LINMA2472-Final Homework

Group 5
Cédric Antoine, Romain Graux, Lionel Lamy

March 6, 2021

1 Introduction

Info: Scripts used to realise our project can be found on GitHub :
`https://github.com/RomainGrx/LINMA2472-Homeworks/tree/main/Homework%204`

Moreover, our Tensorboard is available via :
`http://raspcloud.romaingraux.xyz:6006`

This project was carried out as part of the LINMA2472 course (Algorithms in data science) at UCLouvain. The objective of this project is to chose a dataset and to analyse it with non-trivial algorithms.

2 Project idea

In the search for the type of project we were going to carry out, we repeatedly came across projects by people who used 'generative adversial networks (GAN)' to generate fictitious faces that were as realistic as possible. We then asked ourselves the question, if it was possible to do the same operation but with cartoon caracters. Not finding any project that had tried this approach, we decided to make it our final project. We then immediately started looking for a dataset to start the project. We hesitated between several dataset (Simpsons, South Park,...) but our final choice was a bitmoji dataset.

3 Dataset

At the beginning of our project, we used a kaggle dataset composed of 4000 png images of bitmojis heads (`https://www.kaggle.com/mostafamozafari/bitmoji-faces`). However, we soon realised that on one hand the dataset did not contain enough images for the use we were going to make of it and on the other end that it contained quite a few defaults such as text or badly cropped images which didn't fit our needs.We therefore tried to enlarge our dataset

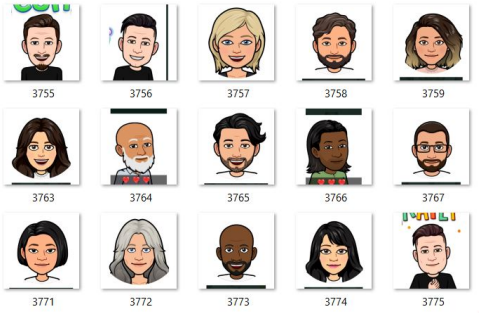

Figure 1: Sample of the initial dataset (input)

Figure 2: 32x32 output sample (generated with inital data)

After looking for other datasets we discovered a GitHub repository (`https://github.com/JoshCheek/bitmoji`) where they showed how bitmojis could be generated using a kind of bitmoji render api (in fact, using the url format). Therefore, we implemented a python script to retrieve/generate bitmojis ourselves with requests to the render-url. This way we were able to generate an unlimited amount of images, which were obviously better suited to our needs.

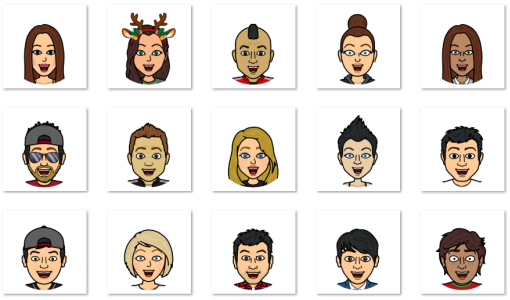

Figure 3: Sample of the dataset we generated and used

4 Generative adversial network (GAN)

Let's now quickly explain what a 'generative adversial network (GAN)' is. A generative adversarial network is a class of machine learning frameworks. Given a training set, this algorithm learns to generate new data with the same characteristics as the training set. The core idea of a GAN is based on a conflict between a generator and a discriminator. This basically means that the generator is not trained to minimize the distance to a specific image, but rather to fool the discriminator and the discriminator is trained to distinguish real from false images. This enables the model to learn in an unsupervised manner.

More precisely, to learn the generator's distribution p_g over data x, we define a prior on input noise variables $p_z(z)$, then represent a mapping to data space as $G(z;\theta_g)$, where G is a differentiable function represented by a multilayer perceptron (MLP) with parameters θ_g. We also define a second MLP $D(x,\theta_d)$ that outputs the probability that x belongs to the real data. We train D to maximize the probability of assigning the correct label to both training examples and samples from G. For a *basic* GAN, we train G to minimize $log(1-D(G(z)))$ and train D to maximize this last value plus $log(D(x))$, in other words:

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

4.1 Generator structure

In this case, as we produce images, we use a structure with convolution layers, the GAN is thus called DCGAN for Deep Convolutional Generative Adversarial Network. The generator is composed of 3 types of block : *dense block*, *convtranspose block* and a final *convolutional block*.

