INF5400 — MACHINE LEARNING FOR IMAGE ANALYSIS
DEPARTMENT OF INFORMATICS, UNIVERSITY OF OSLO
2019

Exercise week 13

# Generative Adversarial Networks

In this programming exercise, we are going to implement DCGAN (*Deep Convolutional Generative Adversarial Networks*) introduced in (Radford et al., 2016). The implementation will be in PyTorch, and a similar tutorial, which this one is heavily influenced by, is available at `https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html`.

- The goal is to implement a standalone python program that can be used for training and generating images.
- This guide will show you the essentials of the implementation. For a more complete version, see the accompanying solution proposal.
- We will train on the *Labeled Faces in the Wild* dataset, available from `http://vis-www.cs.umass.edu/lfw/`. This is a relatively small dataset, and fits well for this purpose.
- The implementation will run on both CPU and GPU, but is not implemented to make use of multiple GPUs.
- The exercise was developed on, and runs with Python 3.6 and PyTorch 1.1, but have not been tested on other versions.
- Feel free to experiment with different approaches, this implementation is just a suggestion, and not necessarily the best.

## 1 Import data

Use the function in listing 1 to download the dataset.

```
import subprocess

def maybe_download_lfw(download_root_dir):
    download_dir = download_root_dir.joinpath('lfw')
    if (
          download_dir.exists() and
          download_dir.is_dir() and
          list(download_dir.iterdir())
        ):
        print("LFW dataset already downloaded")
        return download_dir

    tar_path = download_root_dir.joinpath('lfw.tgz')
    if not tar_path.exists():
        url = 'http://vis-www.cs.umass.edu/lfw/lfw.tgz'
        cmd = ['curl', url, '-o', str(tar_path)]
        print("Downloading LFW from {} to {}".format(url, tar_path))
        subprocess.call(cmd)

    if not download_dir.exists():
        download_dir.mkdir()

    print("Unpacking")
    cmd = ['tar', 'xzf', str(tar_path), '-C', str(download_dir)]
    subprocess.call(cmd)

    return download_dir
```

Listing 1: Download data

Create a `data_loader` that we can use to iterate through the training dataset. We will resize the images to a shape of $(c, h, w) = (3, 64, 64)$, and normalize their values to a range of $[-1, 1]$ in accordance with the original paper (the input image values are in the range $[0, 1]$). As in the original paper, we are going to use a batch-size of 128.

```python
import Path
import torch
import torchvision

data_dir = Path('where you would like to keep your downloaded data')
download_dir = maybe_download_lfw(data_dir)
target_size = 64
batch_size = 128
data_loader = torch.utils.data.DataLoader(
    torchvision.datasets.ImageFolder(
        str(download_dir),
        transform=torchvision.transforms.Compose(
            [
                torchvision.transforms.Resize((target_size, target_size)),
                torchvision.transforms.ToTensor(),
                torchvision.transforms.Normalize(
                    (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)
                    ),
                ]
            )
        ),
    batch_size=batch_size,
    shuffle=True,
    )

# Example how to iterate through one epoch
for image_batch, label_batch in data_loader:
    # Do something with the images and labels
```

Listing 2: Import data

## 2 Training

This section will describe the discriminator network and the generator network. It will then go on to describe how to train them together.

### 2.1 Network implementation

The discriminator is a network that (in our case) takes a $64 \times 64 \times 3$ input image, and outputs a number with a value in (0, 1) that we will interpret as the probability that the input image is a real image. Listing 3 display one way to implement the discriminator network in PyTorch.

