9_GAN_students

July 13, 2019

1 Hands-On Machine Learning

1.1 Session 9: Generative Adversarial networks

by Christoph Haarburger, Laxmi Gupta

1.1.1 Goals of this Session

In this session you will... * learn how to implement a vanilla "Goodfellow-like" generative adversarial network * train a GAN * learn how to implement a convolutional GAN

We'll be working with the MNIST dataset, which you already know from previous sessions.

```
In [1]: import torch
        import torch.nn as nn
        import torchvision
        import matplotlib.pyplot as plt
        import numpy as np
        import os
        os.environ['CUDA_VISIBLE_DEVICES'] = '6'
        %matplotlib inline
```

```
In [2]: torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False
        torch.manual_seed(0)
        np.random.seed(0)
```

1.2 Hyperparameters

```
In [3]: latent_size = 100
        hidden_size = 256
        image_size = 784
        num_epochs = 30
```

Task: Set up the necessary transforms that are applied to the input images. You've already done this in the previous sessions.

We'd like to transform our data to `Tensors` and normalize it with 0.5 mean and 0.5 variance.

After defining the transforms, we can initialize the dataset.

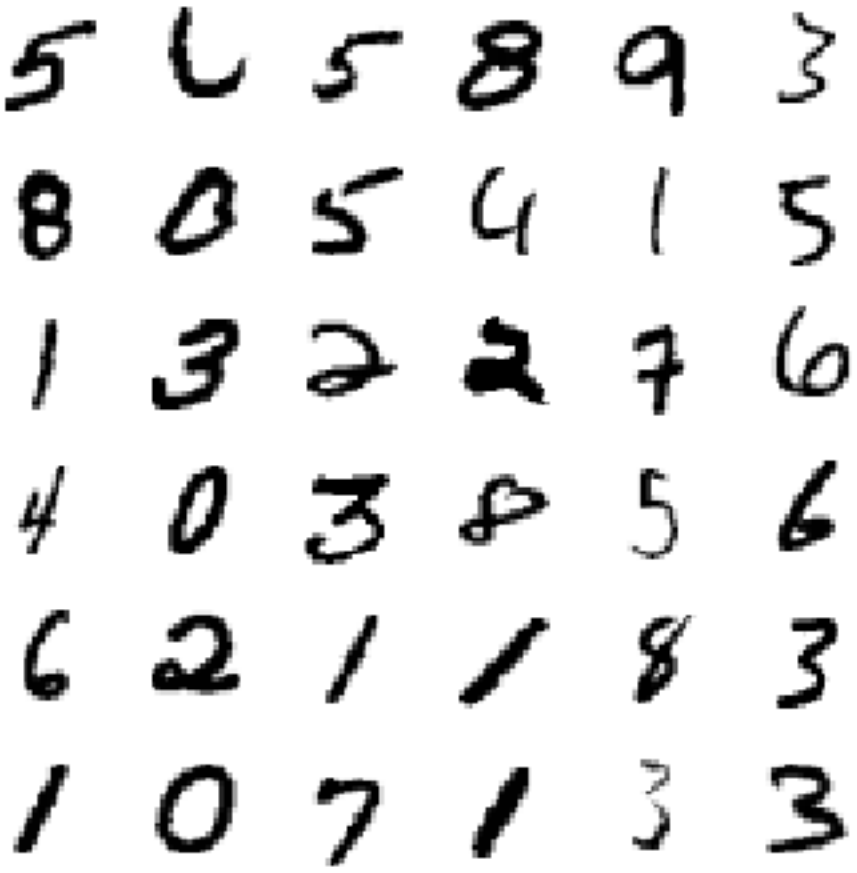
```
In [4]: from torchvision import transforms
        from torchvision.datasets import MNIST
        transform = transforms.Compose([
            #transforms.Resize(image_size),
            #transforms.CenterCrop(image_size),
            transforms.ToTensor(),
            transforms.Normalize([0.5], [0.5])
        ])
        dataset = MNIST(root='./data', train=True, download=True, transform=transform)
```

Now we can set up the `DataLoader`:

```
In [5]: batch_size = 128
        data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                                  batch_size=batch_size,
                                                  shuffle=True, drop_last=True)
```

Task: Visualize some samples of a single batch using the `plot_batch()` helper function.

```
In [6]: from utils import plot_batch

        for i_batch, sample_batched in enumerate(data_loader):
            plot_batch(sample_batched[0])
            break
```



1.3 Build Discriminator

Task: Implement the Discriminator as a sequential model consisting of two `nn.Linear`, `nn.ReLU` layers and a binary output with `nn.Sigmoid` activation.

We will feed the images into the network as flattened arrays.

```
In [7]: discriminator = nn.Sequential(
            nn.Linear(image_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 10),
            nn.ReLU(),
            nn.Linear(10, 1),
            nn.Sigmoid()
        )
```

1.4 Build Generator

Task: Implement the Generator as a sequential model of two `nn.Linear` layers with `nn.ReLU()` activation and a final `nn.Linear` layer with `nn.Tanh` activation that maps from a random vector to the original flattened image dimension.

```
In [8]: generator = nn.Sequential(
            nn.Linear(latent_size, hidden_size),
            nn.LeakyReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.LeakyReLU(),
            nn.Linear(hidden_size, image_size),
            nn.Tanh()
        )
```

1.4.1 General Setup for Training

By the follwing line we can dynamically determine whether we want to run our model and optimization on CPU or GPU

```
In [9]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Task: Send both `generator` and `discriminator` to the `device`. You can do that simply by calling `{mymodule}.to(device)`

```
In [10]: netG = generator.to(device)
         netD = discriminator.to(device)
```

Task: Now we can initialize the optimizers for both generator and discriminator. We'll use `Adam` with a learning rate of `0.0005` here.

```
In [11]: from torch.optim import Adam
         d_optimizer = Adam(netD.parameters(), lr=0.0005)
         g_optimizer = Adam(netG.parameters(), lr=0.0005)
```

Now we only need to initialize the cross entropy loss before getting started with the training loop

Compute BCE_Loss using real images where `BCE_Loss = - y * log(D(x)) - (1-y) * log(1 - D(x))`. The second term of the loss is always zero since real_labels == 1.

```
In [12]: criterion = nn.BCELoss()
```

1.4.2 Training Loop

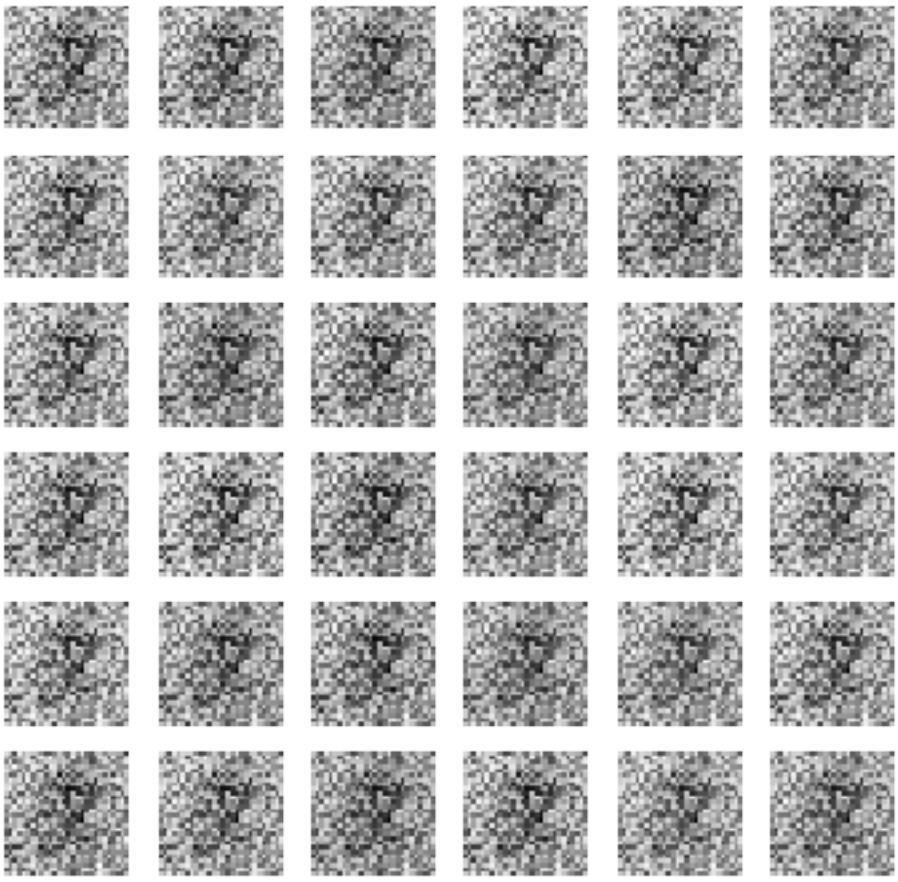
Finally, we can write the training loop and look at actual results. Since this is quite a bit of code, please perform the implementation in the follwing steps:

Task: Train your GAN and watch your fake images get better and better. It might take up to 20 Minutes to produce good results.

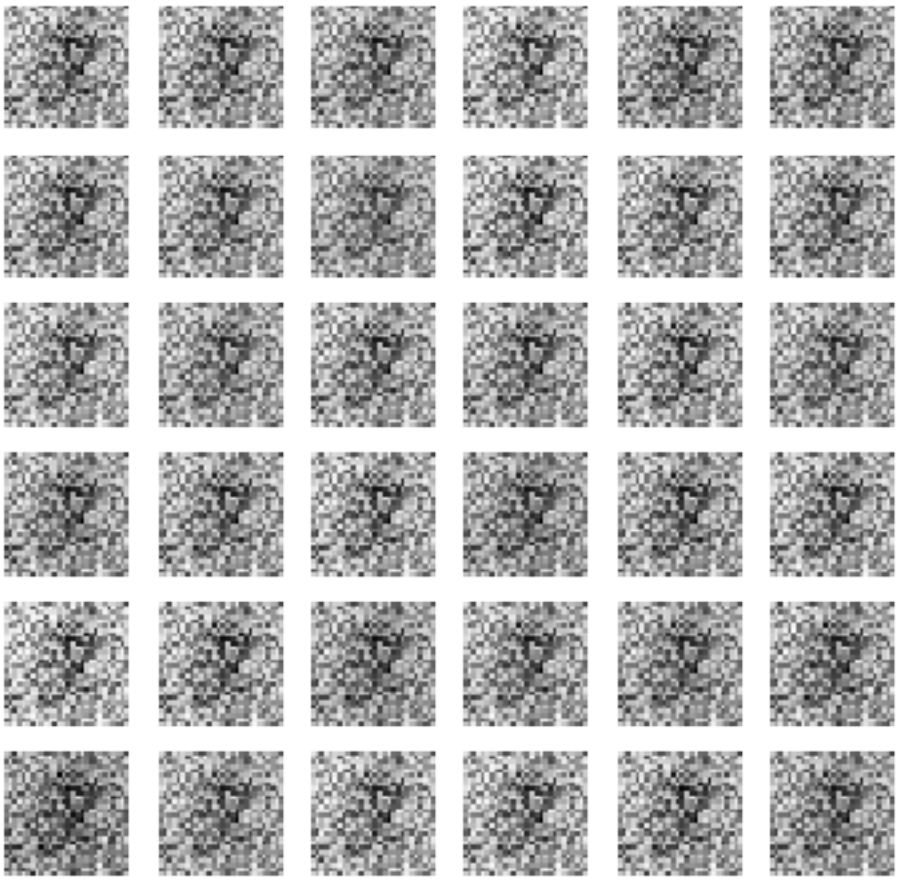
```
In [13]: total_step = len(data_loader)
         for epoch in range(num_epochs):
             for i, (images, _) in enumerate(data_loader):
                 images = images.reshape((images.size(0), -1)).to(device)

                 # Create the labels which are later used as input for the BCE loss
                 real_labels = torch.ones(batch_size, 1).to(device)
                 fake_labels = torch.zeros(batch_size, 1).to(device)

                 # ================================================================== #
                 #                      Train the discriminator                       #
                 # ================================================================== #

                 outputs = discriminator(images)
                 d_loss_real = criterion(outputs, real_labels)
                 real_score = outputs

                 # Compute BCELoss using fake images
                 z = torch.randn(batch_size, latent_size).to(device)
                 fake_images = generator(z)
                 outputs = discriminator(fake_images)
                 d_loss_fake = criterion(outputs, fake_labels)
                 fake_score = outputs

                 # Backprop and optimize
                 d_loss = d_loss_real + d_loss_fake
                 d_optimizer.zero_grad()
                 g_optimizer.zero_grad()
                 d_loss.backward()
                 d_optimizer.step()
```

```
                # ================================================================== #
                #                         Train the generator                        #
                # ================================================================== #

                # Compute loss with fake images
                z = torch.randn(batch_size, latent_size).to(device)
                fake_images = generator(z)
                outputs = discriminator(fake_images)

                # We train G to maximize log(D(G(z)) instead of minimizing log(1-D(G(z)))
                g_loss = criterion(outputs, real_labels)

                # Backprop and optimize
                d_optimizer.zero_grad()
                g_optimizer.zero_grad()
                g_loss.backward()
                g_optimizer.step()

