Project report

Optimal Transport project: OT-GAN implementation

Author :
Charles Roux

Course taught at ENSAE by :
Marco Cuturi

Contents

1 Introduction

1.1 Introduction

Generative adversarial networks -or GANs- are generative models that generate random examples of mostly high-dimensional data, such as images. The idea is that, given a training dataset (even unlabelled), we want train a model such that the distribution of the generated samples will be as close as possible to the distribution of the real training data. This is done by an architecture relying on two adversarial networks: the generator and the discriminator. Our interest lies in the generator, which generates (for example) an image from a noise vector z, and is thus similar to a decoder. However, to make the generated image look like a real image from our training dataset, we need to train the discriminator (or critic) to identify images as being fake or real. To achieve this, both networks are trained at the same time, and try to foul each other.

GANs were first introduced by Ian Goodfellow et al. [1] in 2014, but since then many variations have been proposed, such as DCGAN [2] or Wasserstein GAN [3]. As stated previously, an ideal generator would produce samples with the same distribution as the one of real data. The critic allows this by measuring the distance between those two distributions. This is where optimal transport (OT) comes into play, since OT metrics allows the comparison between distributions with non-overlapping supports.

For this project, we are interested in OT-GAN [4], which is obviously a GAN relying on optimal transport. Proposed by Tim Salimans et al. in 2018, they consider the primal formulation of OT, instead of the dual formulation, used in Wasserstein GAN [3]. The latter relies on the dual formulation for better tractability, when computing the Earth-Mover distance (also called Wasserstein-1 distance). But it also requires some assumptions. Prior to the work of Tim Salimans et al., Genevay et al. [5] considered the primal formulation instead of the dual one. By introducing the Sinkhorn distance [6], they proposed a new distance between two batches of data, which is reused in the studied article. OT-GAN also draws inspiration from Cramer GAN [7] and its Cramer distance.

1.2 Theoritical motivation of OT-GAN: the Mini-batch Energy Distance

Firstly, one can consider OT-GAN as the combination of the Sinkhorn AutoDiff [5] and Cramer GAN [7]. Indeed, it relies on a new distance for the critic, which they call mini-batch energy distance. Starting from the Cramer distance, they first consider mini-batches instead of single samples ([5], [8]). Then, they generalize it to any distance function d, and obtain the following formulation:

$$D_{\text{Generalized Energy Distance}}(p, g) = \sqrt{2\mathbb{E}[d(X, Y)] - \mathbb{E}[d(X, X')] - \mathbb{E}[d(Y, Y')]}$$

where in practice $X, X' \sim p$ will be independently sampled from real data, and $Y, Y' \sim g$ will be generated independently. Then, by plug-in the Sinkhorn distance introduced in [6] for mini-batches, and by taking the square for practicality, the mini-batch energy distance (MED) is given by:

$$D_{\text{Mini-batch Energy Distance}}(p, g)^2 = 2\mathbb{E}[W_c(X, Y)] - \mathbb{E}[W_c(X, X')] - \mathbb{E}[W_c(Y, Y')]$$

By incorporating the primal form of OT in the metric (through W_c), they claim that the resulting OT-GAN uses a more discriminative critic, resulting in a more stable generator.

Remark: a study of the theoretical foundations would have been quite instructive. Some interesting aspects (the biased gradient problem, the fact that the resulting mini-batch gradients are unbiased,...) have only been briefly mentioned. However, since it was not the core of the project, and due to time constraints, I only summarized the crucial theoretical aspects to have an overall understanding of the publication.

1.3 Outline of OT-GAN

In this subsection, we explain how the mini-batch energy is incorporated into a GAN architecture to obtain the OT-GAN.

First of all, we can take a look at the overall algorithm and architecture, presented below:

Algorithm 1 Optimal Transport GAN (OT-GAN) training algorithm with step size α, using mini-batch SGD for simplicity

```
Require: $n_{gen}$, the number of iterations of the generator per critic iteration
Require: $\eta_0$, initial critic parameters. $\theta_0$, initial generator parameters
1: for $t = 1$ to $N$ do
2:     Sample $\mathbf{X}, \mathbf{X}'$ two independent mini-batches from real data, and $\mathbf{Y}, \mathbf{Y}'$ two independent mini-batches from the generated samples
3:     $\mathcal{L} = \mathcal{W}_c(\mathbf{X}, \mathbf{Y}) + \mathcal{W}_c(\mathbf{X}, \mathbf{Y}') + \mathcal{W}_c(\mathbf{X}', \mathbf{Y}) + \mathcal{W}_c(\mathbf{X}', \mathbf{Y}') - 2\,\mathcal{W}_c(\mathbf{X}, \mathbf{X}') - 2\,\mathcal{W}_c(\mathbf{Y}, \mathbf{Y}')$
4:     if $t \bmod n_{gen} + 1 = 0$ then
5:         $\eta \leftarrow \eta + \alpha \cdot \nabla_\eta \mathcal{L}$
6:     else
7:         $\theta \leftarrow \theta - \alpha \cdot \nabla_\theta \mathcal{L}$
8:     end if
9: end for
```

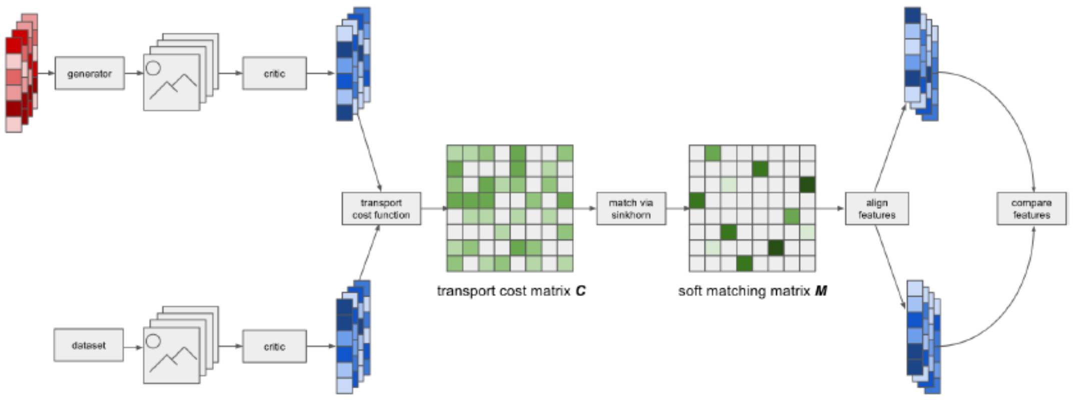

Figure 1: Overall architecture of OT-GAN, and its pseudo-code.

Looking at the illustration, OT-GAN has a critic and a generator -like any GAN-, which are deep neural networks that we want to optimize by gradient descent (ascent for the former). We then feed noise vectors to the generator to obtain fake images, and we pass real and generated images to the critic. If the first steps are usual steps for a GAN, OT-GAN then differs from a classical GAN. First, the critic returns vectors in a latent space instead of scalars indicating the probability for each sample to be fake or real (such as in DCGAN for instance). These features are then used to compute a cost matrix C, and a soft-matching matrix M. We finally "align features" and "compare them". Of course, the last major difference is the use of 2 real and fake mini-batches, due to the formulation of the Mini-batch Energy Distance. Let us clarify these differences.