The generator is (in our case) a network that takes an one-dimensional vector of length `latent_length` and outputs an image with shape $64 \times 64 \times 3$. A suggestion for the generator network is implemented in listing 4.

```python
class Discriminator(torch.nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.leaky_relu = torch.nn.LeakyReLU(0.2)
        # Input shape (c, h, w): (3, 64, 64)
        self.conv_1 = torch.nn.Conv2d(
            in_channels=3,
            out_channels=64,
            kernel_size=4,
            stride=2,
            padding=1,
            bias=False,
            )
        # Input shape (c, h, w): (64, 32, 32)
        self.conv_2 = torch.nn.Conv2d(
            in_channels=64,
            out_channels=128,
            kernel_size=4,
            stride=2,
            padding=1,
            bias=False,
            )
        self.bn_1 = torch.nn.BatchNorm2d(128)
        # Input shape (c, h, w): (128, 16, 16)
        self.conv_3 = torch.nn.Conv2d(
            in_channels=128,
            out_channels=256,
            kernel_size=4,
            stride=2,
            padding=1,
            bias=False,
            )
        self.bn_2 = torch.nn.BatchNorm2d(256)
        # Input shape (c, h, w): (256, 8, 8)
        self.conv_4 = torch.nn.Conv2d(
            in_channels=256,
            out_channels=512,
            kernel_size=4,
            stride=2,
            padding=1,
            bias=False,
            )
        self.bn_3 = torch.nn.BatchNorm2d(512)
        # Input shape (c, h, w): (512, 4, 4)
        self.conv_5 = torch.nn.Conv2d(
            in_channels=512,
            out_channels=1,
            kernel_size=4,
            stride=1,
            padding=0,
            bias=False,
            )

    def forward(self, x):
        x = self.leaky_relu(self.conv_1(x))
        x = self.leaky_relu(self.bn_1(self.conv_2(x)))
        x = self.leaky_relu(self.bn_2(self.conv_3(x)))
        x = self.leaky_relu(self.bn_3(self.conv_4(x)))
        x = torch.sigmoid(self.conv_5(x))
        return x
```

Listing 3: Discriminator network

```python
latent_length = 100
class Generator(torch.nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        # Input shape (c, h, w): (latent_length, 1, 1)
        self.convtr_1 = torch.nn.ConvTranspose2d(
            in_channels=latent_length,
            out_channels=512,
            kernel_size=4,
            stride=1,
            padding=0,
            bias=False,
            )
        self.bn_1 = torch.nn.BatchNorm2d(num_features=512)
        # Input shape (c, h, w): (512, 4, 4)
        self.convtr_2 = torch.nn.ConvTranspose2d(
            in_channels=512,
            out_channels=256,
            kernel_size=4,
            stride=2,
            padding=1,
            bias=False,
            )
        self.bn_2 = torch.nn.BatchNorm2d(num_features=256)
        # Input shape (c, h, w): (256, 8, 8)
        self.convtr_3 = torch.nn.ConvTranspose2d(
            in_channels=256,
            out_channels=128,
            kernel_size=4,
            stride=2,
            padding=1,
            bias=False,
            )
        self.bn_3 = torch.nn.BatchNorm2d(num_features=128)
        # Input shape (c, h, w): (128, 16, 16)
        self.convtr_4 = torch.nn.ConvTranspose2d(
            in_channels=128,
            out_channels=64,
            kernel_size=4,
            stride=2,
            padding=1,
            bias=False,
            )
        self.bn_4 = torch.nn.BatchNorm2d(num_features=64)
        # Input shape (c, h, w): (64, 32, 32)
        self.convtr_5 = torch.nn.ConvTranspose2d(
            in_channels=64,
            out_channels=3,
            kernel_size=4,
            stride=2,
            padding=1,
            bias=False,
            )
        # Output shape (c, h, w): (3, 64, 64). NOTE: The spatial shape should
     match target_size

    def forward(self, x):
        x = torch.nn.functional.relu(self.bn_1(self.convtr_1(x)))
        x = torch.nn.functional.relu(self.bn_2(self.convtr_2(x)))
        x = torch.nn.functional.relu(self.bn_3(self.convtr_3(x)))
        x = torch.nn.functional.relu(self.bn_4(self.convtr_4(x)))
        x = torch.tanh(self.convtr_5(x))
        return x
```

Listing 4: Generator network

## 2.2 Parameter initialisation

All weights are initialised from a zero-centered Normal distribution with standard deviation 0.02, while the Batch Norm bias parameters are initialised to zero. This can be achieved in PyTorch by implementing the function

```python
def initialise_weights(submodule):
    if (
            isinstance(submodule, torch.nn.Conv2d) or
            isinstance(submodule, torch.nn.ConvTranspose2d)
        ):
        # Initialise from a random normal distribution with mean 0.0 and
        # stdev 0.02
        torch.nn.init.normal_(submodule.weight.data, 0.0, 0.02)
    elif isinstance(submodule, torch.nn.BatchNorm2d):
        # Initialise from a random normal distribution with mean 1.0 and
        # stdev 0.02
        torch.nn.init.normal_(submodule.weight.data, 1.0, 0.02)
        # Initialise all bias parameters to zero
        torch.nn.init.constant_(submodule.bias.data, 0.0)
```

