AI Lab: Computer Vision and NLP

Magne Hovdar (2037815), Andry Ambinintsoa Randrianantoandro (1891313)

May - June, 2022

SAPIENZA
UNIVERSITÀ DI ROMA

Contents

1 Introduction

1.1 The topic chosen

The topic, generation of *Pokémon* by using previous designs as data input is a self-chosen one. The main idea behind this was that old *Pokémon* designs are relatively simple and (in terms of pixels) small enough that a student's laptop could to process them through a neural network. Availability is another advantage that we will return to in the next subsection.
Originally, we wanted to use a VAR (*variational autoencoder*) but upon recommendation from professor Pannone, a Generative Adversarial Network was chosen instead. These two approaches are quite different from one another.

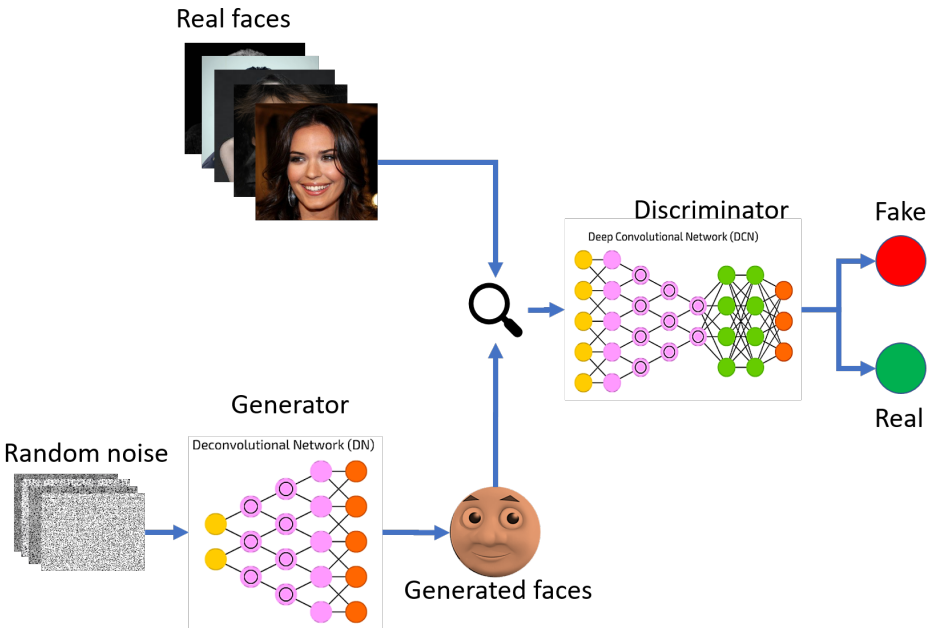

Figure 1: A Generative adversarial network (GAN).

A GAN works by pitching two different neural networks up against one another, a **generator**, whose role is to generate images similar to those in the dataset, and a **discriminator**, which tries to figure out whether a given image comes from the generator or the dataset . This latter has a *loss function* that rewards weighting if the network correctly classify fake (i.e.:generated) images from real (i.e.: from the dataset) images . On the other hand, the generator has a loss function that rewards weighting if the generator successfully tricks the discriminator into thinking that the generated picture is from the dataset. Ideally, there should be a fair competition between the networks as they improve together and eventually reach what is known from

Game Theory as a *Nash Equilibrium*. When this is achieved, images from the generator are indistinguishable from real images, thus the discriminator will have a 50% chance of guessing correctly.
The main challenge with a GAN is training it. As mentioned above, the two networks are dependent on a fair competition in order to improve together. If either network manages to completely outsmart the other, or in other words if the discriminator starts guessing correctly close to either 100% or 0% of the time then the learning potential of the losing network is reduced to the point of further training having no effect. Training the network in a way that prohibits this divergence proved to be the biggest challenge of this project.

1.2 Data availability

One of the reasons we chose *Pokémon* in particular was because of the availability of data. *Pokémon* designs used to get revamped for every new game which means there are several different sprites for every *Pokémon* available. Intuitively, we thought this could be advantageous since it would add some variety to every interpretation of a *Pokémon*. Professor Pannone recommended looking for existing datasets on *Kaggle*. Several datasets existed, but most of them used *artwork* instead of *spritework*. The difference being that spritework is created in low definition to be used on Nintendo handheld consoles and therefore does not require resizing. Using only spritework would also help to differentiate this project from other existing ones. The dataset used for this project was therefore created exclusively by images found at [10]. Tthe tool used to get a list of all *Pokémon* is found at [9].

