Lab 3 - Computer Vision

Diego Torres

November 2023

1 Introduction

In this lab exercise we will implement GANs (Generative Adversarial Networks), which are a type of deep generative model that used to be state-of-the-art until the arrival of diffusion based models. I will explain how they work, their components, loss function and training procedure, as well as provide my implementation and results on a simple dataset. Most of the code that I wrote is heavily inspired by the tutorial on GANs provided by the Pytorch documentation, available at `https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html`.

2 GANs

As the name suggests, this architecture consists of two networks, which are adversaries of one another. As shown in figure 1, the first element of the network is the generator, which is in charge of generating samples as realistic as possible from random noise. How good the generator is will be determined by another network, the discriminator, whose job is to tell apart real samples from generator ones. Therefore, the goal of the generator is to trick the discriminator into classifying its data as real. When training both in the right manner, improvement in the discriminator will make the generator improve as well. GANs are most widely known for generating images, but their principle is independent on the type of data one wishes to generate.

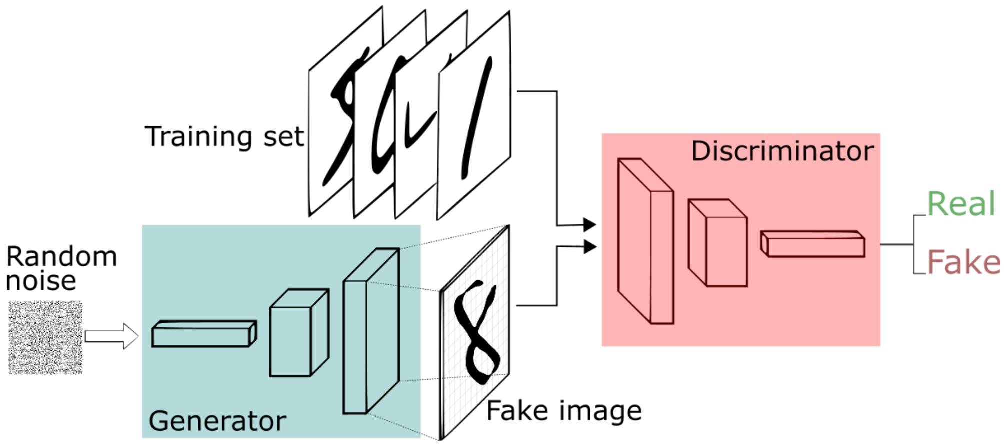

Figure 1: Diagram of a generative adversarial network

The mathematical formulation of GANs is not as complicated as one might think. The discriminator is a simple classifier that tries to distinguish between real and generated images. This means that its loss function will be the well known cross entropy loss. On the other hand, the performance of the generator is evaluated through the discriminator, so the same loss is applicable. In this case, however, we look to maximize the loss, as the generator's goal is to confuse the discriminator. Therefore, there is only one loss function, which the discriminator tries to minimize and the generator to maximize:

$$\min_G \max_D \mathbb{E}_{x\sim \text{data}}[\log(D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z))]. \tag{1}$$

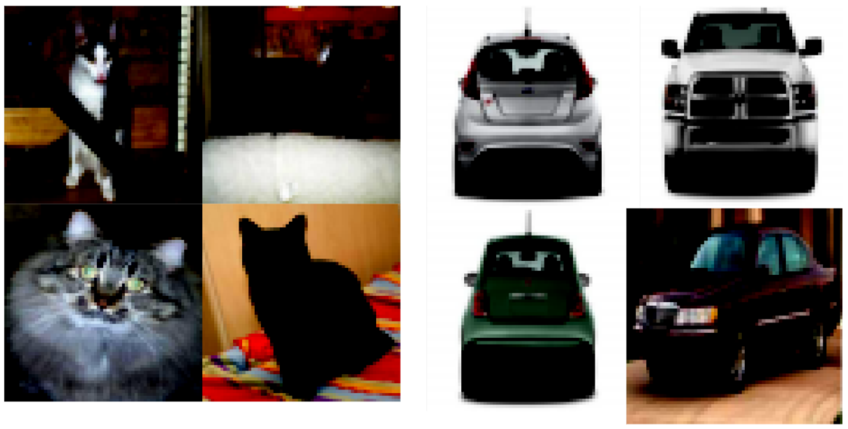

Figure 2: Comparison of both datasets. On the left, the cat dataset, with much more variability. On the right, the car dataset, more uniform.