To understand why we compute cost and matching matrices, we need to detail an aspect previously overlooked: the computation of W_c. W_c is defined as:

$$W_c(X,Y) = \inf_{M \in \mathcal{M}} \mathrm{Tr}[MC^T]$$

where we recover our matching and cost matrices. Now, to obtain C, we have to specify a cost function c, which is here chosen as the cosine distance:

$$c_\eta(x,y) = 1 - \frac{v_\eta(x) v_\eta(y)}{||v_\eta(x)||_2 ||v_\eta(y)||_2}$$

Here, we denote by $v_\eta(x)$ the representation of a (real) sample x into the latent space generated by the critic. So we now also see why the critic returns vectors instead of scalars. So to obtain C, we calculate the cosine distance between the vectors of the two mini-batches. For the matching matrix, we use the Sinkhorn algorithm that we have seen in the course (also described in [5]). The matrix M obtained is defined as

$$M_\gamma = \arg\min_{M \in \mathcal{M}} < M, C > -\gamma E(M)$$

where an entropic penalty term appears. γ is a regularization parameter for the regularized Wasserstein distance,

$$W_\gamma(\mu,\nu) = \min_{M \in \mathcal{M}} < M, C > -\gamma E(M)$$

and corresponds to the entropy penalty of alignments described in Annex B. We invite the reader to directly check the code to see how the Sinkhorn algorithm works.

Now that we know how we compute C and M, we shall clarify the part about "aligning features and comparing them". Actually, this is just about the computation of the loss, which is the squared Mini-batch Energy Distance. This explains the previous computation of C and M, to obtain W_c. Still, looking at the pseudo-code, the loss is slightly different of the EMD:

$$\mathcal{L} = W_c(X,Y) + W_c(X,Y') + W_c(X',Y) + W_c(X',Y') - 2W_c(X,X') - 2W_c(Y,Y')$$

We recover $\mathcal{L}$ by multiplying the EMD by 2 (because there are 2 $\times$ 2 mini-batches) and using that $\mathbb{E}[W_c(X,Y)] = \mathbb{E}[W_c(X',Y)]$ since X, X' are iid (same idea for Y, Y').

The last point we need to mention about OT-GAN is that we do not update the critic as often as we update the generator. This is to avoid that the critic becomes degenerate, which would allow the generator to take advantage of this, leading to poorly generated images.

In the next sections, we will try to code the OT-GAN, and we will challenge some of the statements presented in the article.

2 Implementation

In this section, we will mostly comment the code and some choices made during the implementation.

2.1 OT-GAN

2.1.1 Generator

The OT Generator class allows us to build a generator following the architecture provided Annex B (see table bellow). It should be able to generate images of any size and channel number.

Layers	**Parameters**	**Output dimension**	**Default values**
Input	-	(B, dim_z)	(B, 100)
Linear, GLU	-	(B, 8×n_features_G×4×4)	(B, 16384)
Reshape	-	(B, 8×n_features_G, 4, 4)	(B, 1024, 4, 4)
Upsample	Factor: 2 Nearest-neighbor	(B, 8×n_features_G, 8, 8)	(B, 1024, 8, 8)
Conv, GLU	Kernel: 5×5 Stride: 1	(B, 4×n_features_G, 8, 8)	(B, 512, 8, 8)
Upsample	Factor: 2 Nearest-neighbor	(B, 4×n_features_G, 16, 16)	(B, 512, 16, 16)
Conv, GLU	Kernel: 5×5 Stride: 1	(B, 2×n_features_G, 16, 16)	(B, 256, 16, 16)
Upsample	Size: img[1], img[2] Nearest-neighbor	(B, 2×n_features_G, img[1], img[2])	(B, 256, 32, 32)
Conv, GLU	Kernel: 5×5 Stride: 1	(B, n_features_G, img[1], img[2])	(B, 128, 32, 32)
Conv, Tanh.	Kernel: 5×5 Stride: 1	(B, channels, img[1], img[2])	(B, 3, 32, 32)

Table 1: Our architecture of the OT-GAN Generator (B stands for mini-batch size, img[1] and img[2] for the height and the width)

We performed some changes to reuse the architecture for different image sizes. Another solution would have been to resize the input images to 32x32 pixels, but we preferred the flexibility of our solution. It only requires a slight change in the last up-sampling layer, which has no parameters to learn, so the overall architecture should remain the same.

In terms of parameters, these are the *image shape* (channels, height, width), *dim z* the dimension of the noise vector from which we generate images, and finally *n features G* which allows us to scale a bit the architecture. By default, the latter is set to 128 (to match the original OT-GAN described for CIFAR-10), but we could assume that for simpler images, a lower dimension could be sufficient, while speeding up the training.

The class also incorporates a function to load the weights of a previously trained generator. If no weight path is provided, it will call another function, which performs some weight initialization. This initialization is not mentioned in the article, so we re-used the one proposed in a

Pytorch tutorial for DCGAN, from which the studied architecture is inspired.

Also, one small note about the GLU activation, which is relatively unusual. It is formulated as:

$$GLU(a, b) = a \otimes \sigma(b)$$

where σ is the usual sigmoid function. In the Pytorch implementation, we only pass a single x as input, which is then split into half to replace a and b.

Finally, we reuse the optimizer proposed Annex B: Adam with a learning rate of $3e^{-4}$, $\beta_1 = 0.5$ and $\beta_2 = 0.999$

2.1.2 Critic

The OT Critic class allows us to build a critic/discirminator following the architecture provided Annex B (see table bellow). It should be able to work with most image sizes and any number of channel.

Layers	**Parameters**	**Output dimension**	**Default values**
Input	-	(B, channels, img[1], img[2])	(B, 3, 32, 32)
Conv., CReLU	Kernel: 5×5 Stride: 1	(B, 2×n_features_C, img[1], img[2])	(B, 256, 32, 32)
Conv., CReLU	Kernel: 5×5 Stride: 2	(B, 4×n_features_C, img[1]/2, img[2]/2)	(B, 512, 16, 16)
Conv., CReLU	Kernel: 5×5 Stride: 2	(B, 8×n_features_C, img[1]/4, img[2]/4)	(B, 1024, 8, 8)
Conv., CReLU	Kernel: 5×5 Stride: 2	(B, 8×n_features_C, img[1]/8, img[2]/8)	(B, 2048, 4, 4)
Reshape, L2	-	(B, 8×n_features_C×img[1]/8×img[2]/8)	(B, 32768)

Table 2: Our architecture of the OT-GAN Critic (B stands for mini-batch size, img[1] and img[2] for the height and the width)

Inspiring ourselves from the modified generator, the critic should work with different image shape. However this time, the architecture is the same as the one described in the article. There are only 2 parameters that we have already mentioned for the generator: the *image shape* (channels, height, width), and *n features C* which allows us to scale a bit the architecture (also set to 128 by default). We use the same optimizer as the generator.

This time we use a CReLU activation, which is also relatively unusual. The expression is the following:

$$CReLU(x) = [ReLU(x), ReLU(-x)]$$

There was seamingly no Pytorch implementation, so we created our own CReLU class.