1.3 Previous work done on the topic

There have been previous attempts at doing something similar. A notable one is the one found at [8] which has the same goal in mind as our project but uses high definition hand-drawn artwork in the dataset instead of pixelart.
In addition, some useful resources on how to train our GAN for this specific purpose were: Research on how to train GANs on limited datasets [5] and general principles on how to tune GAN parameters starting from a default implementation [6].

2 Method

2.1 The dataset

The process started with getting the dataset. Luckily, there are extensive online databases for *Pokémon* spritework online. Creating an offline dataset for training was done by using the script [10]. The spritework has varying quality, with dimentions of 64×64, 80×80 or 96×96. For this project, the database was downscaled to 64×64 for all pictures.
A sample is shown in fig. 2.

Figure 2: Sample of training images.

2.2 Default DCGAN implementation

The next step was implementing a GAN as described in section 1.1. We needed a robust and simple starting point and because we wanted to use *PyTorch*, the default implementation found in the *PyTorch* examples [1] was adopted as a starting point for the project. According to [12], there is no reason not to use a *Deep convolutional GAN (DCGAN)* unless you have a specific reason not to. This sentiment is consistent with the *PyTorch* default GAN example being precisely a DCGAN.
By simply switching from the default dataset to our self-made one, we now had a running DCGAN with a generator and a discriminator. However, since the project was not yet tailored to our needs, the generator would only produce noise using this implementation.

2.3 Simple tuning and GPU utility

The first step taken from the default implementation was to separate discriminator and generator training cycles. We began by giving the generator a few batches of "head start" allowing it to improve a little before the discriminator starts it's training. According to [6], separating training cycles like this can be viewed as a "quick fix" that allows us to see if the architecture actually works or not. However, final results will be worse than if the problem is fixed in a more permanent way where the generator and the discriminator are trained in an equal cycle. This was also evident from our results.
Once the noise from the generator started to obtain some kind of structure, the next logical step was to increase the number of training batches to see if the structured noise could be enhanced further into something that resembled pixelart. For this to be feasible we needed to use a GPU which meant moving the project to *Google Colab* and the dataset to *Google Drive*. With the free GPU supplied by *Google Colab*, we were able to train on up to 500 batches with a batch size of 64. The results are detailed in section 3.2.

2.4 Re-implementation and further tuning

In order to have complete control over our model and, initially, to also be able to convert it into a Wasserstein GAN (*WGAN*) as described in [14].[1], we decided to rewrite the code entirely ourselves. This was done with the help of [2], [3], [7], and [11], all materials on writing a DCGAN. In this implementation, the generator no longer had a head start on the discriminator. From there, we tried several methods to improve performance:

1. The amount of convolutional layers in the generator was doubled from the default Pytorch implementation. The idea came from [3], where the number of layers was the same as recommended in the paper ([4]). This resulted in higher output quality (see fig. 7) in a fewer number of epochs.

2. Following this success, we mirrored this architecture in the discriminator, doubling its convolutional layers. However, this did not produce any noticeable change, most likely due to the fact that the discriminator was already good at its job and thus was not affected by the increase in potential learning capabilities.

3. Until now, we had used a batch size of 64, but once again following the recommendations of the paper, we set it to 128. As a result, training speed was increased.

[1] in short, a DCGAN that uses a different loss function, see section 3.5

4. We experimented with setting all backgrounds to black for real sprites before they were sent to the discriminator, mainly as the different colours seemed to cause some noise in early runs. Although we had some positive effects on sprite sharpness, having a uniform background seemed to increase the chances of *mode collapse*. See fig. 9. As indicated by [12], data augmentation counteracts mode collapse so we can theorize that the variety in background color is helpful for the model. Thus, we reverted to not editing the backgrounds in the dataset.

5. Another attempted improvement was by halving the generator's and discriminator's amount of layers, w.r.t. the default implementation. This was simply an experiment based on the principle that simpler is better, but ultimately the results were far from satisfactory. Because of the small size of the networks, training was much faster, so we once again tried running for 500 epochs, but the results (see fig. 10) were about the same as what we had obtained with the previous architecture running for less than a quarter of that number of epochs.

