Introduction to Generative Modeling

Deep neural networks are used mainly for supervised learning: classification or regression. Generative Adversarial Networks or GANs, however, use neural networks for a very different purpose: Generative modeling

> Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset. - Source

To get a sense of the power of generative models, just visit thispersondoesnotexist.com. Every time you reload the page, a new image of a person's face is generated on the fly. The results are pretty fascinating:

While there are many approaches used for generative modeling, a Generative Adversarial Network takes the following approach:

There are two neural networks: a *Generator* and a *Discriminator*. The generator generates a "fake" sample given a random vector/matrix, and the discriminator attempts to detect whether a given sample is "real" (picked from the training data) or "fake" (generated by the generator). Training happens in tandem: we train the discriminator for a few epochs, then train the generator for a few epochs, and repeat. This way both the generator and the discriminator get better at doing their jobs.

GANs however, can be notoriously difficult to train, and are extremely sensitive to hyperparameters, activation functions and regularization. In this tutorial, we'll train a GAN to generate images of anime characters' faces.

We'll use the Anime Face Dataset, which consists of over 63,000 cropped anime faces. Note that generative modeling is an unsupervised learning task, so the images do not have any labels. Most of the code in this tutorial is based on this notebook.

```
project_name = 'anime-dcgan'

# Uncomment and run the appropriate command for your operating system, if required
# No installation is reqiured on Google Colab / Kaggle notebooks

# Linux / Binder / Windows (No GPU)
# !pip install numpy matplotlib torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==0.7.0 -f https://download.pytorch.org/whl/torch_stable.html
```

```
# Linux / Windows (GPU)
# pip install numpy matplotlib torch==1.7.1+cu110
torchvision==0.8.2+cu110 torchaudio==0.7.2 -f
https://download.pytorch.org/whl/torch_stable.html

# MacOS (NO GPU)
# !pip install numpy matplotlib torch torchvision torchaudio
```

Downloading and Exploring the Data

We can use the `opendatasets` library to download the dataset from Kaggle. `opendatasets` uses the Kaggle Official API for downloading datasets from Kaggle. Follow these steps to find your API credentials:

1. Sign in to https://kaggle.com/, then click on your profile picture on the top right and select "My Account" from the menu.

2. Scroll down to the "API" section and click "Create New API Token". This will download a file `kaggle.json` with the following contents:

```
{"username":"YOUR_KAGGLE_USERNAME","key":"YOUR_KAGGLE_KEY"}
```

1. When you run `opendatsets.download`, you will be asked to enter your username & Kaggle API, which you can get from the file downloaded in step 2.

Note that you need to download the `kaggle.json` file only once. On Google Colab, you can also upload the `kaggle.json` file using the files tab, and the credentials will be read automatically.

```
!pip install opendatasets --upgrade --quiet

import opendatasets as od

dataset_url = 'https://www.kaggle.com/splcher/animefacedataset'
od.download(dataset_url)

Please provide your Kaggle credentials to download this dataset. Learn
more: http://bit.ly/kaggle-creds
Your Kaggle username: sid3503
Your Kaggle Key: ··········
Dataset URL: https://www.kaggle.com/datasets/splcher/animefacedataset
```

The dataset has a single folder called `images` which contains all 63,000+ images in JPG format.

```
import os

DATA_DIR = './animefacedataset'
print(os.listdir(DATA_DIR))
```

```
['images']
```

```
print(os.listdir(DATA_DIR+'/images')[:10])
```

```
['23703_2008.jpg', '39426_2012.jpg', '37908_2012.jpg',
'42249_2013.jpg', '49181_2015.jpg', '8801_2004.jpg', '53237_2016.jpg',
'13030_2005.jpg', '10352_2004.jpg', '16249_2006.jpg']
```

Let's load this dataset using the `ImageFolder` class from `torchvision`. We will also resize and crop the images to 64x64 px, and normalize the pixel values with a mean & standard deviation of 0.5 for each channel. This will ensure that pixel values are in the range `(-1, 1)`, which is more convenient for training the discriminator. We will also create a data loader to load the data in batches.

```
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
import torchvision.transforms as T

image_size = 64
batch_size = 128
stats = (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)

train_ds = ImageFolder(DATA_DIR, transform=T.Compose([
    T.Resize(image_size),
    T.CenterCrop(image_size),
    T.ToTensor(),
    T.Normalize(*stats)]))

train_dl = DataLoader(train_ds, batch_size, shuffle=True,
num_workers=3, pin_memory=True)
```

```
/usr/local/lib/python3.11/dist-packages/torch/utils/data/
dataloader.py:624: UserWarning: This DataLoader will create 3 worker
processes in total. Our suggested max number of worker in current
system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get
DataLoader running slow or even freeze, lower the worker number to
avoid potential slowness/freeze if necessary.
  warnings.warn(
```

Let's create helper functions to denormalize the image tensors and display some sample images from a training batch.

```
import torch
from torchvision.utils import make_grid
import matplotlib.pyplot as plt
%matplotlib inline

def denorm(img_tensors):
    return img_tensors * stats[1][0] + stats[0][0]
```

```
def show_images(images, nmax=64):
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(make_grid(denorm(images.detach()[:nmax]),
nrow=8).permute(1, 2, 0))

def show_batch(dl, nmax=64):
    for images, _ in dl:
        show_images(images, nmax)
        break

show_batch(train_dl)
```

Using a GPU

To seamlessly use a GPU, if one is available, we define a couple of helper functions (`get_default_device` & `to_device`) and a helper class `DeviceDataLoader` to move our model & data to the GPU, if one is available.

```
def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')

def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)

class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        """Number of batches"""
        return len(self.dl)
```

Based on where you're running this notebook, your default device could be a CPU (`torch.device('cpu')`) or a GPU (`torch.device('cuda')`).

