ArtGan - Generating Art

Thomas Gilles, Sarah Perrin

***Abstract*—We apply different Generative Adversarial Network techniques to generate art. We also train the network to generate specific art styles, not limiting ourselves to simply generate content as portraits, plant pots or landscapes. In particular, we interest ourselves to the different outputs in style for a same input, and the potential mix between art styles. Our code can be found here.**

I. Introduction

Generative Adversarial Networks are a powerful class of generative models that cast generative modelling as a game between two networks : a generator network produces synthetic data given some noise source and a discriminator network discriminates between the generators output and true data. GANs can produce very visually appealing samples, but are often hard to train.

They are many different possible applications for GANs, however most of their known results are applied on images. One frequent example is faces, where the GAN is trained on a dataset of faces, celebrities for examples, and generates new faces that seem familiar but are different from any face in the training dataset. One of the main hurdle is the resolution of images, sometimes restricting the network to low resolution, leading to mildly appealing results.

We interested ourselves in the idea of creative generation, and wondered if the model would be efficient in creating art paintings. Moreover, we studied the ability of GANs to generate specific styles, and what would happen if two classes were mixed together.

II. Background and Related Work

A. GANs

GANs were first introduced in the famous paper Generative Adversarial Nets [2] written by Ian Goodfellow. Generally, the goal is to generate images that seem real. In our case, we want to generate paintings of specific art styles. To do so, we need to approximate the distribution of the data, i.e to learn the generators distribution p_g over data x. Thus, we define G the generator and D the discriminator, as two neural networks (originally multilayer perceptrons) that compete against each other to solve the optimization game :

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

We train D to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train G to minimize $\log(1-D(G(z)))$, that is to say to fool D. $p_z(z)$ is a prior on input noise variables. For instance, to generate numbers, a prior could be a random vector of dimension 10.

Training GANs is not easy and is known to be not very stable. In fact, early in learning, when G is poor, D can reject samples with high confidence because they are clearly different from the training data, and it can be hard for G to improve itself. Similarly, if D is very bad, G can fool D too easily and will stagnate and generate poor quality samples. This is why the learning of both networks must be synchronized.

In practice, the learning process is divided in two steps, one for each network. First, the generator generates fake images from a random noise of the latent space. These images are concatenated with a random batch of real images and the discriminator is trained over the whole sample. Then, another random noise generates other fake images. The discriminator weights are frozen, and the generator is updated to fool the discriminator. These steps are repeated many times over the whole dataset until the quality outputed by the generator is deemed sufficient.

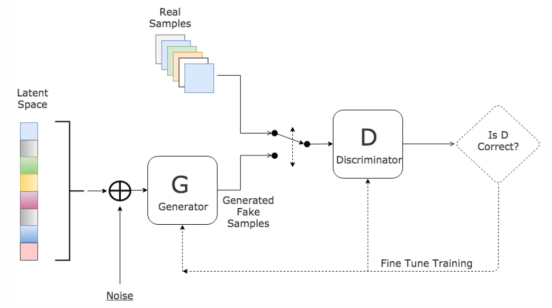

Fig. 1: Scheme representing how a GAN works

B. DC-GANs

DC-GANs (Deep Convolutional GANs) [7], are GANs for which multilayer perceptrons have been replaced by Convolutional Networks, as Convolutional Networks have proved to be very efficient on images. More precisely, Convolutional Networks of DC-GANs have some specificities :

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

C. Wasserstein GANs

1) Wasserstein GANs theory: To improve the stability and learning of GANs, Wasserstein GANs have been introduced in [1]. The main idea is that classical GANs as the ones presented before cannot approximate well the distribution of the data because they are based on the Jensen-Shannon distance. In fact, if we suppose that we have the optimal discriminator and fixe it in equation 1, then the optimization problem 1 can be written as (see [2] for the full demonstration):

$$\max_D V(D,G) = -\log(4) + 2 \cdot JSD(p_{data} \| p_g) \tag{2}$$

Wasserstein GANs are based on another distance called the Earth-Mover distance and defined as:

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}[\|x - y\|] \tag{3}$$

where $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ denotes the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively $\mathbb{P}_r$ and $\mathbb{P}_g$. Intuitively, $\gamma(x, y)$ indicates how much mass must be transported from x to y in order to transform the distributions $\mathbb{P}_r$ into the distribution $\mathbb{P}_g$. The EM distance then is the cost of the optimal transport plan.