3 Results

3.1 Default implementation results

The default implementation was only able to produce noise. Since the generator uses Gaussian noise as input, getting noise as an output meant that the weights in the generator network were not being trained at all or they were being trained randomly. By tracking the values of the loss functions in both networks, it was apparent that the loss of the discriminator very quickly approached 0. In other words, the discriminator was very quickly able to sort out the real samples from the fake ones without making any mistakes. This stops the learning process of the generator since no matter how the weights are tuned, the result is the same. The main problem to solve was therefore: Stop the networks from diverging and instead make them converge into a Nash Equilibrium.

```
Loss_D: 1.6306  Loss_G: 4.0068  D(x): 0.4196    D(G(z)): 0.4229 / 0.0224
Loss_D: 0.0334  Loss_G: 29.3709 D(x): 0.9764    D(G(z)): 0.0000 / 0.0000
Loss_D: 0.3660  Loss_G: 19.9719 D(x): 0.9649    D(G(z)): 0.2643 / 0.0000
Loss_D: 0.0001  Loss_G: 35.6428 D(x): 0.9999    D(G(z)): 0.0000 / 0.0000
Loss_D: 0.0001  Loss_G: 35.1634 D(x): 0.9999    D(G(z)): 0.0000 / 0.0000
Loss_D: 0.0182  Loss_G: 37.1623 D(x): 0.9892    D(G(z)): 0.0000 / 0.0000
Loss_D: 0.0004  Loss_G: 36.9652 D(x): 0.9996    D(G(z)): 0.0000 / 0.0000
Loss_D: 0.0002  Loss_G: 33.8933 D(x): 0.9998    D(G(z)): 0.0000 / 0.0000
```

Figure 3: Discriminator dominates and bottlenecks generator.

3.2 Results after small changes and GPU

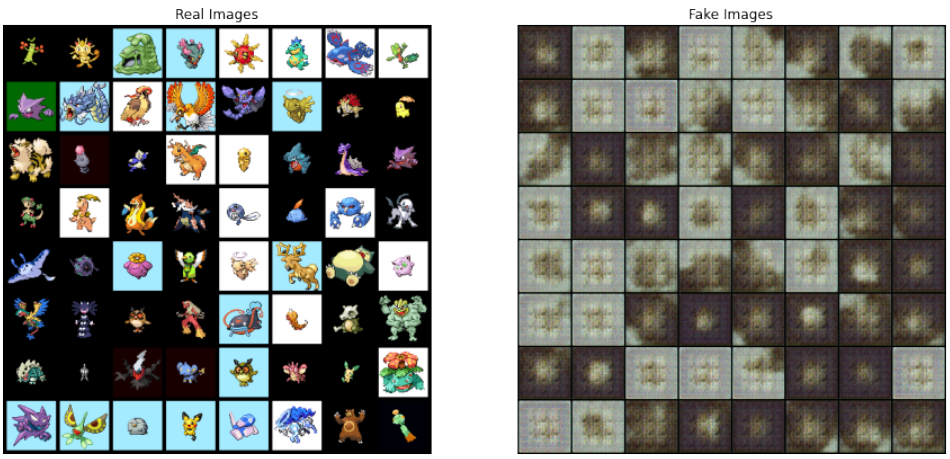

Figure 4: Structured noisy output.

After applying the changes described in section 2.3, training runs with a low amount of epochs would produce noise with some sort of structure

Figure 5: Result from a diverging training session.

as seen figure 4. This indicated that the generator network was learning something. By increasing the amount of training, the generator was able to produce something that resembled pixelart. This result can be seen in fig. 5. However, giving the generator network a "head start" only works as a quick-fix and does not stop the networks from eventually diverging. The next cause of action was then to make the network approach a stable equilibrium. The divergence is illustrated in figure 6.