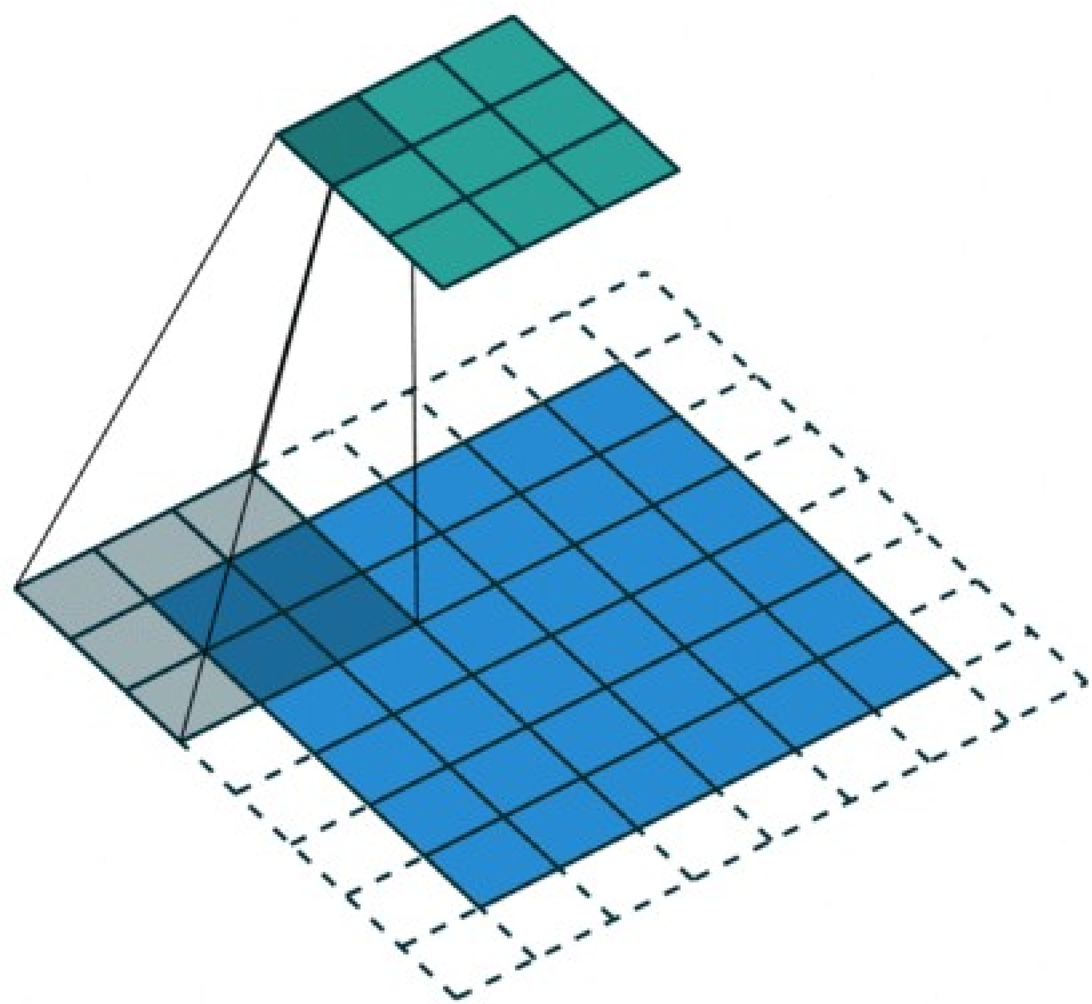
```
device = get_default_device()
device

device(type='cuda')
```

We can now move our training data loader using `DeviceDataLoader` for automatically transferring batches of data to the GPU (if available).

```
train_dl = DeviceDataLoader(train_dl, device)
```

Discriminator Network

The discriminator takes an image as input, and tries to classify it as "real" or "generated". In this sense, it's like any other neural network. We'll use a convolutional neural networks (CNN) which outputs a single number output for every image. We'll use stride of 2 to progressively reduce the size of the output feature map.



```
import torch.nn as nn

discriminator = nn.Sequential(
    # in: 3 x 64 x 64

    nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 64 x 32 x 32
```

```
    nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1,
bias=False),
    nn.BatchNorm2d(128),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 128 x 16 x 16

    nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1,
bias=False),
    nn.BatchNorm2d(256),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 256 x 8 x 8

    nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1,
bias=False),
    nn.BatchNorm2d(512),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 512 x 4 x 4

    nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0, bias=False),
    # out: 1 x 1 x 1

    nn.Flatten(),
    nn.Sigmoid())
```

Note that we're using the Leaky ReLU activation for the discriminator.

> Different from the regular ReLU function, Leaky ReLU allows the pass of a small gradient signal for negative values. As a result, it makes the gradients from the discriminator flows stronger into the generator. Instead of passing a gradient (slope) of 0 in the back-prop pass, it passes a small negative gradient. - Source

Just like any other binary classification model, the output of the discriminator is a single number between 0 and 1, which can be interpreted as the probability of the input image being real i.e. picked from the original dataset.

Let's move the discriminator model to the chosen device.

```
discriminator = to_device(discriminator, device)
```

Generator Network

The input to the generator is typically a vector or a matrix of random numbers (referred to as a latent tensor) which is used as a seed for generating an image. The generator will convert a latent tensor of shape `(128, 1, 1)` into an image tensor of shape `3 x 28 x 28`. To achive this, we'll use the `ConvTranspose2d` layer from PyTorch, which is performs to as a *transposed convolution* (also referred to as a *deconvolution*). Learn more

```
latent_size = 128

generator = nn.Sequential(
    # in: latent_size x 1 x 1

    nn.ConvTranspose2d(latent_size, 512, kernel_size=4, stride=1,
padding=0, bias=False),
    nn.BatchNorm2d(512),
    nn.ReLU(True),
    # out: 512 x 4 x 4

    nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1,
bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(True),
    # out: 256 x 8 x 8

    nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1,
bias=False),
    nn.BatchNorm2d(128),
    nn.ReLU(True),
    # out: 128 x 16 x 16

    nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1,
bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(True),
    # out: 64 x 32 x 32

    nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1,
bias=False),
    nn.Tanh()
    # out: 3 x 64 x 64
)
```

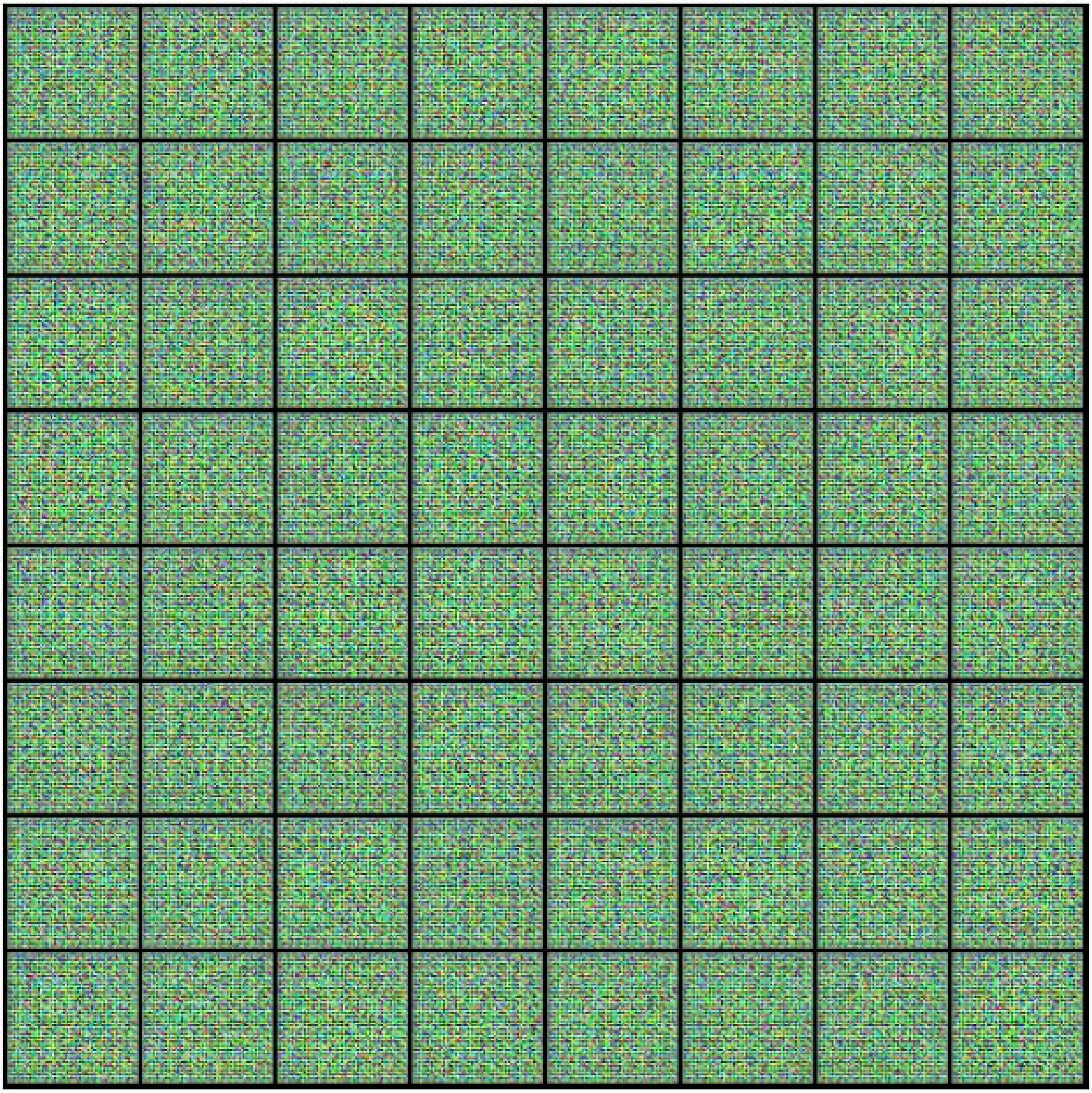
We use the TanH activation function for the output layer of the generator.

