Deep Learning : Final Project
Generative Adversarial Networks and Cycle-GAN

Maxime Berillon and Rémy Deshayes

April 28th 2021

When studying an unsupervised learning problem, data at hand supposedly follow a distribution $\mathbb{P}_r$ that we want to discover and learn.

Section 1 introduces two approaches to tackle the problem of probability distribution learning and make a brief introduction to a model - the GAN - that leverages one of these approaches. Then, section 2 builds on the remarks made in the previous part and introduces the Wasserstein GAN. Section 3 and section 4 propose an in-depth review of two GANs and their implementations[1], namely the DCGAN and the CycleGAN.

1 Introduction

1.1 Intuitions on learning a probability distribution

1.1.1 The classical maximum likelihood approach

This approach leverages the probability densities. Namely, one defines a parametric family of densities $(P_\theta)_{\theta \in \mathbb{R}^d}$ and finds the density that maximizes the likelihood of the data at hand.
Asymptotically, when the real data distribution admits a density, this approach amounts to minimizing the Kullback-Leibler divergence between $\mathbb{P}_r$ the real data distribution and $\mathbb{P}_\theta$ the distribution of the density P_θ

However, minimizing the KL divergence can be challenging given that in rather common situations the KL divergence can be undefined. This notably is the case when the topological spaces studied are of low dimensions.

In order to make the maximum likelihood approach work and overcome the above complications, the classical trick is to add noise, sometimes a lot - which can be to the detriment of the models' outputs - to the model distribution $\mathbb{P}_\theta$.

1.1.2 Focusing on an alternative approach

These complications triggered the need for another option which resides in avoiding the estimation of the density of the real data distribution.

The method is to define a random variable Z with a fixed distribution $p(z)$ and then use it as an input for a parametric function $g_\theta : \mathcal{Z} \to \mathcal{X}$ that generates samples following a distribution $\mathbb{P}_\theta$.
Varying the parameter θ will allow to change $\mathbb{P}_\theta$ and make it as close as possible to the real distribution.

This approach has two major advantages over the one introduced in 1.1.1 :

1. The potential low dimensions of the spaces is no longer a problem

[1] notebook can be found on GitHub here or on Google Colab here

2. Having samples at hand rather than density values is more convenient in various use-cases

As we will see in part 2.2, GANs, that we are studying in our project, leverage this alternative approach. Let us then briefly introduce the GAN model in part 1.2.

1.2 Brief introduction to the GAN model

Definition : GAN

A Generative Adversarial Network is an unsupervised **generative model** where **two networks compete** with each other in a game theory setting.

The first network is called a **generator**, it generates a sample. Its adversary, the other network, is called a **discriminator** and tries to detect if a sample is genuine or if its an output from the generator. The learning procedure can be modeled as a zero sum game - one model's gain is the other's loss.

The generator and the discriminator are multilayer perceptrons meaning they can be any usual deep learning framework, independently. For instance, later in section 3, we will study in more detail a GAN implementation using CNN that are well suited to image processing and to our use case on the MNIST images dataset.

There is a lot of use-cases for GANs notably in image upsampling, artistic creation and reinforcement learning. Let us use of this use case, namely the art forger, to better understand the GAN's operational principle. A common analogy is that of an art forger - the generator - who tries to forge paintings and an investigator - the discriminator - who tries to detect imitations.

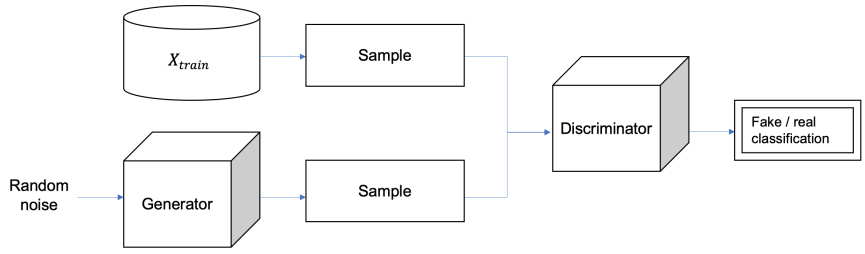


Figure 1: GAN's framework high level view - an interesting takeout from this framework is that the generator has no knowledge of the real data, the only message it will get is the discriminator's output.