2.1.3 OT-GAN

The OT GAN class allows us to build the OT-GAN following the architecture described in the article and explicated previously. To build it, we need to provide an instance of a generator, critic, and data loader (described later). Also, you can choose another update ratio for the critic than the default one (4). The computation of the cost, the sampling of the noise z, and the Sinkhorn algorithm are performed by their respective methods in the class.

We shall notice that for the Sinkhorn algorithm, we implemented a version allowing us to do less than *max iter* iterations if we are close to a fixed point (stopping criterion controlled by *delta*). This version may be more efficient (in practice, the Sinkhorn algorithm required *max iter*iterations only half of the time). Also, a hyper-parameter of the algorithm is the entropy regularisation term *ent pen*. In the paper, they propose a value of 500, but in practice, the results seemed to be less satisfactory than the ones obtained with 100. So we chose 100 as the default value.

In the cost function, we do not perform the L2 normalization, since it has been already done in the last layer of the critic.

Looking at the train function, the steps should be quite clear given the previous explanations. Nonetheless, there is one small change in the computation of W_c: we compute the sum of an element-wise product instead of $\mathrm{Tr}[MC^T]$. At first, we did this naturally since it was how we computed the cost of the optimal plan during the labs. But we can manually check that these two methods give the same result in our case (for example with a 3x3 matrix). Moreover, the element-wise multiplication is slightly faster.

2.2 DCGAN

To have a better insight into the performances of OT-GAN, it is useful to compare it with another baseline: the DCGAN. Indeed, we chose to compare it to the DCGAN since OT-GAN draws inspiration from it for its generator and critic. Their results are also directly compared in the article after many epochs, so ideally we would need a DCGAN to check their assertions.

The implementation has been done by directly referring to [2], and we used the Pytorch tutorial on DCGAN to double-check our implementation and complement some unclear aspects.

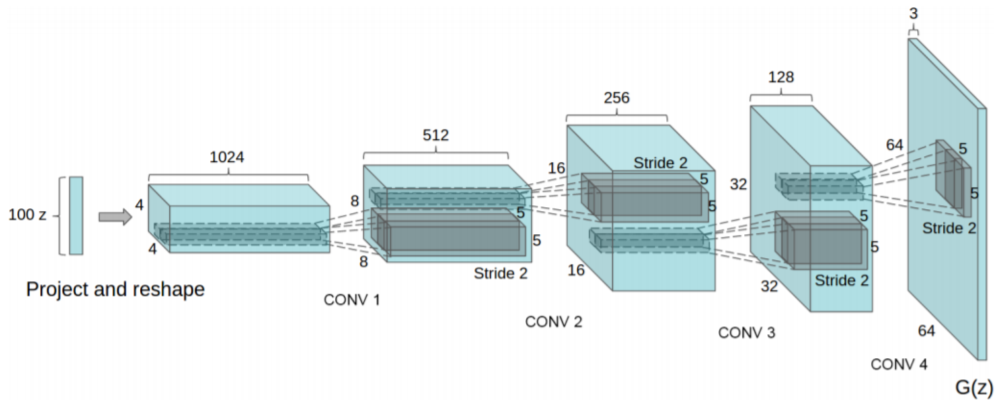

Figure 2: Generator of DCGAN.

2.2.1 Generator

The *DCGAN Generator* class has the same properties as the *OT Generator* class. We will just present here its architecture, which is different. Especially, there is no dense layer, and it uses batch normalization layers.

Layers	**Parameters**	**Output dimension**	**Default values**
Input	-	(B, dim_z)	(B, 100)
ConvTr+BN+ReLU	Kernel: 4×4 Stride: 1	(B, 8×n_features_G, 4, 4)	(B, 1024, 4, 4)
ConvTr+BN+ReLU	Kernel: 4×4 Stride: 2	(B, 4×n_features_G, 8, 8)	(B, 512, 8, 8)
ConvTr+BN+ReLU	Kernel: 4×4 Stride: 2	(B, 2×n_features_G, 16, 16)	(B, 256, 16, 16)
ConvTr+BN+ReLU	Kernel: 4×4 Stride: 2	(B, n_features_G, 32, 32)	(B, 128, 32, 32)
ConvTr	Kernel: 4×4 Stride: 2	(B, channels, 64, 64)	(B, channels, 64, 64)
Upsample	img[1], img[2]	(B, channels, img[1], img[2])	(B, channels, 64, 64)

Table 3: Architecture of our DCGAN Generator

We also added an upsample layer for compatibility with other image shapes (described for 64x64 pixels images). Also, one major difference between DCGAN and OT-GAN is the upsampling method. The Upsampling used in OT-GAN is just a simple scaling-up of the image (here: using nearest neighbour upsampling). ConvTranspose is a convolution, with a kernel learnt during training. Of course, the former is faster. Also, Upsampling + Convolution is not equivalent to ConvTranspose.

2.2.2 Critic

The *DCGAN Critic* class has the same properties as the *OT Critic* class. Once again, the following table summarizes the architecture of the discriminator:

Layers	**Parameters**	**Output dimension**	**Default values**
Input	-	(B, channels, img[1], img[2])	(B, 3, 64, 64)
Upsample	(64, 64)	(B, channels, 64, 64)	(B, 3, 64, 64)
Conv +LeakyReLU	Kernel: 4×4 Stride: 1	(B, n_features_C, 32, 32)	(B, 128, 4, 4)
Conv+BN +LeakyReLU	Kernel: 4×4 Stride: 1	(B, 2×n_features_C, 16, 16)	(B, 256, 16, 16)
Conv+BN +LeakyReLU	Kernel: 4×4 Stride: 1	(B, 4×n_features_C, 8, 8)	(B, 512, 8, 8)
Conv+BN +LeakyReLU	Kernel: 4×4 Stride: 1	(B, 8×n_features_C, 4, 4)	(B, 1024, 4, 4)
Conv +Sigmoid	Kernel: 4×4 Padding: 1 Stride: 0	(B, 1, 1, 1)	(B, 1, 1, 1)

Table 4: Architecture of our DCGAN Critic

To allow the use of different image sizes, we start by an upsampling layer (instead of doing a resize directly on images). Also, unless stated, the padding is set to 2 (since the kernel size is 4) and the leaking parameter in the LeakyReLU is set to 0.2.

2.2.3 DCGAN

The *DCGAN* class has the same properties as the *OTGAN* class.

This time, there are two different losses for the critic and the generator. First, for the discriminator, we aim at maximizing the probability of correctly classifying an input as fake or real. So we want to maximize the following quantity:

$$log(C(x)) + log(1 - C(G(z)))$$

This is actually the opposite of the binary cross-entropy $-y_n log(x_n) - (1 - y_n) log(1 - x_n)$, with $y_n = 1$ and $x_n = C(x)$ when the image x is real, and $y_n = 0$ and $x_n = C(x)$ otherwise (since x is fake, it has been generated: $x = G(z)$). In practice, we compute the first part of the BCE on a real batch, and the second part on a fake batch. For the generator, we wish to minimize

$$log(1 - C(G(z)))$$

(we can see here why the discriminator and the critic are competing with each other). in practice, we instead maximize

$$log(C(G(z)))$$

apparently for stability issues.

Finally, the parameters of the optimizers are slightly different (the learning rate is actually lower, at $2e^{-4}$). Also, there we update the critic as often as we update the generator.

