Gertner Victor
gertnervictor@gmail.com

Implementing a GAN for MNIST Digit Synthesis

1 Data and package

So I took the data from the package datasets of torchvision. And for all the test I used pytorch as it's really nice in order to have a good understanding of what is happening.

2 Step 1: Basic GAN Implementation for MNIST Digit Synthesis

Theory So first we are going to implement a GAN with a basic approach a generator and a decoder. So in order to implement them, we have to understand what they are and how do we optimize them.

- Generator : it's a generative model that tries to capture the data distribution by creating data from noise.
- Discriminator : It's a discriminative model that estimate the probability that a sample came from the training data rather than a sample created by G.

So they are trained together, for G we maximise it's chance to trick the Discriminator. And for D it's simply minimizing the error of classification. So we have a minimax two player game.
We can write the Value function we are trying to optimize :

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

In practice, it gives us two loss functions :

- **discriminator** : maximise $\mathscr{L}_D = logD(x) + log(1 - D(\hat{x}))$ having x being real data and $\hat{x}$ being generated data.
- **generator** : minimize $\mathscr{L}_G = log(1 - D(G(z)))$ where z is the noise we sampled. Here we trying to trick the discriminator into thinking generated data is real data.

But in reality as stated in [1] it's better to maximize log D(G(z)). It gives the same result but the gradient is much stronger in the early phase of training.

Implementation :
So to implement it I considered that G and D are MLPs. Both have 3 layers, we could do more but for computation issue I stayed with 3. I added dropout for our model not to over fit to the training data.
We want G to go from a certain hidden dim to 784 (the flatten size of a MNIST digit) and we want D to go from 784 to 1 (classifying if real or not). So for D I used leaky ReLU for the activation of inner layer as it permits to avoid vanishing gradient and neuron dying, but I used sigmoid for the output layer as we are doing binary classification. On the other hand, I used a Tanh activation function on the output for G. To do so, we have to re scale our data between -1 and 1. (This idea was from internet, I started by using sigmoid activation, but I had a huge issue : the generator was collapsing, at the end of training it would only generate 0's. And even by trying to change the architecture of my model or try to initialise differently my generator, it wouldn't not collapse.)

Results :
So the results are not really good with my implementation, we can see that it tries to generate but there is often a mix in the digits.

3 Step 2 : conditionnal GAN implementation

So now we are going to implement a conditional GAN : to do so we add information to what the generator is fed, we add the digit in a one hot encoding format. Thus the generator is going to able to generate specific digits when it is been given a specific label. More over the discriminator is also fed with the labels for it to have more information on how to detect generated samples from the real one.

Implementation :
I kept the same architecture in terms of MLPs, but on the first layer i just add to the input dimension the number of class we want to be generated. And I just concatenate the flatten image and the class to be fed to the generator. The same process is made also for the discriminator.

Results :
So the conditional GAN has two real advantages that are demonstrated in the plotted results, first it permits to be able to generate a specific class. This was an issue with the classic GAN. Moreover, since we feed our two main components with additional information, they achieve to perform better, and the generator is able to produce way better samples of data than previously.

4 Step 3 : Adding an encoder to the C-GAN architecture

I had never encountered this kind of GAN before. Hence I tried to implement it the way it's pictured here :

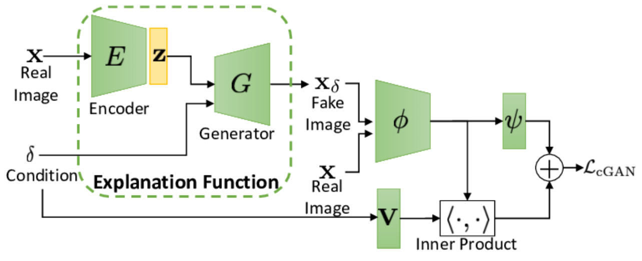

Figure 1: Encoder C-GAN

This idea between this is adding information to the generator. We are taking this idea from an auto-encoder. now the latent space is not going to be fully random but we will have a glance at the empiric distribution of the data : we learn a structured latent space.

Implementation :
So in order to implement it we have to add the encoder to our architecture .I took the decision to make it a part of the generator. Hence the new attribute, in the class definition of my generator. So now, the generator takes 3 arguments in the forward : noise, label and real image. Then the real image is passed through the encoder to be mapped into the latent space our generator picks from.Then the generator takes the concatenation of the noise, the label and the encoded real image in order to generate a sample. This adds information about what sample has to be generated because you now control your latent space. We can also think that this architecture will provide a better convergence for the generator : the generator is going to converge faster, hence we can think that there will be less case of the discriminator being too good or a collapsing generator.

Results :
Yet, in my implementation, I'm struggling to really get the whole thing to work. I think it as to do how I want to train the encoder. I might believe that we need to train it before and then freeze it's weights for the training of the rest of the C-GAN. But I'm not able to do so. And same if I try to train it at the same time than the GAN, I don't achieve to make it work...

References

[1] Mehdi Mirza Bing Xu David Warde-Farley Sherjil Ozair Aaron Courville Yoshua Bengio Ian J. Goodfellow, Jean Pouget-Abadie. Generative adversarial nets. 2014.