Listing 5: Weight initialisation

and applying it on the `discriminator` network and the `generator` network

```python
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
discriminator = Discriminator().to(device)
discriminator.apply(initialise_weights)
generator = Generator().to(device)
generator.apply(initialise_weights)
```

Listing 6: Weight initialisation

## 2.3 Adversarial training

For each update step we are going to update the discriminator network $D$, and the generator network $G$ once. First, consider the discriminator network with the associated loss

$$L_D = -\frac{1}{m}\sum_{i=1}^{m}[\log(D(x_i)) + \log(1 - D(G(z_i)))] \tag{2.1}$$

over a mini-batch of size $m$, as shown in the lecture slides for week 13. This is a sum of two binary-cross-entropy loss functions; one where the discriminator output on real images $x_i$ is compared against 1, and one where the discriminator output on generated images $G(z_i)$ is compared against 0. For the discriminator network, we use the Adam optimisation method with the same hyperparameter values as in the DCGAN paper. The discriminator update can be implemented as.

```
 1 binary_ce = torch.nn.BCELoss()
 2 discriminator_optimiser = torch.optim.Adam(
 3     discriminator.parameters(),
 4     lr=0.0002,
 5     betas=(0.5, 0.999),
 6     )
 7 for image_batch, _ in data_loader:
 8     discriminator.zero_grad()
 9     image_batch = image_batch.to(device)
10     batch_size = image_batch.shape[0]
11 
12     # Disriminator wrt true images
13     real_label_batch = torch.full((batch_size, ), real_label, device=device)
14     discriminator_output_real = discriminator.forward(image_batch).view(-1)
15     discriminator_loss_real = binary_ce(
16         discriminator_output_real, real_label_batch
17         )
18     discriminator_loss_real.backward()
19 
20     # Disriminator wrt generated images
21     noise = torch.randn(batch_size, latent_length, 1, 1, device=device)
22     fake_batch = generator.forward(noise)
23     fake_label_batch = torch.full((batch_size, ), fake_label, device=device)
24     discriminator_output_fake = discriminator\
25         .forward(fake_batch.detach())\
26         .view(-1)
27     discriminator_loss_fake = binary_ce(
28         discriminator_output_fake, fake_label_batch
29         )
30     discriminator_loss_fake.backward()
31 
32     discriminator_optimiser.step()
```

Listing 7: Discriminator network update

Note that he binary cross entropy for a single example between a label $y \in \{0,1\}$ and a prediction $\hat{y} \in [0,1]$ is

$$l = -[y \log \hat{y} + (1-y)\log(1-\hat{y})].$$

Since, for the discriminator loss, the labels for real examples are 1 and for generated examples are 0, we end up with the expression in eq. (2.1). Also note the `fake_batch.detach()` on line 25 in the listing 7, this tells PyTorch not to compute gradients for the generator here.

The generator will be updated similarly. Note that we already have computed the forward pass of the generator (line 22 in listing 7). Since we have updated the discriminator, we evaluate it again on the generated images (line 8 in listing 8). From the lectures, we have that the generator loss is

$$L_G = -\frac{1}{m}\sum_{i=1}^{m} \log D(G(z_i)). \tag{2.2}$$

This is the binary cross entropy loss comparing generated images $G(z_i)$ with the label 1. Remember that the objective is to guide the generator to generate images that looks like they are from the same distribution as the real images.

```python
generator_optimiser = torch.optim.Adam(
    generator.parameters(),
    lr=0.0002,
    betas=(0.5, 0.999),
    )
for image_batch, _ in data_loader:
    generator.zero_grad()
    discriminator_output_fake = discriminator.forward(fake_batch).view(-1)
    generator_loss = binary_ce(discriminator_output_fake, real_label_batch)
    generator_loss.backward()

    generator_optimiser.step()
```