2.3 Datasets

In this section, we quickly evoke the datasets used, and how we built our data loaders. Concerning the datasets, we will mostly use MNIST for its simplicity. The images have a shape of (1, 28, 28), explaining why we had to performs some changes compared to the original size of (3, 32, 32).

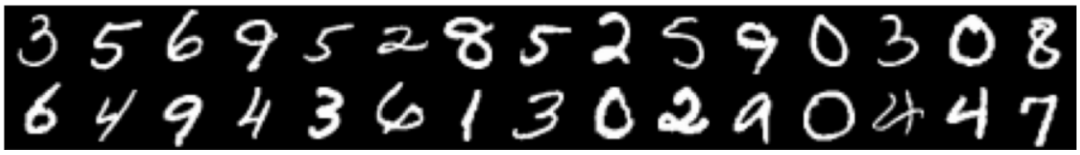
Figure 3: 30 samples from MNIST.

Then, we also tried to generate images using CIFAR-10, which is the dataset presented in the article. However, this dataset is much more challenging, since the content is much more diverse. The images have a shape of (3, 32, 32). For CIFAR-10, we actually only tried to generate one single class at a time.

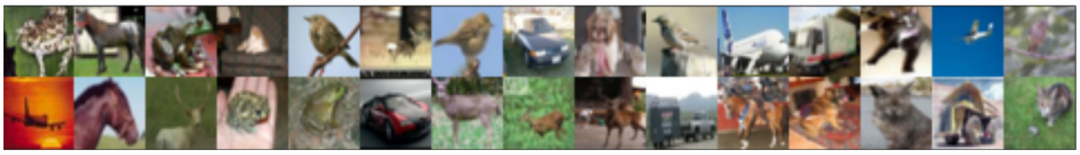
Figure 4: 30 samples from CIFAR-10.

For more flexibility, we provide a *Data Loader* class, which automatically loads one of the two studied datasets, download them if required and transform them. We can also filter it, thus keeping only the desired classes. More importantly, we specify the batch size when instantiating the data loader. Keep in mind that the batch size is the total batch size, so for OT-GAN, the mini-batch size will be two times lower.

3 Empirical study

3.1 Results of OT-GAN

3.1.1 MNIST

Here, we show some results obtained with the whole MNIST dataset. As parameters, we used a batch size of 512 (giving a mini batch size of 256), which was the maximum value we could use, due to memory limitations, on the GPU provided by Google Colab (12Go of memory). The number of epochs was set to 25.

After these 25 epochs, we obtain results similar to the ones displayed bellow:

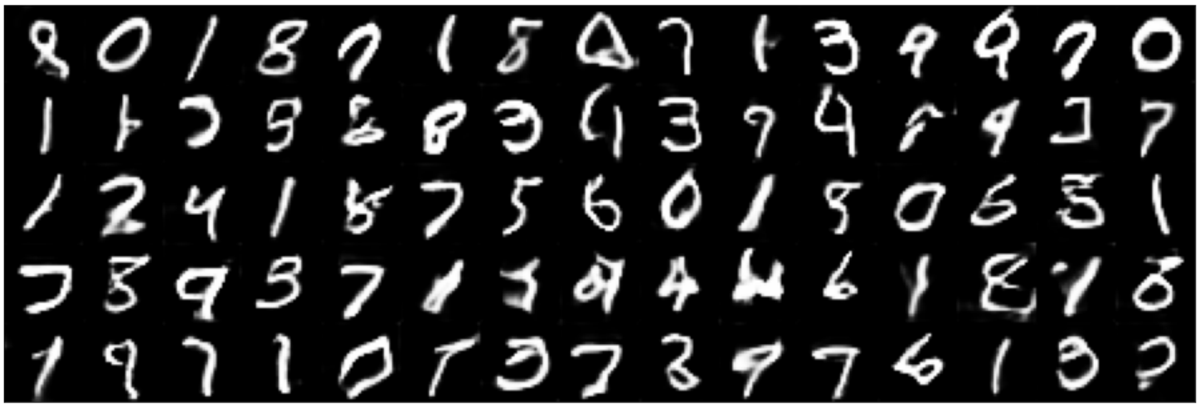

Figure 5: 75 samples generated by the OT generator.

The result is satisfactory. One could have hoped for more "refined" figures: we would clearly identify most of the results as fake images. Still, we can find one instance of each figure which is recognisable. However, there are some figures that we cannot identify.

Also, our results contrast with those presented at the end of the article on CIFAR-10, which is much more complex. Of course, they trained their model on many epochs, with a higher batch size, with several GPUs, and during several days.

It may be interesting to see the evolution of the figures generated, epoch after epoch. Only after 5 epochs, we already start to recognize some figures. The improvements in the following epochs are less visible, especially after 15 epochs.

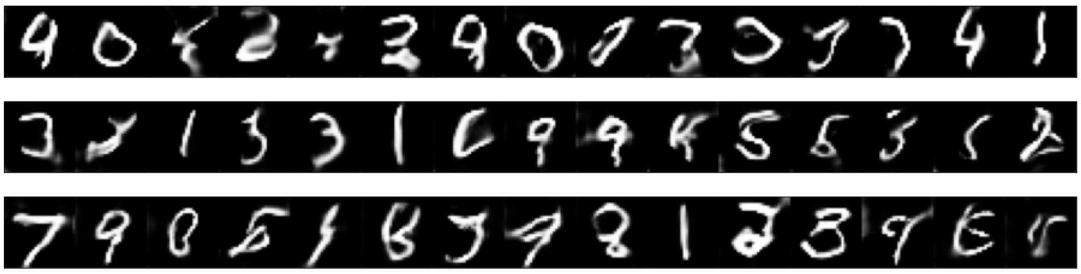

Figure 6: Samples generated by the OT generator at the 10th, 15th and 20th epoch.

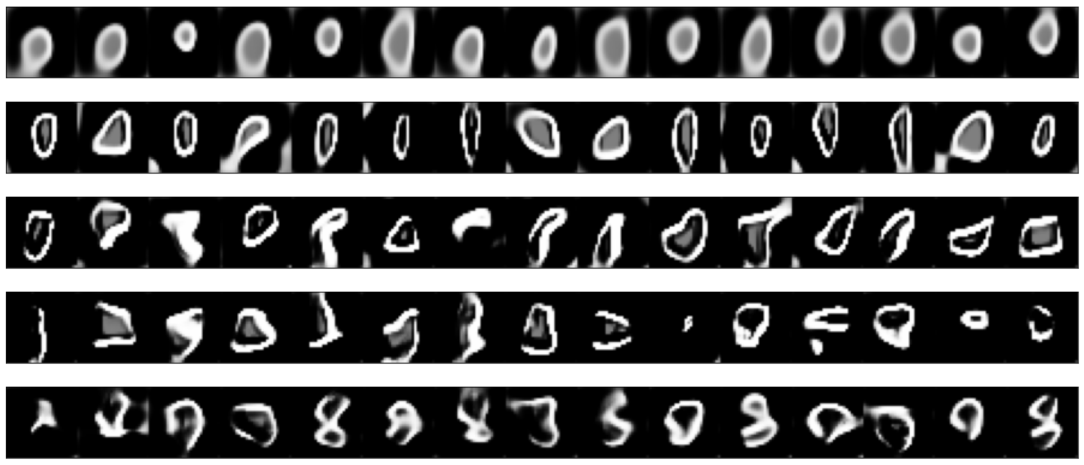
Figure 7: Samples generated by the OT generator from the 1st to the 5th epoch.