Here $p(z)$ is the distribution of the noise, which is usually a standard normal distribution. This equation also reveals that for the discriminator training both real and fake images are needed, whereas the generator only needs its generated images.

3 Data

I initially tried to train the model with images of cats, available on Kaggle at the following link `https://www.kaggle.com/datasets/chetankv/dogs-cats-images`. The results were very discouraging, generating what seemed to be cat silhouettes but without any details that made them look like actual cats. Some of them had very bad lightning conditions and were mostly black. I tried to play with the hyperparameters to see if I could get some improvement, but they were marginal. By doing some manual exploration of the data, I found out that it had too much variability, with cats in different poses and scenarios.

Accordingly, I decided to change the dataset, selecting one composed of car images (also on Kaggle, available at `https://www.kaggle.com/datasets/chetankv/dogs-cats-images`). They have less variability because they are stock images of specific angles of the cars: from behind, from the side, the wheels, the mirrors, etc. Additionally, the dataset is bigger, leading to more information that the network can learn.

4 Implementation

As state before, my implementation closely follows that of the official documentation of Pytorch. The code of the generator and discriminator is available in code 1, whereas the optimizers and other training hyperparameters are shown in code 2. As for the training loop, with the exception that I train for num_epochs=40, it is the same as in the tutorial, so it is not provided here.

5 Results

In figure 3 are shown the discriminator and generator losses. We note that they are mostly stable due to to the generator step making it harder for the discriminator to perform better, and viceversa. During some of my experiments I often encountered the problem of the discriminator learning much faster than the generator, reaching an accuracy of more than 99%. This broke the training procedure and some adjustments were needed. In this scenario a constant loss for either model is not a bad indicator, as it is measure with respect to a better model in subsequent training steps. Most likely some additional tuning is needed, since at the end the discriminator seems to be doing better than the generator. In figure 4 we can see the generated cars with the fixed 64 random vectors (as described in the tutorial). These are quite satisfactory, reaching

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(n_latent, n_features * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(n_features * 8),
            nn.ReLU(True),
            # state size. ``(ngf*8) x 4 x 4``
            nn.ConvTranspose2d(n_features * 8, n_features * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_features * 4),
            nn.ReLU(True),
            # state size. ``(ngf*4) x 8 x 8``
            nn.ConvTranspose2d( n_features * 4, n_features * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_features * 2),
            nn.ReLU(True),
            # state size. ``(ngf*2) x 16 x 16``
            nn.ConvTranspose2d(n_features * 2, n_features, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_features),
            nn.ReLU(True),
            # state size. ``(ngf) x 32 x 32``
            nn.ConvTranspose2d(n_features, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. ``(nc) x 64 x 64``
        )

    def forward(self, x):
        return self.main(x)

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            # input is ``(nc) x 64 x 64``
            nn.Conv2d(3, n_features, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf) x 32 x 32``
            nn.Conv2d(n_features, n_features * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_features * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*2) x 16 x 16``
            nn.Conv2d(n_features * 2, n_features * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_features * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*4) x 8 x 8``
            nn.Conv2d(n_features * 4, n_features * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_features * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*8) x 4 x 4``
            nn.Conv2d(n_features * 8, 1, 4, 1, 0, bias=False)
        )

    def forward(self, x):
        return self.main(x)
```

Code 1: Generator and discriminator code, n_latent=100 and n_features=64, as suggested in the tutorial. One of the improvements that I made upon the documentation code is not applying sigmoid on the model, but rather computing the loss directly from the logits.

```
# Initialize the ``BCELoss`` function
criterion = nn.BCEWithLogitsLoss()

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(discriminator.parameters(), lr=0.001, betas=(0.5, 0.999))
optimizerG = optim.Adam(generator.parameters(), lr=0.001, betas=(0.5, 0.999))