Listing 8: Generator network update

## 3 Restore a trained model

The point of this exercise is to train the above adversarial network so that we can use the trained generator to generate new examples from the training data distribution. For us to be able to restore the generator, we need to save its parameter values. This can be achieved by

```python
torch.save(generator.state_dict(), "Path where you want to save your model")
```

Listing 9: Save generator parameter values

which needs to be written inside the `for`-loop in listing 7, and activated at appropriate times, for example at every $n$ steps. The generator can then be restored with

```python
noise = torch.randn(64, latent_length, 1, 1, device=device)
generator.load_state_dict(torch.load("Path with desired checkpoint"))
generator.eval()
generated_images = generator.forward(noise)
```

Listing 10: Restore generator

## 4 Example results

You can display the generated images with the code in listing 11.

```python
def plot_mosaic(images, filename):
    num_cols = 8
    num_rows = min(images.shape[0] // num_cols, 8)
    plt.figure()
    images = images[:num_rows*num_cols, :, :, :]
    images = images * 0.5 + 0.5 # Invert normalisation
    image_grid = np.transpose(
        torchvision.utils.make_grid(images,padding=2).cpu().detach().numpy(),
        (1, 2, 0)
        )
    plt.axis('off')
    plt.imsave(filename, image_grid)
plot_mosaic(generated_images, "Location you want to put output images")
```