                if (i+1) % 200 == 0:
                    print('Epoch [{}/{}], Step [{}/{}], d_loss: {:.4f}, g_loss: {:.4f}, D(x):
                          .format(epoch, num_epochs, i+1, total_step, d_loss.item(), g_loss.it
                                real_score.mean().item(), fake_score.mean().item()))
            fake_images = fake_images.reshape(fake_images.size(0), 1, 28, 28).cpu().detach().r
            plot_batch(fake_images)
```

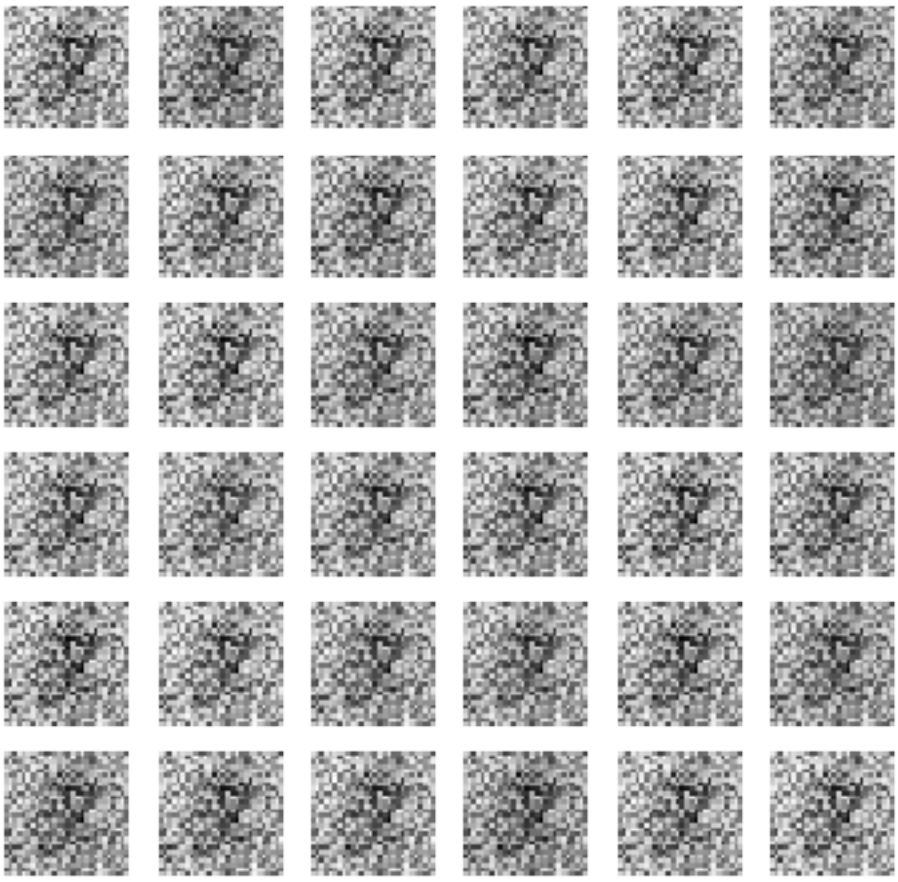
```
Epoch [0/30], Step [200/468], d_loss: 1.3867, g_loss: 0.7146, D(x): 0.49, D(G(z)): 0.49
Epoch [0/30], Step [400/468], d_loss: 1.3866, g_loss: 0.7102, D(x): 0.49, D(G(z)): 0.49
```



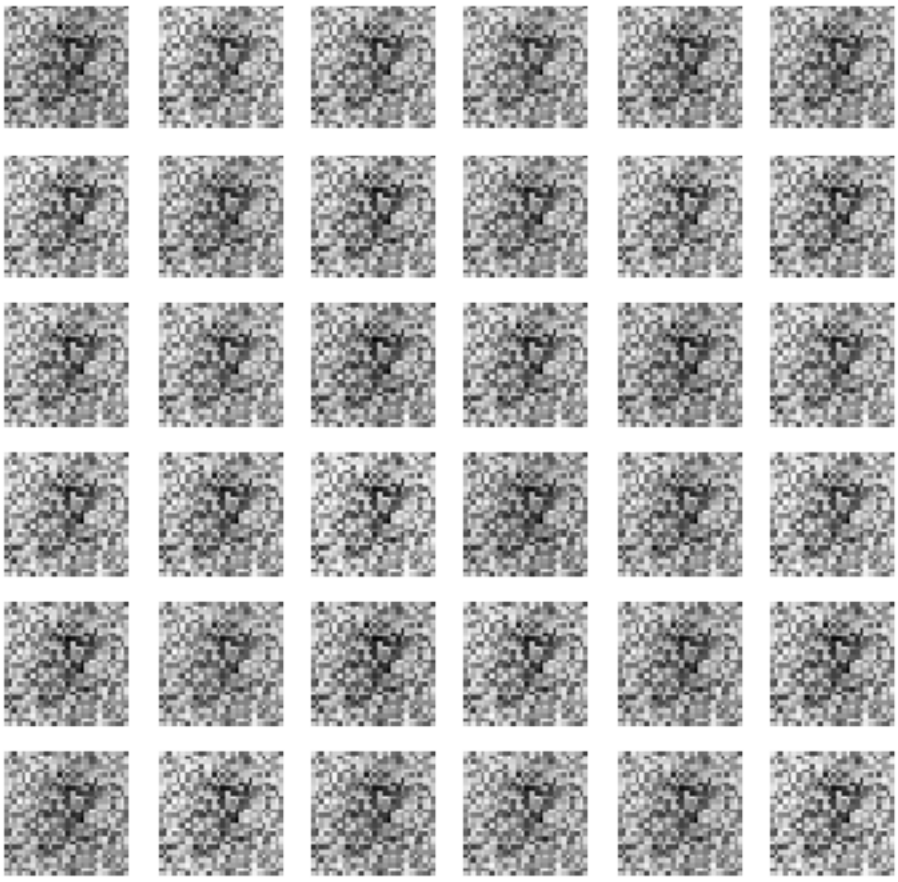
```
Epoch [1/30], Step [200/468], d_loss: 1.3864, g_loss: 0.7045, D(x): 0.49, D(G(z)): 0.49
Epoch [1/30], Step [400/468], d_loss: 1.3864, g_loss: 0.7008, D(x): 0.50, D(G(z)): 0.50
```



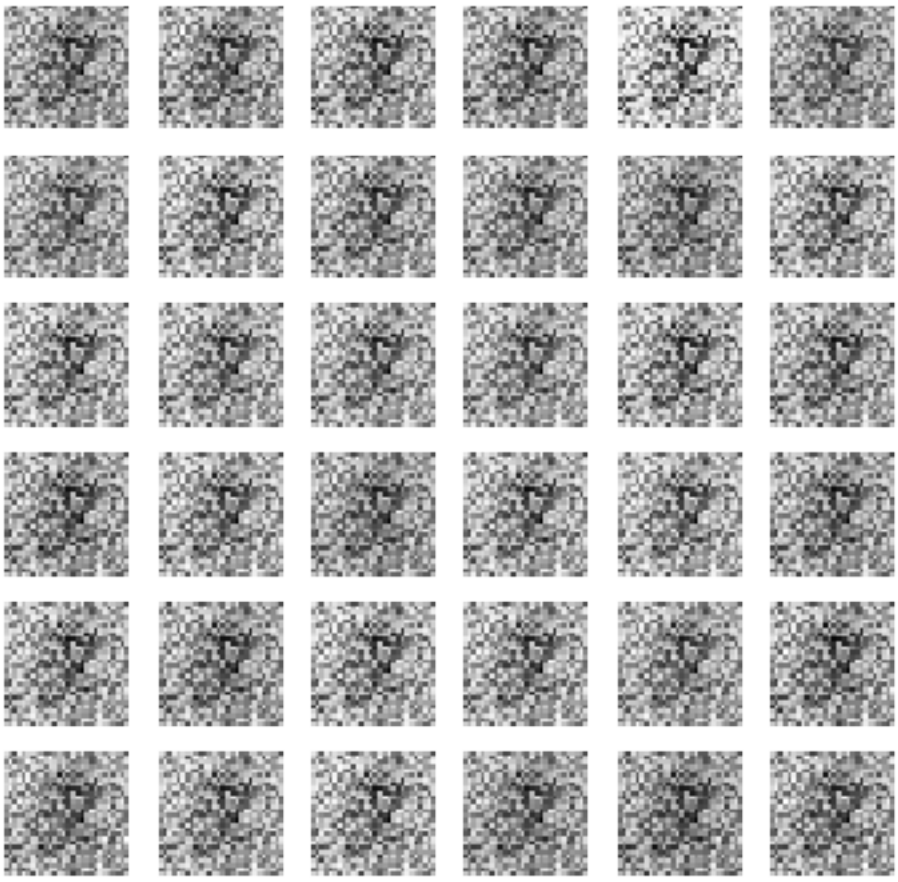
```
Epoch [2/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6972, D(x): 0.50, D(G(z)): 0.50
Epoch [2/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6954, D(x): 0.50, D(G(z)): 0.50
```



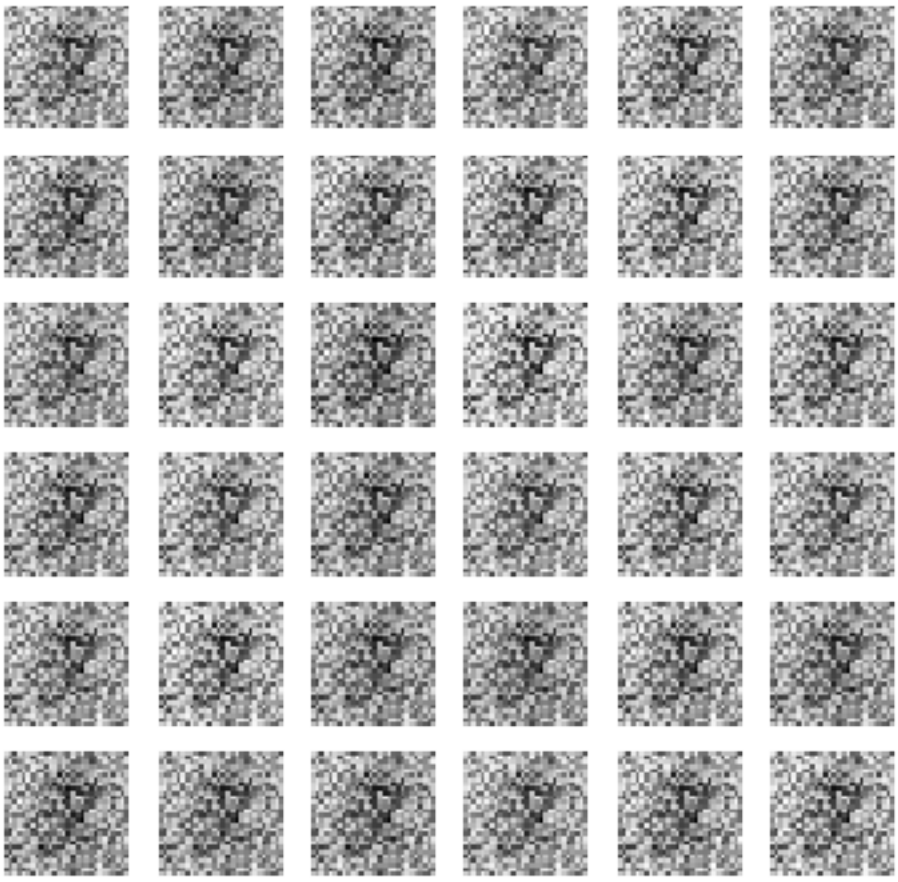
```
Epoch [3/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6941, D(x): 0.50, D(G(z)): 0.50
Epoch [3/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6936, D(x): 0.50, D(G(z)): 0.50
```



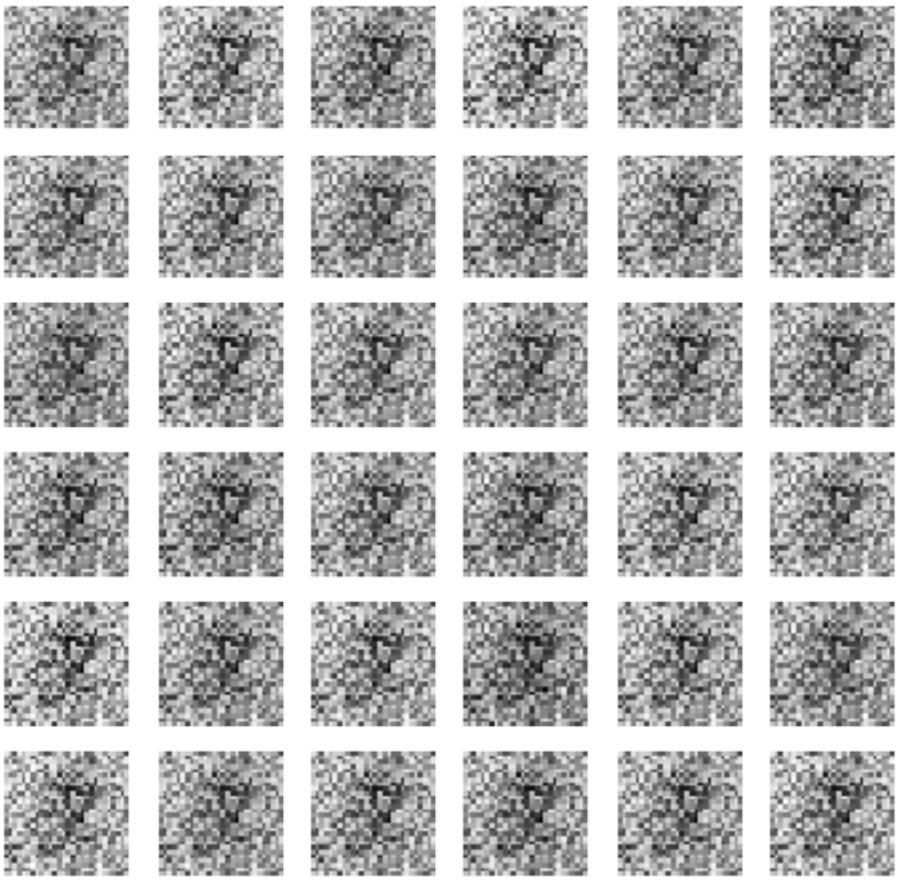
```
Epoch [4/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6933, D(x): 0.50, D(G(z)): 0.50
Epoch [4/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6932, D(x): 0.50, D(G(z)): 0.50
```



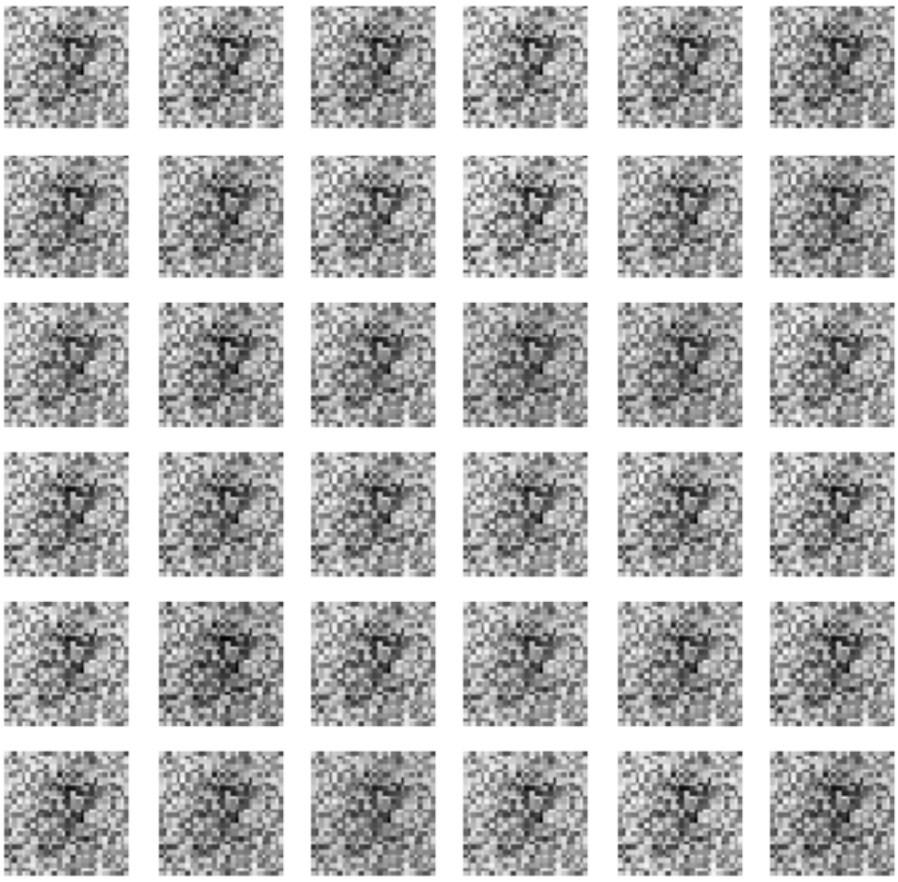
```
Epoch [5/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6932, D(x): 0.50, D(G(z)): 0.50
Epoch [5/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



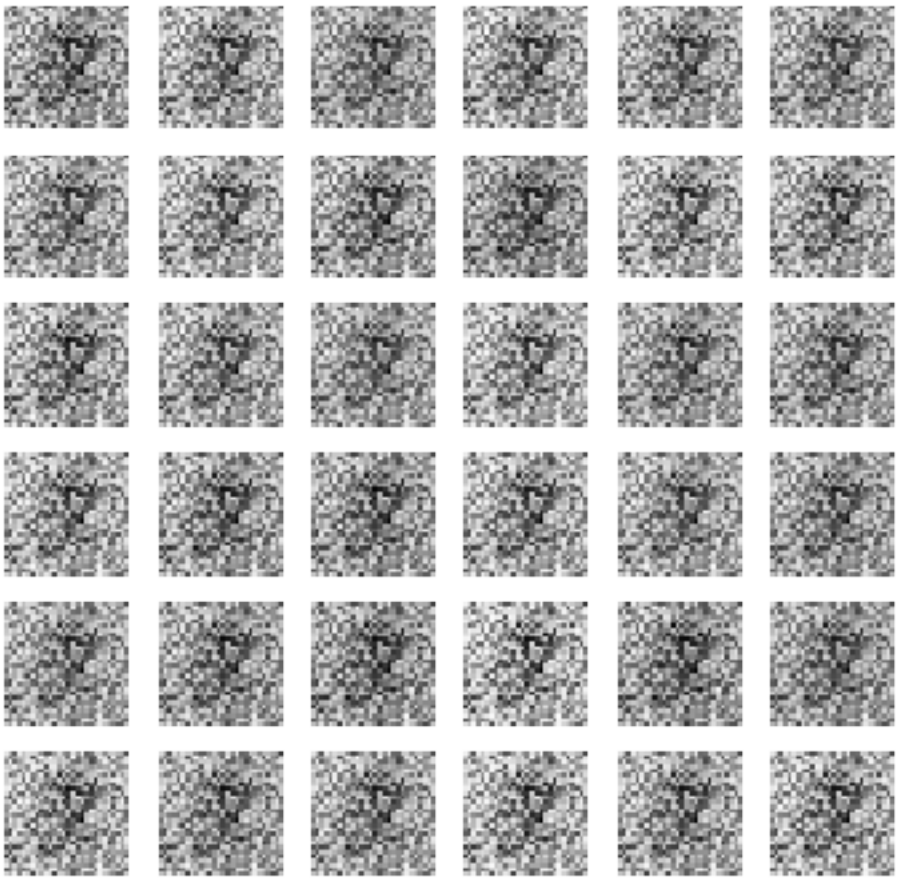
```
Epoch [6/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [6/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



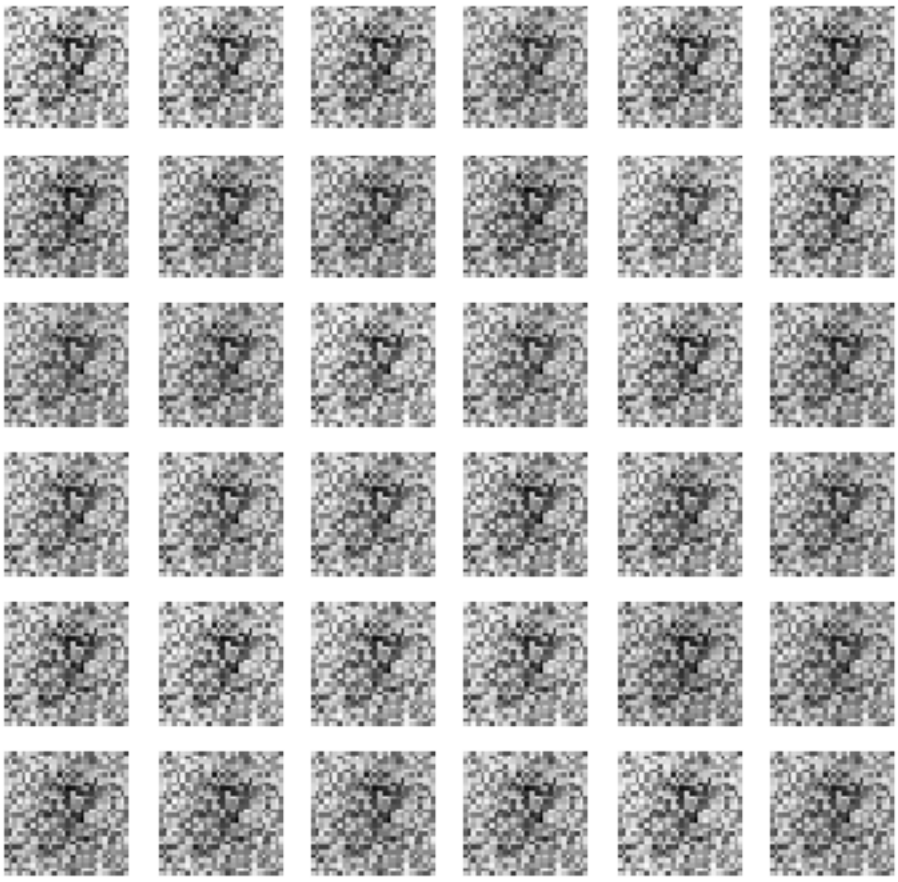
```
Epoch [7/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [7/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



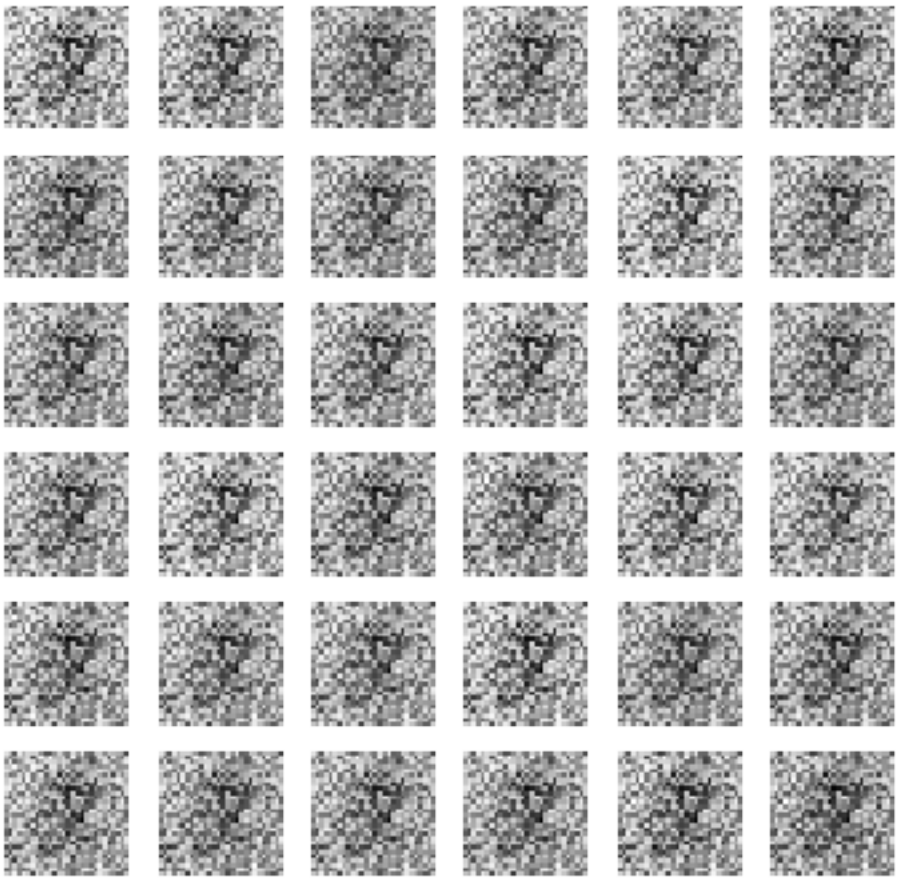
```
Epoch [8/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [8/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



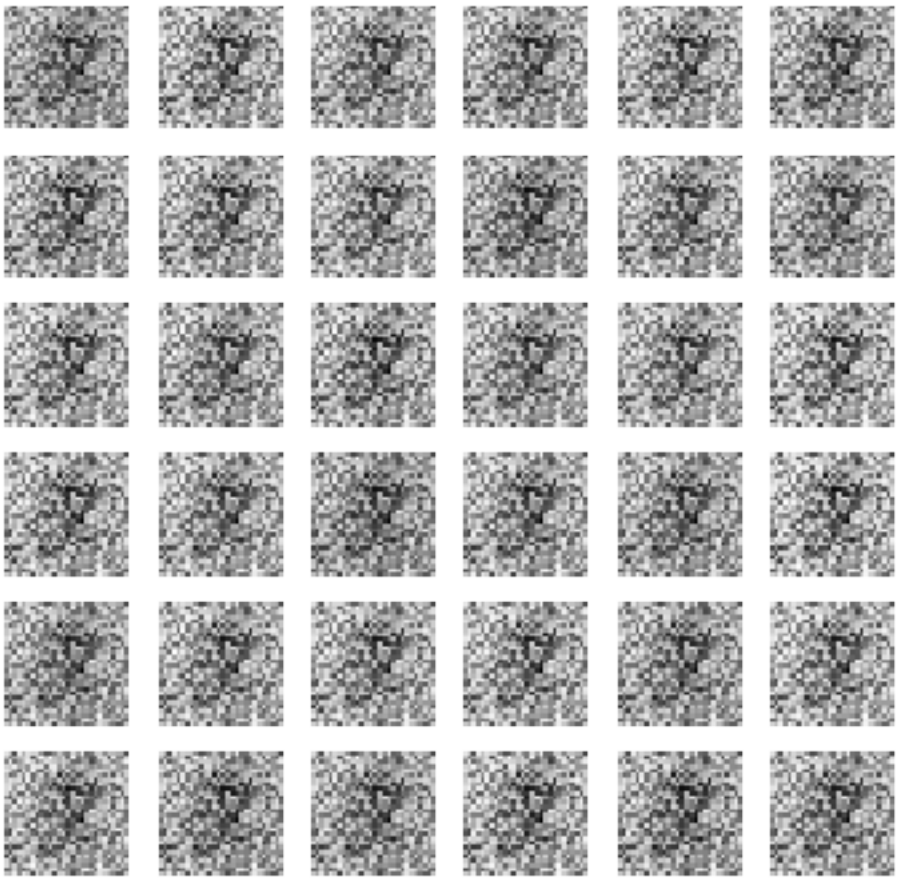
```
Epoch [9/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [9/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