The *dense block* is composed of a Dense layer of the good number of neurons to match the first convtranspose number of neurons, a Reshape layer to match image dims (B,W,H,C) and then a LeakyRelu activation.

The *convtranspose block* is composed of a ConvTranspose2D layer, a BatchNormalization layer and then a LeakyRelu activation.

And then the *convolutional block* is composed of a Conv2D layer and then a tanh activation to match the $[-1,1]$ range of the images.

For the 64×64 example, here is the global structure:

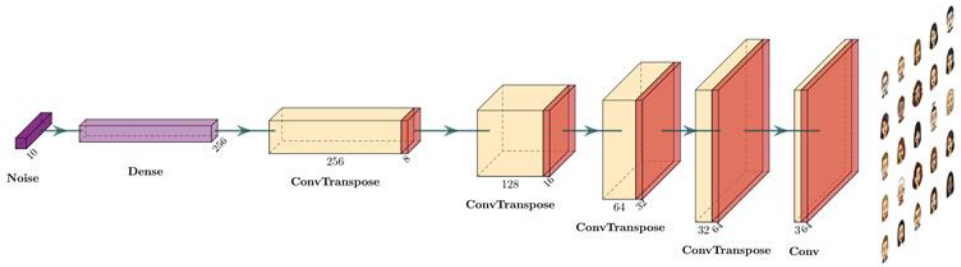

Figure 4: Generator structure

4.2 Discriminator structure

The discriminator is composed of 2 types of blocks : *convolutional block* and a *dense block*.

The *convolutional block* is composed of a Conv2D layer, a BatchNormalization layer and then a LeakyRelu activation.

And the *Dense block* is composed of a first Dense layer, a Flatten layer and finally a last Dense layer as output.

For the 64×64 example, here is the global structure:

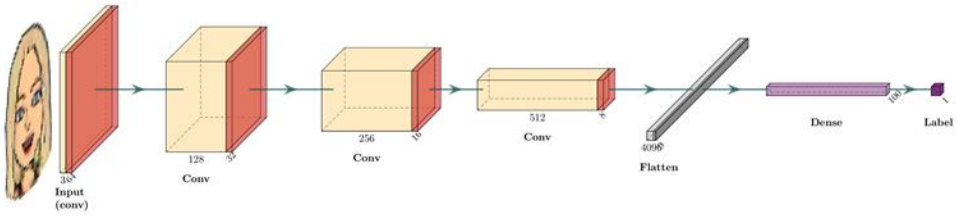

Figure 5: Discriminator structure

5 Training stages

5.1 Classical GAN

At first we started by implementing a classical GAN structure. We provided this GAN with the images of the initial dataset and started by trying to generate 32x32 bitmojis (Figure 8).

This GAN is composed of the previous explained Generator structure and Discriminator structure but we add a Sigmoid activation to the discriminator to force the output to be between 0 and 1 (e.g. fake and true data) and we then used a BCE (Binary crossentropy) loss to compute how well does the discriminator classify these data.

This worked more or less well and we then tried to generate 64x64 bitmojis. However, GAN very quickly began to mode collapse. All the generated images were the same.
A mode collapse happens when the data is multi-modal but the generator gets stuck in one mode or more generally in a local minima. This means that the generator learns to fool the discriminator

by producing only a small variety of data compared to the training set (Here is a visual example : `https://www.youtube.com/watch?v=r58tgUy9r_s`).

Multiple solutions were available to us, such as :

- Change the loss function
- More training for the discriminator than the generator

We then implement both these solutions in the next section.

5.2 Wasserstein GAN

5.2.1 Loss function

A problem with the KL-divergence/Maximum Likelihood loss is that if the the distribution of the real data and the distribution of the generated data are very different (does not overlap), the computed gradients are close to zero in the non-overlapping region and thus the network does not learn anything.

The solution proposed by the WGAN is to compute a loss that can lead to non-zero gradients even if the distributions are totally different from each other. It turns out that it can be done by computing a *Wasserstein* loss that is basically a continuous generalization of the Earthmover distance. It is defined as the minimal amount of work to move the probability mass of one distribution to make it match to the other.

