Artificial intelligence

Project 2: GAN to generate faces

Alexander Beauregard Bravo

169206

Introduction

Generative Adversarial Networks (GAN's) are an approach to generative modeling using deep learning methods, such as convolutional neural networks. These are comprised of node layers, containing an input layer, one or more hidden layers, and an output layer. Each node connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network.

In this project, the purpose was to implement a model and train it with data from the known 'celebA' dataset, which contains around 200,000 images of celebrities. The point is to get an output which is comprised of fake faces generated by the model. In this case, I used a pre-processed version of the 'celebA' dataset simply for computational purposes. This pre-processed version contains around 40,000 images, which are the inputs that the model was trained in.

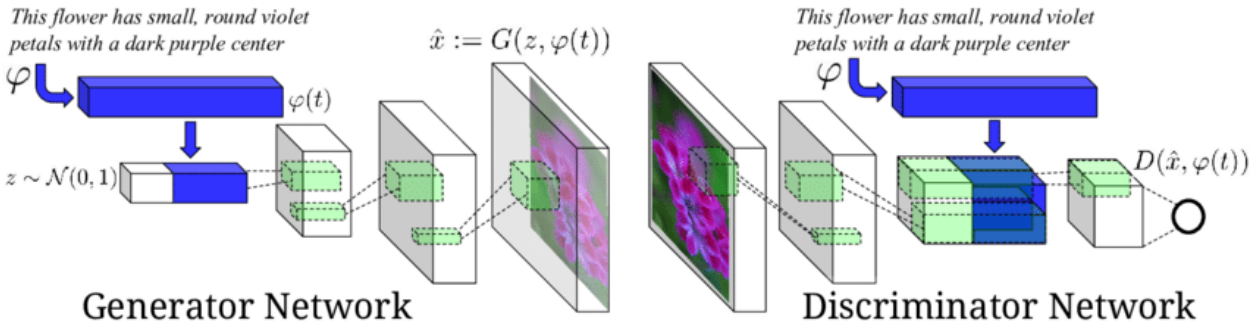

Figure 1: General GAN Architecture

The model is based on the "discriminator vs generator" conundrum. The job of the discriminator is to look at an image and output whether or not it is a real training image or a fake image from the generator. During training, the generator is constantly trying to outsmart the discriminator by generating better and better fakes, while the discriminator is working to become a better detective and correctly classify the real and fake images. The equilibrium of this game is when the generator is generating perfect fakes that look as if they came directly from the training data, and the

discriminator is left to always guess at 50% confidence that the generator output is real or fake.

The discriminator is made up of strided convolution layers, batch norm layers, and LeakyReLU activations. The input is a 3x64x64 input image and the output is a scalar probability that the input is from the real data distribution. The generator is comprised of convolutional-transpose layers, batch norm layers, and ReLU activations. The input is a latent vector z, that is drawn from a standard normal distribution and the output is a 3x64x64 RGB image.

Explanation

First, importing the correct libraries is important, I used the ones shown in Figure 2. As a side note, I also set a manual seed so that the resulting output will stay the same, although this is optional and can be removed.

```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

Figure 2: Necessary libraries

Then, several variables were instantiated. These include things such as the amount of epochs, the learning rate, the amount of GPU's used (I used 4. If none then the program will just use the CPU), etc. Also, the directory where the training data can be found is declared here as well.

Next, the dataset is imported and a data loader is created and a few images are displayed, as shown in Figure 3.

Figure 3: CelebA images

In several papers about GAN's, it is mentioned that all model weights have to be randomly initialized from a Normal distribution with parameters $\mu = 0$ and $\sigma = 0.02$. In Figure 4, the *weights_init(m)* function can be seen, which takes an initialized model as input and reinitializes all convolutional, convolutional-transpose, and batch normalization layers. Next, two classes are created: the Generator class and the Discriminator class. After this, a generator is instantiated and the *weights_init(m)* function is applied to it.

Now, the Discriminator is a binary classification network that takes an input, in this case images, and it outputs a number that represents a probability that the input image is fake or real. After a series several layers, the Discriminator outputs this probability through a Sigmoid activation function. Again, we use the *weights_init(m)* and apply it.