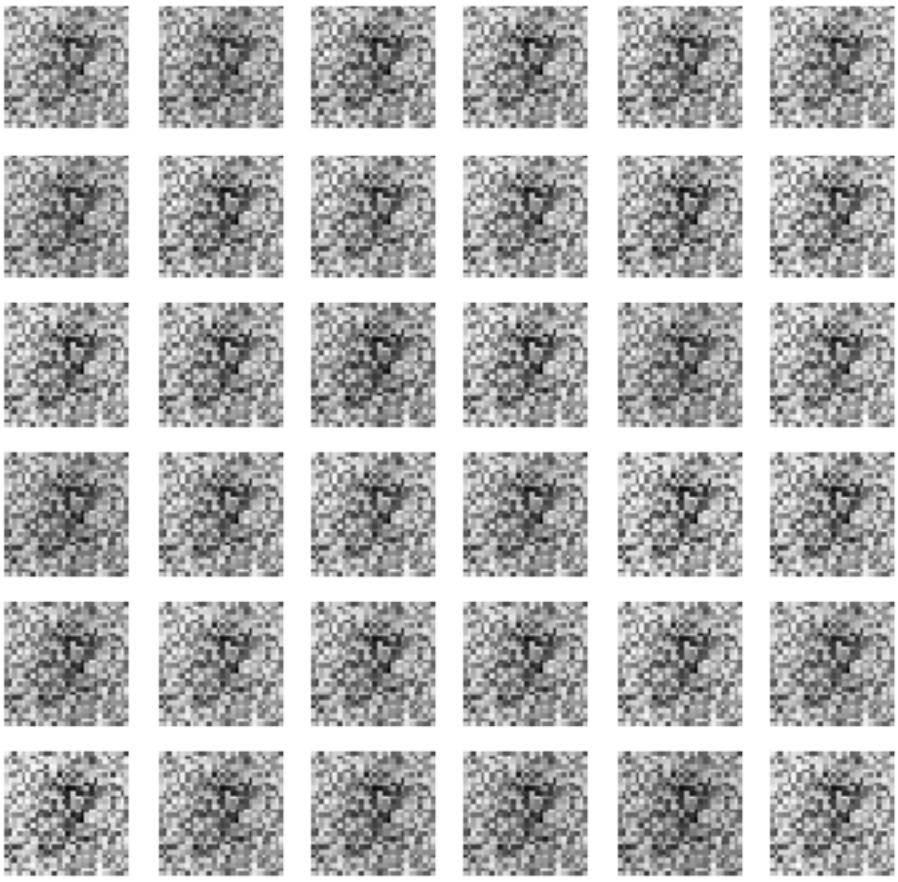
```
Epoch [10/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [10/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



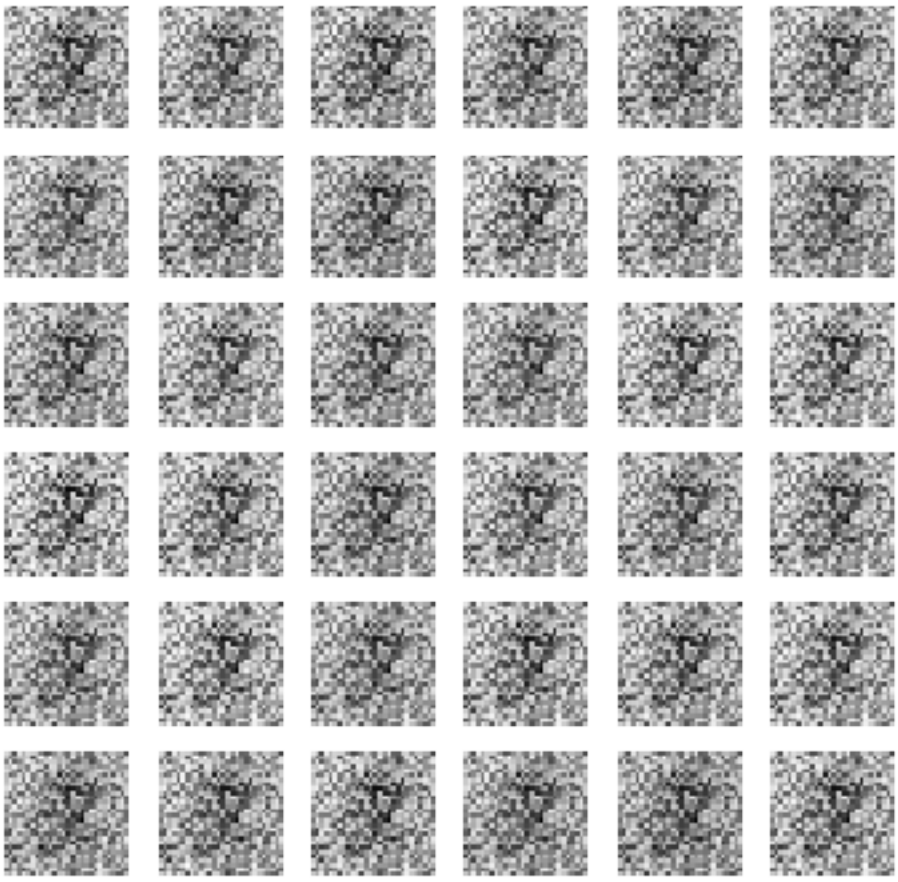
```
Epoch [11/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [11/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



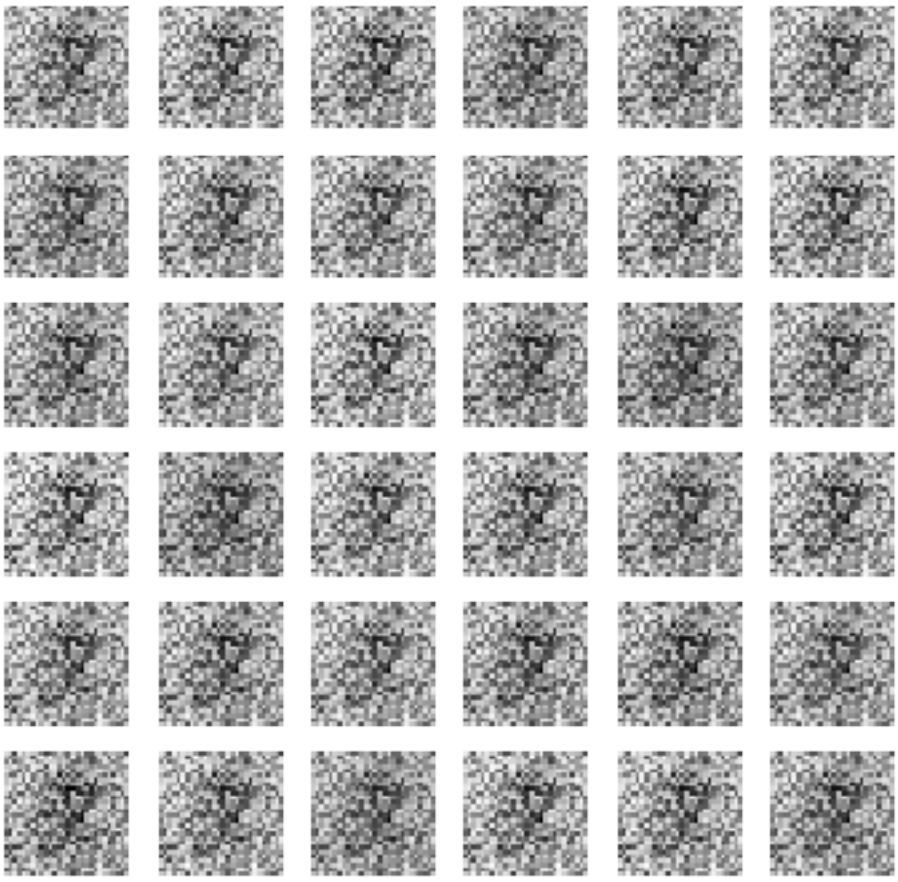
```
Epoch [12/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [12/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



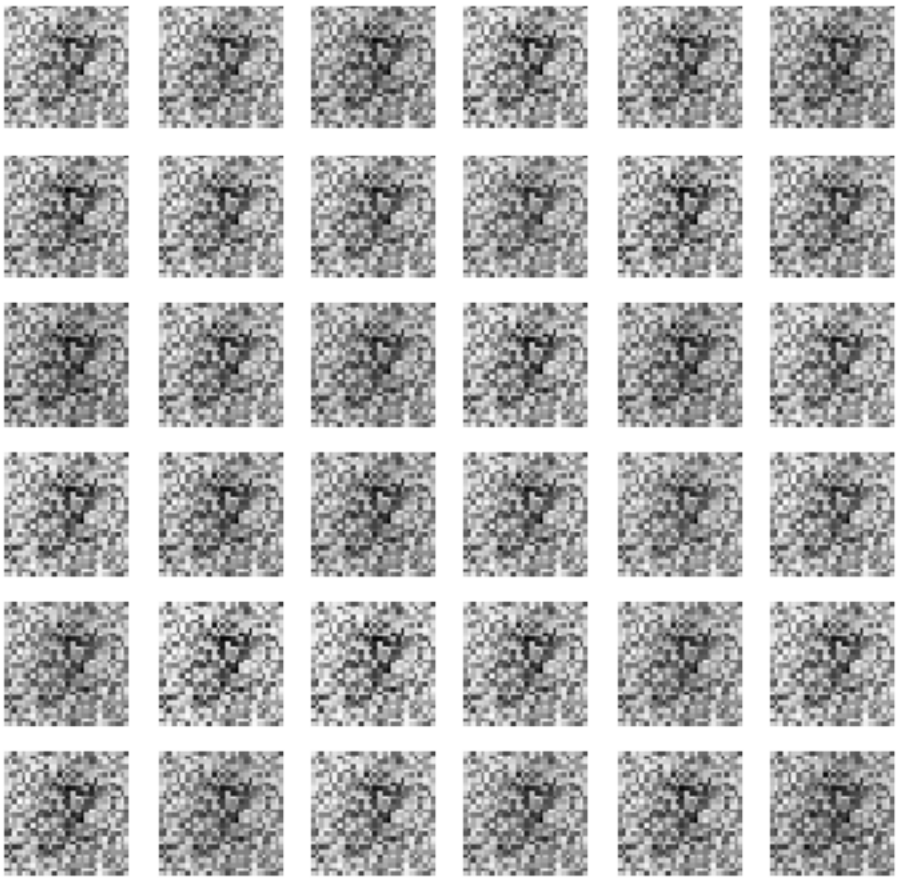
```
Epoch [13/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [13/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



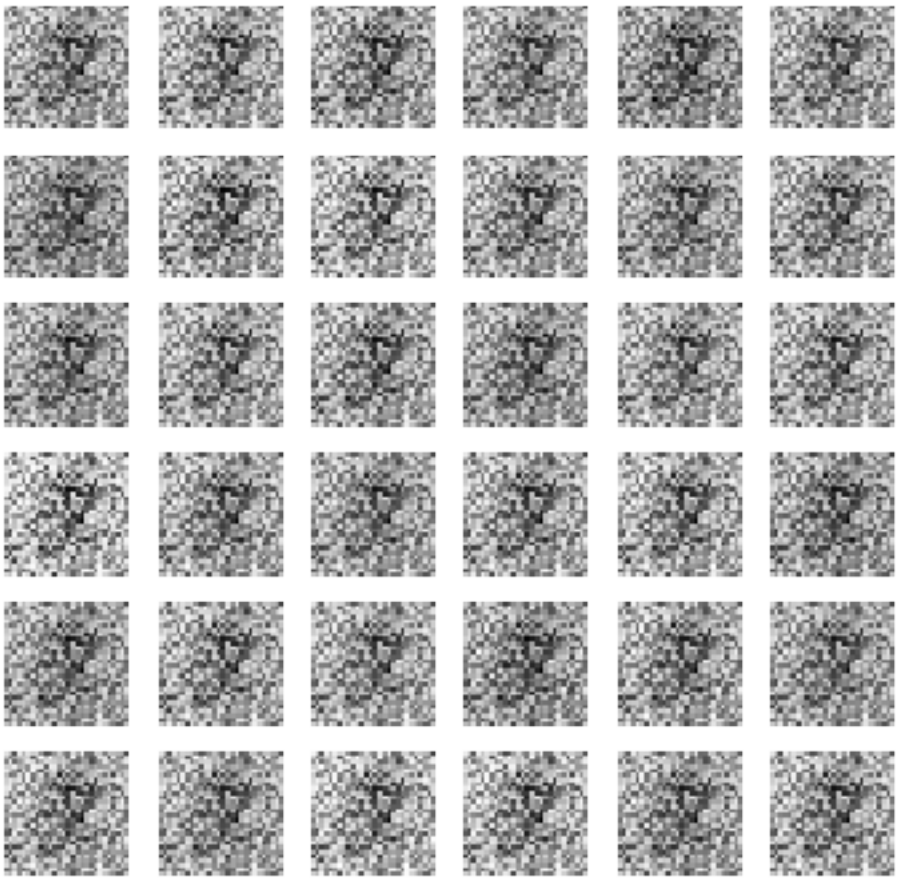
```
Epoch [14/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [14/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



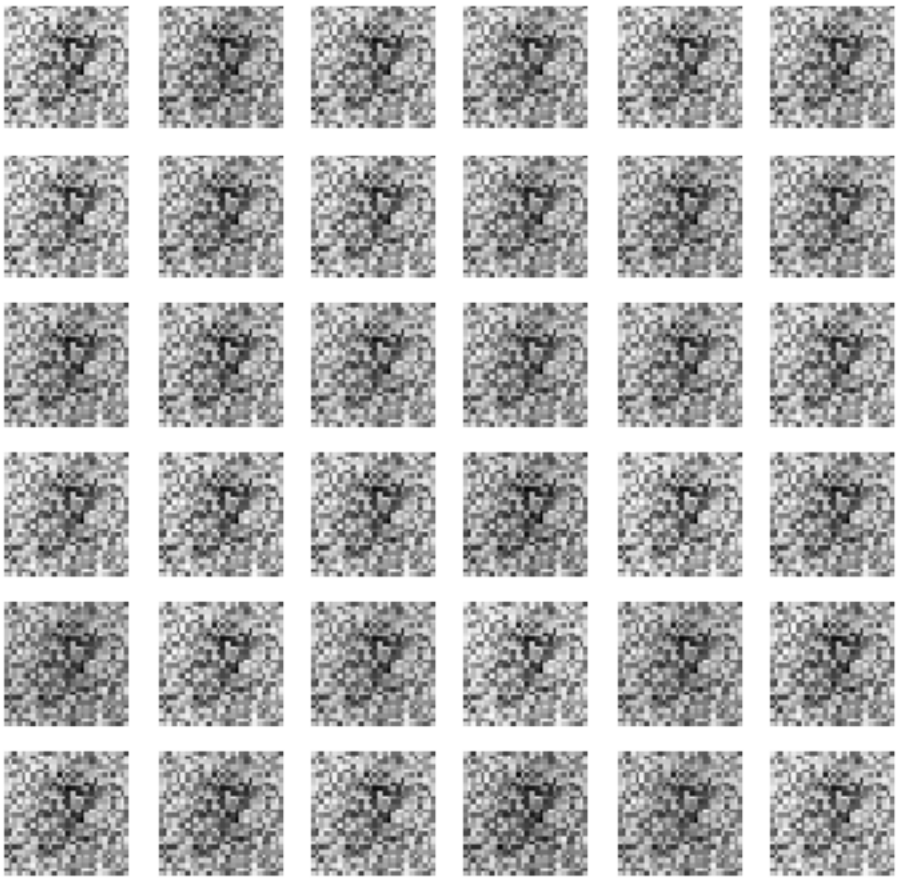
```
Epoch [15/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [15/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



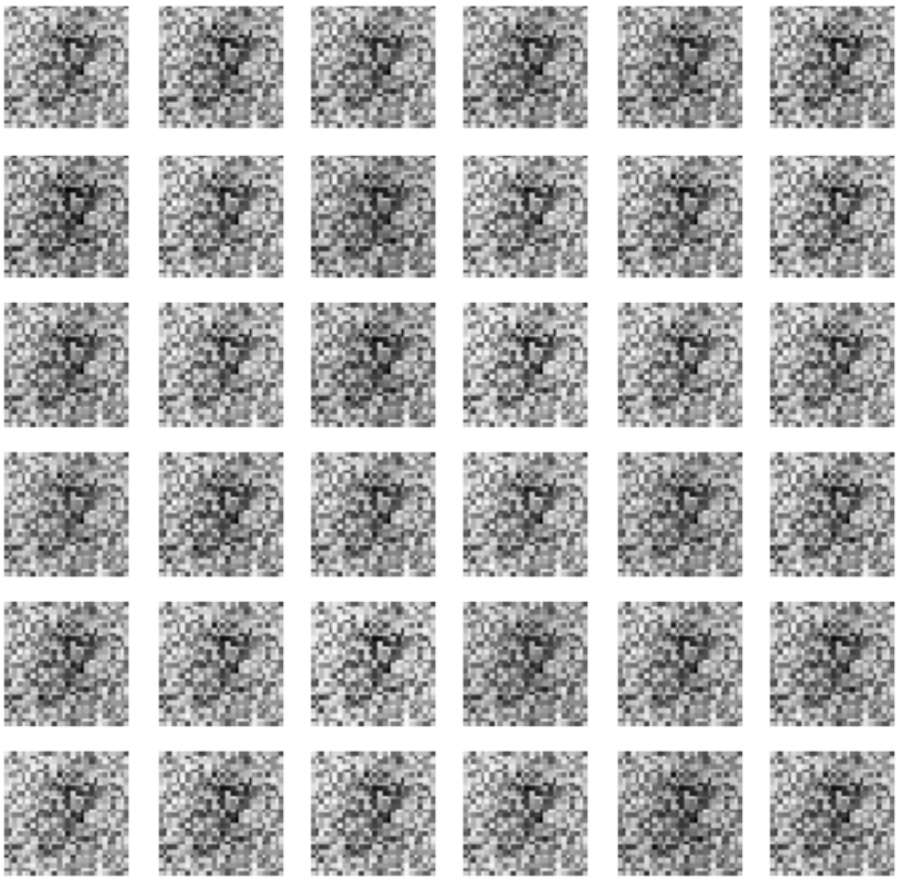
```
Epoch [16/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [16/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



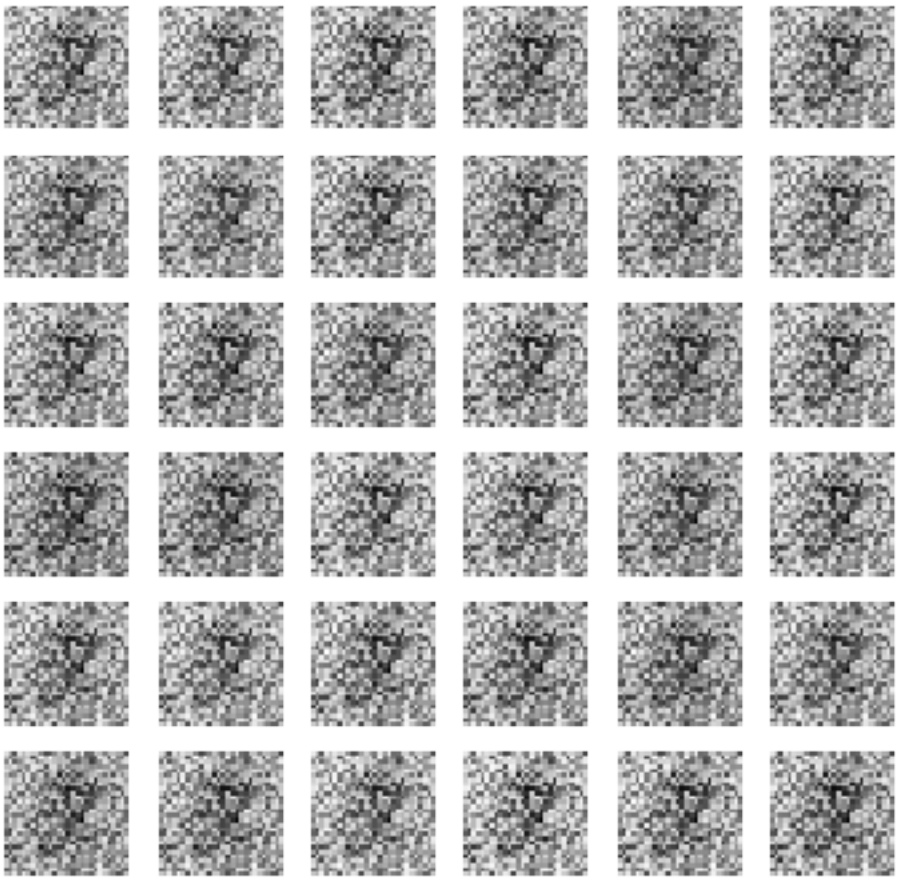
```
Epoch [17/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [17/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



