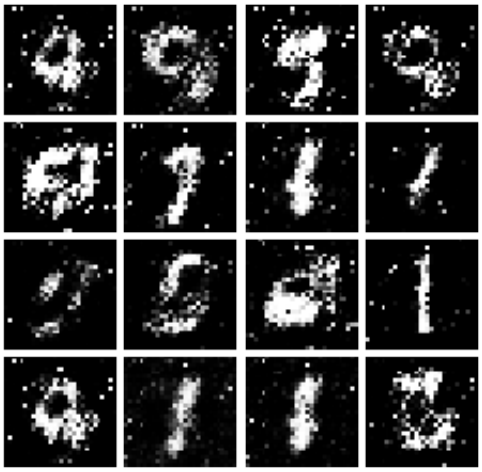
```
Iter: 3000, D: 0.128, G:0.2866
```



```
Iter: 3250, D: 0.1331, G:0.3467
```



```
Iter: 3500, D: 0.1186, G:0.3989
```



```
Iter: 3750, D: 0.1621, G:0.228
```

上面我们讲了 最基本的 GAN 和 least squares GAN，最后我们讲一讲使用卷积网络的 GAN，叫做深度卷积生成对抗网络

Deep Convolutional GANs

深度卷积生成对抗网络特别简单，就是将生成网络和对抗网络都改成了卷积网络的形式，下面我们来实现一下

卷积判别网络

卷积判别网络就是一个一般的卷积网络，结构如下

- 32 Filters, 5x5, Stride 1, Leaky ReLU(alpha=0.01)
- Max Pool 2x2, Stride 2
- 64 Filters, 5x5, Stride 1, Leaky ReLU(alpha=0.01)
- Max Pool 2x2, Stride 2
- Fully Connected size 4 x 4 x 64, Leaky ReLU(alpha=0.01)
- Fully Connected size 1

```
class build_dc_classifier(nn.Module):
    def __init__(self):
        super(build_dc_classifier, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 32, 5, 1),
            nn.LeakyReLU(0.01),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(32, 64, 5, 1),
            nn.LeakyReLU(0.01),
            nn.MaxPool2d(2, 2)
        )
        self.fc = nn.Sequential(
```

```
            nn.Linear(1024, 1024),
            nn.LeakyReLU(0.01),
            nn.Linear(1024, 1)
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.shape[0], -1)
        x = self.fc(x)
        return x
```

卷积生成网络

卷积生成网络需要将一个低维的噪声向量变成一个图片数据，结构如下

- Fully connected of size 1024, ReLU
- BatchNorm
- Fully connected of size 7 x 7 x 128, ReLU
- BatchNorm
- Reshape into Image Tensor
- 64 conv2d^T filters of 4x4, stride 2, padding 1, ReLU
- BatchNorm
- 1 conv2d^T filter of 4x4, stride 2, padding 1, TanH

```
class build_dc_generator(nn.Module):
    def __init__(self, noise_dim=NOISE_DIM):
        super(build_dc_generator, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(noise_dim, 1024),
            nn.ReLU(True),
            nn.BatchNorm1d(1024),
            nn.Linear(1024, 7 * 7 * 128),
            nn.ReLU(True),
            nn.BatchNorm1d(7 * 7 * 128)
        )

        self.conv = nn.Sequential(
            nn.ConvTranspose2d(128, 64, 4, 2, padding=1),
            nn.ReLU(True),
            nn.BatchNorm2d(64),
            nn.ConvTranspose2d(64, 1, 4, 2, padding=1),
            nn.Tanh()
        )

    def forward(self, x):
```