Also, to check that the critic did not degenerate, we monitored the W_cs and the loss on the last batch of each epoch. This was intended for debugging at first, and sufficient to check if everything was still working. We also measured the time length of each epoch.

On average, one epoch took about 700 seconds to run, which is slightly more than 11min30s. So it took nearly 5 hours to train on 25 epochs. It is still reasonable, but it restricts our experiments (especially since Colab has a limitation for the usage of its GPUs). The graphs below show that the critic was not degenerated. We can also notice that the loss remains quite low.

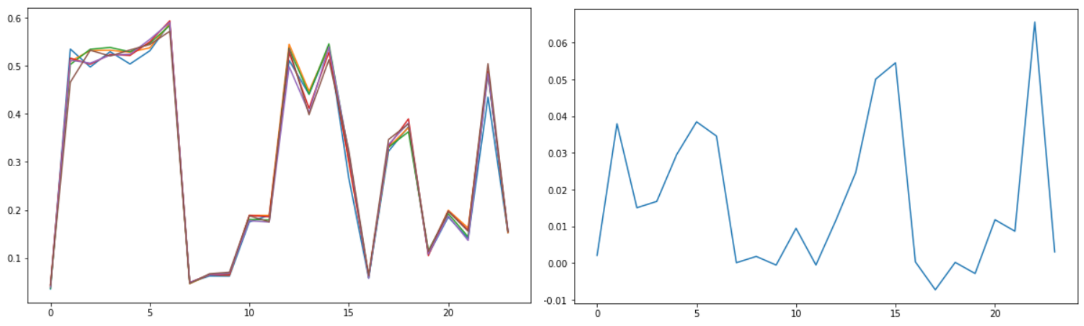
Figure 8: W_cs (top) and loss (bottom) for the last batch of each epoch.

Remark: the version of the code provided should measure the average on the epoch; we could not rerun all the trainings after the changes.

3.1.2 MNIST filtered

Then, we tried to reduce the dataset diversity, hoping that the figures generated would be slightly better for a reasonable number of epochs. We chose to keep only the 2s, since they were poorly generated when we used every figures. To compensate for the lower size of the dataset, we increased the number of epochs to 50. The other parameters remain the same.

As we can see, we can often recognize the 2s, even though the quality is far from being perfect. Finally, the figure bellow illustrates the slower convergence, that we attribute to the lower size of the dataset. Of course, an epoch was 10 times faster than previously.

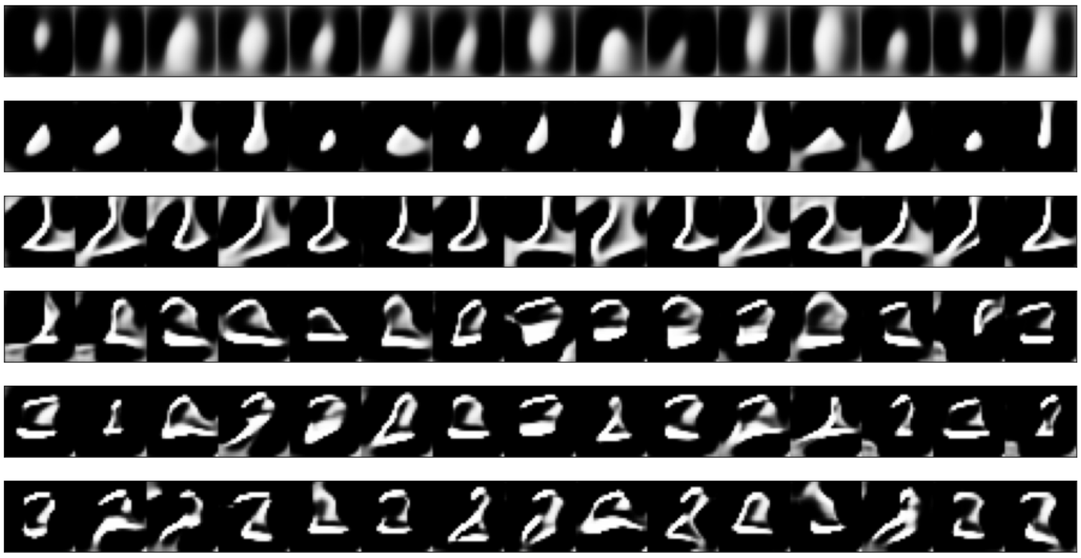

Figure 9: Samples generated every 10 epochs, between the 1st and the 50th epoch.

3.1.3 CIFAR-10 filtered

We actually wanted to visualize the results on CIFAR-10, bearing in mind that we should not have high expectations after the first experiments. To make things easier, we kept only the images of planes, which have a more uniform background. Also, a plane as an approximately "uniform" color. This time, the batch size has to be lower (bigger images, 3 channels instead of 1) to fit in memory: we set it to 128 (mini-batch size of 64).

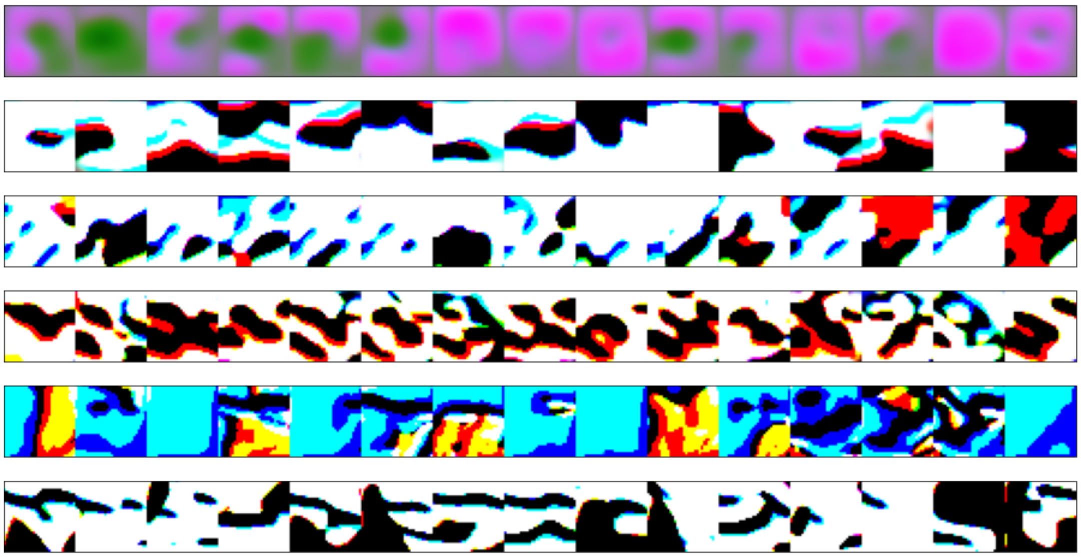

Figure 10: Samples generated every 10 epochs, between the 1st and the 50th epoch.