Advantages of EM distance compared to JS distance are that it has nice continuity properties:

- If g is continuous in θ (weights of the generator), so is $W(\mathbb{P}_r, \mathbb{P}_\theta)$.
- If g is locally Lipschitz and satisfies the previous assumption, then $W(\mathbb{P}_r, \mathbb{P}_\theta)$ is continuous everywhere, and differentiable almost everywhere.

These assumptions can be easily verified in a neural network (see [1]). For instance, it is possible to make G 1-Lipschitz by clipping the weights into $[-1, 1]$. Then, the value function in Wasserstein GANs can be written as:

$$\begin{aligned} \min_G \max_{D \in \mathcal{D}} V(D,G) = \mathbb{E}_{x \sim p_{data}}[\log D(x)] \\ -\mathbb{E}_{z \sim p_z(z)}[\log(D(G(z)))] \end{aligned} \tag{4}$$

Where $\mathcal{D}$ is the set of 1-Lipschitz functions. Under the optimal discriminator, we can show that finding the best generator is equivalent to minimizing the EM distance $W(\mathbb{P}_r, \mathbb{P}_g)$ in $C(G)$.

2) Improved training of Wasserstein GANs: Recently, the paper Improved training of Wasserstein GANs [3] proposed an alternative to clipping weights, which was a very harsh mean to enforce Lipschitz constraint. They rather propose to use gradient penalty, which modifies the value function as follow:

$$\begin{aligned} V(D,G) = \mathbb{E}_{x \sim p_{data}}[\log D(x)] - \mathbb{E}_{z \sim p_z(z)}[\log(D(G(z)))] \\ +\lambda \mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{x}}}[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2] \end{aligned} \tag{5}$$

$\hat{x} \sim \mathbb{P}_{\hat{x}}$ is a random weighted average of real data and fake generated data. We will discuss how we implemented this method - our best results - in the next section.

III. Method

This section describes technical details of our implementation. We built two different types of networks : one inspired from DC-GAN and one inspired from Wasserstein GAN. For Wasserstein GANs, we implemented the version described in [3]. However, they both have a particularity: they are "conditional GANs", as it is possible to specify the output class desired. For instance, on MNIST dataset, it is possible to choose the target number, for CIFAR 10 it is possible to choose the target object, and for Wikiart, it is possible to choose the style. For these three datasets it is also possible to mix the outputs.

A. Discriminator and Generator

1) DC-GAN: For our implementation and adaptation of DC-GAN, we use convolutional neural networks in both the discriminator and the generator.

For the generator, we use a succession of deconvolution layers (Conv2DTranspose in Keras), with Batch Normalization and ReLu activation. The deconvolution layers increase gradually the size of the noise to reach the final size of the target image (64×64 pixels).

For the discriminator, we use classic convolution layers with Batch Normalization and Leaky ReLu activation.

2) Wasserstein: For the discriminator, the network is very similar to ResNet [4]: residual blocks with Layer Normalization instead of Batch Normalization. We did not use Batch Normalization as recommended in [3] because batch normalization changes the form of the discriminators problem from mapping a single input to a single output to mapping from an entire batch of inputs to a batch of outputs.

B. Losses

This section describes the different loss functions used in our GANs. For DC-GAN, we simply used the double loss, and for WGAN-GP, we used the label loss (from double loss) combined with Wasserstein loss (to replace the validity loss of the double loss) and gradient penalty.

1) Double loss: As we implemented a conditional GAN, we need to have two terms in the loss:

- validity: to check if the sample is a "real" or "fake" one. The associated loss is binary cross entropy.
- label: the discriminator tries to predict the label (number, object or style). The associated loss is categorical cross entropy.

The loss is then the sum of these two terms.

2) Wasserstein loss: The Wasserstein loss function is very simple to calculate. In a standard GAN, the discriminator has a sigmoid output, representing the probability that samples are real or generated. In Wasserstein GANs, however, the output is linear with no activation function. Instead of being constrained to [0, 1], the discriminator wants to make the distance between its output for real and generated samples as large as possible. The most natural way to achieve this is to label generated samples -1 and real samples 1, instead of the 0 and 1 used in normal GANs, so that multiplying the outputs by the labels will give you the loss immediately. Note that the nature of this loss means that it can be (and frequently will be) less than 0.