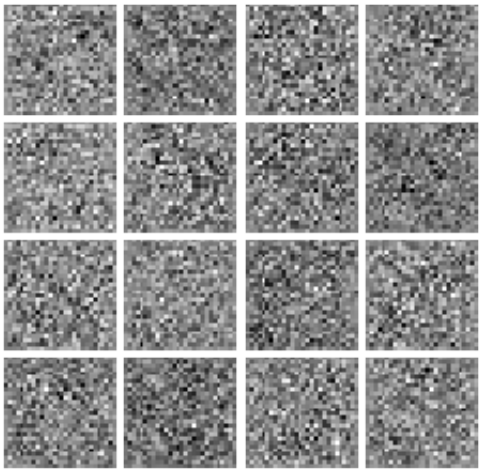
```
        x = self.fc(x)
        x = x.view(x.shape[0], 128, 7, 7) # reshape 通道是 128，大小是 7x7
        x = self.conv(x)
        return x
```

```
def train_dc_gan(D_net, G_net, D_optimizer, G_optimizer, discriminator_loss,
generator_loss, show_every=250,
                 noise_size=96, num_epochs=10):
    iter_count = 0
    for epoch in range(num_epochs):
        for x, _ in train_data:
            bs = x.shape[0]
            # 判别网络
            real_data = Variable(x).cuda() # 真实数据
            logits_real = D_net(real_data) # 判别网络得分

            sample_noise = (torch.rand(bs, noise_size) - 0.5) / 0.5 # -1 ~ 1 的均匀分
布
            g_fake_seed = Variable(sample_noise).cuda()
            fake_images = G_net(g_fake_seed) # 生成的假的数据
            logits_fake = D_net(fake_images) # 判别网络得分

            d_total_error = discriminator_loss(logits_real, logits_fake) # 判别器的
loss
            D_optimizer.zero_grad()
            d_total_error.backward()
            D_optimizer.step() # 优化判别网络

            # 生成网络
            g_fake_seed = Variable(sample_noise).cuda()
            fake_images = G_net(g_fake_seed) # 生成的假的数据

            gen_logits_fake = D_net(fake_images)
            g_error = generator_loss(gen_logits_fake) # 生成网络的 loss
            G_optimizer.zero_grad()
            g_error.backward()
            G_optimizer.step() # 优化生成网络

            if (iter_count % show_every == 0):
                print('Iter: {}, D: {:.4}, G:{:.4}'.format(iter_count,
d_total_error.data[0], g_error.data[0]))
                imgs_numpy = deprocess_img(fake_images.data.cpu().numpy())
                show_images(imgs_numpy[0:16])
                plt.show()
                print()
```

```
            iter_count += 1
```

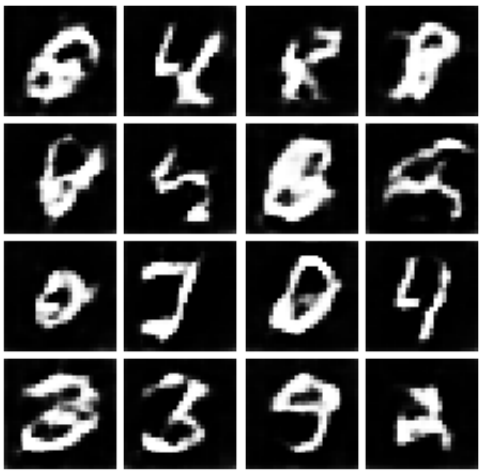
```
D_DC = build_dc_classifier().cuda()
G_DC = build_dc_generator().cuda()

D_DC_optim = get_optimizer(D_DC)
G_DC_optim = get_optimizer(G_DC)

train_dc_gan(D_DC, G_DC, D_DC_optim, G_DC_optim, discriminator_loss, generator_loss,
num_epochs=5)
```

```
Iter: 0, D: 1.387, G:0.6381
```

