UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Master in Innovation and Research in Informatics

Algorithm Data Mining

Generative Adversarial Networks

Fourth Assignment

Mohana Fathollahi

June 26, 2022

1 Introduction

One of the hot topic in deep learning is Generative adversarial. Deep learning algorithms are state-of-the-art solutions to almost any task in the Computer vision area. However, deep networks require large datasets to achieve good performance. Even when large dataset are available, the networks overfit on training dataset and show poor or not great performance on validation/test dataset. Generative Adversarial Network or GAN is a branch in the Deep learning area that tries to generate new images that are similar to real-world dataset.
For example, a GAN can be trained with lots of real face images and generate a new face image that has a combination of attributes or landmarks that are not present in the given dataset. For example if a GAN gets trained on only human faces, we expect that the model generates images like the figure 1. None of these images belong to a real-human. In fact you can see artifacts on some of them. But there are more advanced GANs that can generate high quality images that are hard for even humans to distinguish. In this paper we try to explain a simple GAN and experiments with hyper-parameters.

Figure 1: Example of GAN output

2 Model

Generally, GAN is an unsupervised algorithm to generate synthetic, not real, images that are similar to real images.
Figure 2 shows the diagram of a GAN network. It consists of generators and discriminators. The generator input is a random vector with Gaussian distribution and the output is a synthetic image.
Discriminator is actually a simple classifier that is trained to distinguish between a synthetic and real-world image. In other words, it assigns a binary label to its input image.
Therefore, it should assign a high probability to real-world image and low-probability to a synthetic image. On the other hand the Generator should output images that look like real-world images.

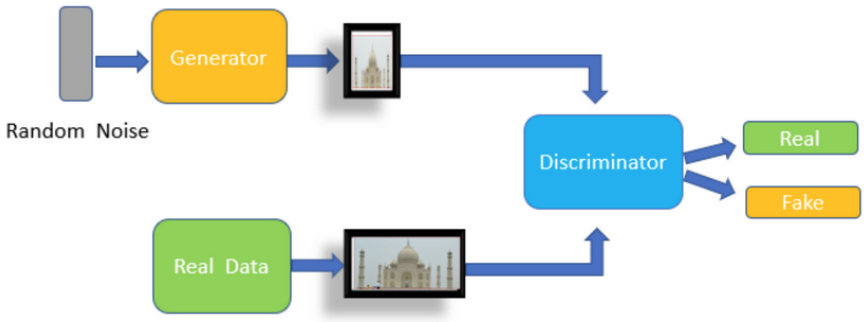

Figure 2: GAN Diagram

2.1 Discriminator Loss

As we dicused before our main goal of discriminative function is classifying the real data and the fake data from the generator.
In the below formula; x is real data and G(z) is data that generated by generator. Therefore, D(x) assign a probability to x and D(G(z)) assign probability to G(z). First probability should be higher than second probability, as a result we should maximize this equation to reach this goal. As a summary, the following cross-entropy teaches the discriminator network to distinguish fake from real images.

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right]$$

2.2 Generator Loss

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 generator gets rewarded if it produces an image that gets classified as real by the discriminator. In other words, it gets rewarded if D(G(z_i)) is close to one and gets penalized otherwise. The generator tries to create images, G(z), that have HIGH probability or minimize the distance of the assigned probability, D(G(z)) to 1. Therefore, we should minimize the below loss function.

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log \left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$

with combining last two loss functions, we get below function that should minimize generator and maximize discriminator.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}\left[logD(x)\right] + \mathbb{E}_{z \sim p_z(z)}\left[log(1 - D(G(z)))\right]$$

2.3 Implementation

2.3.1 Generator

In this report we analyze the GAN model that is presented in this tutorial. The generator should convert the noise vector to an image of size $nc * nc$. Meaning that it should convert a 1D latent vector z to a 2D image. To do so, DCGAN has used Transposed convolution which upsample the input feature map to a desired output using a learnable parameter. Each transpose convolution is followed by a Batch normalization and ReLU.
Figure 5 shows the diagram of the generator, where at each stage the number of feature maps or depth of the block gets divided by two and its spatial resolution gets doubled. "Conv" in this image is actually "transpose conv".