> "The ReLU activation (Nair & Hinton, 2010) is used in the generator with the exception of the output layer which uses the Tanh function. We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the color space of the training distribution. Within the discriminator we found the leaky rectified activation (Maas et al., 2013) (Xu et al., 2015) to work well, especially for higher resolution modeling." - Source

Note that since the outputs of the TanH activation lie in the range `[-1,1]`, we have applied the similar transformation to the images in the training dataset. Let's generate some outputs using the generator and view them as images by transforming and denormalizing the output.

```
xb = torch.randn(batch_size, latent_size, 1, 1) # random latent tensors
print(xb.shape)
fake_images = generator(xb)
print(fake_images.shape)
show_images(fake_images)
```

```
torch.Size([128, 128, 1, 1])
torch.Size([128, 3, 64, 64])
```



As one might expect, the output from the generator is basically random noise, since we haven't trained it yet.

Let's move the generator to the chosen device.

```
generator = to_device(generator, device)
```

Discriminator Training

Since the discriminator is a binary classification model, we can use the binary cross entropy loss function to quantify how well it is able to differentiate between real and generated images.

```python
def train_discriminator(real_images, opt_d):
    # Clear discriminator gradients
    opt_d.zero_grad()

    # Pass real images through discriminator
    real_preds = discriminator(real_images)
    real_targets = torch.ones(real_images.size(0), 1, device=device)
    real_loss = F.binary_cross_entropy(real_preds, real_targets)
    real_score = torch.mean(real_preds).item()

    # Generate fake images
    latent = torch.randn(batch_size, latent_size, 1, 1, device=device)
    fake_images = generator(latent)

    # Pass fake images through discriminator
    fake_targets = torch.zeros(fake_images.size(0), 1, device=device)
    fake_preds = discriminator(fake_images)
    fake_loss = F.binary_cross_entropy(fake_preds, fake_targets)
    fake_score = torch.mean(fake_preds).item()

    # Update discriminator weights
    loss = real_loss + fake_loss
    loss.backward()
    opt_d.step()
    return loss.item(), real_score, fake_score
```

Here are the steps involved in training the discriminator.

- We expect the discriminator to output 1 if the image was picked from the real MNIST dataset, and 0 if it was generated using the generator network.
- We first pass a batch of real images, and compute the loss, setting the target labels to 1.
- Then we pass a batch of fake images (generated using the generator) pass them into the discriminator, and compute the loss, setting the target labels to 0.
- Finally we add the two losses and use the overall loss to perform gradient descent to adjust the weights of the discriminator.

It's important to note that we don't change the weights of the generator model while training the discriminator (`opt_d` only affects the `discriminator.parameters()`)

Generator Training

Since the outputs of the generator are images, it's not obvious how we can train the generator. This is where we employ a rather elegant trick, which is to use the discriminator as a part of the loss function. Here's how it works:

- We generate a batch of images using the generator, pass the into the discriminator.

- We calculate the loss by setting the target labels to 1 i.e. real. We do this because the generator's objective is to "fool" the discriminator.
- We use the loss to perform gradient descent i.e. change the weights of the generator, so it gets better at generating real-like images to "fool" the discriminator.

Here's what this looks like in code.

```
def train_generator(opt_g):
    # Clear generator gradients
    opt_g.zero_grad()

    # Generate fake images
    latent = torch.randn(batch_size, latent_size, 1, 1, device=device)
    fake_images = generator(latent)

    # Try to fool the discriminator
    preds = discriminator(fake_images)
    targets = torch.ones(batch_size, 1, device=device)  # main step
    loss = F.binary_cross_entropy(preds, targets)

    # Update generator weights
    loss.backward()
    opt_g.step()

    return loss.item()
```