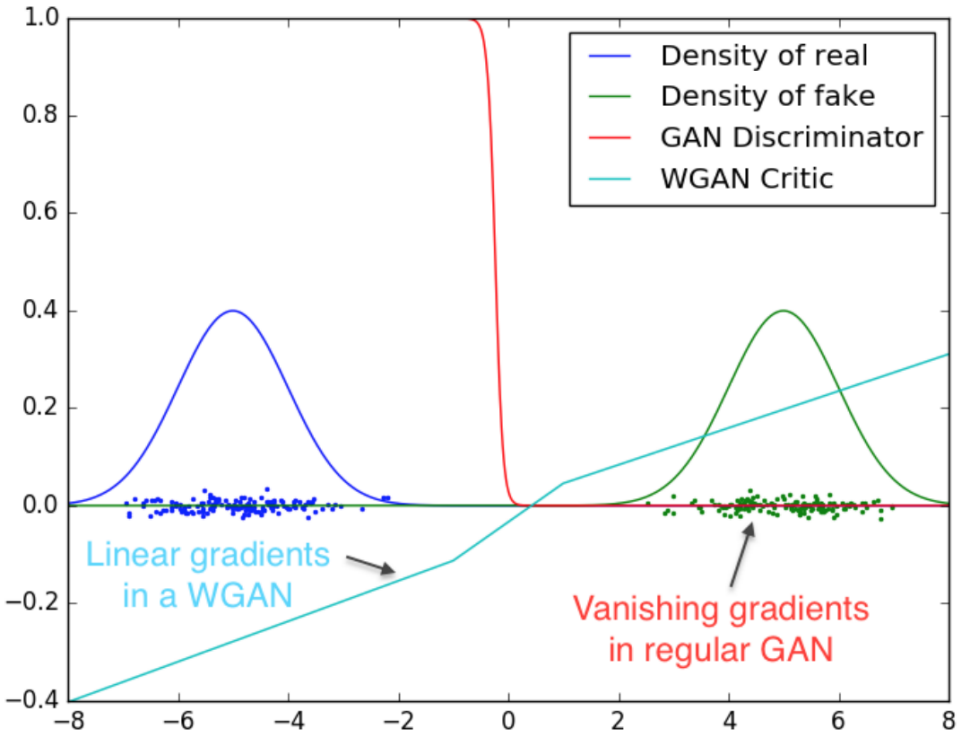

Figure 6: GAN vs WGAN

So as you can see on the Figure 6, the discriminator does not output a class probability real vs fake anymore but just a real number representing a confidence for either of the classes, hence it is called a *critic* instead.

The wasserstein distance is :

$$W\left(\mathbb{P}_r, \mathbb{P}_g\right) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]$$

Where $\mathbb{P}_x$ and $\mathbb{P}_g$ are respectively the probability distribution of real data and generated data. And $\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)$ being all plans with $\sum_y \gamma(x,y) = p(x) \& \sum_x \gamma(x,y) = p(y)$

5.2.2 Lipschitz continuity

As the original formularization of the Wasserstein distance is usually intractable, we need to introduce the Lipschitz continuity.

A real function f:$\mathbb{R} \to \mathbb{R}$ is Lipschitz continuous if

$$|f(x_1) - f(x_2)| \leq K |x_1 - x_2|$$

Therefore the Lipschitz continuity limits the maximal derivative value of the function.

Intuitively, Lipschitz continuity bounds the gradients and beneficial in mitigating gradient explosions in deep learning.

With this new concept, we can change our distance metrics, thanks to the Kantorovich-Rubinstein duality and therefore the distance becomes

$$W(\mathbb{P}_r, \mathbb{P}_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_g}[f(x)]$$

Where *sup* is the least upper bound. We need to find f which is a 1-Lipschitz function that follows the constraint

$$|f(x_1) - f(x_2)| \leq |x_1 - x_2|$$

and maximizes

$$\mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_g}[f(x)]$$

5.2.3 Train the discriminator

As we now use the discriminator as a critic, the steps become:

- For a fixed θ_g (generator variables), train the discriminator until convergence of θ_d regarding to the Wasserstein loss $W(\theta_d, \theta_g)$. In practice we compute it N times.
- Once we find the optimal θ_d, we compute the θ_g gradient of the loss over $D(g(z))$ by sampling several $z \sim Z$.
- Update θ_g and repeat.

The discriminator plays the more important role in the process because it is telling the generator how far it is from the true distribution, we thus update the discriminator N times to help him learn how the generator performs and how well it is doing. Once learned the distribution of the generator over the true distribution, we update one time the generator regarding to the Wasserstein distance.