Listing 11: Restore generator

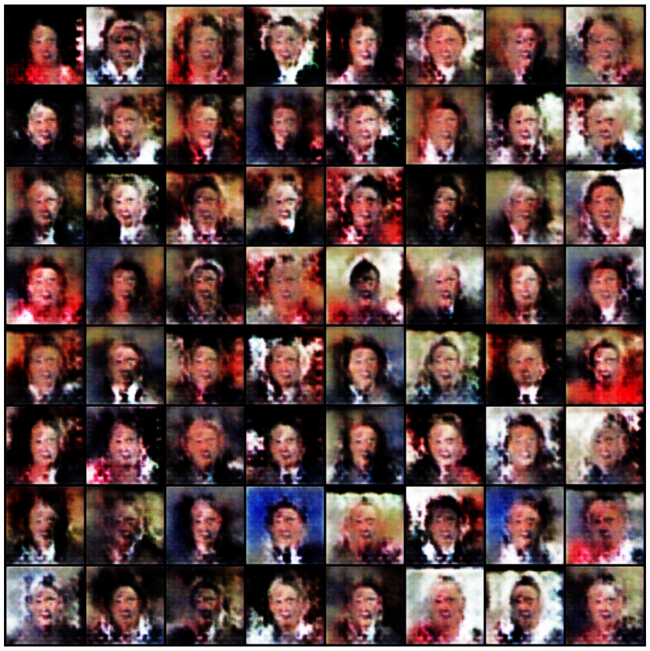

Figure 4.1: Generated images from generator restored after training 1 000 steps

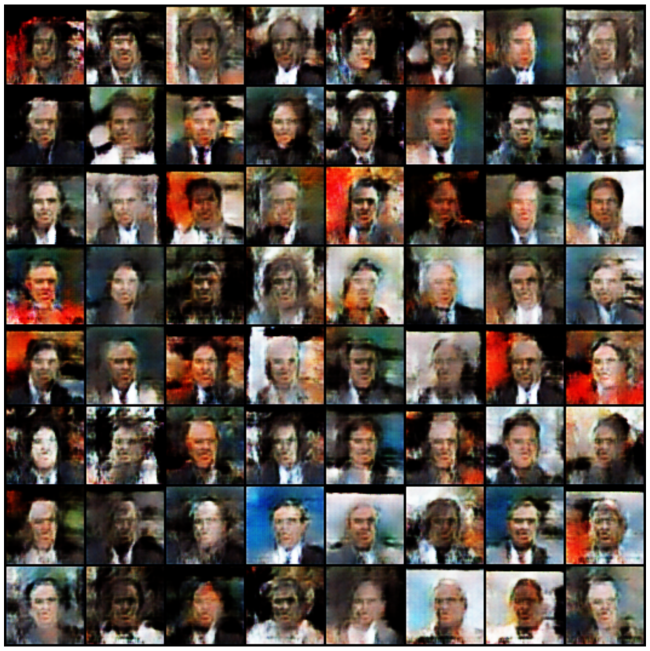

Figure 4.2: Generated images from generator restored after training 2 000 steps

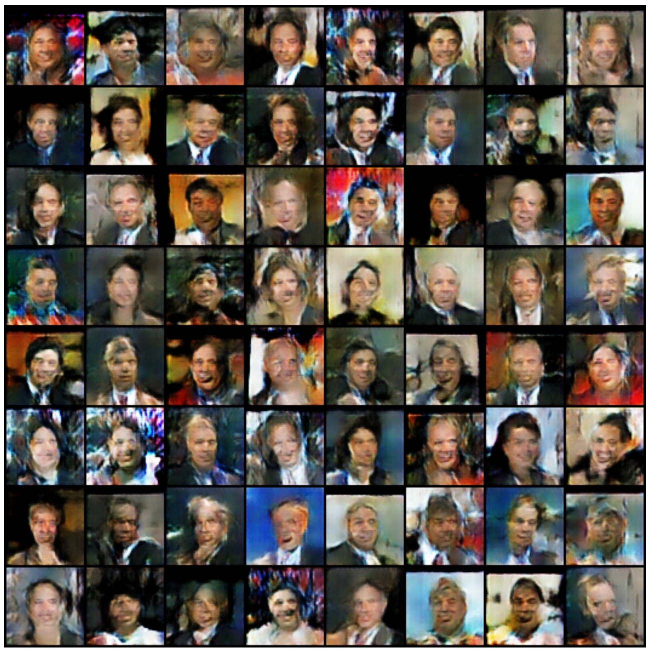

Figure 4.3: Generated images from generator restored after training 3 000 steps

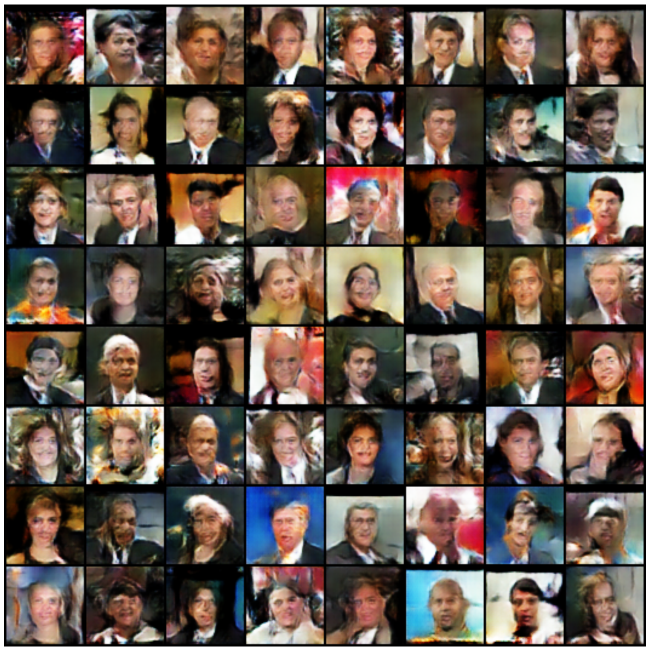