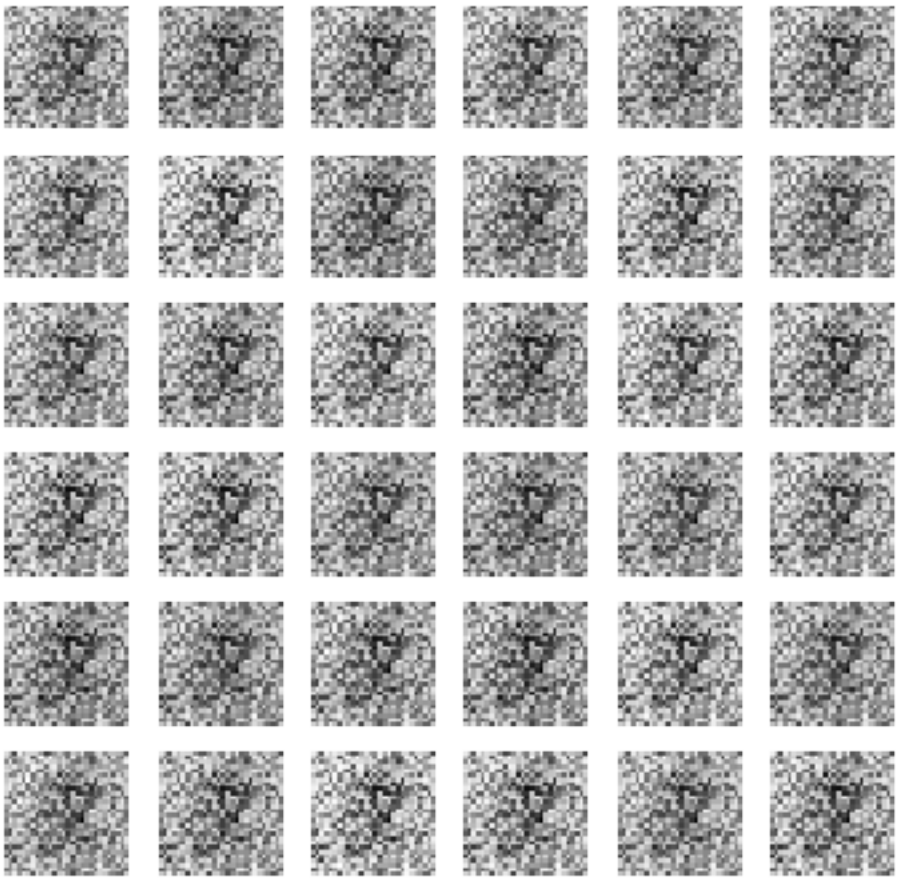
```
Epoch [18/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [18/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



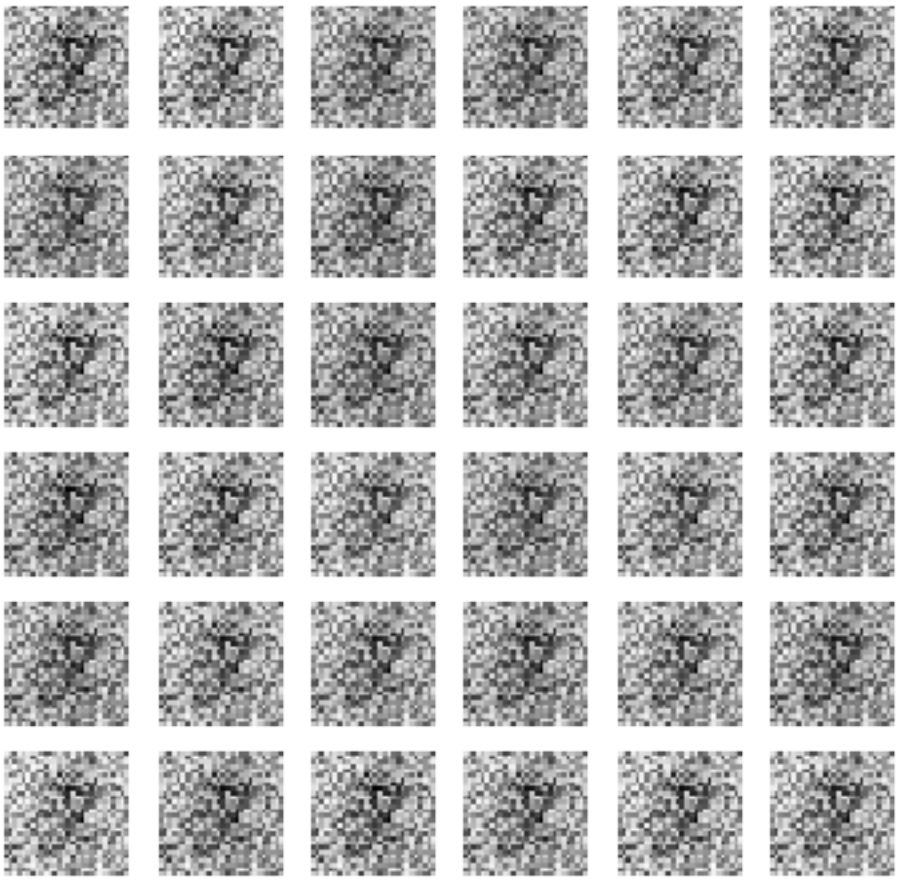
```
Epoch [19/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [19/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



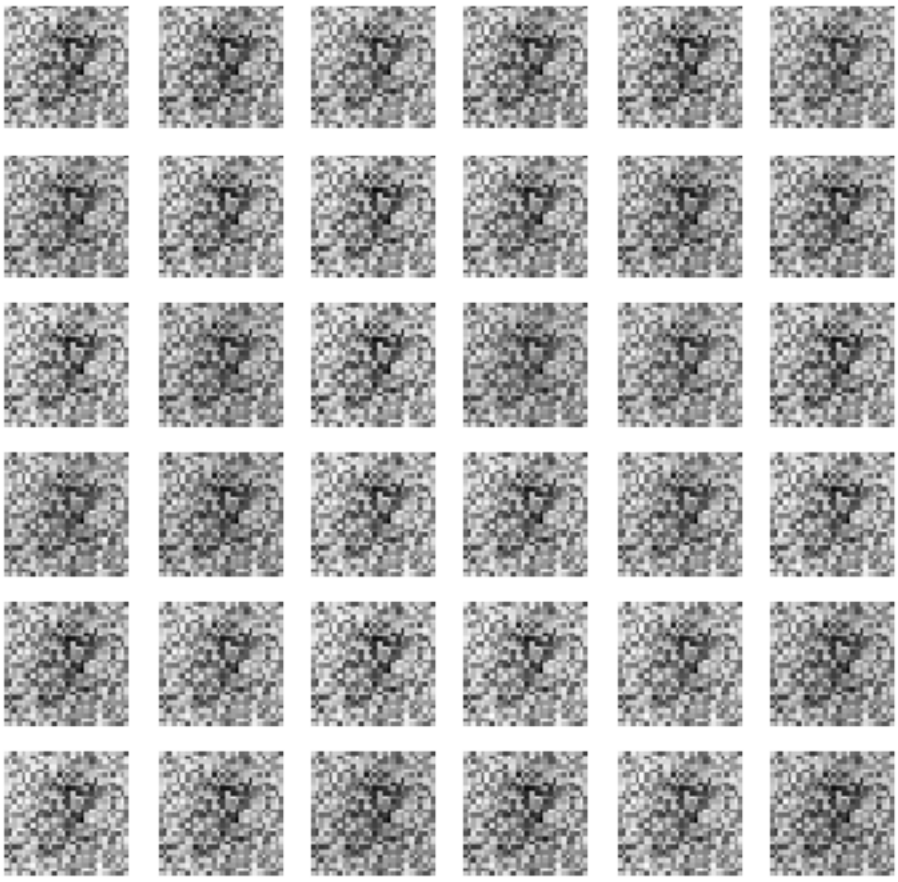
```
Epoch [20/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [20/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



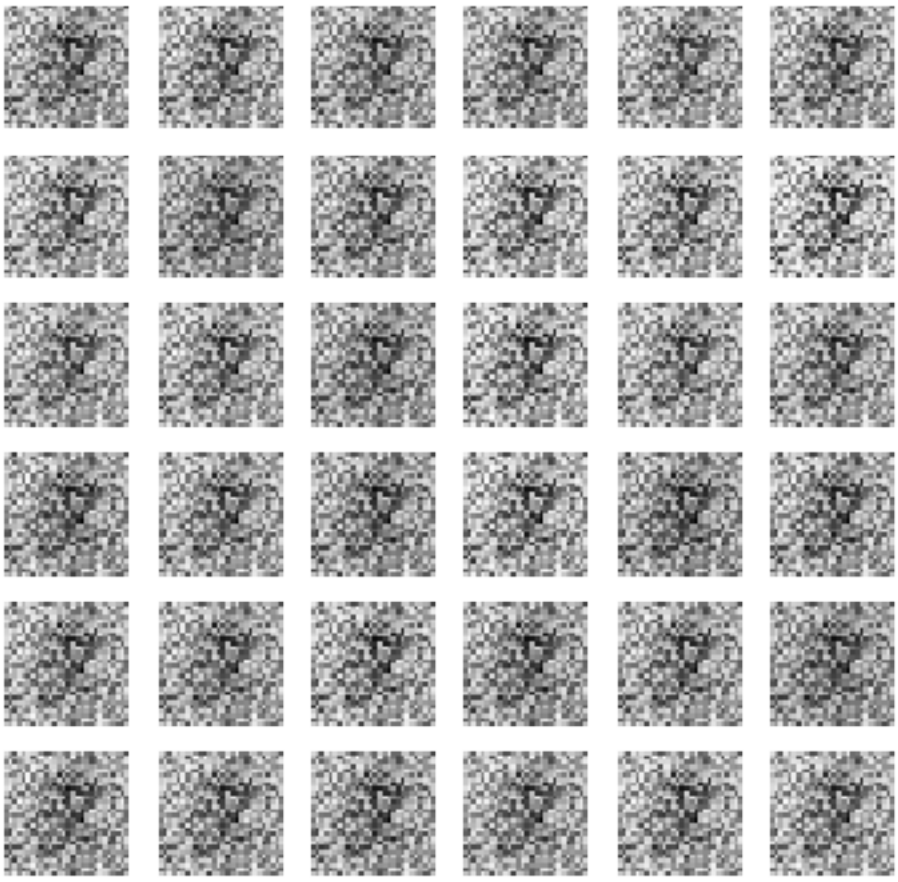
```
Epoch [21/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [21/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



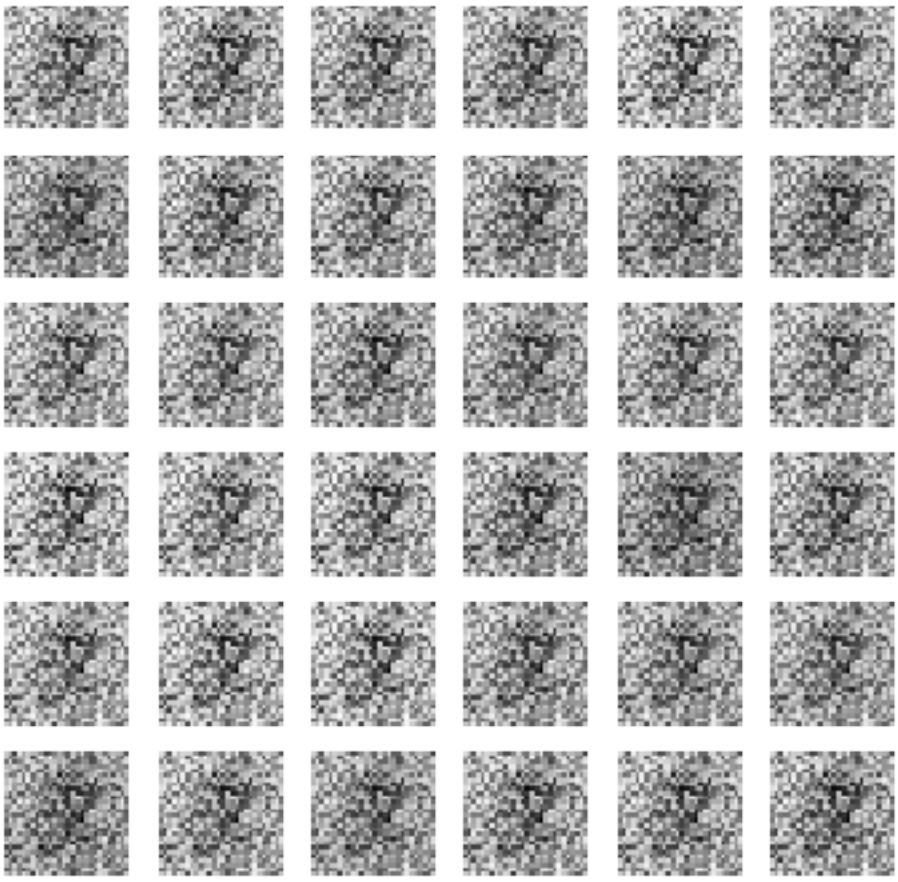
```
Epoch [22/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [22/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



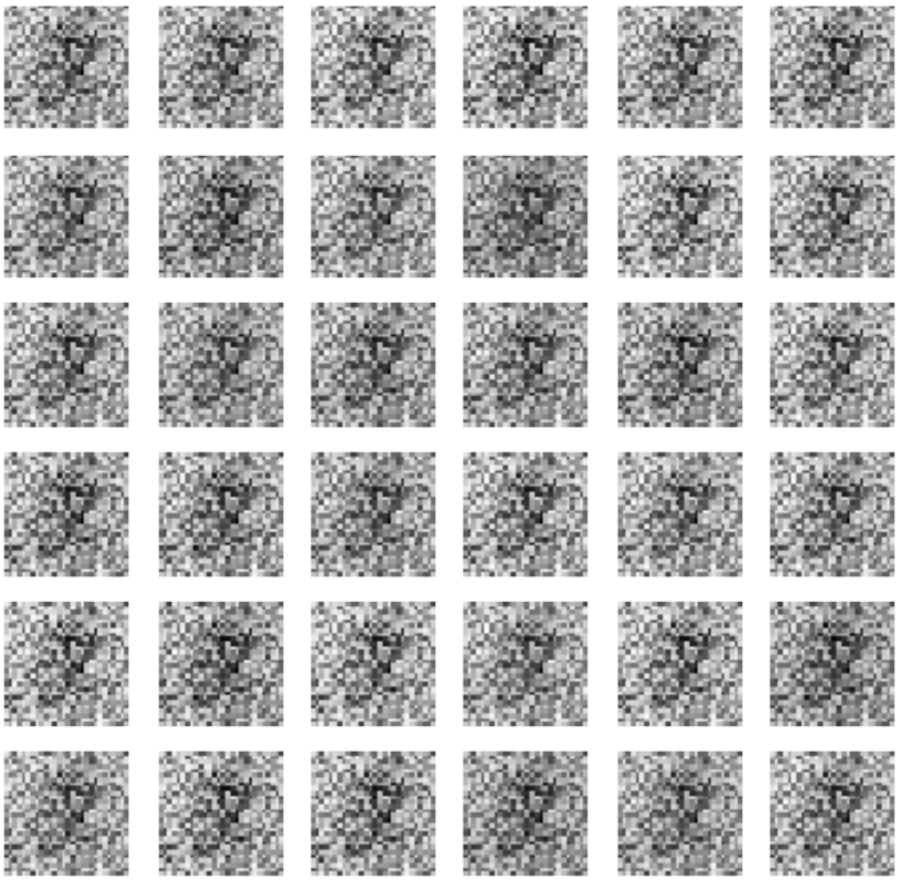
```
Epoch [23/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [23/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



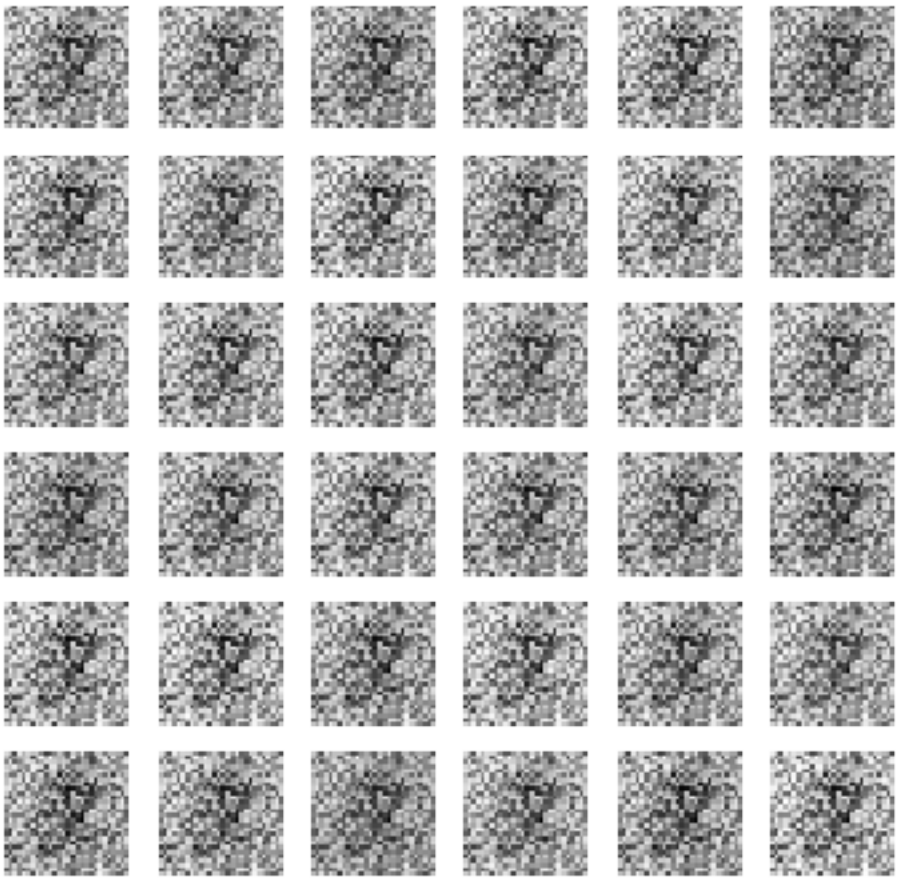
```
Epoch [24/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [24/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



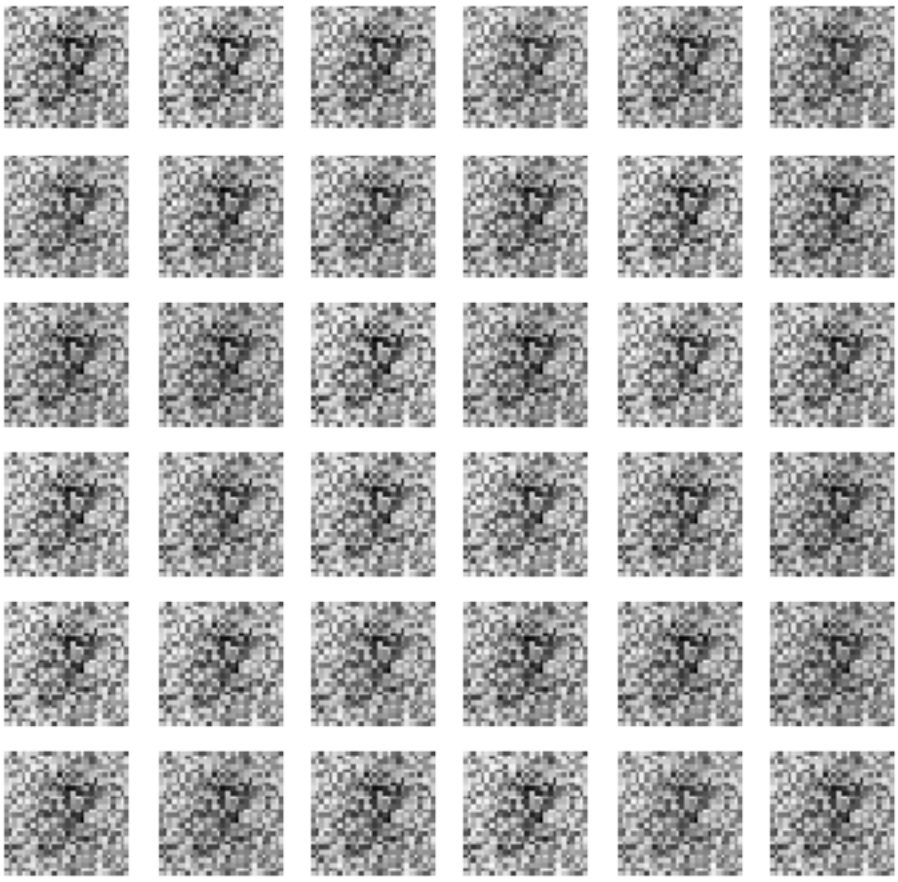
```
Epoch [25/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [25/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