5.2.4 Weight clipping

To enforce the constraint, WGAN applies a very simple clipping to restrict the maximum weight value in f, i.e. the weights of the discriminator must be within a certain range. In our case we clipped the weights between $[-0.001, 0.001]$.

5.3 WGAN-GP (Wasserstein GAN with Gradient penalty)

After training the new model that worked theoretically, we realized that it was still collapsing. In fact the weight clipping was not the best choice to constraint f.

Another way to enforce the Lipschitz constraint is to use a gradient penalty.
A differentiable function f is 1-Lipschitz if and only if it has gradients with norm at most 1 everywhere.

$$f^* \text{ has gradient norm } \underline{1} \text{ almost everywhere under } \mathbb{P}_r \text{ and } \mathbb{P}_g$$

We can find in Appendix A of WGAN-GP paper that

$$\begin{aligned} &\text{if } f^* \text{ is differentiable }, \pi(x = y) = 0, \text{ and } x_t = (1-t)y \text{ with } 0 \leq t \leq 1 \ , \\ &\text{it holds that } \mathbb{P}_{(x,y)\sim\pi}\left[\nabla f^*\left(x_t\right) = \frac{y - x_t}{\|y - x_t\|}\right] = 1 \end{aligned}$$

So, points interpolated between the real and generated data should have a gradient nrom of 1 for f

Therefore, instead of applying weights clipping to enforce Lipschitz constraint, WGAN-GP penalizes the model if the norm of the gradients moves away from 1.

The final loss becomes now

$$L = \underbrace{\mathop{\mathbb{E}}_{\tilde{x}\sim\mathbb{P}_g}[D(\tilde{\boldsymbol{x}})] - \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathbb{P}_r}[D(\boldsymbol{x})]}_{\text{Original critic loss}} + \underbrace{\lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}}\sim\mathbb{P}_{\hat{\boldsymbol{x}}}}\left[\left(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1\right)^2\right]}_{\text{Gradient penalty}}$$

where $\hat{x}$ sampled from $\tilde{x}$ and x with t uniformly sampled between 0 and 1

$$\hat{\boldsymbol{x}} = t\tilde{\boldsymbol{x}} + (1-t)\boldsymbol{x} \text{ with } 0 \leq t \leq 1$$

5.4 Speed up

After finally finding the right model (WGAN-GP), we needed to speed up the process, 2 RTX 2080 Ti was not enough. So we rented on vast.ai, the services of 4 GPUs (RTX 3090) to speed up the train phase and to be able to increase the number of epochs made. We then let our model run on these GPUs for 17 hours to generate 128x128 bitmojis. However, even this was not enough, we would have had to let the model run longer.

6 Final results

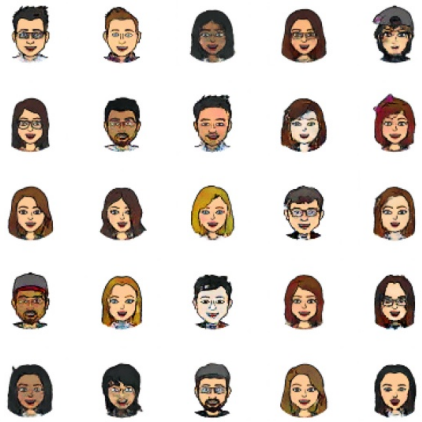

Figure 7: 64x64 WGAN-GP generated bitmojis

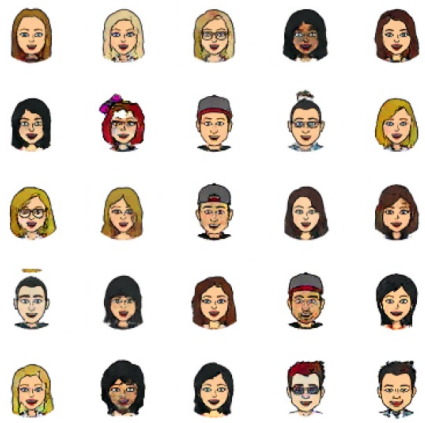

Figure 8: 64x64 WGAN-GP generated bitmojis

7 Conclusion

We can conclude by saying that our final model, the Wasserstein GAN with gradient penalty, is well capable of efficiently generating a wide range of bitmojis. It is more efficient in the diversity of bitmojis created and therefore does not collapse like the previous models we used.
On the other hand, we have also observed that at our level, generating 128x128 bitmojis or larger is almost impossible because of the computational power and the time it requires. However, with the right hardware this should not be a problem.