Figure 4.4: Generated images from generator restored after training 4 000 steps

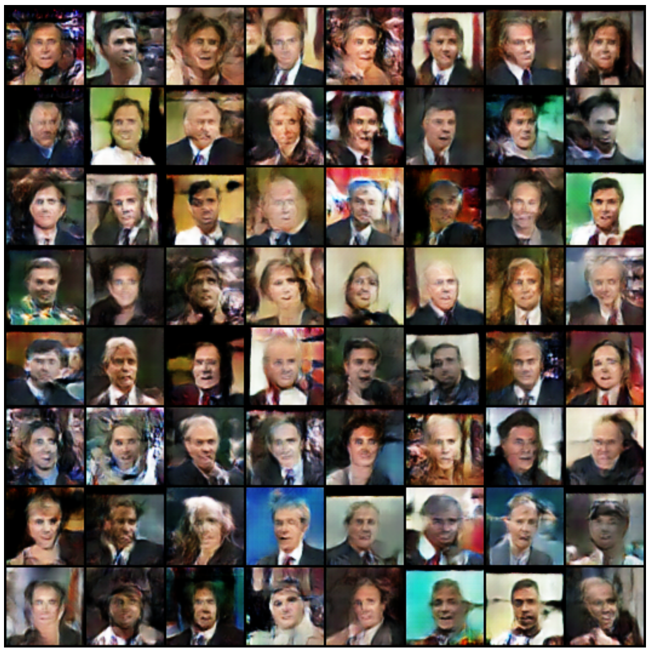

Figure 4.5: Generated images from generator restored after training 5 000 steps

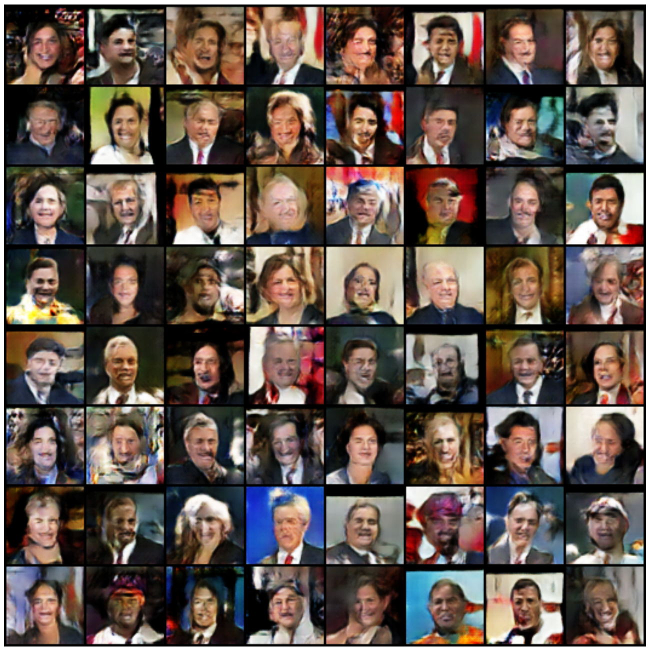

Figure 4.6: Generated images from generator restored after training 6 000 steps

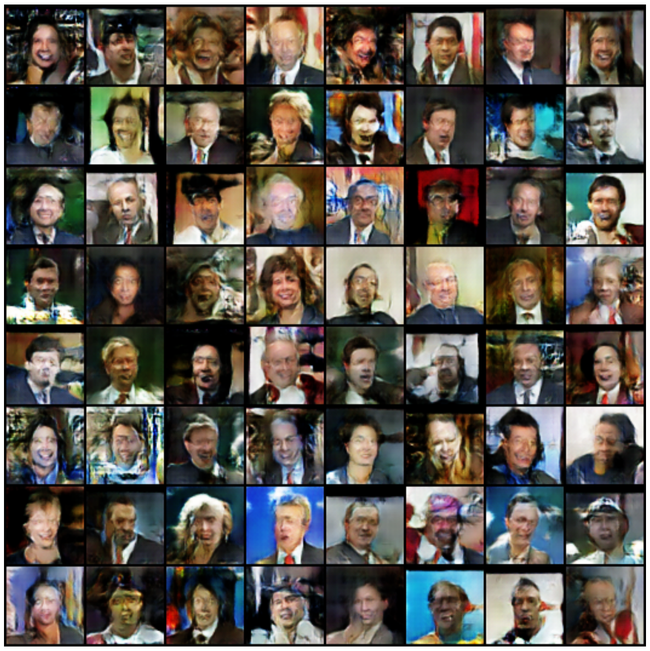

Figure 4.7: Generated images from generator restored after training 7 000 steps

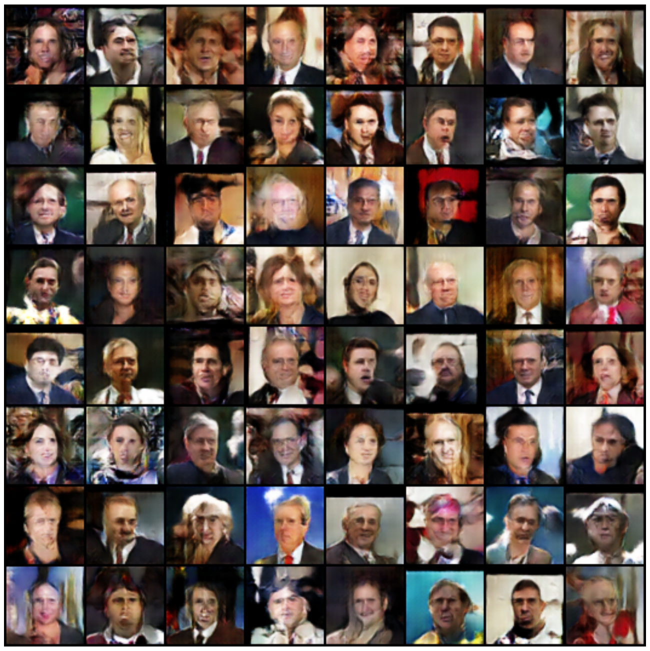