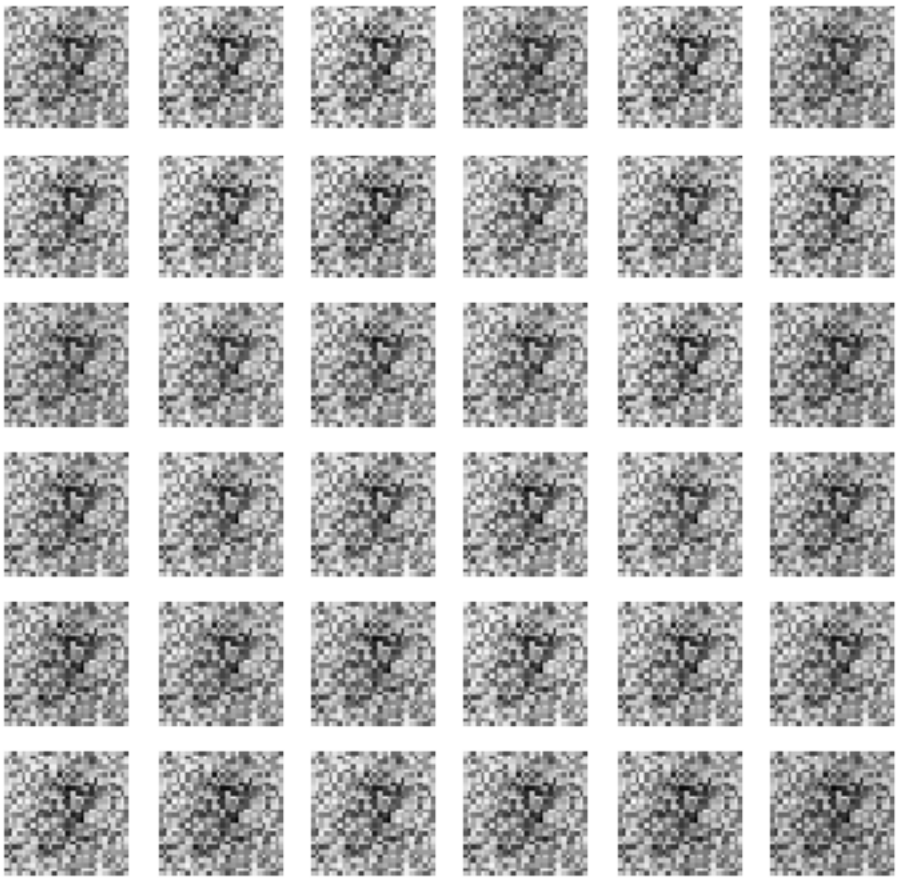
```
Epoch [26/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [26/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



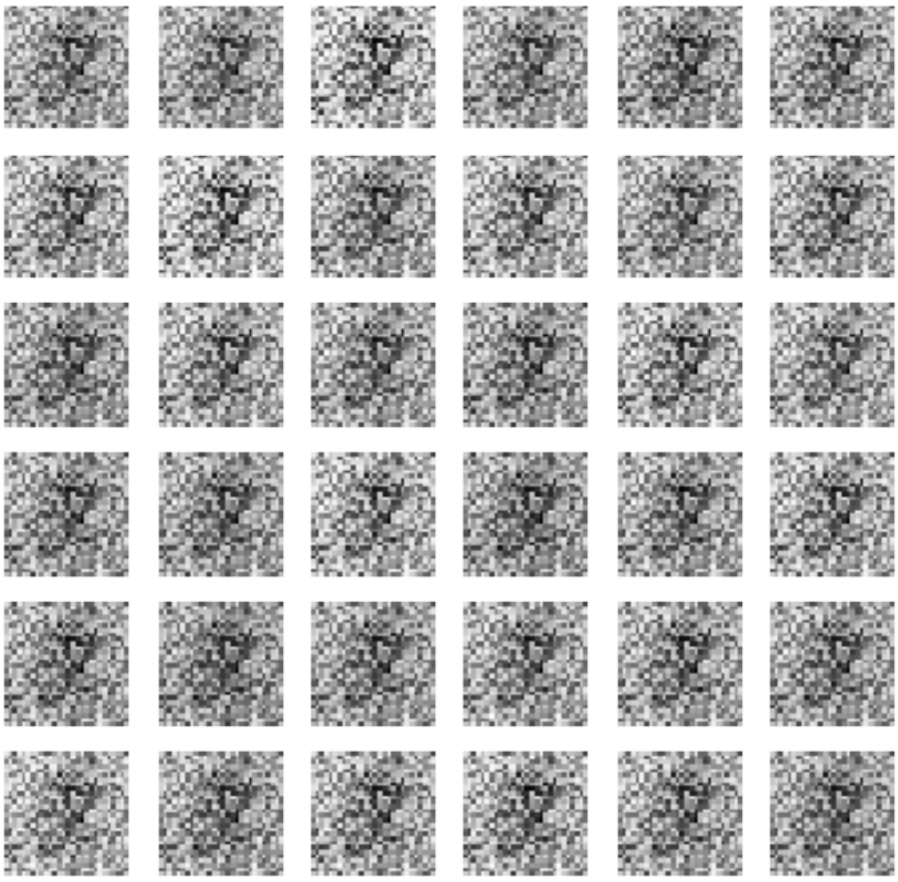
```
Epoch [27/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [27/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



```
Epoch [28/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [28/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



```
Epoch [29/30], Step [200/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
Epoch [29/30], Step [400/468], d_loss: 1.3863, g_loss: 0.6931, D(x): 0.50, D(G(z)): 0.50
```



Vanilla GANs as implemented above are difficult to train because of the high number of the many feedforward connections. DCGAN aims to overcome this by using Convolutional instead of Fully-connected layers.

In the next cells we will implement the same Training loop and setup as above, but using a DCGAN.

1.5 Build DCGAN

Task: Implement the generator of a DCGAN. To allow you to train the DCGAN also with a different dataset later, we change the input resolution from `28x28` as before to `64x64`.

1.5.1 Generator

```
In [14]: g_c = torch.nn.Sequential(
             #nn.ConvTranspose2d(latent_size, 1024, kernel_size=(4, 4), stride=(1, 1), bias=Fa
             nn.ConvTranspose2d(latent_size, 512, kernel_size=(4, 4), stride=(1, 1), bias=Fals
             nn.BatchNorm2d(512),
             nn.ReLU(),
             nn.ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=1),
             nn.BatchNorm2d(256),
```

```
            nn.ReLU(),
            nn.ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 1, kernel_size=(4, 4), stride=(2, 2), padding=1),
            nn.Tanh()
        )
```

1.5.2 Discriminator

Task: Implement the discriminator of a DCGAN. To allow you to train the DCGAN also with a different dataset later, we change the input resolution from 28x28 as before to 64x64.

```
In [19]: d_c = torch.nn.Sequential(
            nn.Conv2d(1, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(negative_slope=0.2, inplace=True),
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(negative_slope=0.2, inplace=True),
            nn.Conv2d(128, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(negative_slope=0.2, inplace=True),
            nn.Conv2d(256, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(negative_slope=0.2, inplace=True),
            nn.Conv2d(512, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

In [21]: test_input = torch.randn((128, 100, 1, 1))
         out = g_c(test_input)

In [22]: out.shape

Out[22]: torch.Size([128, 1, 64, 64])
```

Task: Once again, we have to send the networks to our device and initialize the optimizers. Use the same learning rate as above.

```
In [23]: criterion = nn.BCELoss()

         # send to device
         netG_c = g_c.to(device)
         netD_c = d_c.to(device)

         # optimizer
         d_optimizer_c = Adam(netD_c.parameters(), lr=0.0002, betas=(0.5, 0.999))
         g_optimizer_c = Adam(netG_c.parameters(), lr=0.0002, betas=(0.5, 0.999))
```

1.5.3 Transforms

Task: To adapt the MNIST images to the new expected input dimension of `64x64`, we need to adapt our `transforms` accordingly. Find the right transform to resize all images to `64x64` and initialize dataset and data loader with the new transforms.

```
In [24]: new_image_size = 64*64
         transform_c = transforms.Compose([
             transforms.Resize((64, 64)),
             #transforms.CenterCrop(image_size),
             transforms.ToTensor(),
             transforms.Normalize([0.5], [0.5])
         ])
         dataset = torchvision.datasets.MNIST(root='.',
                                              train=True,
                                              transform=transform_c,
                                              download=True)
         data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                                   batch_size=batch_size,
                                                   shuffle=True, drop_last=True)
```

1.5.4 Training Loop Reloaded

Now we can build the training loop for the DCGAN. But don't worry, we can mostly copy our code from above. Take care of the changed tensor dimensions, though.

Task: Implement the Training loop from above and visualize a batch of fake data after every epoch. How does the training differ from the vanilla GAN training?

```
In [25]: total_step = len(data_loader)
         for epoch in range(num_epochs):
             for i, (images, _) in enumerate(data_loader):
                 # Create the labels which are later used as input for the BCE loss
                 real_labels = torch.ones(batch_size, 1).to(device)
                 fake_labels = torch.zeros(batch_size, 1).to(device)
                 images = images.to(device)
                 # ================================================================== #
                 #                      Train the discriminator                       #
                 # ================================================================== #
                 outputs = d_c(images)
                 d_loss_real = criterion(outputs, real_labels)
                 real_score = outputs

                 # Compute BCELoss using fake images
                 z = torch.randn(batch_size, latent_size, 1, 1).to(device)
                 fake_images = g_c(z)
                 outputs = d_c(fake_images)
                 d_loss_fake = criterion(outputs, fake_labels)
                 fake_score = outputs
```

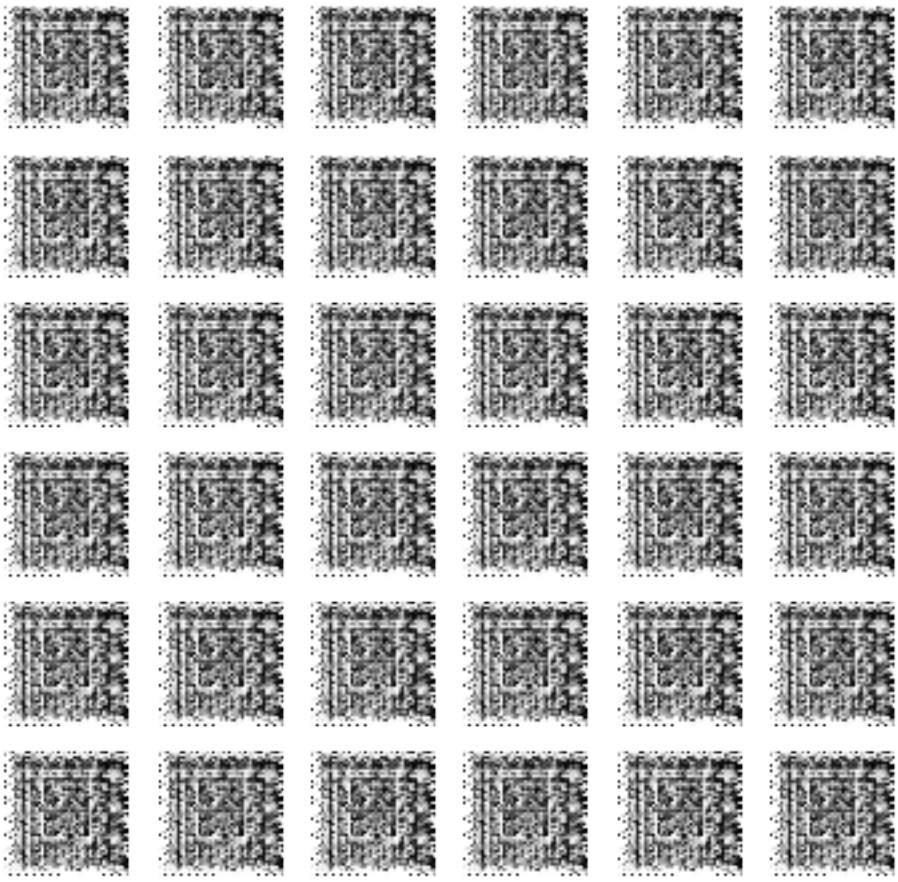
```
                # Backprop and optimize
                d_loss = d_loss_real + d_loss_fake
                d_optimizer_c.zero_grad()
                g_optimizer_c.zero_grad()
                d_loss.backward()
                d_optimizer_c.step()

                # ================================================================== #
                #                        Train the generator                         #
                # ================================================================== #

                # Compute loss with fake images
                z = torch.randn(batch_size, latent_size, 1, 1).to(device)
                fake_images = g_c(z)
                outputs = d_c(fake_images)

                # We train G to maximize log(D(G(z)) instead of minimizing log(1-D(G(z)))
                g_loss = criterion(outputs, real_labels)

                # Backprop and optimize
                d_optimizer_c.zero_grad()
                g_optimizer_c.zero_grad()
                g_loss.backward()
                g_optimizer_c.step()

                if (i+1) % 200 == 0:
                    print('Epoch [{}/{}], Step [{}/{}], d_loss: {:.4f}, g_loss: {:.4f}, D(x):
                          .format(epoch, num_epochs, i+1, total_step, d_loss.item(), g_loss.it
                                real_score.mean().item(), fake_score.mean().item()))
            fake_images = fake_images.reshape(fake_images.size(0), 1, 64, 64).cpu().detach().r
            plot_batch(fake_images)
```

```
/usr/local/anaconda3/envs/hamlet-pytorch-1-1/lib/python3.7/site-packages/torch/nn/modules/loss
  return F.binary_cross_entropy(input, target, weight=self.weight, reduction=self.reduction)


Epoch [0/30], Step [200/468], d_loss: 0.1821, g_loss: 4.7172, D(x): 0.86, D(G(z)): 0.02
Epoch [0/30], Step [400/468], d_loss: 0.6930, g_loss: 2.6134, D(x): 0.79, D(G(z)): 0.34
```

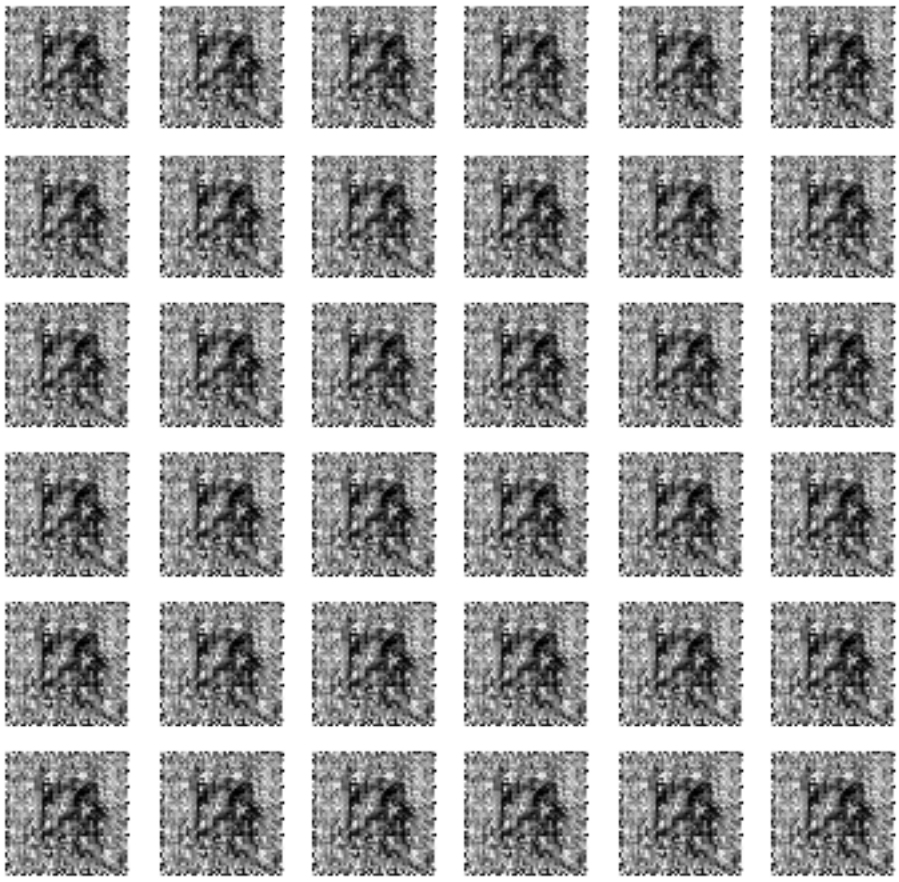
```
Epoch [1/30], Step [200/468], d_loss: 0.6200, g_loss: 2.2121, D(x): 0.80, D(G(z)): 0.32
Epoch [1/30], Step [400/468], d_loss: 0.5635, g_loss: 2.4475, D(x): 0.80, D(G(z)): 0.28
```