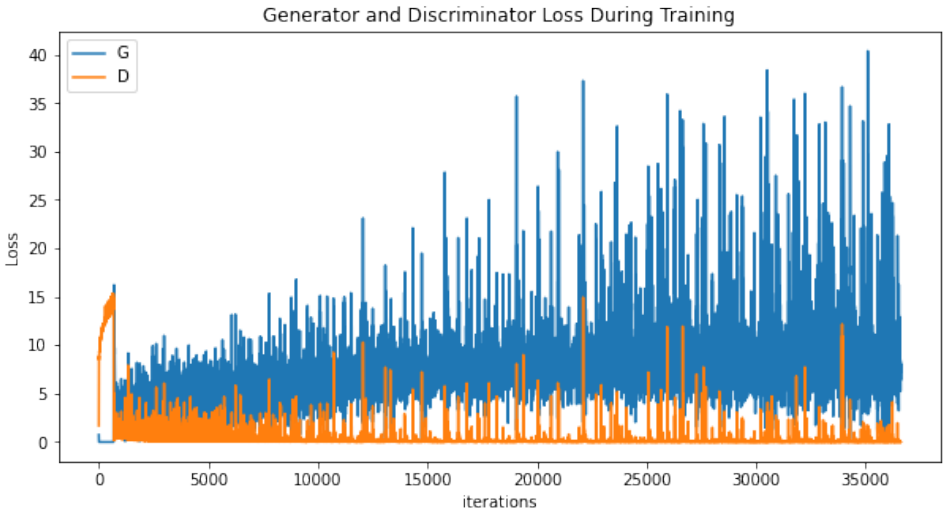

Figure 6: Illustration of the divergence.

3.3 Results after re-implementation

The most significant change we have made for our model is the architectural one, where we doubled the generator's layer size. The images we have obtained, while of comparable quality to those in fig. 5, were obtained in about half the number of epochs, and without giving the generator any head start. An example is shown in fig. 7.

Figure 7: Result after further improvements.

Some sprites obtained from setting the background to black appeared sharper, we can venture to say that this is because the generator, not having to learn about the different background colours, could dedicate itself more to the *Pokémon* themselves. In fig. 8, for example, it outputted one of the better looking looking fakes so far. However, as one can see, this has also led the generator to output a lot of plain black images. Our method for changing the background colour was a very simple one that resulted in colours matching the background colour being changed as well. Since the effect of this augmentation is hard to predict, we are not surprised to see some negative effects on the generated outputs. In the end, although some results of this were promising, we revoked this change to avoid empty sprites fooling the discriminator.

The other attempted improvements did not yield any better results, however some figures that we have obtained are shown in appendix A.

3.4 Final results

The final result is a GAN model that is able to create pixel art with a concise form and a limited colour palette. It might still be easy for a human to differentiate which are real and which are fake, but some of the results are meaningful nonetheless. The model seems to understand that a *Pokèmon*

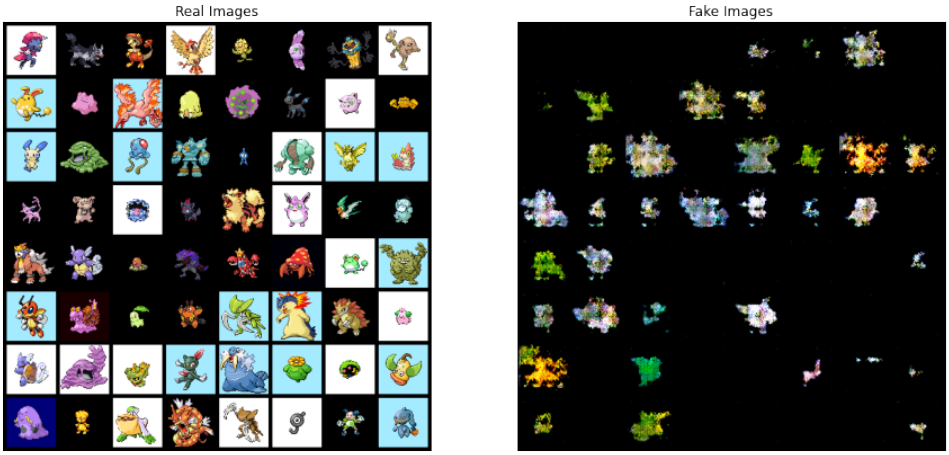

Figure 8: A pseudo-Charizard on the upper right.

normally consists of a few colours grouped together in a structured fashion. Although the final results are a little lacking compared to human creativity, they can be used as templates to improve on manually.