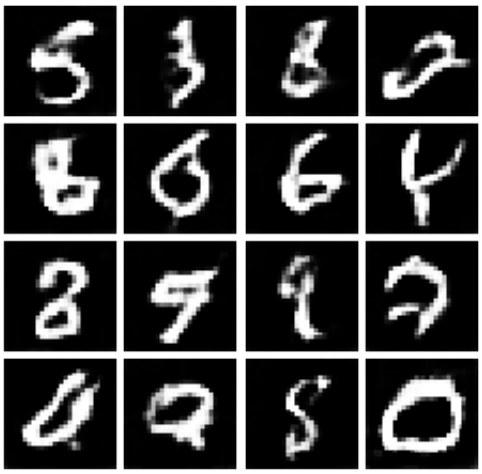


```
Iter: 250, D: 0.7821, G:1.807
```

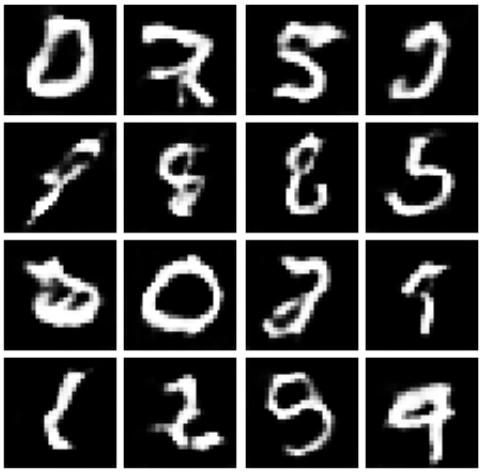
```
Iter: 500, D: 1.058, G:0.8468
```



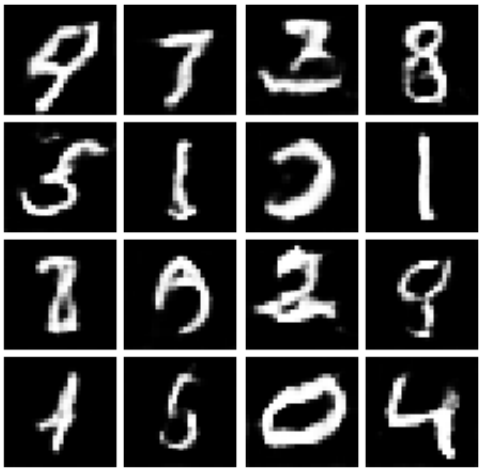
```
Iter: 750, D: 1.163, G:0.8711
```



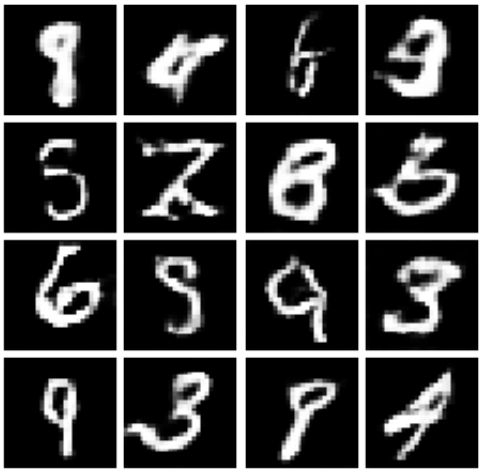
```
Iter: 1000, D: 1.293, G:1.201
```



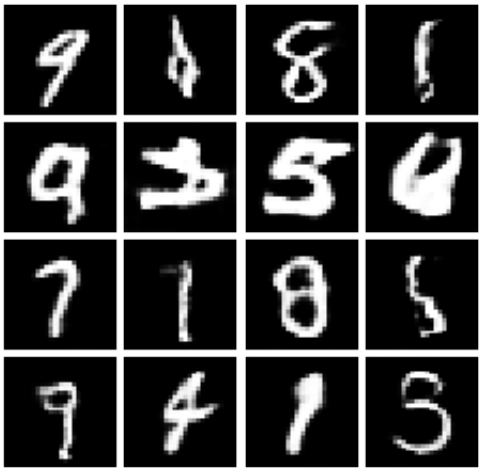
```
Iter: 1250, D: 1.182, G:0.961
```



```
Iter: 1500, D: 1.216, G:0.7218
```



```
Iter: 1750, D: 1.143, G:1.092
```



可以看到，通过 DCGANs 能够得到更加清楚的结果