```
Epoch [2/30], Step [200/468], d_loss: 0.5096, g_loss: 3.1736, D(x): 0.87, D(G(z)): 0.29
Epoch [2/30], Step [400/468], d_loss: 0.7519, g_loss: 2.3375, D(x): 0.81, D(G(z)): 0.39
```

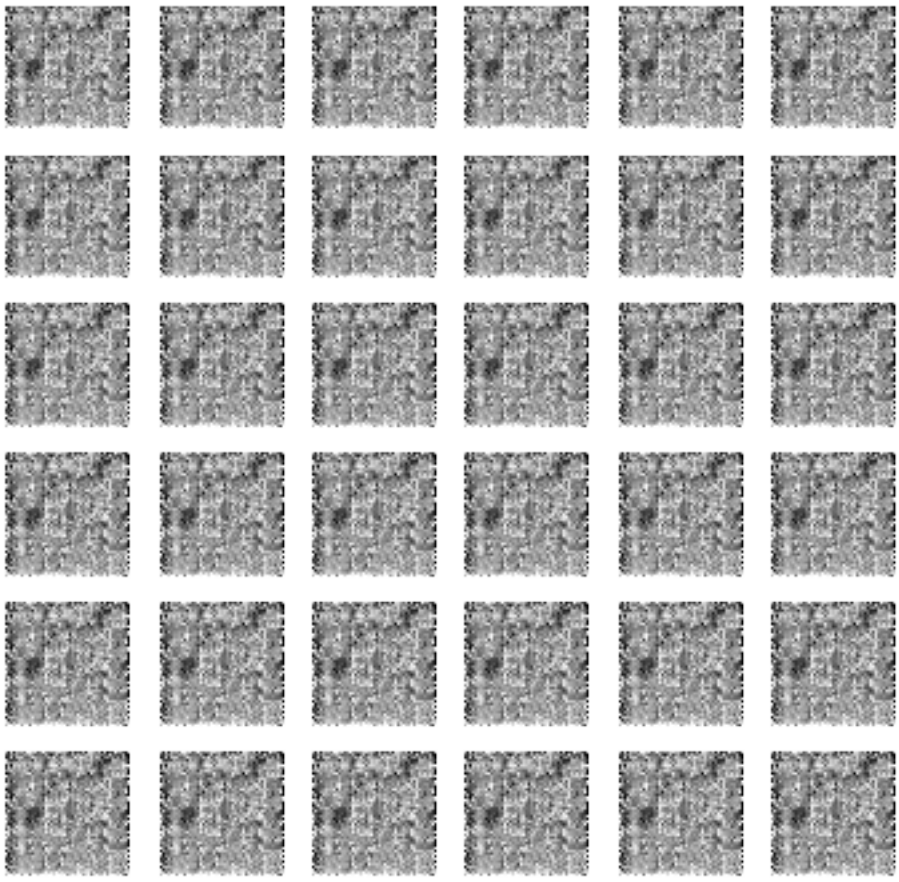
```
Epoch [3/30], Step [200/468], d_loss: 0.6667, g_loss: 1.5133, D(x): 0.76, D(G(z)): 0.29
Epoch [3/30], Step [400/468], d_loss: 0.0138, g_loss: 6.7604, D(x): 1.00, D(G(z)): 0.01
```



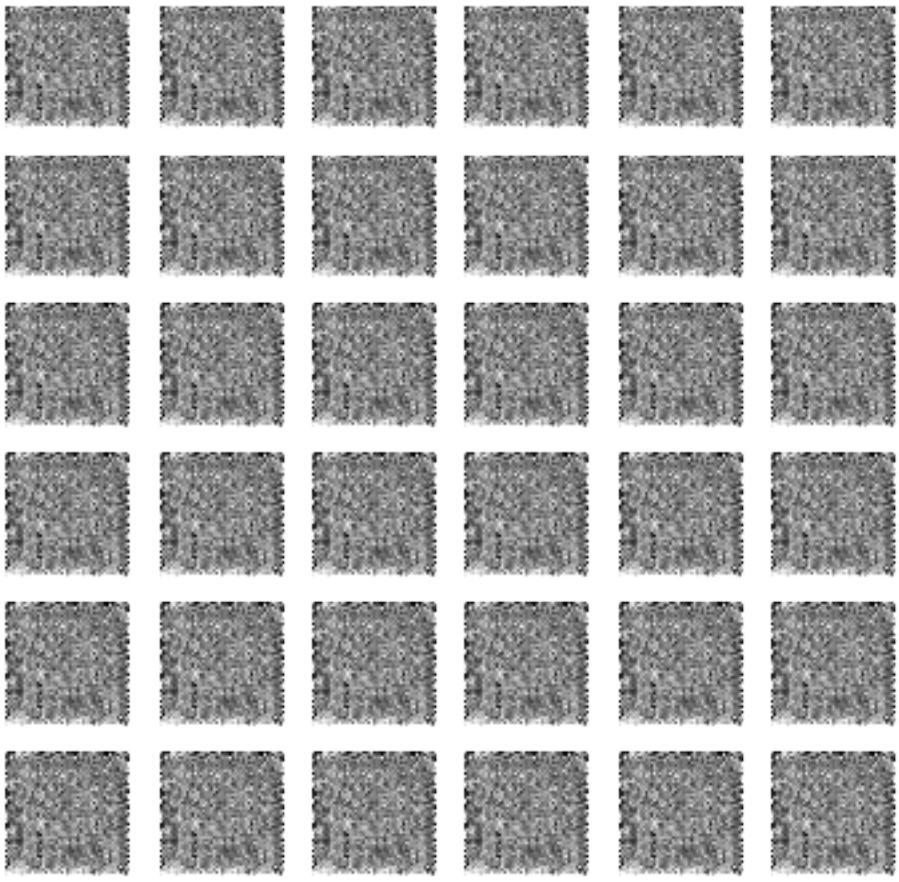
```
Epoch [4/30], Step [200/468], d_loss: 0.0015, g_loss: 7.0911, D(x): 1.00, D(G(z)): 0.00
Epoch [4/30], Step [400/468], d_loss: 0.0016, g_loss: 6.9488, D(x): 1.00, D(G(z)): 0.00
```



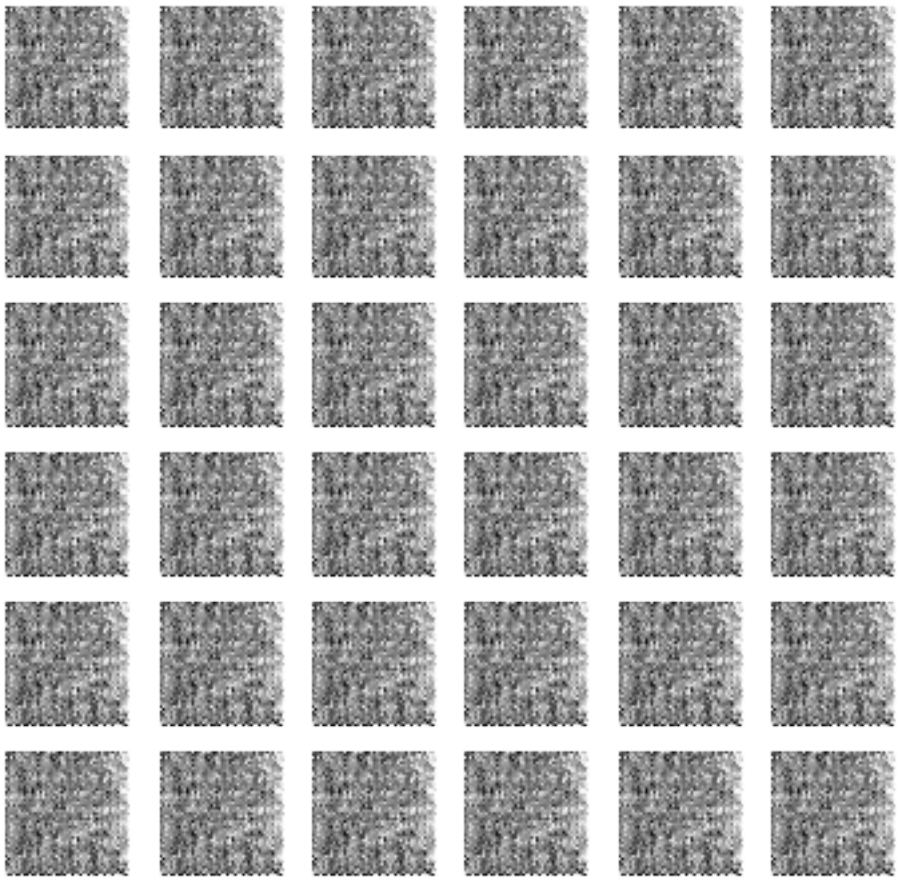
```
Epoch [5/30], Step [200/468], d_loss: 0.0003, g_loss: 8.7054, D(x): 1.00, D(G(z)): 0.00
Epoch [5/30], Step [400/468], d_loss: 0.0004, g_loss: 8.5067, D(x): 1.00, D(G(z)): 0.00
```



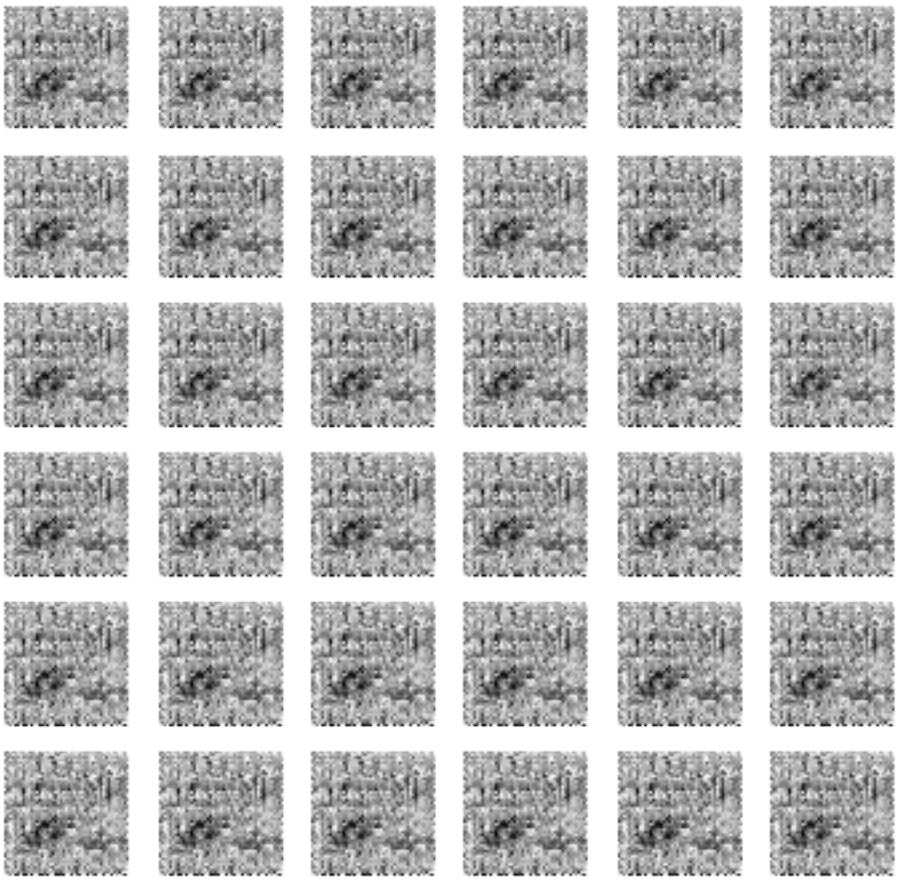
```
Epoch [6/30], Step [200/468], d_loss: 0.0002, g_loss: 8.6291, D(x): 1.00, D(G(z)): 0.00
Epoch [6/30], Step [400/468], d_loss: 0.0002, g_loss: 8.8919, D(x): 1.00, D(G(z)): 0.00
```



```
Epoch [7/30], Step [200/468], d_loss: 0.0002, g_loss: 8.7816, D(x): 1.00, D(G(z)): 0.00
Epoch [7/30], Step [400/468], d_loss: 0.0001, g_loss: 9.6698, D(x): 1.00, D(G(z)): 0.00
```



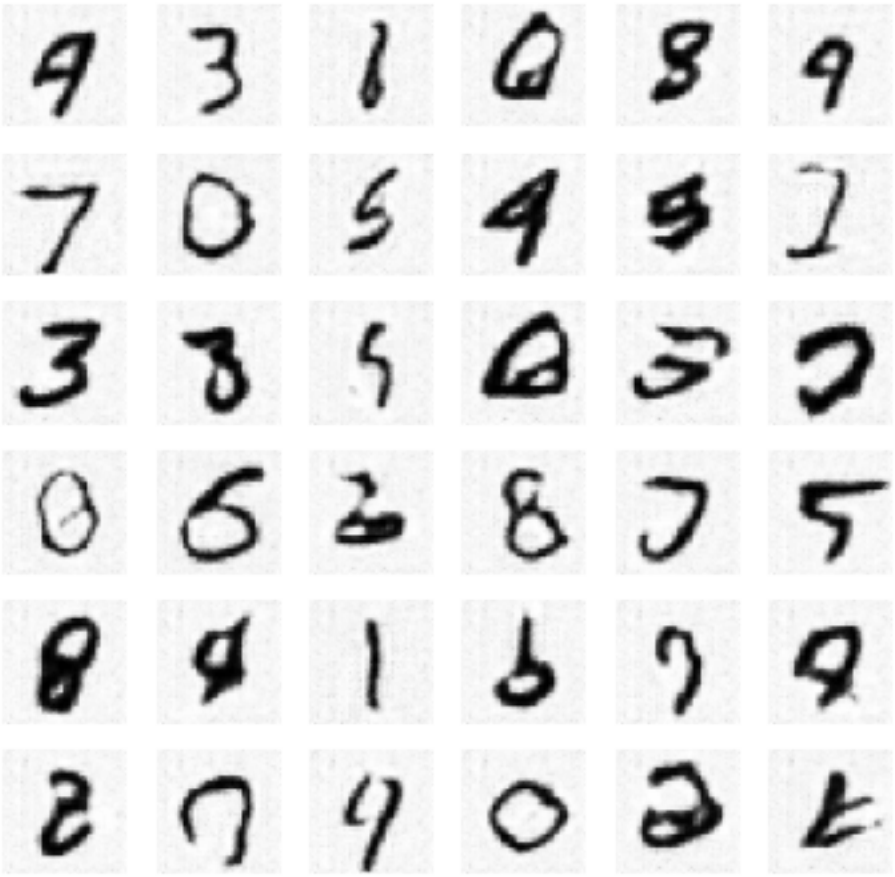
```
Epoch [8/30], Step [200/468], d_loss: 0.0001, g_loss: 10.1828, D(x): 1.00, D(G(z)): 0.00
Epoch [8/30], Step [400/468], d_loss: 0.0001, g_loss: 10.0996, D(x): 1.00, D(G(z)): 0.00
```



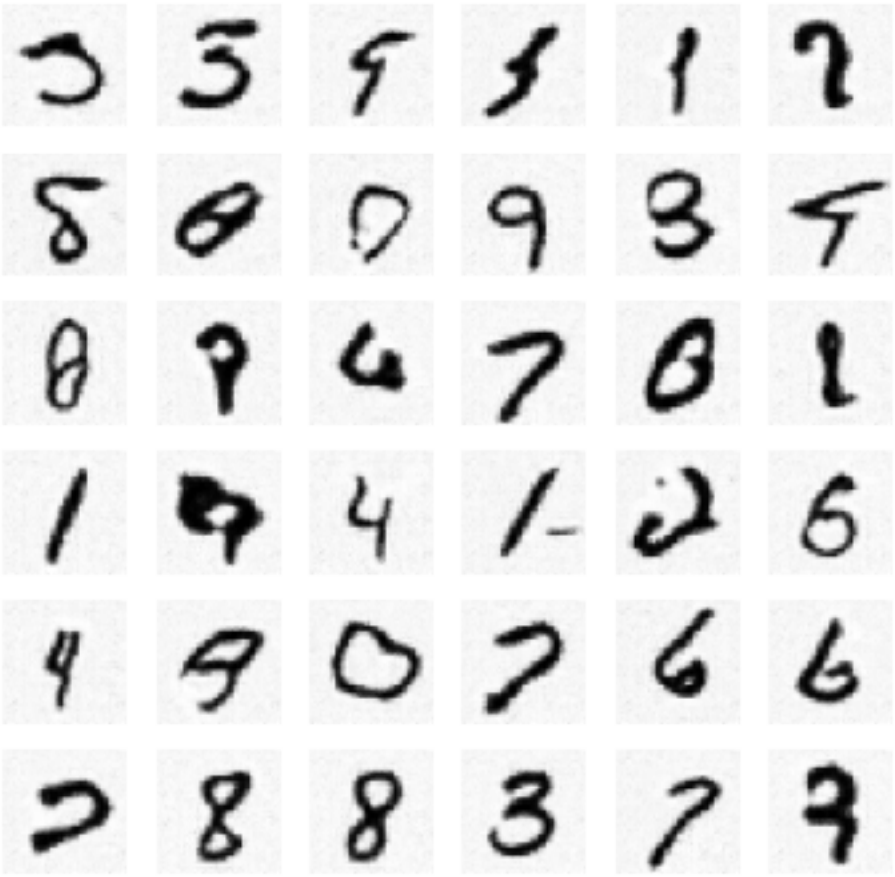
```
Epoch [9/30], Step [200/468], d_loss: 0.0710, g_loss: 7.1816, D(x): 0.94, D(G(z)): 0.00
Epoch [9/30], Step [400/468], d_loss: 1.4461, g_loss: 2.0296, D(x): 0.31, D(G(z)): 0.00
```