Let's create a directory where we can save intermediate outputs from the generator to visually inspect the progress of the model. We'll also create a helper function to export the generated images.

```
from torchvision.utils import save_image

sample_dir = 'generated'
os.makedirs(sample_dir, exist_ok=True)

latent_tensors = torch.randn(128, 128, 1, 1)
fake_images = generator(latent_tensors)

----------------------------------------------------------------------
-----
RuntimeError                              Traceback (most recent call
last)
<ipython-input-29-e7f43639d3ca> in <cell line: 0>()
      1 latent_tensors = torch.randn(128, 128, 1, 1)
----> 2 fake_images = generator(latent_tensors)

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in
_wrapped_call_impl(self, *args, **kwargs)
   1737             return self._compiled_call_impl(*args, **kwargs)
```

```
# type: ignore[misc]
   1738         else:
-> 1739             return self._call_impl(*args, **kwargs)
   1740 
   1741     # torchrec tests the code consistency with the following
code

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in
_call_impl(self, *args, **kwargs)
   1748                 or _global_backward_pre_hooks or
_global_backward_hooks
   1749                 or _global_forward_hooks or
_global_forward_pre_hooks):
-> 1750             return forward_call(*args, **kwargs)
   1751 
   1752         result = None

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/container.py
in forward(self, input)
    248     def forward(self, input):
    249         for module in self:
--> 250             input = module(input)
    251         return input
    252 

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in
_wrapped_call_impl(self, *args, **kwargs)
   1737             return self._compiled_call_impl(*args, **kwargs)
# type: ignore[misc]
   1738         else:
-> 1739             return self._call_impl(*args, **kwargs)
   1740 
   1741     # torchrec tests the code consistency with the following
code

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in
_call_impl(self, *args, **kwargs)
   1748                 or _global_backward_pre_hooks or
_global_backward_hooks
   1749                 or _global_forward_hooks or
_global_forward_pre_hooks):
-> 1750             return forward_call(*args, **kwargs)
   1751 
   1752         result = None

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/conv.py in
forward(self, input, output_size)
   1160         )
   1161 
-> 1162         return F.conv_transpose2d(
```

```
   1163                input,
   1164                self.weight,

RuntimeError: Input type (torch.FloatTensor) and weight type
(torch.cuda.FloatTensor) should be the same or input should be a
MKLDNN tensor and weight is a dense tensor
```

```
fake_images[0], fake_images[0].max(), fake_images[0].min()
```

```
(tensor([[[ 0.1089,  0.0494,  0.2467,  ..., -0.0087, -0.0056, -
0.0951],
          [-0.2068, -0.1683, -0.4545,  ...,  0.3168,  0.1050, -
0.1328],
          [-0.0422, -0.1157,  0.1233,  ..., -0.4035,  0.3853,
0.3623],
          ...,
          [-0.1884, -0.0179, -0.8904,  ..., -0.1275, -0.2109, -
0.1564],
          [ 0.1762, -0.1402,  0.3429,  ..., -0.0359, -0.0306,
0.0128],
          [-0.0538,  0.1835, -0.2136,  ...,  0.1130, -0.2923, -
0.1302]],

         [[ 0.0797,  0.3725,  0.2608,  ...,  0.5123,  0.1874,
0.1329],
          [-0.0591,  0.3120,  0.1284,  ..., -0.5246,  0.2893,
0.0699],
          [ 0.2953, -0.0368,  0.2352,  ...,  0.0515,  0.0344,
0.0932],
          ...,
          [ 0.2013, -0.3554,  0.6001,  ..., -0.3148,  0.5368,
0.4620],
          [ 0.1891,  0.6673, -0.5462,  ...,  0.8821,  0.0570,
0.1653],
          [-0.0434, -0.0970, -0.0283,  ...,  0.2115, -0.0920, -
0.1315]],

         [[ 0.1175, -0.1395,  0.4358,  ...,  0.4548,  0.5352,
0.0452],
          [-0.3357, -0.3494, -0.1298,  ..., -0.3161, -0.6118,
0.2965],
          [-0.0076, -0.5027,  0.0201,  ...,  0.2585,  0.1159, -
0.2479],
          ...,
          [ 0.0300, -0.5894,  0.0453,  ..., -0.3956, -0.5292,
0.3318],
          [ 0.0256, -0.6338,  0.6502,  ..., -0.1782,  0.6849, -
0.0860],
          [-0.3496, -0.2543, -0.4988,  ..., -0.1861, -0.0478,
0.0572]]],
```

```
        grad_fn=<SelectBackward0>),
 tensor(0.9991, grad_fn=<MaxBackward1>),
 tensor(-0.9985, grad_fn=<MinBackward1>))
```