3) Gradient Penalty: Gradient Penalty loss is calculated over a batch of "averaged" samples (as described in previous section). In WGANs-GP, the 1-Lipschitz constraint is enforced by adding a term to the loss function that penalizes the network if the gradient norm moves away from 1. However, it is impossible to evaluate this function at all points in the input space. The compromise used in the paper is to choose random points on the lines between real and generated samples, and check the gradients at these points. Note that it is the gradient with respect to the input averaged samples, not the weights of the discriminator, that we're penalizing! In order to evaluate the gradients, we must first run samples through the generator and evaluate the loss. Then we get the gradients of the discriminator with respect to the input averaged samples. The L_2 norm and penalty can then be calculated for this gradient. Note that this loss function requires the original averaged samples as input, but Keras only supports passing `y_true` and `y_pred` to loss functions. To get around this, we make a partial() of the function with the `averaged_samples` argument, and use that for model training.

C. Training

To train our GAN, we used a GPU Nvidia RTX 2080. We have used Adam optimizer [6] with learning rate 0.0002. The other parameters are:

- MNIST: Batch size = 64. The training only took a few minutes.
- CIFAR: Batch size = 64. The training of one epoch was around one minute and a half for DC-GAN, 5 minutes for WGAN and we trained the GAN over 100 epochs.
- Wikiart: Batch size = 32. The learning rate for Adam decreases progressively to reach 0 at the 100th epoch. For WGAN, one epoch took approximately 15 minutes, and we trained over 100 epochs.

The training was longer for WGAN than for DC-GAN because of the networks that are based on ResNet. Furthermore, it is necessary to train the discriminator five times at each iteration, which is costly. A description of the training algorithm is available in Figure 2.

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.
while θ has not converged **do**
  for $t = 1, ..., n_{\text{critic}}$ **do**
    for $i = 1, ..., m$ **do**
      Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
      $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

Fig. 2: WGAN-GP algorithm

IV. Experiments

A. MNIST

We first train our GANs on a simple dataset : MNIST, which is made of 60 000 hand-written digits stored in grayscale on 28*28 images. We can observe on Figures 3, 4, 5 and 6 the improvements during the training.

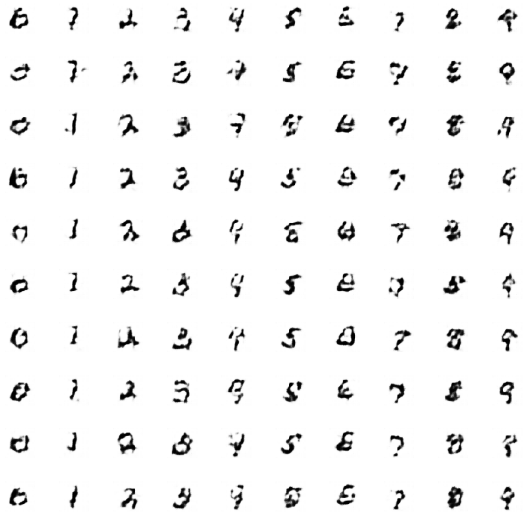

Fig. 3: DC-GAN trained on MNIST after 1 epoch : the digits are still blurry and undefined

Fig. 4: DC-GAN trained on MNIST after 5 epoch : the digits are pretty well-formed apart from a few mistakes on 2s, 3s and 5s

However on Figure 6, artifacts appear as the training is too long : the discriminator has become too powerful and doesn't provide necessary feedback to the generator (the discriminator is right on more than 90% of the cases).

We can also observe what happens when we ask the generator to generate two different digits at the same time : the results are a mix of both numbers, resulting in an hybrid digit (cf Figure 7).

Fig. 5: DC-GAN trained on MNIST after 10 epoch

Fig. 6: DC-GAN trained on MNIST after 20 epoch : artefacts appear, with blurred 6s and points next to 1s

B. CIFAR-10

We next move onto a slightly harder dataset : CIFAR-10, which comprises 60 000 32*32 pictures divided into 10 classes : Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship and Truck. The images are more complex than digits, and colored, generating a higher difficulty for the GAN.

Our first model, a DC-GAN similar to the one used for MNIST, fails to reproduce the images as seen in Figure 8, outputting blurry images where the class cannot even be identified.