The results above clearly show that our algorithm was not able to generate an image even close to a real one. At this point, despite the complexity of the image and the relatively low size of the test dataset, we also suspected the influence of the batch size. After all, the results provided in the article stated that the images were better and the generation more consistent with bigger batch size.

3.2 Comparison of OT-GAN with DCGAN

3.2.1 MNIST

The figure below illustrates the results obtain with the DCGAN, with the same parameters as the one used in OT-GAN. We shall note that the batch size will be 512 for DCGAN, compared to a mini-batch size of 256 for OT-GAN. We could have adjusted the batch size of DCGAN to 256, but for a fair comparison, we preferred to keep a batch size of 512: it is a design choice of OT-GAN to use 2 mini-batches.

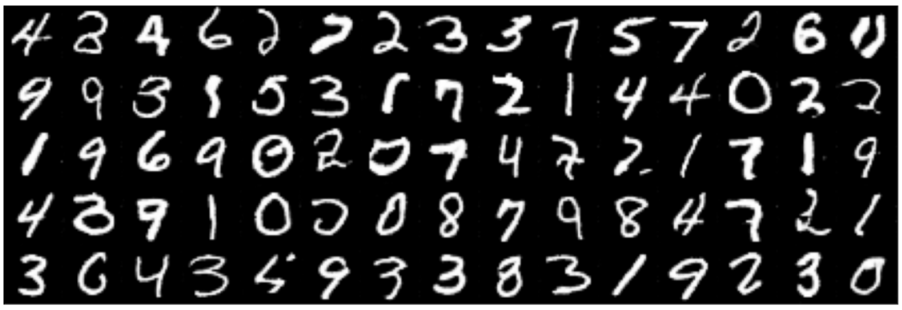
Figure 11: 75 samples generated by the DCGAN generator.

Compared to the OT-GAN, the results are much more convincing. Most of the figures could be real figures. Also, it seems to learn faster than OT-GAN: the results are already acceptable after 5-10 epochs. This may be linked to the higher values of the loss. Previously, the loss on one batch had an order of magnitude of $10e{-2}$. Here, for both critic and generator losses, it is around $10e^0$. As a consequence, the training will be much faster. Additionally, the training was 2 times faster: it only required 6min40s on average to do 1 epoch.

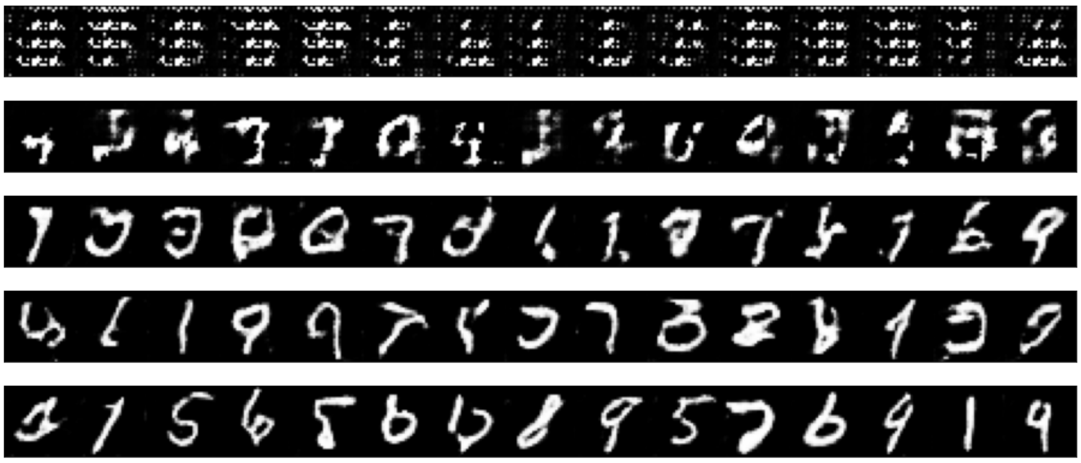
Figure 12: Samples generated by the DCGAN generator from the 1st to the 5th epoch.

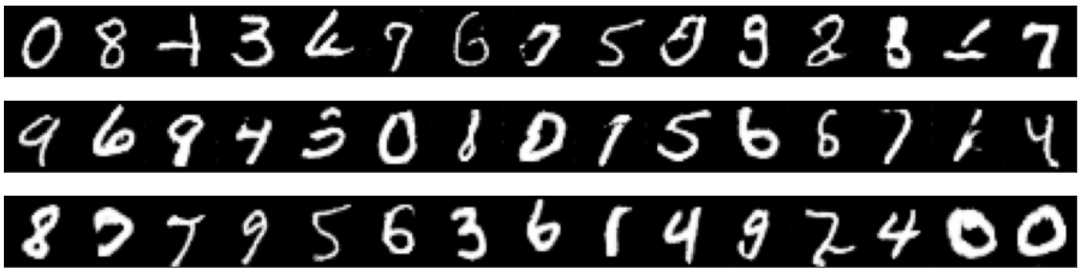

Figure 13: Samples generated by the DCGAN generator at the 10th, 15th and 20th epoch.

3.2.2 MNIST filtered

We also observe that we require more epochs due to the decrease in the dataset size (especially on the first 10 epochs), but the final result is also much more convincing.

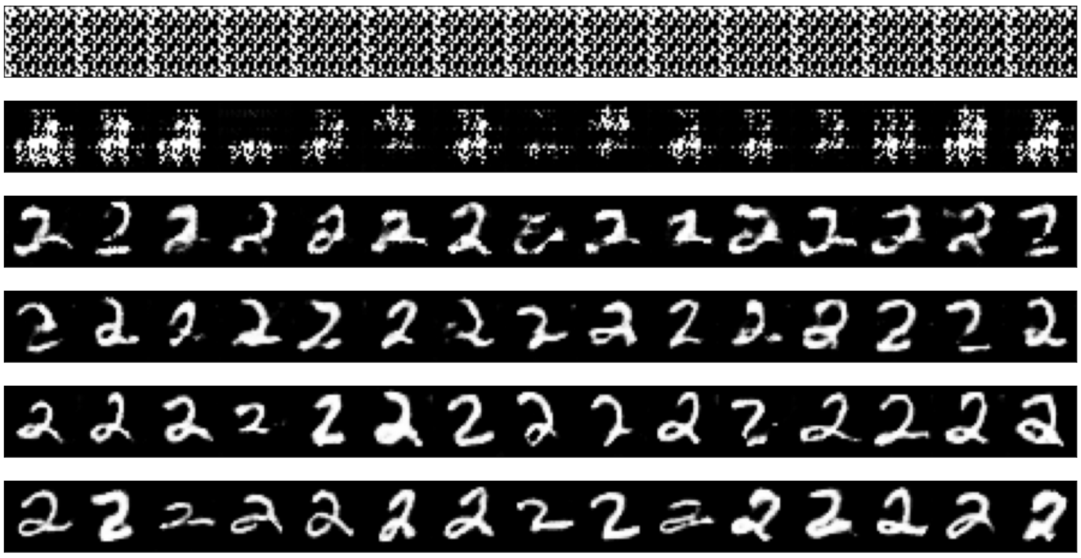

Figure 14: Samples generated every 10 epochs, between the 1st and the 50th epoch.