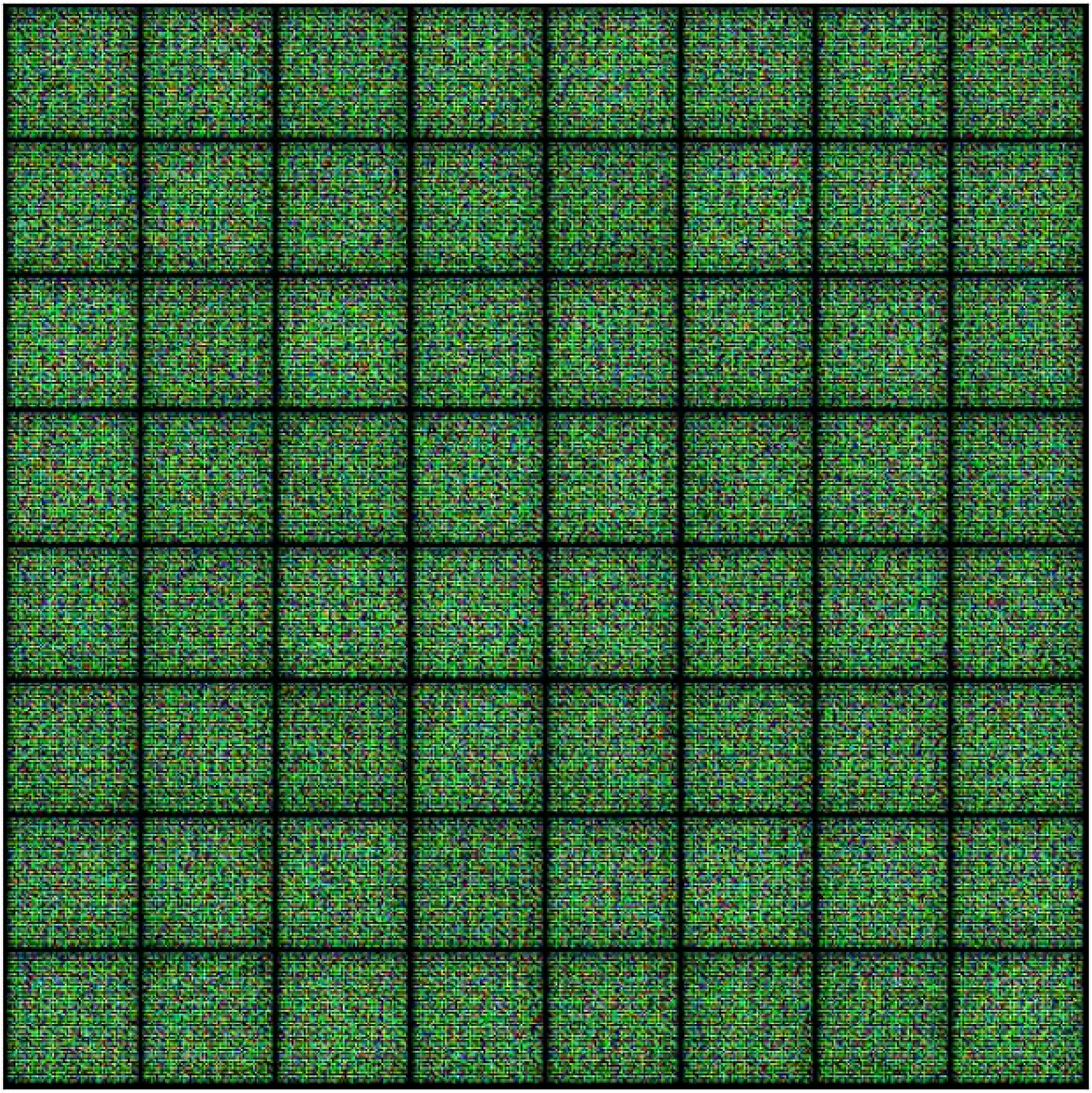
```python
x = denorm(fake_images[0])
x, x.max(), x.min()
```

```
(tensor([[[0.5544, 0.5247, 0.6233,  ..., 0.4956, 0.4972, 0.4524],
          [0.3966, 0.4158, 0.2727,  ..., 0.6584, 0.5525, 0.4336],
          [0.4789, 0.4422, 0.5616,  ..., 0.2983, 0.6926, 0.6811],
          ...,
          [0.4058, 0.4910, 0.0548,  ..., 0.4362, 0.3945, 0.4218],
          [0.5881, 0.4299, 0.6714,  ..., 0.4821, 0.4847, 0.5064],
          [0.4731, 0.5917, 0.3932,  ..., 0.5565, 0.3539, 0.4349]],

         [[0.5399, 0.6863, 0.6304,  ..., 0.7561, 0.5937, 0.5665],
          [0.4705, 0.6560, 0.5642,  ..., 0.2377, 0.6447, 0.5350],
          [0.6477, 0.4816, 0.6176,  ..., 0.5258, 0.5172, 0.5466],
          ...,
          [0.6006, 0.3223, 0.8000,  ..., 0.3426, 0.7684, 0.7310],
          [0.5945, 0.8336, 0.2269,  ..., 0.9410, 0.5285, 0.5827],
          [0.4783, 0.4515, 0.4858,  ..., 0.6057, 0.4540, 0.4343]],

         [[0.5587, 0.4302, 0.7179,  ..., 0.7274, 0.7676, 0.5226],
          [0.3321, 0.3253, 0.4351,  ..., 0.3420, 0.1941, 0.6482],
          [0.4962, 0.2487, 0.5100,  ..., 0.6293, 0.5579, 0.3761],
          ...,
          [0.5150, 0.2053, 0.5226,  ..., 0.3022, 0.2354, 0.6659],
          [0.5128, 0.1831, 0.8251,  ..., 0.4109, 0.8424, 0.4570],
          [0.3252, 0.3729, 0.2506,  ..., 0.4070, 0.4761, 0.5286]]],
        grad_fn=<AddBackward0>),
 tensor(0.9995, grad_fn=<MaxBackward1>),
 tensor(0.0007, grad_fn=<MinBackward1>))
```

```python
nmax=64
fig, ax = plt.subplots(figsize=(8, 8))
ax.set_xticks([]); ax.set_yticks([])
ax.imshow(make_grid(fake_images.cpu()[:nmax], nrow=8).permute(1, 2,
0))
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for
imshow with RGB data ([0..1] for floats or [0..255] for integers). Got
range [-0.9999103..0.99997145].
```

```
<matplotlib.image.AxesImage at 0x77fc1f4e58d0>
```



```
fake_images[0]
```

```
tensor([[[ 0.1089,  0.0494,  0.2467,  ..., -0.0087, -0.0056, -0.0951],
         [-0.2068, -0.1683, -0.4545,  ...,  0.3168,  0.1050, -0.1328],
         [-0.0422, -0.1157,  0.1233,  ..., -0.4035,  0.3853,  0.3623],
         ...,
         [-0.1884, -0.0179, -0.8904,  ..., -0.1275, -0.2109, -0.1564],
         [ 0.1762, -0.1402,  0.3429,  ..., -0.0359, -0.0306,  0.0128],
         [-0.0538,  0.1835, -0.2136,  ...,  0.1130, -0.2923, -
0.1302]],

        [[ 0.0797,  0.3725,  0.2608,  ...,  0.5123,  0.1874,  0.1329],
         [-0.0591,  0.3120,  0.1284,  ..., -0.5246,  0.2893,  0.0699],
```

```
         [ 0.2953, -0.0368,  0.2352,  ...,  0.0515,  0.0344,  0.0932],
         ...,
         [ 0.2013, -0.3554,  0.6001,  ..., -0.3148,  0.5368,  0.4620],
         [ 0.1891,  0.6673, -0.5462,  ...,  0.8821,  0.0570,  0.1653],
         [-0.0434, -0.0970, -0.0283,  ...,  0.2115, -0.0920, -
0.1315]],

        [[ 0.1175, -0.1395,  0.4358,  ...,  0.4548,  0.5352,  0.0452],
         [-0.3357, -0.3494, -0.1298,  ..., -0.3161, -0.6118,  0.2965],
         [-0.0076, -0.5027,  0.0201,  ...,  0.2585,  0.1159, -0.2479],
         ...,
         [ 0.0300, -0.5894,  0.0453,  ..., -0.3956, -0.5292,  0.3318],
         [ 0.0256, -0.6338,  0.6502,  ..., -0.1782,  0.6849, -0.0860],
         [-0.3496, -0.2543, -0.4988,  ..., -0.1861, -0.0478,
0.0572]]],
       grad_fn=<SelectBackward0>)
```

```
fake_images.detach()[0]
```

```
tensor([[[ 0.1089,  0.0494,  0.2467,  ..., -0.0087, -0.0056, -0.0951],
         [-0.2068, -0.1683, -0.4545,  ...,  0.3168,  0.1050, -0.1328],
         [-0.0422, -0.1157,  0.1233,  ..., -0.4035,  0.3853,  0.3623],
         ...,
         [-0.1884, -0.0179, -0.8904,  ..., -0.1275, -0.2109, -0.1564],
         [ 0.1762, -0.1402,  0.3429,  ..., -0.0359, -0.0306,  0.0128],
         [-0.0538,  0.1835, -0.2136,  ...,  0.1130, -0.2923, -
0.1302]],

        [[ 0.0797,  0.3725,  0.2608,  ...,  0.5123,  0.1874,  0.1329],
         [-0.0591,  0.3120,  0.1284,  ..., -0.5246,  0.2893,  0.0699],
         [ 0.2953, -0.0368,  0.2352,  ...,  0.0515,  0.0344,  0.0932],
         ...,
         [ 0.2013, -0.3554,  0.6001,  ..., -0.3148,  0.5368,  0.4620],
         [ 0.1891,  0.6673, -0.5462,  ...,  0.8821,  0.0570,  0.1653],
         [-0.0434, -0.0970, -0.0283,  ...,  0.2115, -0.0920, -
0.1315]],

        [[ 0.1175, -0.1395,  0.4358,  ...,  0.4548,  0.5352,  0.0452],
         [-0.3357, -0.3494, -0.1298,  ..., -0.3161, -0.6118,  0.2965],
         [-0.0076, -0.5027,  0.0201,  ...,  0.2585,  0.1159, -0.2479],
         ...,
         [ 0.0300, -0.5894,  0.0453,  ..., -0.3956, -0.5292,  0.3318],
         [ 0.0256, -0.6338,  0.6502,  ..., -0.1782,  0.6849, -0.0860],
         [-0.3496, -0.2543, -0.4988,  ..., -0.1861, -0.0478,
0.0572]]])
```