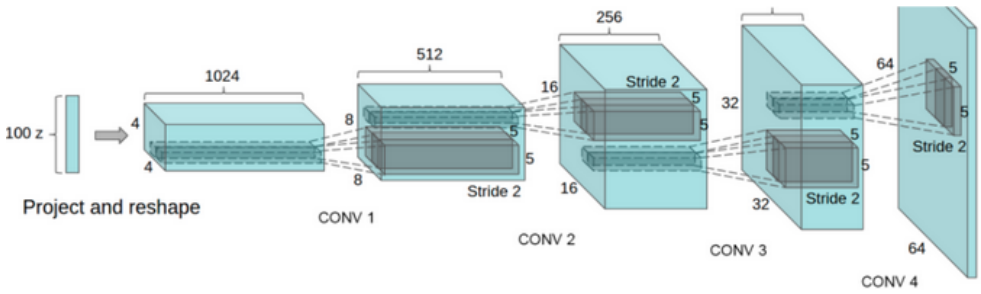

Figure 3: Structure of GAN

2.3.2 Discriminator

In the Discriminator network we have an input that is image and a series of $Conv+batchNorm+relu$ will be applied on them and outputs the final probability through a Sigmoid activation function.
This architecture can be extended with more layers depend on the problem, but there is significance to the use of the strided convolution, BatchNorm, and LeakyReLUs. In the following, there is an example of discriminator model:

```
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 4: Example of Discriminator

2.3.3 Training process

What is really important in GAN is how to train the model. Since the model consists of two sub networks, Discriminator and Generator, it is important how to calculate the loss and update the weights of these two networks. Following is a basic method that is used across most literature. Let's call the Discriminator network, netD, and the Generator network, netG.

- Update netD weights:
  - Create a batch of real images from training dataset and assign label "1" to them.
  - Pass real images through netD Calculate cross-entropy loss → D_realLoss.
  - Calculate the gradient of netD using "D_realLoss"
  - Create a noise vector and pass it through netG to create Fake images.

  - Assign label '0' to images created in step c.
  - Forward Fake images through netD and assign label '0' to these images.
  - Calculate cross-entropy loss for data in step f → D_fakeLoss.
  - Accumulate the gradient in step c by the gradient of netD respect "D_FakeLoss".
  - Update netD weights using gradient in step g.
- Update netG weights:
  - To update weights of generator, we assign label '1' to images created in section d.
  - Calculate cross entropy loss for this batch-> calculate gradient and update netG weights.

The reasoning that we assign label '1' to Fake images is that in equation 2 we would like to maximize log(1-D(G(z))) meaning that the Generator should try to create images, G(z), that deceive the discriminator. Meaning that the probability that the discriminator assigns to fake images, D(G(z)) be close to one. So, the generator should minimize the difference between this probability and '1'. Therefore, it should minimize log(1-D(G(z))). This is equivalent to maximizing log(D(G(z))) or assigning '1' label to this data in the cross-entropy loss function.

2.4 Results

In this section I use mnist dataset to train a GAN to generate 'Fake' images that are not present in the dataset but are similar to the training dataset. Since the model is unsupervised we don't use anylabel that is provided along with the dataset. Specifically the model uses 60000 images to trian generator and discriminator. The images are gray scale and I use the following parameters.

- nz= 100 dimension of noize vector to generate Fake images, Latent vector.
- ngf= 64 Number of feature map in the generator
- ndf= 64 Number of feature map in the discriminator
- Bs = 64 batch size

In this report I only qualitatively evaluate the results. In other words, every 200 batches, I save the images that the generator has created. Visually the more these images are similar to the training dataset the better. For example following is "real" samples from training dataset.

Figure 5: Real sample of MNIST

In the figures 6 and 7 there are "fake" images that the generator has created. We can see that in higher iterations where the network has trained more the "fake" images are getting more similar to the training images.

Figure 6: Iteration = 200

Figure 7: iteration = 400

2.4.1 Effect of Increasing input latent dimension

Tables 8 and 9 show the training logs for two different values of input noise latent dimension. Specifically the format of each row in the table is as follows. For example if one row of the table looks like this:

$[24/25][450/469]Loss\_D: 15.9771 Loss\_G: 5.4332 D(x): 0.0000 D(G(z)): 0.0000/0.0169$

- Item 1 is the epoch number, 24.
- Total number of epochs, 25.
- sample index, 450.
- Total number of epochs 469.
- Cross entropy loss of Discriminator: loss_G.
- Cross entropy loss of Generator: Loss_D.
- D(x) is the estimated probability of "real" images.
- $D(G(z_1))/D(G(z_2))$ is the estimated probability of generating a "fake" image before and after training the discriminator.

In an ideal world, we expect D(x) starts from a value close to 1 and by training the network for longer iterations it reduces to 0.5. That's because after training for a long time, the generator produces realistic images that make it harder for the discriminator network to distinguish "Fake" and "real" images. Based on the following tables, we can see this trend when we increase the latent dimension from 100 to 200.
Although it is still not perfect. To investigate what parameters would help to reduce D(x) to 0.5, there is a need to run more experiments with changing hyper-parameters.
Similarly, we expect D(G(z)) probability of the Discriminator network to generate images, stars from something small and increase to 0.5. Again we observe this trend more, when we increase the latent dimension from 100 to 200.

Latent dimension = 200