# Learning rate schedulers
schedulerD = optim.lr_scheduler.StepLR(optimizerD, step_size=500, gamma=0.6)
schedulerG = optim.lr_scheduler.StepLR(optimizerG, step_size=500, gamma=0.6)
```

Code 2: Loss, optimizers and learning rate schedulers for the model. The values of the hyperparameters were found after several manual exploration steps.

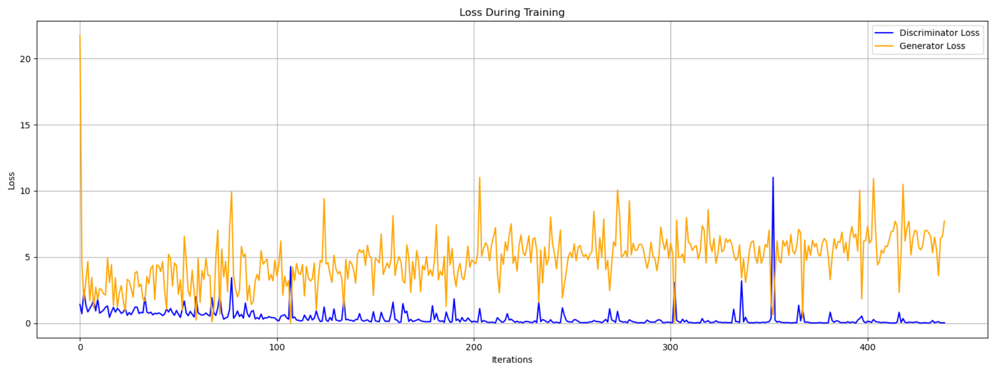

Figure 3: Losses of the discriminator and generator during training

the same level of realism as the original Pytorch code. It's possible to see that the images that correspond to cars in a white background have the highest quality of all, arising from a predictable distribution. Other decent images correspond with the close-up photos of lights. The worst ones seem to be of the cars' interiors, having bigger artifacts than the rest. Further data is needed to reach better quality, but this is already good, considering the size of the network and that it only took 2 hours to train.

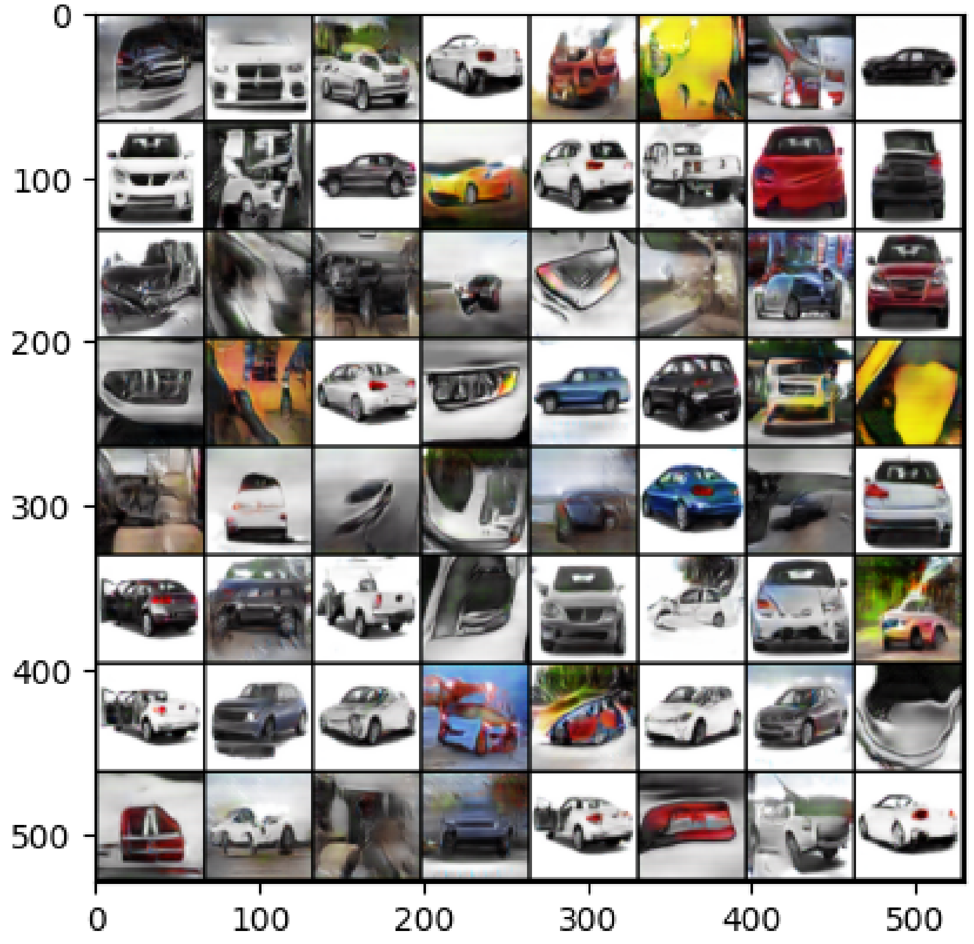

Figure 4: Generated images at the final training step