3.5 Future work

During development we had an unsuccessful attempt at implementing a *WGAN*. By replacing the binary discriminator with a stochastic evaluator of "realness", a critic, *WGAN* is supposed to increase stability and rate of convergence. This is achieved, among other changes, mainly by using a different loss function, namely the Wasserstein loss, also known as the earth-mover distance (more details in [13]). We were not able to implement this in a way where this was the case, the generator ended up completely dominated[2] by the critic (or discriminator) and was unable to learn. This could have been experimented a lot more with.
The relatively limited dataset could be improved using "stochastic discriminator augmentation" as presented in [5]. We imagine potential improvements by augmenting the dataset in a way that makes sure the generator does not learn the augmentation. Sadly, we did not have the time to implement this in the model.

[2] see figure 11

A Experimental results

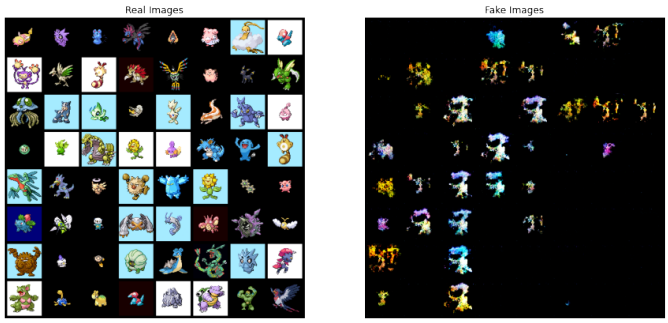

Figure 9: Complete mode collapse with black background.

Figure 10: Result after 500 epochs with simplified generator.

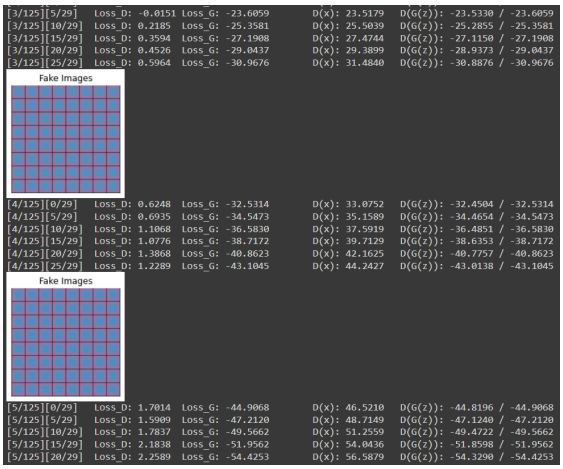

Figure 11: Unsuccessful attempt at implementing a *WGAN* that led to complete divergence and a noisy output.

References

[1] *DCGAN examples*. https://github.com/pytorch/examples/blob/main/dcgan/README.md. Accessed: 2022-05-11.

[2] *DCGAN github*. https://github.com/aladdinpersson/Machine-Learning-Collection.

[3] *DCGAN implementation video*. https://www.youtube.com/watch?v=IZtv9s_Wx9I.

[4] *DCGAN paper*. https://arxiv.org/abs/1511.06434.

[5] *GAN limited datasets*. https://arxiv.org/abs/2006.06676. Accessed: 2022-05-11.

[6] *GAN training principles*. https://towardsdatascience.com/10-lessons-i-learned-training-generative-adversarial-networks-gans-for-a-year-c9071159628. Accessed: 2022-05-11.

[7] *MonsterGAN*. https://medium.com/@yvanscher/using-gans-to-create-monsters-for-your-game-c1a3ece2f0a0.

[8] *Pokemon GAN 2018*. https://github.com/llSourcell/Pokemon_GAN. Accessed: 2022-05-11.

[9] *Pokemon list generator*. https://www.dragonflycave.com/resources/pokemon-list-generator. Accessed: 2022-05-11.

[10] *Pokemon Sprites Database*. https://pokemondb.net/sprites. Accessed: 2022-05-10.

[11] *Pytorch DCGAN*. https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html.

[12] *Training stable GANs*. https://machinelearningmastery.com/how-to-train-stable-generative-adversarial-networks/. Accessed: 2022-05-11.

[13] *Wassertein loss*. https://en.wikipedia.org/wiki/Wasserstein_metric.

[14] *WGAN paper*. https://arxiv.org/abs/1701.07875.