```
[24/25][450/469] Loss_D: 15.9771 Loss_G: 5.4332 D(x): 0.0000 D(G(z)): 0.0000 / 0.0169
[24/25][451/469] Loss_D: 2.9109 Loss_G: 0.0623 D(x): 0.1246 D(G(z)): 0.0196 / 0.9543
[24/25][452/469] Loss_D: 5.5724 Loss_G: 1.4996 D(x): 0.9819 D(G(z)): 0.9670 / 0.3649
[24/25][453/469] Loss_D: 1.3909 Loss_G: 3.4489 D(x): 0.5900 D(G(z)): 0.3681 / 0.0514
[24/25][454/469] Loss_D: 1.8820 Loss_G: 0.4668 D(x): 0.2319 D(G(z)): 0.0952 / 0.6606
[24/25][455/469] Loss_D: 1.3257 Loss_G: 2.0133 D(x): 0.9005 D(G(z)): 0.6322 / 0.1960
[24/25][456/469] Loss_D: 1.0045 Loss_G: 2.3613 D(x): 0.6703 D(G(z)): 0.3682 / 0.1299
[24/25][457/469] Loss_D: 1.3886 Loss_G: 0.6267 D(x): 0.3490 D(G(z)): 0.1504 / 0.5708
[24/25][458/469] Loss_D: 1.1711 Loss_G: 1.5717 D(x): 0.8011 D(G(z)): 0.5601 / 0.2475
[24/25][459/469] Loss_D: 0.8510 Loss_G: 1.8330 D(x): 0.6674 D(G(z)): 0.3129 / 0.1862
[24/25][460/469] Loss_D: 0.7621 Loss_G: 1.3350 D(x): 0.6261 D(G(z)): 0.2105 / 0.3039
[24/25][461/469] Loss_D: 0.9289 Loss_G: 1.6459 D(x): 0.7369 D(G(z)): 0.4144 / 0.2240
[24/25][462/469] Loss_D: 0.8721 Loss_G: 1.4301 D(x): 0.6073 D(G(z)): 0.2553 / 0.2862
[24/25][463/469] Loss_D: 0.7024 Loss_G: 1.7885 D(x): 0.7652 D(G(z)): 0.3185 / 0.2037
[24/25][464/469] Loss_D: 0.6836 Loss_G: 1.5804 D(x): 0.6855 D(G(z)): 0.2319 / 0.2454
[24/25][465/469] Loss_D: 0.8644 Loss_G: 1.6620 D(x): 0.7306 D(G(z)): 0.3801 / 0.2177
[24/25][466/469] Loss_D: 0.7238 Loss_G: 1.6043 D(x): 0.6785 D(G(z)): 0.2447 / 0.2348
[24/25][467/469] Loss_D: 0.6412 Loss_G: 1.4863 D(x): 0.7145 D(G(z)): 0.2246 / 0.2586
[24/25][468/469] Loss_D: 0.8812 Loss_G: 2.1087 D(x): 0.7713 D(G(z)): 0.4149 / 0.1552
```

Figure 8: Result of Latent dimension = 200

Latent dimension = 100