```python
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

Figure 4: weights_init() function

Then, after the Discriminator and the Generator are created, we can define how they learn through some functions, such as the so called Loss function and with optimizers. Several similar projects, when using PyTorch, use a Binary Cross Entropy loss function. After that, the labels are defined (1 for real images and 0 for fake images) and two optimizers are created (both are Adam optimizers with learning rate $lr = 0.0002$ and $\beta_1 = 0.5$. Finally, we train our model and look at the resulting output.

```python
# Generator Code

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

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

Figure 5: Generator class

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

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

Figure 6: Discriminator class

Results & Conclusion

In this case, only one epoch was used. This is just because when using several epochs, the kernel on my virtual environment died and did not finish the training. One epoch may not be a lot, but the results were satisfactory.

In Figure 7 we can see a plot of the Discriminator and Generator losses versus training iterations and in Figure 8 we can see the final result. The faces are a bit weird, but it is possible to make out a human resemblance.

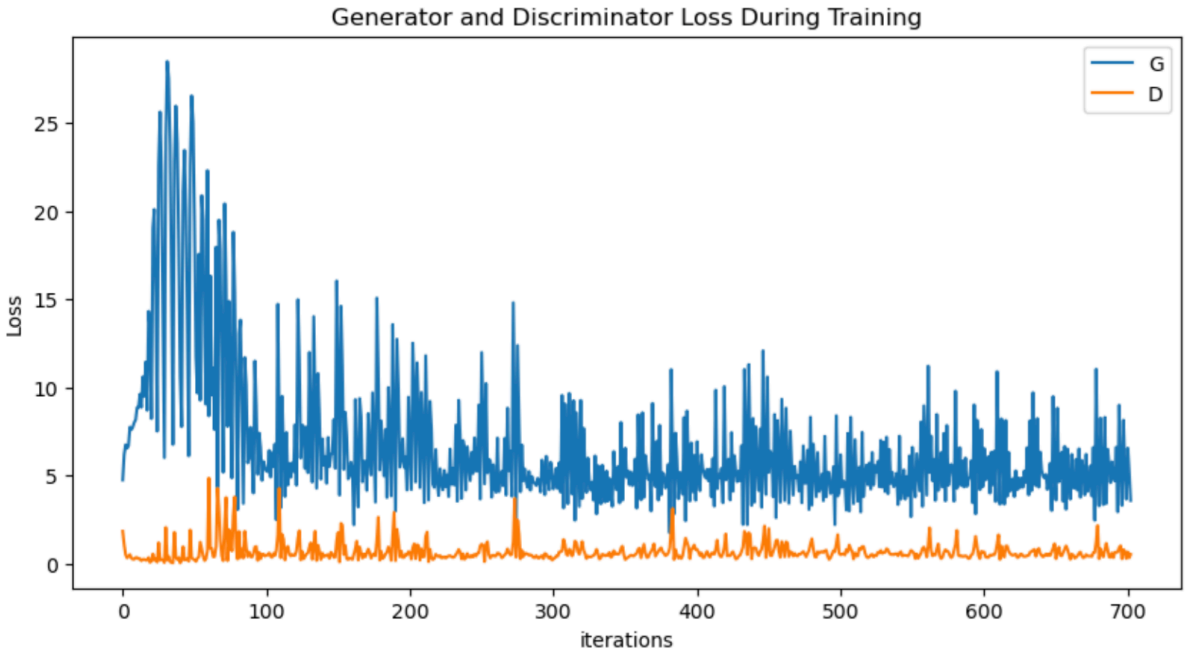

Figure 7: D&G loss

I am convinced that if I were to run more epochs of this program, the result could be better, however, given the time constraints (hours and hours of training) I feel the final output is adequate. It is true that the output presented is not the optimal one, nor even a “good” one maybe. It would be good to experiment with several different parameters, such as the learning rate, to see how the output would react.

Figure 8: Final Result

In conclusion, this project was a great learning experience. I had to delve a little bit into some research papers and a lot of obscure webpages and GitHub posts with already-completed projects to gain better understanding of how a GAN works. One thing I can say is that understanding the mathematical framework in which GAN's operate is essential to the full realization of this project. A lot of times I got stuck because I just did not understand in depth how the model was supposed to work. Also, the pre-process of the input data was one of the most difficult and important parts about this project.

Citations

IBM. "What Are Convolutional Neural Networks? | IBM." *Www.ibm.com*, www.ibm.com/topics/convolutional-neural-networks. Accessed 2 Apr. 2023.

Inkawhich, Nathan. "DCGAN Tutorial — PyTorch Tutorials 1.6.0 Documentation." *Pytorch.org*, pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html. Accessed 2 Apr. 2023.

Jason Brownlee. "A Gentle Introduction to Generative Adversarial Networks (GANs)." *Machine Learning Mastery*, 16 June 2019, machinelearningmastery.com/what-are-generative-adversarial-networks-gans/. Accessed 2 Apr. 2023.

Nekamiche, Noha. "Implementing Deep Convolutional Generative Adversarial Network (DCGAN) Using Celeba Dataset." *Medium*, 9 Nov. 2021, medium.com/@hn_nekamiche/implementing-deep-convolutional-generative-adversarial-network-dcgan-using-celeba-dataset-b06e54bb0b44. Accessed 3 Apr. 2023.