However, our other model based on WGAN-GP provides better results (cf. figure 9).

Fig. 7: DC-GAN trained on MNIST : mixing matrix : input asked is the superposition of 2 numbers

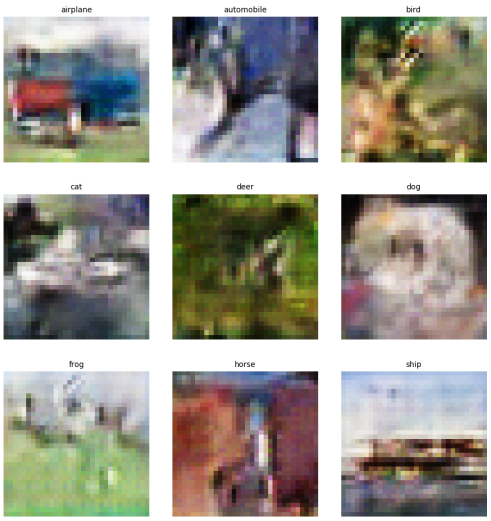

Fig. 8: DC-GAN trained on CIFAR : images and blurred and not significantly linked to the classes

C. Wikiart

The last dataset, Wikiart, is composed of around 66000 artworks from 25 different styles and different sizes. We resized every artwork before the training to match 64*64 target size. This dataset was clearly the hardest one because of the wide variety of styles in the database. Figure 10 shows some of our results for different styles.

It is also possible to mix styles for Wikiart (Figure 11). Furthermore, we have obtained very satisfying results for styles that were the most represented in the dataset: Impressionism (10000 artworks), Expressionism (5000 artworks) and Romanticism (5000 artworks).

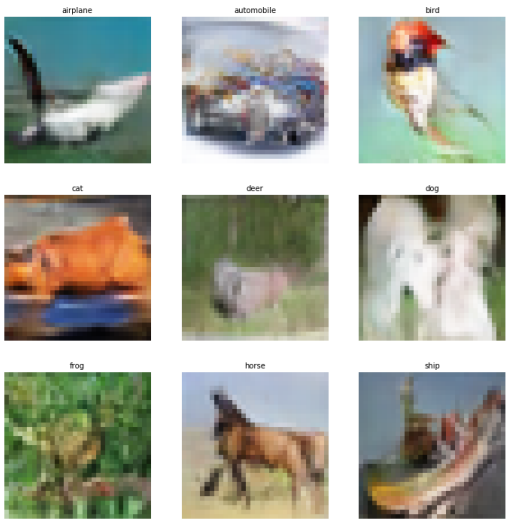

Fig. 9: WGAN trained on CIFAR

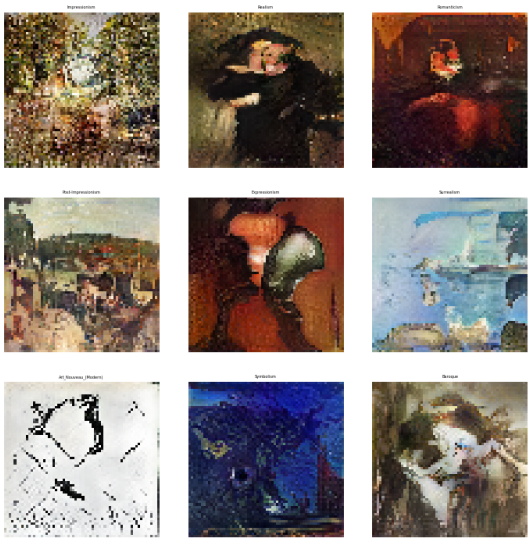

Fig. 10: WGAN trained on Wikiart : different art styles

V. Conclusion and Future Work

We have implemented two types of GANs: DC-GANs, which uses Convolutional Networks, and Wasserstein GANs with penalty gradient, which uses another distance that have nice continuity properties for the optimization, combined with a penalty to ensure these properties. DC-GANs provides very satisfying results on simple datasets like MNIST. However, to generate consistent images on more complicate datasets, we needed to implement Wasserstein GANs, in order to get good results. Thanks to them, we could generate art of specific styles. Our work was a challenge because GANs used to make mainly artworks of specific content, and not of specific style. Thus, we think that we obtained very satisfying results given the difficulty of the initial problem.