3.2.3 CIFAR-10 filtered

The final test is CIFAR-10, for which we did not obtain anything with OT-GAN. This time, we finally obtain images that we can interpret as planes after 50 epochs. And even after 30 epochs, the results start to look a bit more credible.

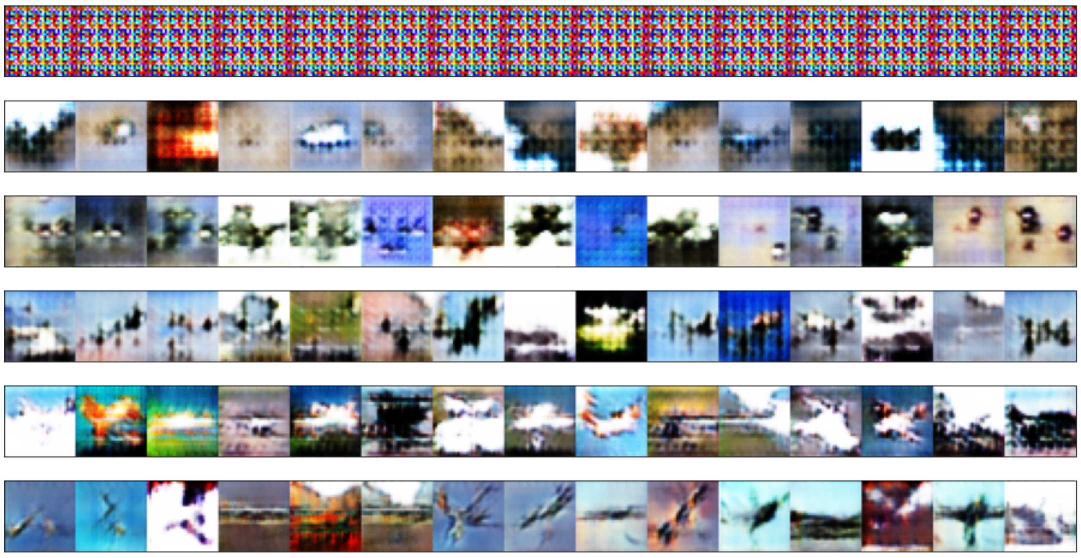

Figure 15: Samples generated every 10 epochs, between the 1st and the 50th epoch.

3.2.4 Some differences linked to the architecture

In this section, we compare the number of parameters between OT-GAN and DCGAN. We previously saw that the loss was much lower for OT-GAN, but the number of parameters could have an additional effect on the slow training of the model. This explain why we have calculated the number of trainable parameters for both models in the tables below.

As we can see, there are 3 times more parameters in OT-GAN than DC-GAN. It explains the higher duration of an epoch and may affect the slow training of the model. Also, we need to keep in mind that the critic is only updated once every 5 batches, and the generator 4 times every 5 batches. One could argue (arbitrarily, this is an intuition) that since the critic is updated so rarely, we need 5 times more epochs to reach the same result like the one obtained with DCGAN.

Layers	**Number of parameters (default)**
Input	0
Linear, GLU	$32.768 \times 100 + 32.768 = 3.309.568$
Reshape	0
Upsample	0
Conv, GLU	$1024 \times 1024 \times 5 \times 5 = 26.214.400$
Upsample	0
Conv, GLU	$512 \times 512 \times 5 \times 5 = 6.553.600$
Upsample	0
Conv, GLU	$256 \times 256 \times 5 \times s\ 5 = 1.638.400$
Conv, Tanh.	$1\ (3) \times 128 \times 5 \times 5 = 3200\ (9600)$
TOTAL	37.719.168 (37.725.568)

Table 5: Trainable parameters of the OT-GAN Generator with MNIST (CIFAR-10)

Layers	**Number of parameters (default)**
Input	0
Conv., CReLU	$128 \times 1\ (3) \times 5 \times 5 = 3200\ (9600)$
Conv., CReLU	$256 \times 256\ (3) \times 5 \times 5 = 1.638.400$
Conv., CReLU	$512 \times 512 \times 5 \times 5 = 6.553.600$
Conv., CReLU	$1024 \times 1024 \times 5 \times 5 = 26.214.400$
Reshape, L2	0
TOTAL	34.409.600 (34.416.000)

Table 6: Trainable parameters of the OT-GAN Critic with MNIST (CIFAR-10)

Layers	**Number of parameters (default)**
Input	0
ConvTr+BN+ReLU	$100 \times 1024 \times 4 \times 4 + 1024 + 1024 = 1.640.448$
ConvTr+BN+ReLU	$1024 \times 512 \times 4 \times 4 + 512 + 512 = 8.389.632$
ConvTr+BN+ReLU	$512 \times 256 \times 4 \times 4 + 256 + 256 = 2.097.664$
ConvTr+BN+ReLU	$256 \times 128 \times 4 \times 4 + 128 + 128 = 524.544$
ConvTr	$128 \times 1\ (3) \times 4 \times 4 = 2048\ (6144)$
Upsample	0
TOTAL	12.654.336 (1.2658.432)

Table 7: Trainable parameters of our DCGAN Generator

Layers	**Number of parameters (default)**
Input	0
Upsample	0
Conv +LeakyReLU	$128 \times 1\ (3) \times 4 \times 4 = 2048$
Conv+BN +LeakyReLU	$256 \times 128 \times 4 \times 4 + 256 + 256 = 524.800$
Conv+BN +LeakyReLU	$512 \times 256 \times 4 \times 4 + 512 + 512 = 2.098.176$
Conv+BN +LeakyReLU	$1024 \times 512 \times 4 \times 4 + 1024 + 1024 = 8.390.656$
Conv +Sigmoid	$1 \times 1024 \times 4 \times 4 = 16.384$
TOTAL	11.032.064 (11.036.160)

Table 8: Trainable parameters of our DCGAN Critic

3.3 Analysis of OT-GAN

In this section, we will present some results we obtained when we changed some of the parameters.

3.3.1 Batch-size

As stated in the article, the batch size is crucial for stability. And as we have seen with CIFAR-10, it may be the major issue with OT-GAN for "personal" usage. To check how important the batch size is, we performed once again the training on MNIST with all the digits, and the same parameters, except that the batch size was put to 32. This may be a bit drastic, since it is 16 times lower than the original batch size, and it means that the mini-batch size is only 16. But the effect is quite clear:

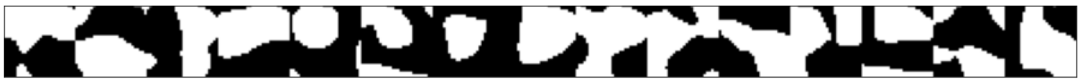

Figure 16: Samples generated after 10 epochs.

We obtain results (at least after 10 epochs) which do not look like at all with the ones obtained in the original conditions. What is also striking is that it looks quite similar to the results obtained with CIFAR-10. Moreover, if we look at the W_c values, they are equal or close to 0 for batches of the same nature (X, X' and Y, Y'), and close to one otherwise. This is true even after the first epoch: it may be the discrepancy of the critic mentioned in the article. We are far from the behaviour observed when the OT-GAN was *a priori* working properly.