Figure 4.8: Generated images from generator restored after training 8 000 steps

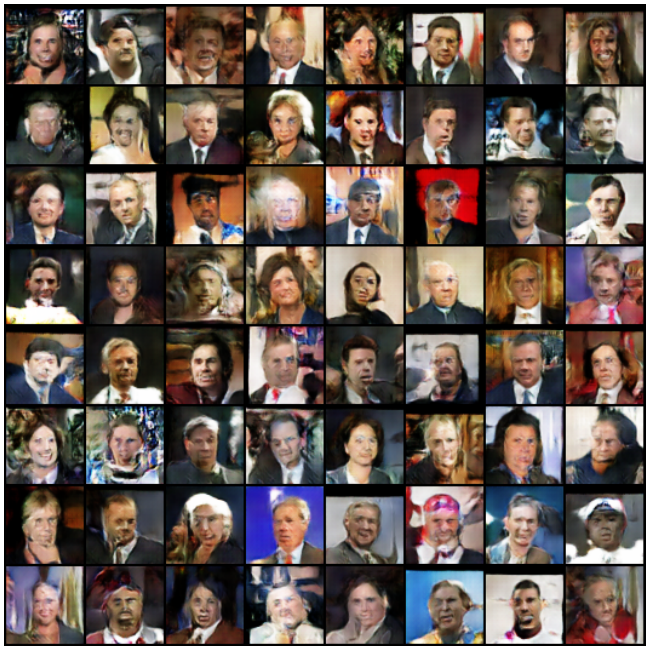

Figure 4.9: Generated images from generator restored after training 9 000 steps

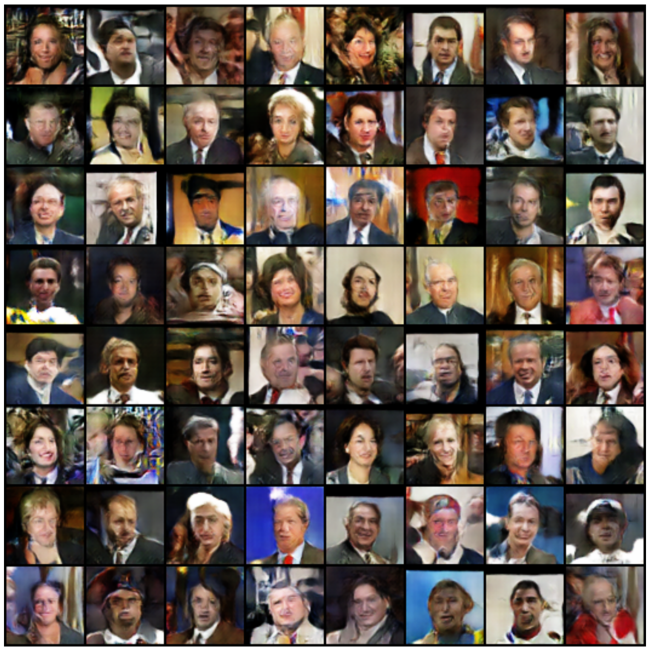

Figure 4.10: Generated images from generator restored after training 10 000 steps

## References

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In *ICLR 2016*, pages 1–16, 2016. URL http://arxiv.org/abs/1511.06434.