```
[24/25][450/469] Loss_D: 0.9578 Loss_G: 2.3186 D(x): 0.8398 D(G(z)): 0.4703 / 0.1420
[24/25][451/469] Loss_D: 0.9205 Loss_G: 1.6579 D(x): 0.5907 D(G(z)): 0.2149 / 0.2448
[24/25][452/469] Loss_D: 0.6821 Loss_G: 2.0209 D(x): 0.7807 D(G(z)): 0.2914 / 0.1783
[24/25][453/469] Loss_D: 0.6223 Loss_G: 2.0312 D(x): 0.7368 D(G(z)): 0.2137 / 0.1851
[24/25][454/469] Loss_D: 0.5361 Loss_G: 2.2144 D(x): 0.8027 D(G(z)): 0.2253 / 0.1427
[24/25][455/469] Loss_D: 0.6257 Loss_G: 2.3401 D(x): 0.7812 D(G(z)): 0.2768 / 0.1227
[24/25][456/469] Loss_D: 0.5895 Loss_G: 1.8779 D(x): 0.7135 D(G(z)): 0.1690 / 0.1978
[24/25][457/469] Loss_D: 0.5772 Loss_G: 2.5523 D(x): 0.8368 D(G(z)): 0.2860 / 0.1106
[24/25][458/469] Loss_D: 0.5315 Loss_G: 1.9391 D(x): 0.7143 D(G(z)): 0.1225 / 0.1888
[24/25][459/469] Loss_D: 0.5140 Loss_G: 2.8875 D(x): 0.9000 D(G(z)): 0.2992 / 0.0764
[24/25][460/469] Loss_D: 0.4985 Loss_G: 2.2803 D(x): 0.7348 D(G(z)): 0.1206 / 0.1487
[24/25][461/469] Loss_D: 0.4016 Loss_G: 2.5789 D(x): 0.8746 D(G(z)): 0.2050 / 0.1051
[24/25][462/469] Loss_D: 0.3297 Loss_G: 2.3579 D(x): 0.8236 D(G(z)): 0.1033 / 0.1267
[24/25][463/469] Loss_D: 0.3929 Loss_G: 3.1811 D(x): 0.9073 D(G(z)): 0.2345 / 0.0571
[24/25][464/469] Loss_D: 0.3796 Loss_G: 2.1890 D(x): 0.7840 D(G(z)): 0.0963 / 0.1502
[24/25][465/469] Loss_D: 0.3079 Loss_G: 2.7755 D(x): 0.9088 D(G(z)): 0.1728 / 0.0877
[24/25][466/469] Loss_D: 0.3576 Loss_G: 2.6457 D(x): 0.8461 D(G(z)): 0.1500 / 0.1014
```

Figure 9: Result of Latent dimension = 100

2.5 Summary

In this project, I explained the fundamentals of Generative adversarial networks, GANs. I used an official Pytorch example to train GANs on a MNIST dataset. I qualitatively explained the network outputs. I explained the logs and the trend we should observe in each of the loss functions during the training. I changed the dimension of input noise and study its effect on the probability of generated fake images.