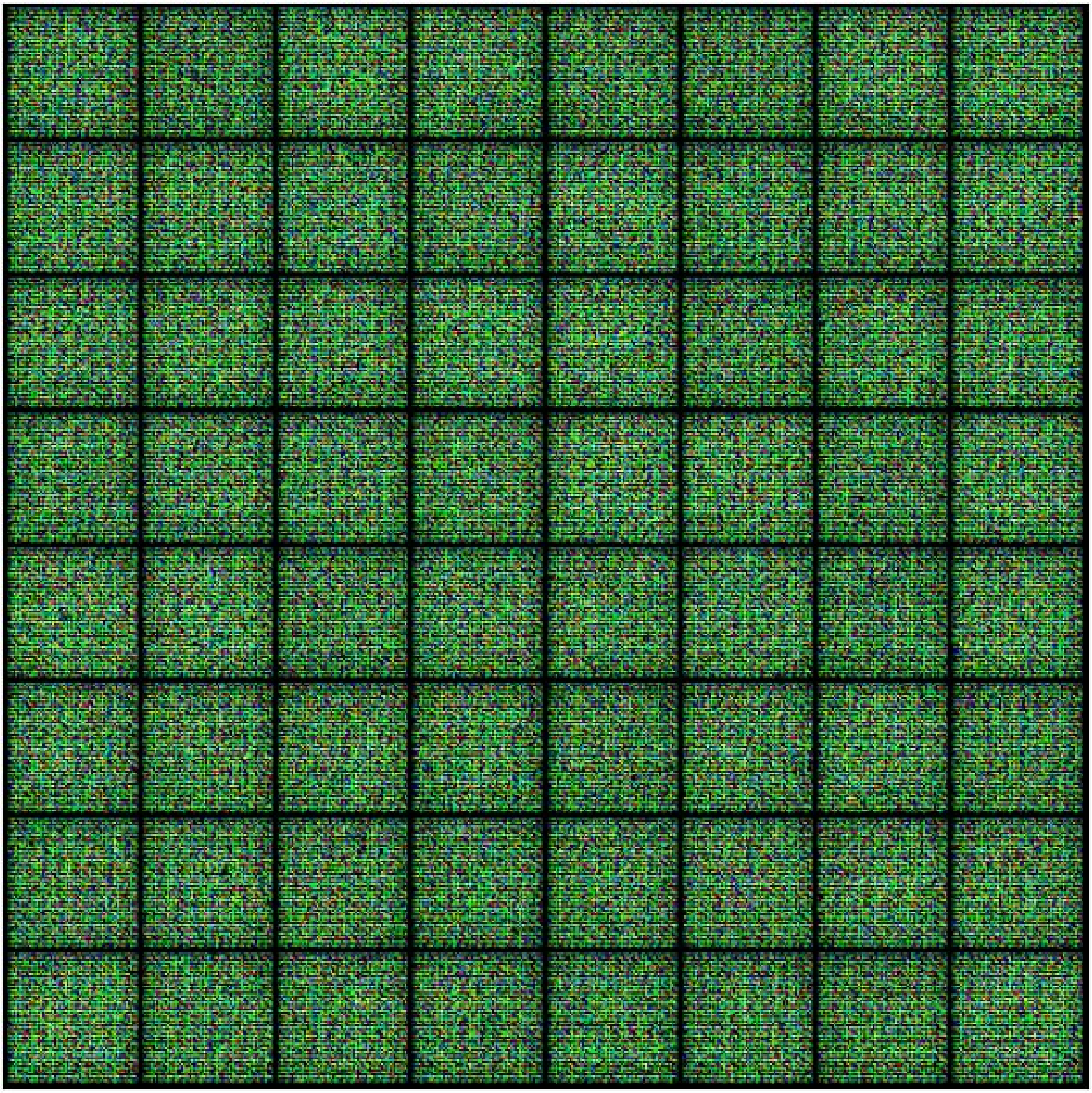
In summary, `detach()` is useful for creating a tensor that shares the same data as the original but is not involved in the computation graph, making it suitable for operations where gradient tracking is unnecessary.

```
def save_samples(index, latent_tensors, show=True):
    fake_images = generator(latent_tensors)
    fake_fname = 'generated-images-{0:0=4d}.png'.format(index)
    save_image(denorm(fake_images), os.path.join(sample_dir,
fake_fname), nrow=8)
    print('Saving', fake_fname)
    if show:
      nmax=64
      fig, ax = plt.subplots(figsize=(8, 8))
      ax.set_xticks([]); ax.set_yticks([])
      ax.imshow(make_grid(fake_images.cpu().detach()[:nmax],
nrow=8).permute(1, 2, 0))
```

We'll use a fixed set of input vectors to the generator to see how the individual generated images evolve over time as we train the model. Let's save one set of images before we start training our model.

```
fixed_latent = torch.randn(64, latent_size, 1, 1, device=device)

save_samples(0, fixed_latent)
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for
imshow with RGB data ([0..1] for floats or [0..255] for integers). Got
range [-0.9999211..0.9999165].

Saving generated-images-0000.png
```



Full Training Loop

Let's define a `fit` function to train the discriminator and generator in tandem for each batch of training data. We'll use the Adam optimizer with some custom parameters (betas) that are known to work well for GANs. We will also save some sample generated images at regular intervals for inspection.

```
from tqdm.notebook import tqdm
import torch.nn.functional as F
```

```
def fit(epochs, lr, start_idx=1):
    torch.cuda.empty_cache()

    # Losses & scores
    losses_g = []
    losses_d = []
    real_scores = []
    fake_scores = []

    # Create optimizers
    opt_d = torch.optim.Adam(discriminator.parameters(), lr=lr, 
betas=(0.5, 0.999))
    opt_g = torch.optim.Adam(generator.parameters(), lr=lr, 
betas=(0.5, 0.999))

    for epoch in range(epochs):
        for real_images, _ in tqdm(train_dl):
            # Train discriminator
            loss_d, real_score, fake_score = 
train_discriminator(real_images, opt_d)
            # Train generator
            loss_g = train_generator(opt_g)

        # Record losses & scores
        losses_g.append(loss_g)
        losses_d.append(loss_d)
        real_scores.append(real_score)
        fake_scores.append(fake_score)

        # Log losses & scores (last batch)
        print("Epoch [{}/{}], loss_g: {:.4f}, loss_d: {:.4f}, 
real_score: {:.4f}, fake_score: {:.4f}".format(
            epoch+1, epochs, loss_g, loss_d, real_score, fake_score))

        # Save generated images
        save_samples(epoch+start_idx, fixed_latent, show=False)

    return losses_g, losses_d, real_scores, fake_scores
```

We are now ready to train the model. Try different learning rates to see if you can maintain the fine balance between the training the generator and the discriminator.

```
lr = 0.0001
epochs = 40

history = fit(epochs, lr)
```

{"model_id":"0b03ab64ae42459a860a8cd7fe22b583","version_major":2,"version_minor":0}

```
Epoch [1/40], loss_g: 7.8827, loss_d: 0.0220, real_score: 0.9841, fake_score: 0.0050
Saving generated-images-0001.png
```

{"model_id":"a53792edf283478a961eaaa27f3ef5c0","version_major":2,"version_minor":0}

```
Epoch [2/40], loss_g: 7.6889, loss_d: 0.0131, real_score: 0.9953, fake_score: 0.0081
Saving generated-images-0002.png
```

{"model_id":"fc3bc52a961541ea8677ada97e646ce3","version_major":2,"version_minor":0}

```
Epoch [3/40], loss_g: 5.5302, loss_d: 0.0392, real_score: 0.9752, fake_score: 0.0101
Saving generated-images-0003.png
```

{"model_id":"61a8acb4e9d749dbaa68ca5be64ce32f","version_major":2,"version_minor":0}

```
Epoch [4/40], loss_g: 12.3595, loss_d: 0.0325, real_score: 0.9796, fake_score: 0.0030
Saving generated-images-0004.png
```

{"model_id":"f04454a64dfd4746b85587b3c29eaadd","version_major":2,"version_minor":0}

```
Epoch [5/40], loss_g: 6.7670, loss_d: 0.0529, real_score: 0.9861, fake_score: 0.0279
Saving generated-images-0005.png
```

{"model_id":"adb1f0a7df8f4e90a5e921252caa6f3e","version_major":2,"version_minor":0}

```
Epoch [6/40], loss_g: 6.7374, loss_d: 0.0327, real_score: 0.9725, fake_score: 0.0020
Saving generated-images-0006.png
```