```
Epoch [10/30], Step [200/468], d_loss: 0.3965, g_loss: 2.1246, D(x): 0.77, D(G(z)): 0.11
Epoch [10/30], Step [400/468], d_loss: 0.3616, g_loss: 3.4598, D(x): 0.90, D(G(z)): 0.22
```

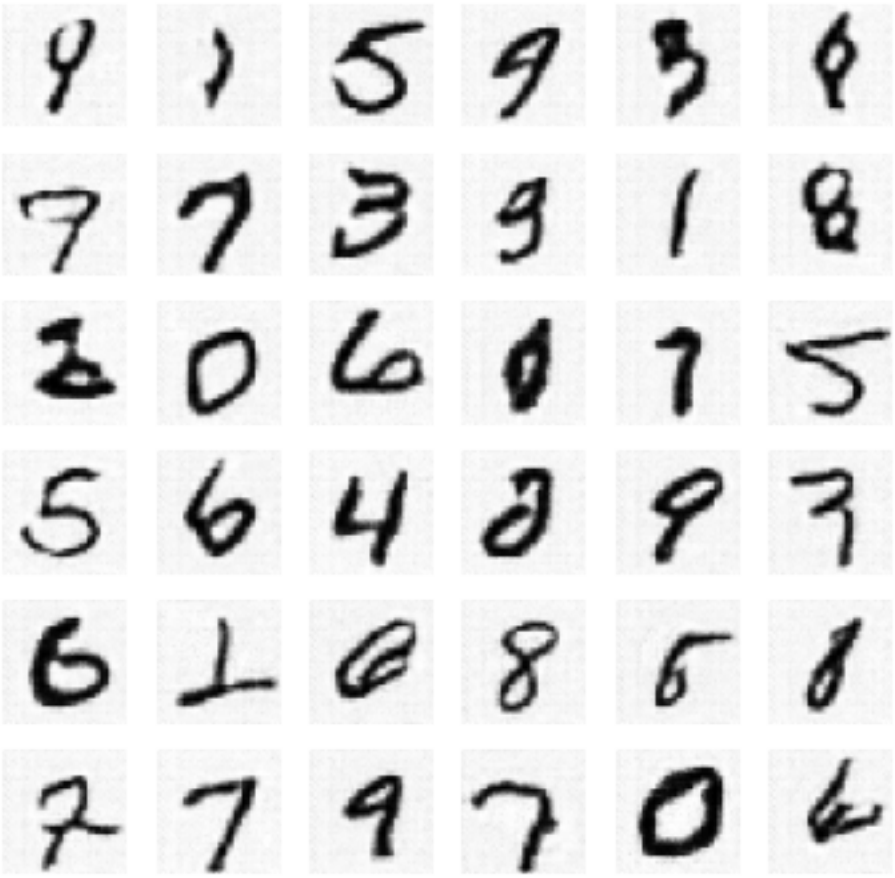
```
Epoch [11/30], Step [200/468], d_loss: 1.5872, g_loss: 0.7308, D(x): 0.27, D(G(z)): 0.02
Epoch [11/30], Step [400/468], d_loss: 1.4905, g_loss: 1.3133, D(x): 0.84, D(G(z)): 0.67
```



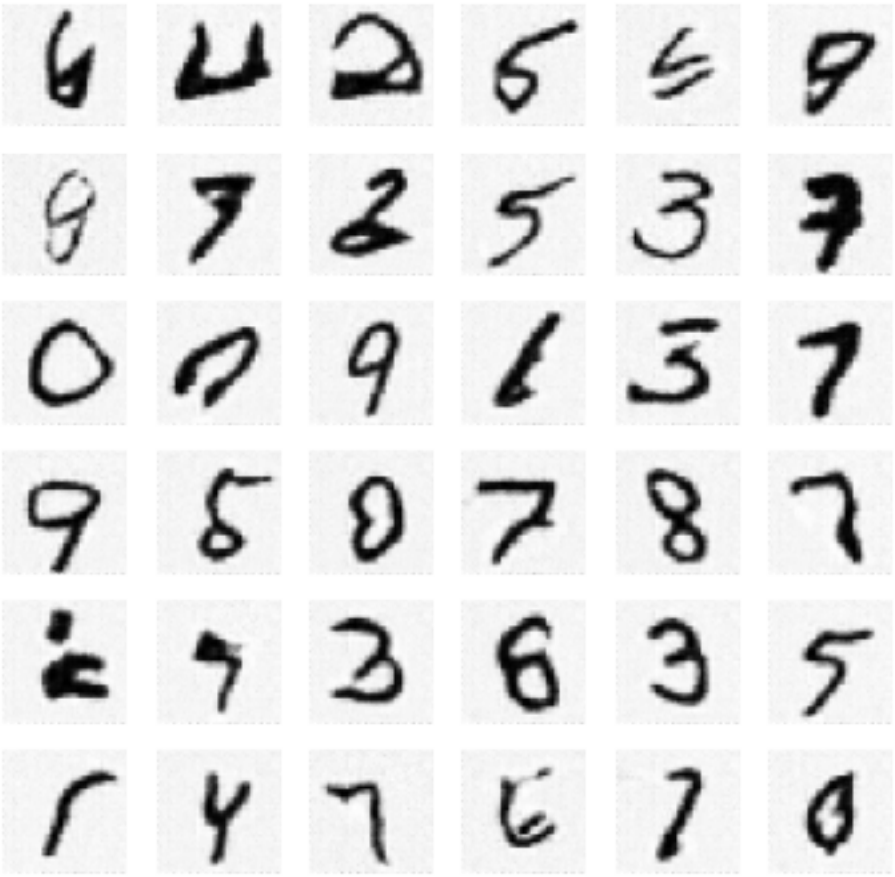
```
Epoch [12/30], Step [200/468], d_loss: 0.1648, g_loss: 2.2819, D(x): 0.89, D(G(z)): 0.04
Epoch [12/30], Step [400/468], d_loss: 0.4075, g_loss: 1.5605, D(x): 0.71, D(G(z)): 0.01
```



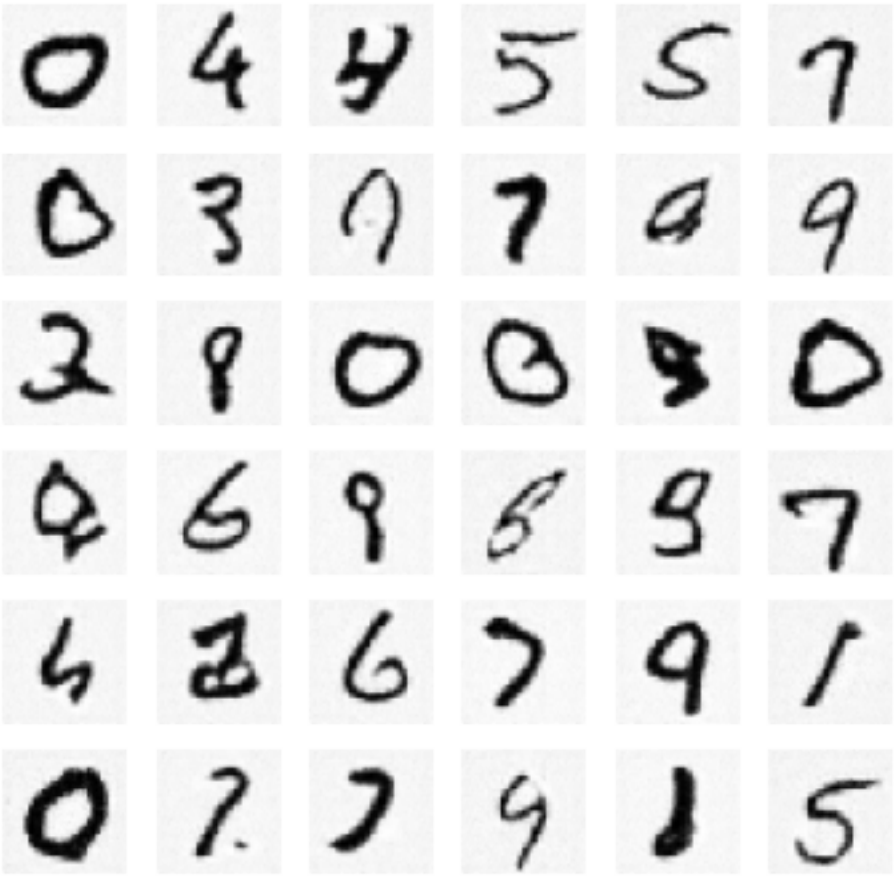
```
Epoch [13/30], Step [200/468], d_loss: 0.0768, g_loss: 3.7021, D(x): 0.97, D(G(z)): 0.04
Epoch [13/30], Step [400/468], d_loss: 0.0620, g_loss: 4.0260, D(x): 0.96, D(G(z)): 0.01
```



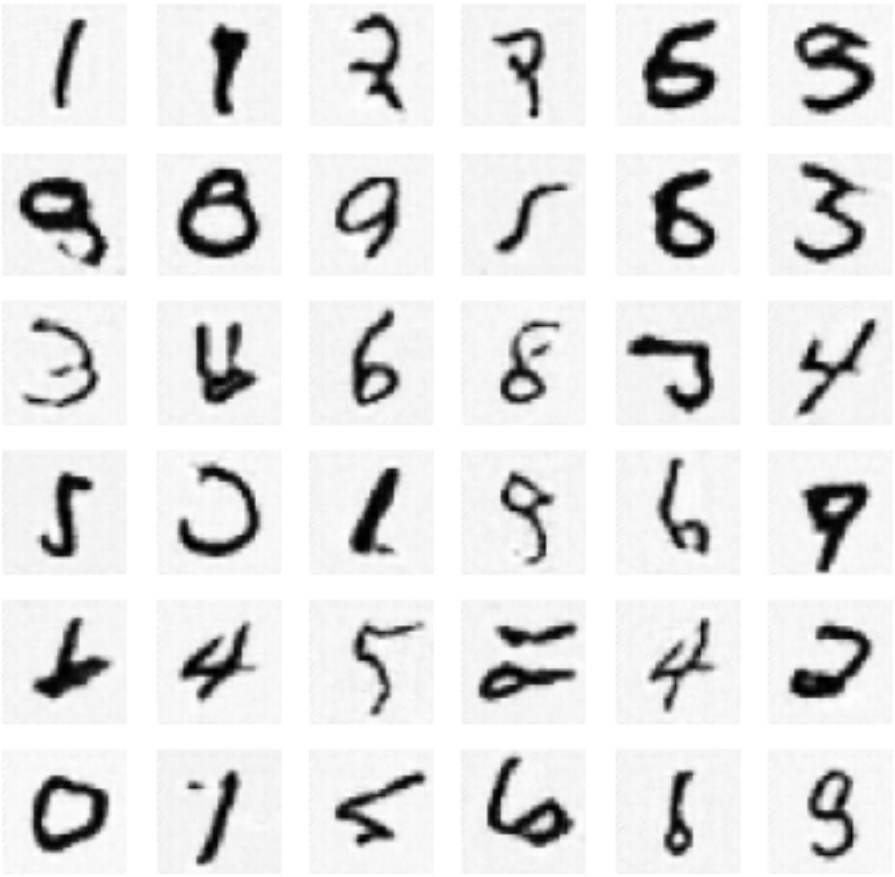
```
Epoch [14/30], Step [200/468], d_loss: 1.6406, g_loss: 6.9093, D(x): 1.00, D(G(z)): 0.74
Epoch [14/30], Step [400/468], d_loss: 4.9307, g_loss: 6.9496, D(x): 1.00, D(G(z)): 0.99
```



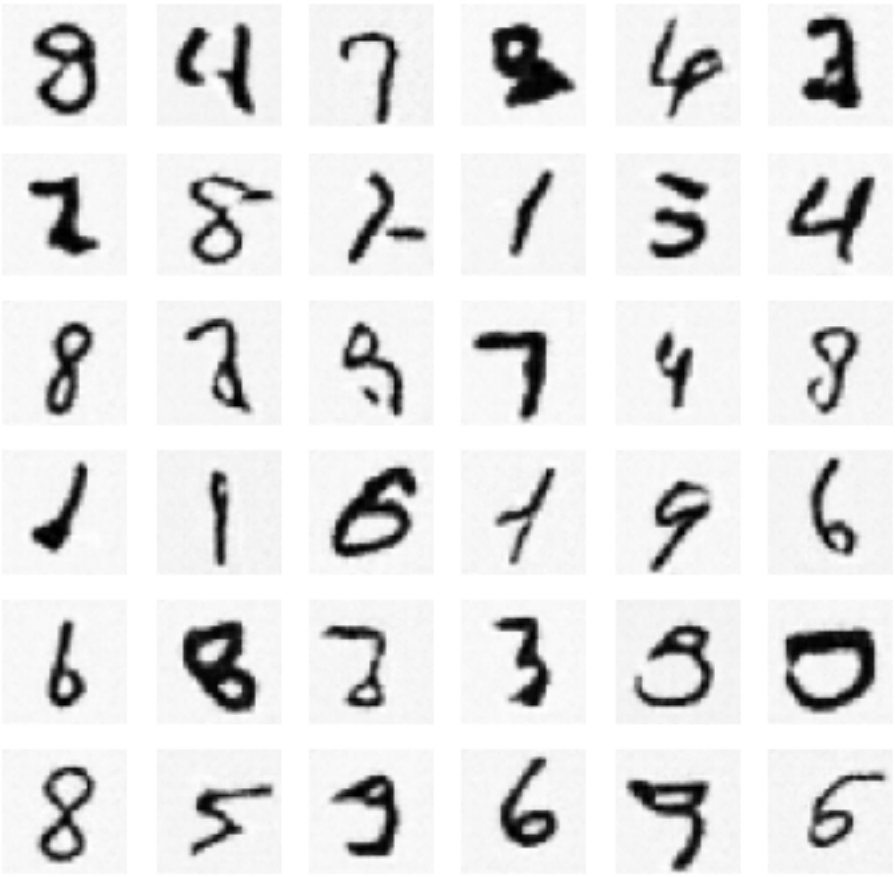
```
Epoch [15/30], Step [200/468], d_loss: 0.0896, g_loss: 3.9697, D(x): 0.95, D(G(z)): 0.03
Epoch [15/30], Step [400/468], d_loss: 0.1241, g_loss: 3.2350, D(x): 0.97, D(G(z)): 0.09
```



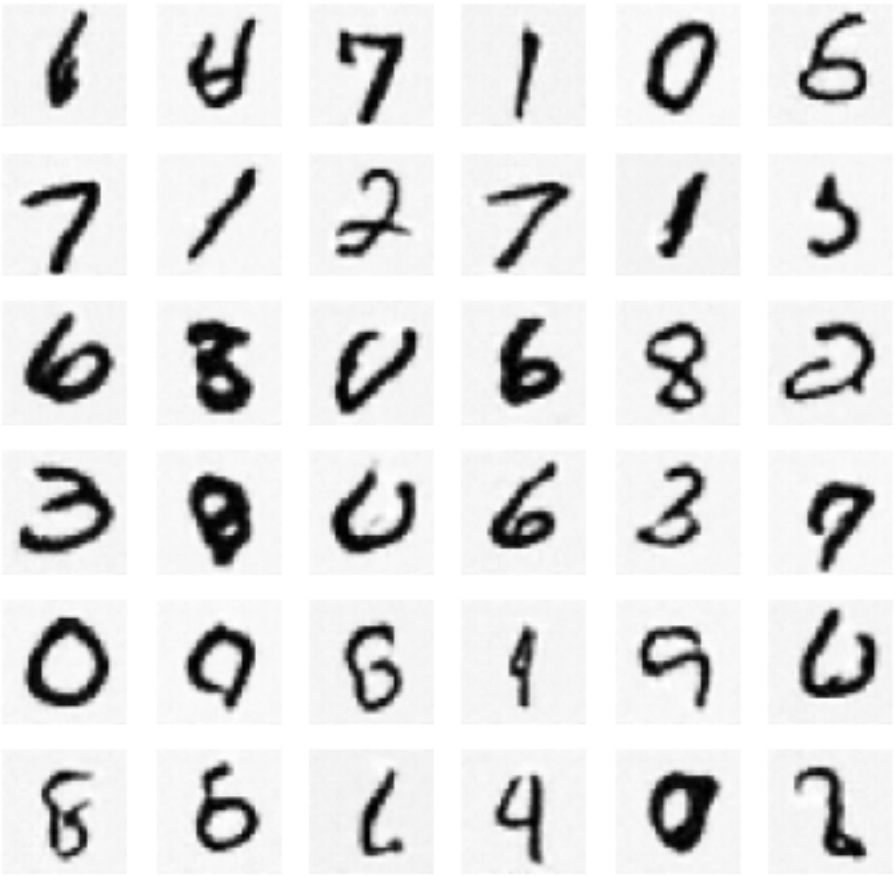
```
Epoch [16/30], Step [200/468], d_loss: 0.1439, g_loss: 3.2796, D(x): 0.89, D(G(z)): 0.02
Epoch [16/30], Step [400/468], d_loss: 0.2105, g_loss: 3.2038, D(x): 0.90, D(G(z)): 0.09
```



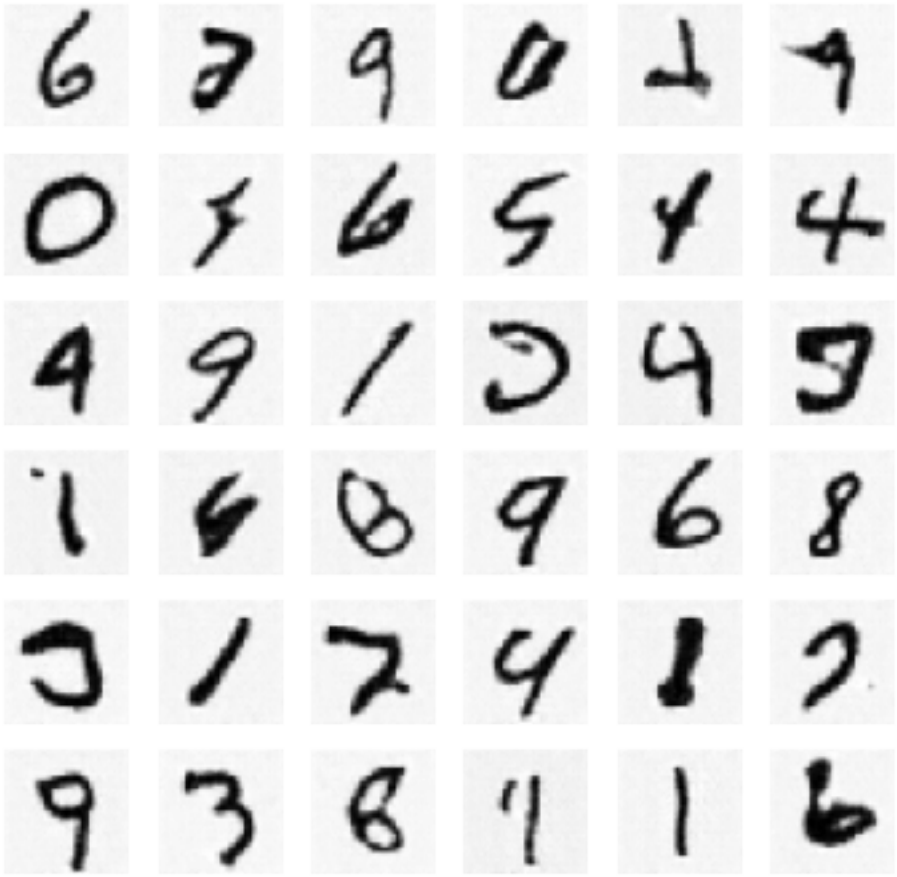
```
Epoch [17/30], Step [200/468], d_loss: 0.0258, g_loss: 4.3833, D(x): 0.99, D(G(z)): 0.02
Epoch [17/30], Step [400/468], d_loss: 0.2890, g_loss: 4.4560, D(x): 0.96, D(G(z)): 0.20
```



```
Epoch [18/30], Step [200/468], d_loss: 1.9407, g_loss: 4.2014, D(x): 0.99, D(G(z)): 0.78
Epoch [18/30], Step [400/468], d_loss: 6.3546, g_loss: 0.1717, D(x): 0.00, D(G(z)): 0.00
```



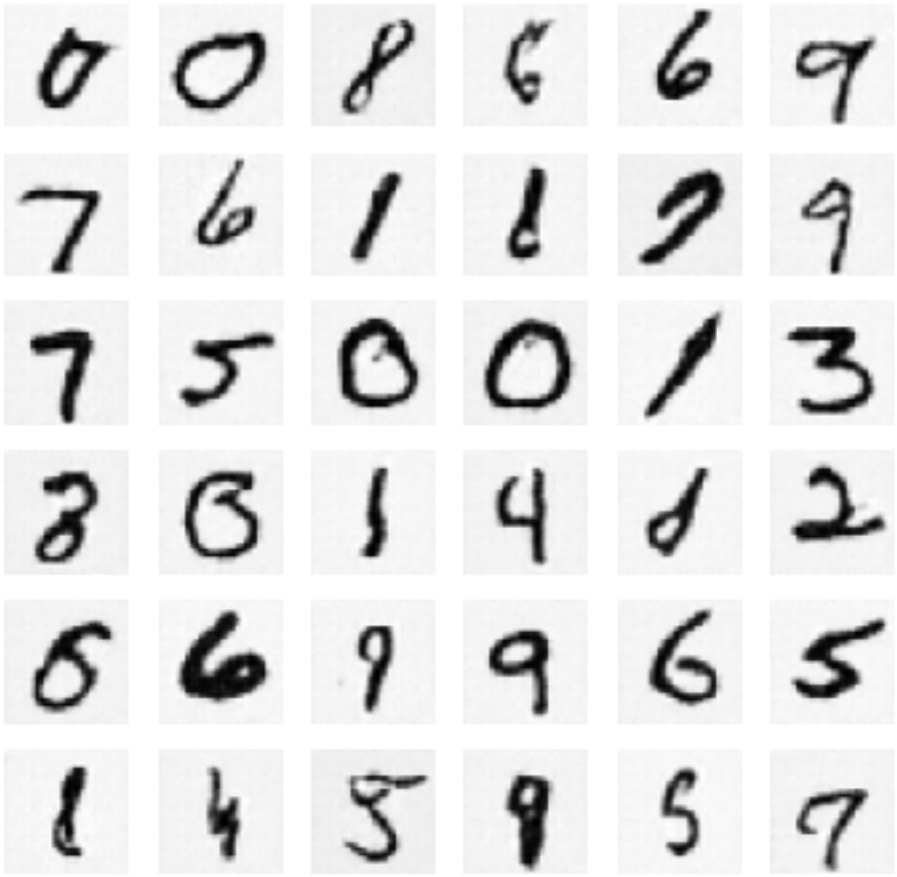
```
Epoch [19/30], Step [200/468], d_loss: 0.0346, g_loss: 4.1575, D(x): 0.98, D(G(z)): 0.02
Epoch [19/30], Step [400/468], d_loss: 0.0306, g_loss: 4.7252, D(x): 0.99, D(G(z)): 0.02
```



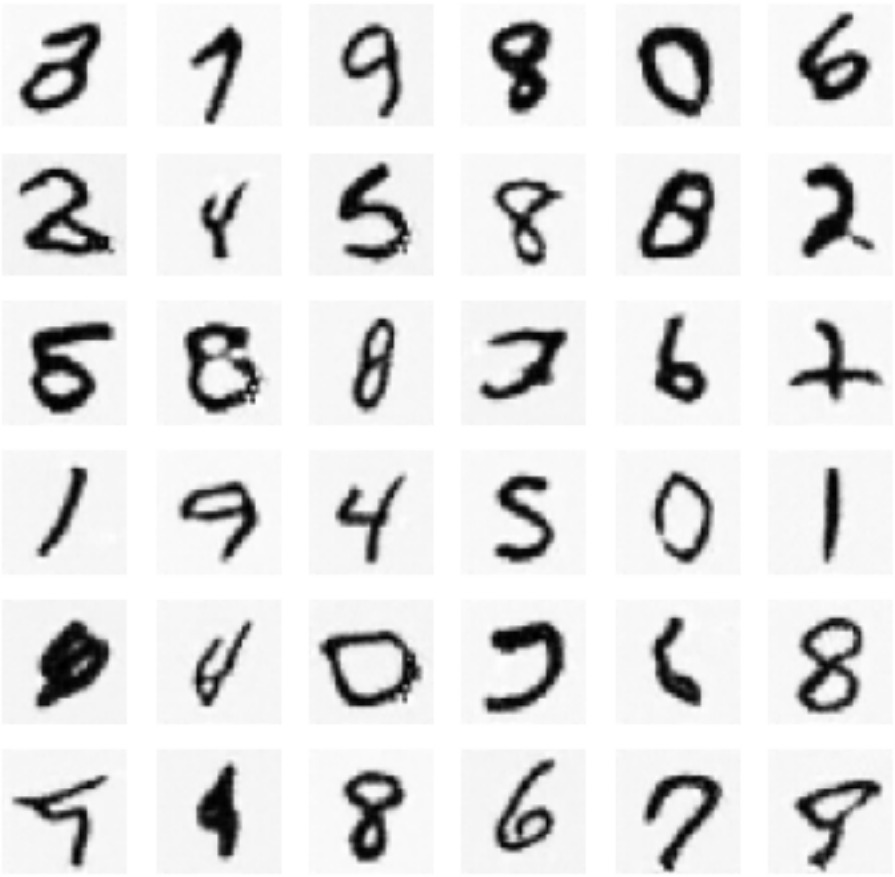
```
Epoch [20/30], Step [200/468], d_loss: 0.1457, g_loss: 3.4130, D(x): 0.96, D(G(z)): 0.09
Epoch [20/30], Step [400/468], d_loss: 0.0549, g_loss: 4.5773, D(x): 0.95, D(G(z)): 0.01
```