Our conclusion is that the batch-size is indeed crucial, and that the results would be certainly better if we could overcome the memory limitation linked to the higher batch sizes.

3.3.2 Generator-critic update ratio

Then, we wanted to test other values for the update ratio, by either increasing it to 1/1, or decreasing it to 1/8 (instead of 1/4). Non-surprisingly, a ratio of 1/1 leads to a critic discrepancy, and we only generate noise. Actually, we do not even generate anything apart from noise. With a ratio of 1/8, there is no discrepancy, the model is just much more slow to train (in terms of number of epochs required). The results after 15 epochs are comparable to those after 3 or 4 epochs in the original setting.

Figure 17: Samples generated after 5 epochs with an update ratio of 1/1.

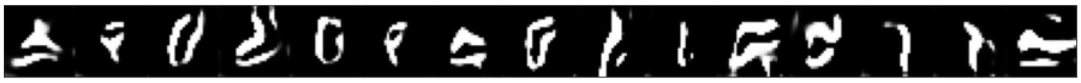

Figure 18: Samples generated after 15 epochs with an update ratio of 1/8.

3.3.3 Sinkhorn

For the Sinkhorn algorithm, we have two parameters we can play with: the entropy penalty, and the maximum number of iterations. We forgot to save the results for a low maximum of iterations, but when we set it to 10, the results were not great at all (expected). We also tried the proposed value of 500, but the gain was not noticeable after about 10 to 20 epochs. Still, the figure bellow gives us the results after 15 epochs, with an entropy penalty of 500 (as suggested in the paper).

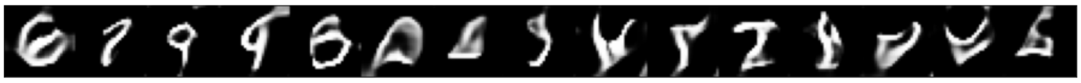
Figure 19: Samples generated after 15 epochs with the suggested entropy penalty of 500.

The results have a worse quality than the originals, after 15 epochs. But the OT-GAN seems to be working. At the beginning, we also tested lower values (between 1 and 100) which gave satisfactory results on the first few epochs. Nonetheless, putting it to values such as $10e{-}2$ completely hindered the learning (we only generated noisy pictures).

3.3.4 Lower Z dimension

The dimension of the noise vector was set to 100 for CIFAR-10, so there was no reason to not decrease it in the case of MNIST (simpler images). We tried a z dimension of 25, which may be a bit low. During our experiment, we were still able to learn and generate images, as we can see bellow. But the process was much slower (in terms of epochs):

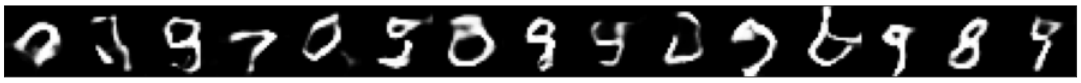
Figure 20: Samples generated after 23 epochs with a Z dimension of 25.

The results are indeed less satisfactory than in the original setting. Also, we forgot to display more images, to see whether the variety of digits generated was impacted (it may be one of the negative consequences). On the few samples of the previous epochs, we would tend to say that this issue is not present for MNIST. Notice that the gain in terms of number of parameters is quite low (about 2 million "only").

3.3.5 Lower generator and critic dimension

The final parameter with which we had the time to "play" was the generator and critic dimension, set to 128 by default. By cutting it to half, we greatly reduce the number of parameters (G: 35.261.568 to 9.029.184 and C: 34.409.600 to 8.603.200), reducing it by a factor 4.

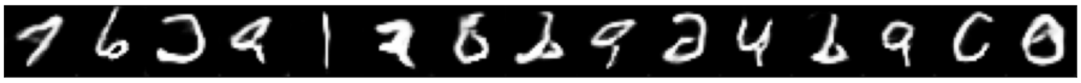
Figure 21: Samples generated after 15 epochs with a generator and critic dimension of 64.

On these few samples, the quality seems to be equivalent. To confirm this, we would need to train it on more epochs. Also, the duration of an epoch was just over 3min, which is much better than before ($\geq$ 11min).

4 Remarks and conclusion

The remarks will evoke mostly what we considered as (often small) unclear aspects or issues, in the article and the implementation.

4.1 About the article

Concerning the article, some clarification could have been made about the Sinkhorn algorithm and the matching process. Of course, we had already seen it during the course, and it is well described in [5]. But a reminder would have made things clearer. Even then, in the illustration of OT-GAN, the part about ”aligning the features” is still unclear to me. Indeed, it seems to be in contradiction with the description: the matching matrix M and the cost matrix C are multiplied (dot product and trace in the article, element-wise then sum in our code) to then compute the W_cs. However, the illustration shows feature vectors, looking like they are the ones obtained after the critic, but modulated according to the matching matrix. Then, the comparison could have been anything. As a consequence, we stuck to the description and the pseudo-code. But this was an issue on which we spent a lot of time.

Remark: I briefly checked the official code provided by OpenAI, and the matching seems to be different from the one I proposed. I would need to double-check this, but there may be a slight difference between the code I implemented, and the ”true” version of OT-GAN (maybe explaining the poorer results).

Other than that, they mention ”weight normalization and data-dependent initialization” which are not specified, but it only limits the reproducibility.

4.2 About the implementation

Many improvements could be added to the code. The pixel-wise cost mentioned could have been tested, to see whether the difference was noticeable after a few epochs only. Computing the inception score would have given a mean to compare generated images other than the 'visual evaluation' we did. The conditional generation is interesting too, and it would allow us to compare the results between digits (for instance, we empirically saw that the 2s had a poorer quality).

Of course, increasing the number of epochs would have been a plus (especially when running the experiments, such as the one with a lower generator and critic dimension). The same goes for the batch size, but this issue is harder to bypass since it requires better hardware. Linked to this issue, the Sinkhorn algorithm is performed on the CPU, slowing down the training. Data augmentation was not evoked, so we decided not to use it. But it would certainly yield better results. Apparently, no weight decay was used either, which is surprising.

4.3 Conclusion

This project was the opportunity to discover yet another surprising application of Optimal Transport, which seems at first far from the problems formulated by Monge and Kantorovich. Even though we were unable to illustrate the advantages of OT-GAN in practice -a higher stability and better sharpness with large batches-, the results were already quite satisfying.

I am looking forward to the upcoming optimal transport applications, for GAN or any other application.

References

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

[2] Alec Radford; Luke Metz; Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. 2016.

[3] Martin Arjovsky; Soumith Chintala; Léon Bottou. Wasserstein gan. 2017.

[4] Tim Salimans; Han Zhang; Alec Radford; Dimitris Metaxas. Improving gans using optimal transport. 2018.

[5] Aude Genevay; Gabriel Peyré; Marco Cuturi. Learning generative models with sinkhorn divergences. 2017.

[6] Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. 2013.

[7] Marc G. Bellemare; Ivo Danihelka; Will Dabney; Shakir Mohamed; Balaji Lakshminarayanan; Stephan Hoyer; Rémi Munos. The cramer distance as a solution to biased wasserstein gradients. 2017.

[8] Tim Salimans; Ian J. Goodfellow; Wojciech Zaremba; Vicki Cheung; Alec Radford; and Xi Chen. Improved techniques for training ganss. 2016.