{"model_id":"edb0b67e1e4043bf9ddac66956f8b0ff","version_major":2,"version_minor":0}

```
Epoch [7/40], loss_g: 6.0444, loss_d: 0.0540, real_score: 0.9705, fake_score: 0.0067
Saving generated-images-0007.png
```

{"model_id":"86c369716cb24e54b64885bb324f749a","version_major":2,"version_minor":0}

```
Epoch [8/40], loss_g: 8.2132, loss_d: 0.0172, real_score: 0.9918, fake_score: 0.0085
Saving generated-images-0008.png
```

{"model_id":"38018ac64b814b229b4434cddc379584","version_major":2,"version_minor":0}

```
Epoch [9/40], loss_g: 12.6639, loss_d: 0.0648, real_score: 0.9973,
fake_score: 0.0566
Saving generated-images-0009.png
```

{"model_id":"3f3e6fb4337c420cbed0b46eba1c085f","version_major":2,"version_minor":0}

```
Epoch [10/40], loss_g: 9.8951, loss_d: 0.0460, real_score: 0.9941,
fake_score: 0.0376
Saving generated-images-0010.png
```

{"model_id":"aea25f5310e0498daebc8a37eaaf8dbb","version_major":2,"version_minor":0}

```
Epoch [11/40], loss_g: 10.0694, loss_d: 0.0161, real_score: 0.9856,
fake_score: 0.0005
Saving generated-images-0011.png
```

{"model_id":"4c8a66442fb64aae90e3154941fc9490","version_major":2,"version_minor":0}

```
Epoch [12/40], loss_g: 6.7159, loss_d: 0.1129, real_score: 0.9386,
fake_score: 0.0005
Saving generated-images-0012.png
```

{"model_id":"bbed995f6c4b451c897ea8bf2917b20a","version_major":2,"version_minor":0}

```
Epoch [13/40], loss_g: 21.9588, loss_d: 0.0111, real_score: 0.9899,
fake_score: 0.0000
Saving generated-images-0013.png
```

{"model_id":"5320f43fb93048a0930ac9501dba5272","version_major":2,"version_minor":0}

```
Epoch [14/40], loss_g: 7.5201, loss_d: 0.0127, real_score: 0.9967,
fake_score: 0.0092
Saving generated-images-0014.png
```

{"model_id":"457145b8600742c5904db1df299993fd","version_major":2,"version_minor":0}

```
Epoch [15/40], loss_g: 7.8541, loss_d: 0.0082, real_score: 0.9962,
fake_score: 0.0044
Saving generated-images-0015.png
```

{"model_id":"201857882264452eaca9805a25496286","version_major":2,"version_minor":0}

```
Epoch [16/40], loss_g: 6.8867, loss_d: 0.0200, real_score: 0.9917,
fake_score: 0.0105
Saving generated-images-0016.png
```

{"model_id":"7f4cdb32ec2646f49818d68f985f3515","version_major":2,"version_minor":0}

```
Epoch [17/40], loss_g: 8.5822, loss_d: 0.0062, real_score: 0.9992,
fake_score: 0.0053
Saving generated-images-0017.png
```

{"model_id":"073dade0e87d4a08a374bb4eac3e0ecb","version_major":2,"version_minor":0}

```
Epoch [18/40], loss_g: 16.1016, loss_d: 0.1407, real_score: 0.9073,
fake_score: 0.0000
Saving generated-images-0018.png
```

{"model_id":"d5804060b50c4c4cbccf4dd176addfe7","version_major":2,"version_minor":0}

```
Epoch [19/40], loss_g: 9.5890, loss_d: 0.0017, real_score: 0.9995,
fake_score: 0.0011
Saving generated-images-0019.png
```

{"model_id":"f1c997698aef4b48998971fa5e0eb9b8","version_major":2,"version_minor":0}

```
Epoch [20/40], loss_g: 9.3953, loss_d: 0.0528, real_score: 0.9999,
fake_score: 0.0454
Saving generated-images-0020.png
```

{"model_id":"87d883aaff3c4dbebfbe62693439db3f","version_major":2,"version_minor":0}

```
Epoch [21/40], loss_g: 7.2898, loss_d: 0.0187, real_score: 0.9900,
fake_score: 0.0079
Saving generated-images-0021.png
```

{"model_id":"40a5e8a3eee74060b32e855e28bcf79b","version_major":2,"version_minor":0}

```
Epoch [22/40], loss_g: 10.6903, loss_d: 0.1204, real_score: 0.9181,
fake_score: 0.0002
Saving generated-images-0022.png
```

{"model_id":"d01024ea26ec451daa13e6fef65cfa43","version_major":2,"version_minor":0}

```
Epoch [23/40], loss_g: 5.8867, loss_d: 0.0180, real_score: 0.9879,
fake_score: 0.0041
Saving generated-images-0023.png
```

{"model_id":"3a38c66d4e34428eb6a0261190d15c3d","version_major":2,"version_minor":0}

```
Epoch [24/40], loss_g: 6.7452, loss_d: 0.0117, real_score: 0.9992, fake_score: 0.0107
Saving generated-images-0024.png
```

{"model_id":"9174c67f43a74a0e8d8b4c698ef04e6c","version_major":2,"version_minor":0}

```
Epoch [25/40], loss_g: 6.9552, loss_d: 0.0844, real_score: 0.9793, fake_score: 0.0349
Saving generated-images-0025.png
```

{"model_id":"b65929251df14b4983b16e74f0481845","version_major":2,"version_minor":0}

```
Epoch [26/40], loss_g: 5.8772, loss_d: 0.0269, real_score: 0.9812, fake_score: 0.0027
Saving generated-images-0026.png
```

{"model_id":"91febad180504f3eadca409f3384a207","version_major":2,"version_minor":0}

```
Epoch [27/40], loss_g: 6.0610, loss_d: 0.0092, real_score: 0.9930, fake_score: 0.0020
Saving generated-images-0027.png
```

{"model_id":"ef5ae051849141e98dad7d8174156b91","version_major":2,"version_minor":0}

```
Epoch [28/40], loss_g: 7.8308, loss_d: 0.0220, real_score: 0.9815, fake_score: 0.0021
Saving generated-images-0028.png
```

{"model_id":"b262aae2cf2f438da46031d19a9b2b6d","version_major":2,"version_minor":0}

```
Epoch [29/40], loss_g: 16.9645, loss_d: 5.3791, real_score: 0.0680, fake_score: 0.0000
Saving generated-images-0029.png
```

{"model_id":"0dab1ddc7cd84d8896dd6678739b3b53","version_major":2,"version_minor":0}

```
Epoch [30/40], loss_g: 6.8423, loss_d: 0.0326, real_score: 0.9963, fake_score: 0.0279
Saving generated-images-0030.png
```

{"model_id":"e1eabadda91a4fddb4a690d971b9ec61","version_major":2,"version_minor":0}

```
Epoch [31/40], loss_g: 7.6408, loss_d: 0.0031, real_score: 0.9988, fake_score: 0.0018
Saving generated-images-0031.png
```

{"model_id":"6dd631b675d0450f960e28147d996c09","version_major":2,"version_minor":0}

```
Epoch [32/40], loss_g: 6.6382, loss_d: 0.0116, real_score: 0.9956, fake_score: 0.0068
Saving generated-images-0032.png
```