Future work could be to increase the size of images to get a better resolution. Beyond the issue of finding computational power, increase the size is not easy because it leads to more parameters to optimize and is then even harder for the GAN to converge. A possible path to explore would be to try to adapt transfer learning to our problem, as proposed recently in [5].

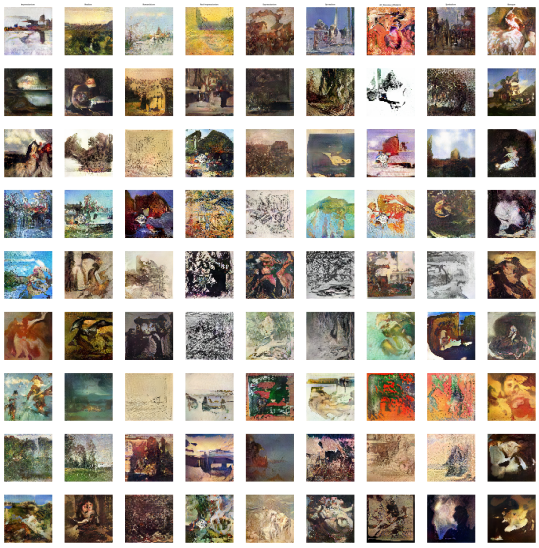

Fig. 11: WGAN trained on Wikiart : mixing matrix : input asked is the superposition of 2 art styles

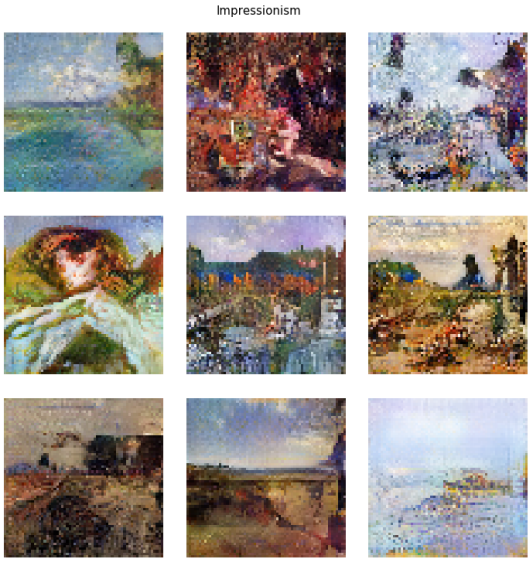

Fig. 12: WGAN trained on Wikiart : different outputs for impressionism

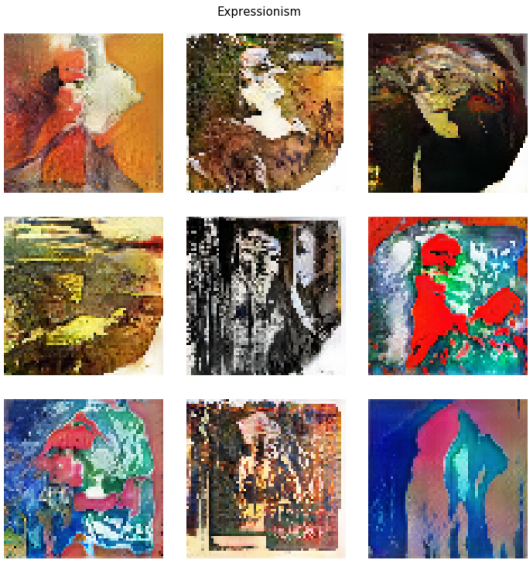

Fig. 13: WGAN trained on Wikiart : different outputs for expressionism

Fig. 14: WGAN trained on Wikiart : different outputs for romanticism. Some images are black and white, white is coherent with black and white engravings present in the dataset

REFERENCES

[1] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan. 2017.

[2] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. 2014.

[3] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville. Improved training of wasserstein gans. In *Advances in Neural Information Processing Systems*, pages 5767–5777, 2017.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. 2015.

Fig. 15: WGAN trained on Wikiart : different style outputs for the same noise input

[5] T. Karras, S. Laine, and T. Aila. A style-based generator architecture for generative adversarial networks. 2018.

[6] D. P. Kingma and J. Lei Ba. Adam: A method for stochastic optimization. 2015.

[7] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. 2016.