```
Epoch [21/30], Step [200/468], d_loss: 0.0332, g_loss: 4.8634, D(x): 0.98, D(G(z)): 0.01
Epoch [21/30], Step [400/468], d_loss: 0.4759, g_loss: 2.8831, D(x): 0.93, D(G(z)): 0.30
```

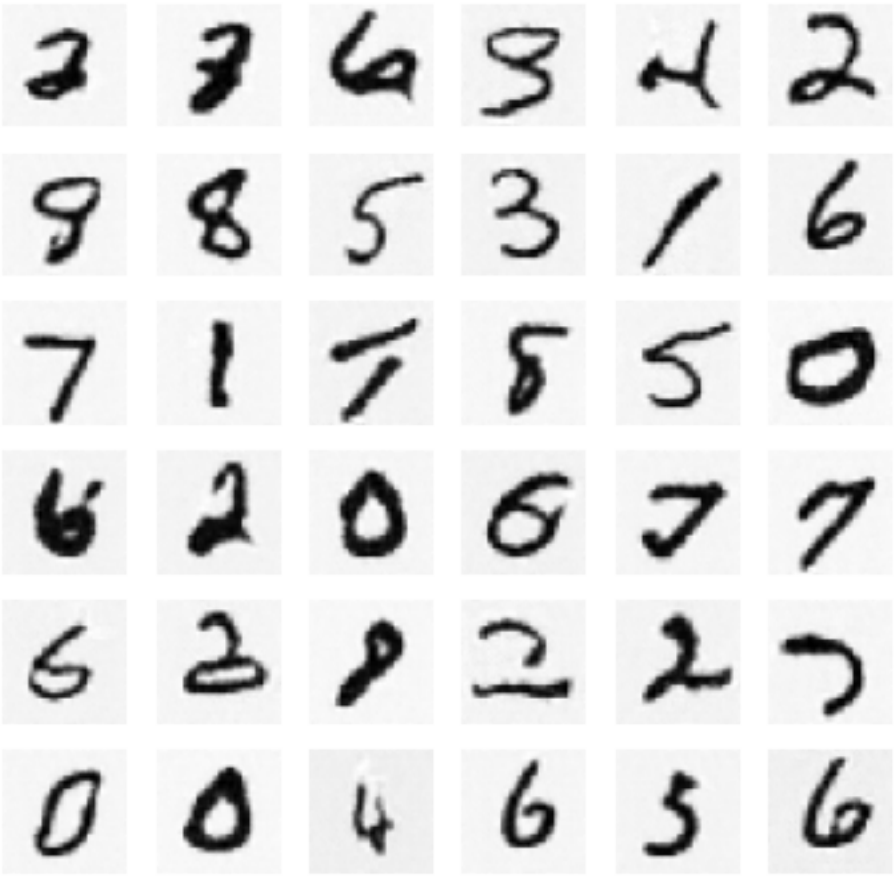
```
Epoch [22/30], Step [200/468], d_loss: 0.0345, g_loss: 4.3618, D(x): 0.98, D(G(z)): 0.01
Epoch [22/30], Step [400/468], d_loss: 0.7046, g_loss: 1.3828, D(x): 0.69, D(G(z)): 0.24
```



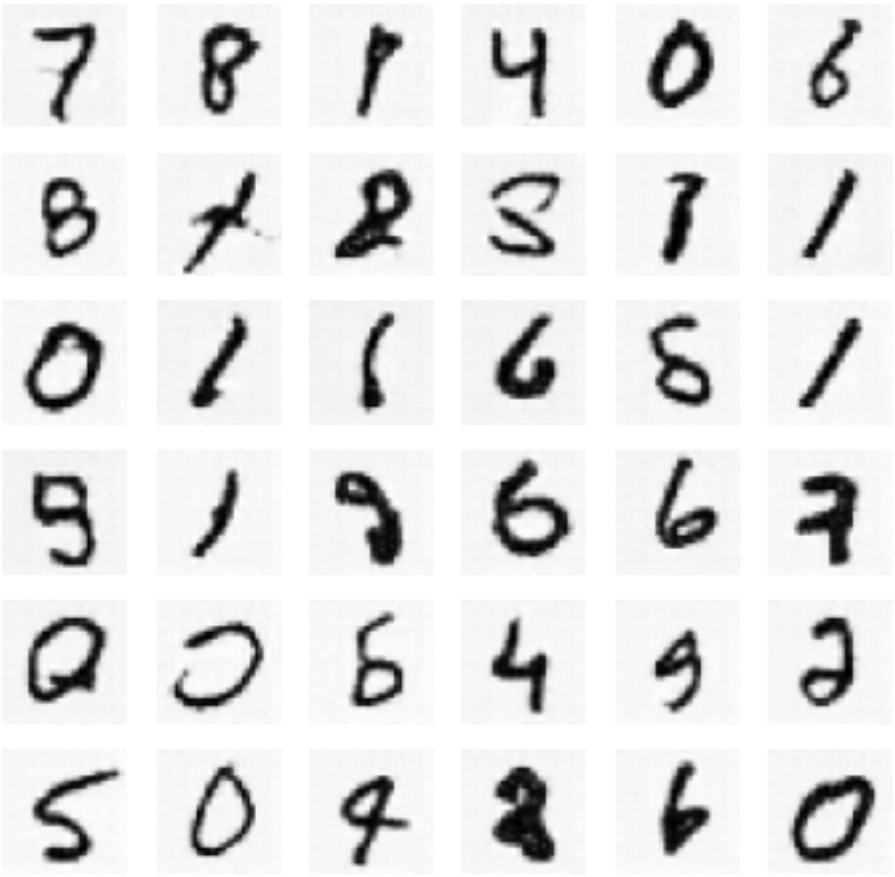
```
Epoch [23/30], Step [200/468], d_loss: 0.0296, g_loss: 5.2937, D(x): 0.98, D(G(z)): 0.01
Epoch [23/30], Step [400/468], d_loss: 0.0142, g_loss: 4.3330, D(x): 0.99, D(G(z)): 0.01
```



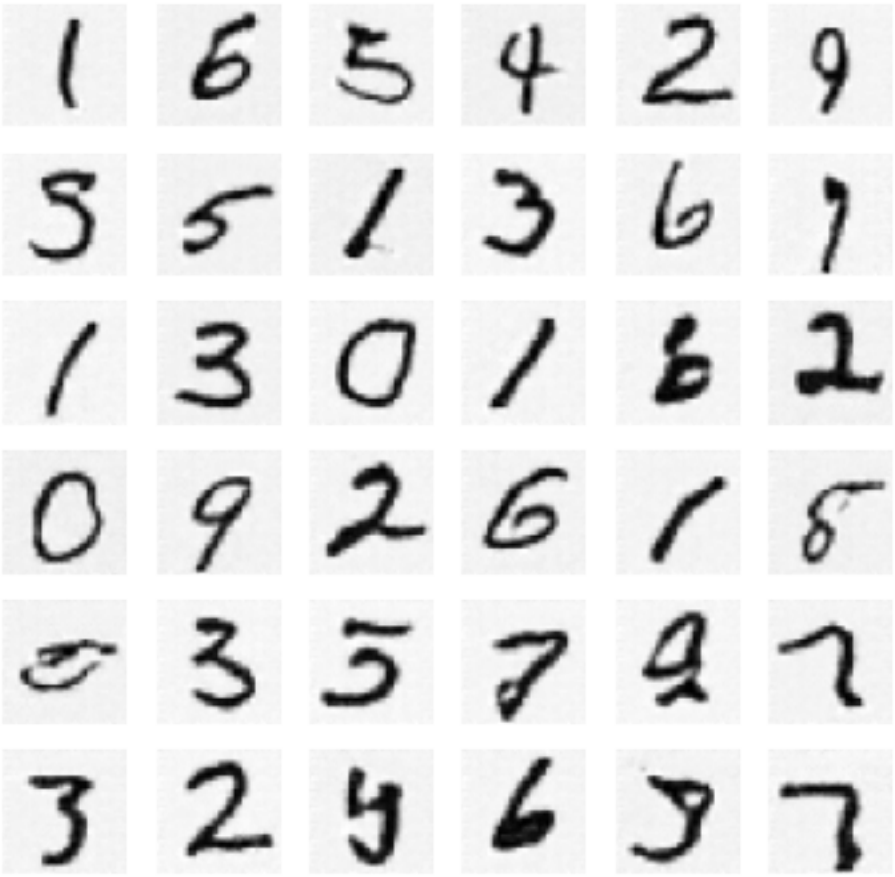
```
Epoch [24/30], Step [200/468], d_loss: 1.0245, g_loss: 1.9638, D(x): 0.82, D(G(z)): 0.49
Epoch [24/30], Step [400/468], d_loss: 0.1030, g_loss: 3.2958, D(x): 0.95, D(G(z)): 0.05
```



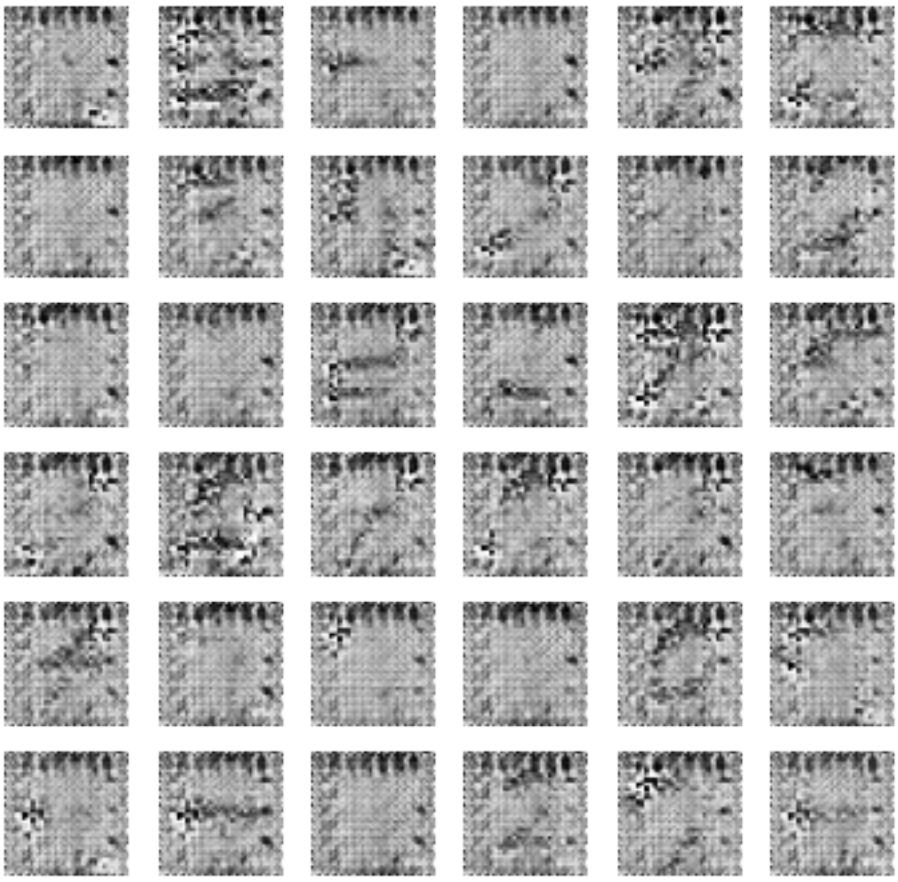
```
Epoch [25/30], Step [200/468], d_loss: 0.0761, g_loss: 3.8846, D(x): 0.97, D(G(z)): 0.05
Epoch [25/30], Step [400/468], d_loss: 0.0130, g_loss: 6.0150, D(x): 0.99, D(G(z)): 0.01
```



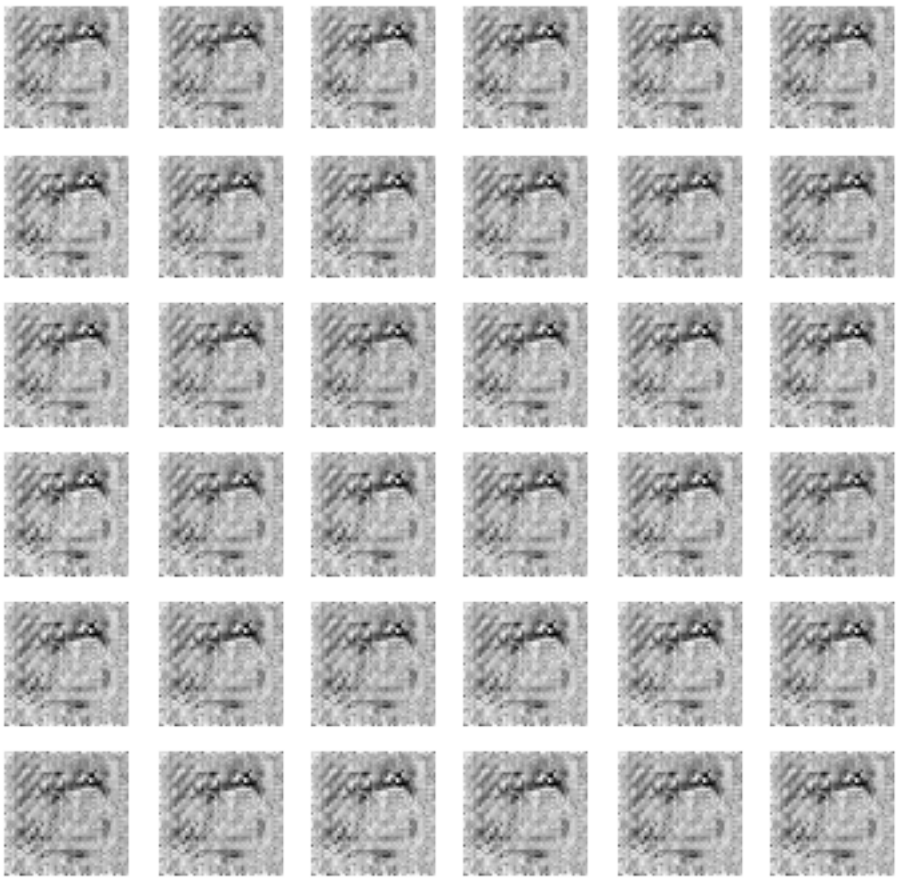
```
Epoch [26/30], Step [200/468], d_loss: 1.0556, g_loss: 1.9765, D(x): 0.85, D(G(z)): 0.52
Epoch [26/30], Step [400/468], d_loss: 0.1076, g_loss: 1.9102, D(x): 0.91, D(G(z)): 0.01
```



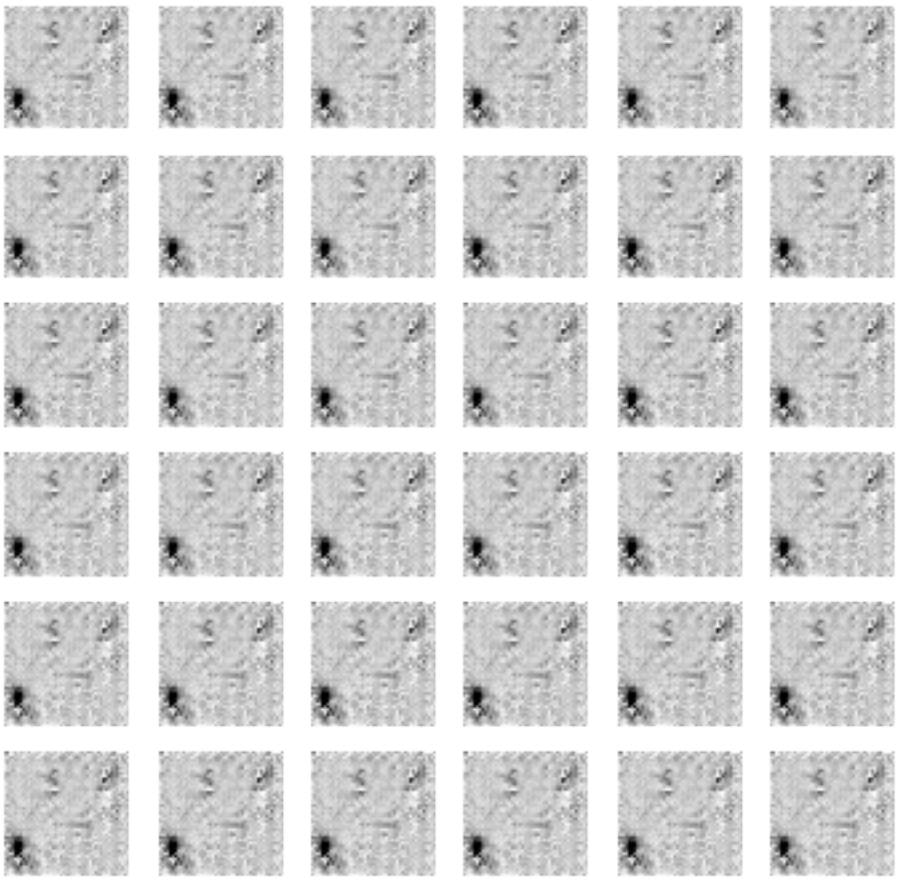
```
Epoch [27/30], Step [200/468], d_loss: 0.0257, g_loss: 4.1618, D(x): 0.99, D(G(z)): 0.01
Epoch [27/30], Step [400/468], d_loss: 0.0129, g_loss: 5.6174, D(x): 0.99, D(G(z)): 0.01
```



```
Epoch [28/30], Step [200/468], d_loss: 0.0039, g_loss: 7.0930, D(x): 1.00, D(G(z)): 0.00
Epoch [28/30], Step [400/468], d_loss: 0.0012, g_loss: 7.5325, D(x): 1.00, D(G(z)): 0.00
```



```
Epoch [29/30], Step [200/468], d_loss: 0.0006, g_loss: 7.8921, D(x): 1.00, D(G(z)): 0.00
Epoch [29/30], Step [400/468], d_loss: 0.0005, g_loss: 8.0703, D(x): 1.00, D(G(z)): 0.00
```



1.6 Bonus Task

Congratulations, you've successfully trained a DCGAN that can generate MNIST numbers. If you are brave, it's time to generate some more complex images.

Task: Have a look at the datasets that are included in `torchvision` and train a DCGAN with another dataset.

1.6.1 Feedback

That's it, we're done

If you have any suggestions on how we could improve this session, please let us know in the following cell. What did you particularly like or dislike? Did you miss any contents?