{"model_id":"8ab55236941147d8a88c7ca50206f5f5","version_major":2,"version_minor":0}

```
Epoch [33/40], loss_g: 46.0095, loss_d: 0.0023, real_score: 0.9978, fake_score: 0.0000
Saving generated-images-0033.png
```

{"model_id":"3b2d29ffd57b4f9fa54abae0ff1b5351","version_major":2,"version_minor":0}

```
Epoch [34/40], loss_g: 42.8991, loss_d: 0.0000, real_score: 1.0000, fake_score: 0.0000
Saving generated-images-0034.png
```

{"model_id":"f78295ae3f1e4fde96c80463131af440","version_major":2,"version_minor":0}

```
Epoch [35/40], loss_g: 42.8378, loss_d: 0.0000, real_score: 1.0000, fake_score: 0.0000
Saving generated-images-0035.png
```

{"model_id":"291992185985414e94db7d01c206b20e","version_major":2,"version_minor":0}

```
Epoch [36/40], loss_g: 40.9549, loss_d: 0.0000, real_score: 1.0000, fake_score: 0.0000
Saving generated-images-0036.png
```

{"model_id":"c2875d515ab94f6eae8434d2ce8b3708","version_major":2,"version_minor":0}

```
Epoch [37/40], loss_g: 39.0423, loss_d: 0.0000, real_score: 1.0000, fake_score: 0.0000
Saving generated-images-0037.png
```

{"model_id":"eab9b11e11674a528b65e4b9c73251ee","version_major":2,"version_minor":0}

```
Epoch [38/40], loss_g: 4.3998, loss_d: 0.0237, real_score: 0.9900, fake_score: 0.0117
Saving generated-images-0038.png
```

{"model_id":"02f969b64ff94e649b1ef29342368f6e","version_major":2,"version_minor":0}

```
Epoch [39/40], loss_g: 6.4680, loss_d: 0.0357, real_score: 0.9827,
fake_score: 0.0169
Saving generated-images-0039.png
```

{"model_id":"60cb175a8e134f3f8e5822e2bf2f40f9","version_major":2,"version_minor":0}

```
Epoch [40/40], loss_g: 8.5992, loss_d: 0.1608, real_score: 0.9112,
fake_score: 0.0002
Saving generated-images-0040.png
```

```
losses_g, losses_d, real_scores, fake_scores = history
```

Now that we have trained the models, we can save checkpoints.

```
# Save the model checkpoints
torch.save(generator.state_dict(), 'G.pth')
torch.save(discriminator.state_dict(), 'D.pth')
```

Here's how the generated images look, after the 1st, 5th and 10th epochs of training.

```
from IPython.display import Image

Image('./generated/generated-images-0001.png')
```

```
Image('./generated/generated-images-0005.png')
```

```
Image('./generated/generated-images-0010.png')
```

```
Image('./generated/generated-images-0020.png')
```

```
Image('./generated/generated-images-0025.png')
```

```
Image('./generated/generated-images-0030.png')
```

```
Image('./generated/generated-images-0035.png')
```

```
Image('./generated/generated-images-0040.png')
```

We can visualize the training process by combining the sample images generated after each epoch into a video using OpenCV.

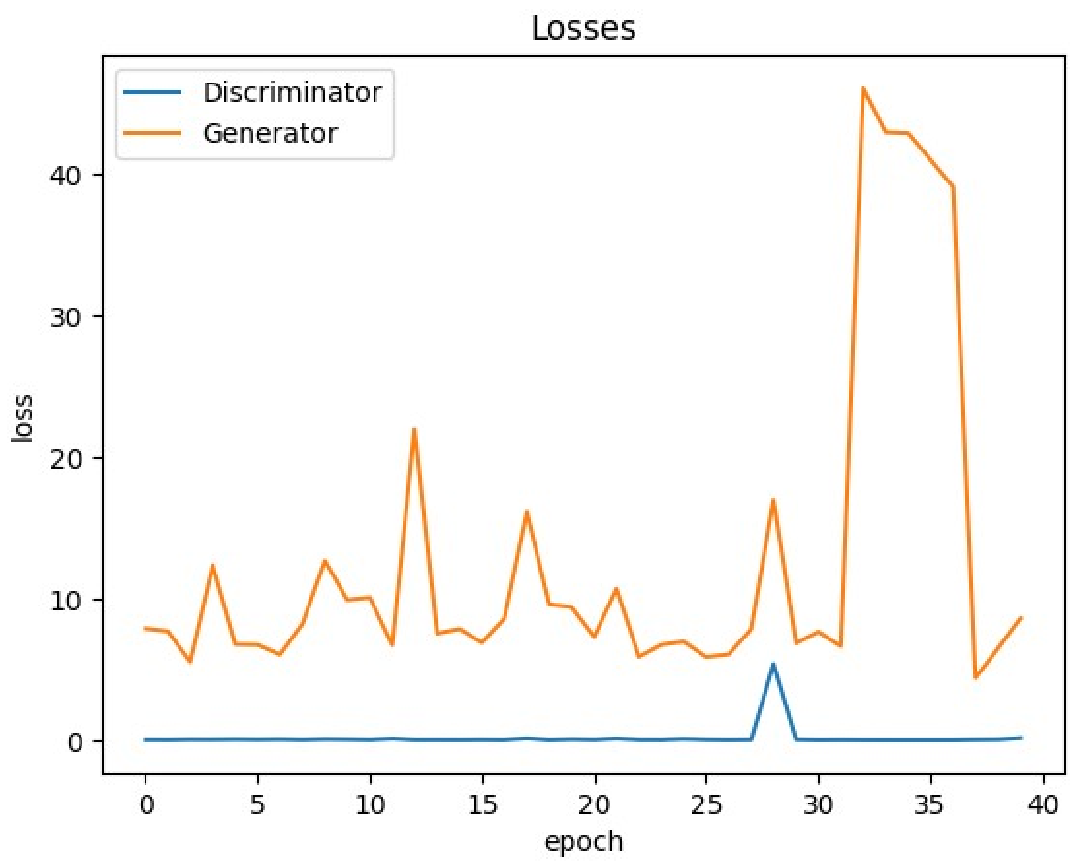
```
import cv2
import os

vid_fname = 'gans_training.avi'

files = [os.path.join(sample_dir, f) for f in os.listdir(sample_dir)
if 'generated' in f]
files.sort()

out = cv2.VideoWriter(vid_fname,cv2.VideoWriter_fourcc(*'MP4V'), 1,
```

```
(530,530))
[out.write(cv2.imread(fname)) for fname in files]
out.release()
```

Here's what it looks like:

We can also visualize how the loss changes over time. Visualizing losses is quite useful for debugging the training process. For GANs, we expect the generator's loss to reduce over time, without the discriminator's loss getting too high.

```
plt.plot(losses_d, '-')
plt.plot(losses_g, '-')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['Discriminator', 'Generator'])
plt.title('Losses');
```




```
plt.plot(real_scores, '-')
plt.plot(fake_scores, '-')
plt.xlabel('epoch')
```

```
plt.ylabel('score')
plt.legend(['Real', 'Fake'])
plt.title('Scores');
```