That said, training a GAN amounts to a simultaneous 2-players minimax game - the generator (G) tries to fool the discriminator (D) and D tries to avoid being fooled by G.

Let us describe a little bit more this minimax game, this will be useful in later sections and notably in 4.

As D produces a binary fake vs. true classification, the binary-cross entropy is chosen as a cost function :

$$V = -(y \log(D(x)) + (1 - y) \log(1 - (D(x)))$$

where y is a class indicator - 0 for false and 1 for true - and $D(x)$ is the probability to be true output by the discriminator.

Let us denote $\hat{x}$ the false samples, D cost function is as follows :

$$V(D) = -[\log D(x) + \log(1 - D(\hat{x}))]$$

Now, adding G we get:

$$V(D) = -[\log D(x) + \log(1 - D(G(z)))]$$

Eventually, as we are considering multiple observations we end up having :

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

where $z \sim p_z(z)$, the chosen distribution.

D optimization program amounts to minimizing its cost function:

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

For its part, G tries to fool D, its cost function is thus the opposite of D's one:

$$V(G) = -V(D)$$

and its optimization program is given by :

$$\min_G V(G) = \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

Hence our 2-players minimax game between D and G:

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

Now that we know a little bit more about the GAN framework, another thing to know is that training GANs can usually suffer from three issues :

- Non-convergence: the model parameters oscillate, destabilize and never converge
- Mode collapse: the generator collapses which produces limited varieties of samples
- Diminished gradient: the discriminator gets too successful that the generator gradient vanishes and learns nothing

Various architectures and alternatives have been proposed as a workaround to these problems. In this context, section 2 introduces one very famous alternative the Wasserstein GAN.

2 Wasserstein GAN - [1]

In this section, part 2.1 will delve into the details of the method adopted by GANs and introduced in 1.1.2, discussing the notion of distances between two probability distributions. Then, taking advantage of the remarks made in part 2.1, part 2.2 introduces an alternative GAN algorithm - the WGAN - and its practical benefits.

2.1 The alternative approach in practice

2.1.1 Impact of the distance measure

Now, recalling the remarks made in 1.1.2, in order to use this new approach and appropriately train g_θ, one needs to define a distance measure between the model and real data distributions which amounts to a crucial discussion on the various ways possible to define a distance $\rho(\mathbb{P}_\theta, \mathbb{P}_r)$.

Indeed, when trying to optimize the θ parameter it is best to have $\mathbb{P}_\theta$ such that $\theta \mapsto \mathbb{P}_\theta$ is continuous, meaning that $\theta_t \to \theta$ yields $\mathbb{P}_{\theta_t} \to \mathbb{P}_\theta$.
However, let us recall the definition of the convergence of a sequence of a probability distributions. $(\mathbb{P}_t)_{t\in\mathbb{N}}$ converges if and only if there is a distribution $\mathbb{P}_\infty$ such that $\rho(\mathbb{P}_t, \mathbb{P}_\infty) \to 0$, meaning that what we defined a few lines above will come down to the definition of ρ.

In their paper, M. Arjovsky, S. Chintala and L. Bottou focus on four different ways to define a distance and even more importantly they study non-negligible impact on sequences of probability distributions convergence.

The four famous distances considered

- The Total Variation distance :

$$\delta(\mathbb{P}_r, \mathbb{P}_g) = \sup_{A\in\Sigma} |\mathbb{P}_r(A) - \mathbb{P}_g(A)| \tag{TV}$$

where Σ denotes the subsets of a compact metric set

- The Kullback-Leibler divergence :

$$KL(\mathbb{P}_r||\mathbb{P}_g) = \int \log\left(\frac{P_r(x)}{P_g(x)}\right) P_r(x)\mathrm{d}\mu(x) \tag{KL}$$

where $\mathbb{P}_r$ and $\mathbb{P}_g$ admit densities as previously noted in 1.1.1

- The Jensen-Shannon divergence :

$$JS(\mathbb{P}_r, \mathbb{P}_g) = KL(\mathbb{P}_r||\mathbb{P}_m) + KL(\mathbb{P}_g||\mathbb{P}_m) \tag{JS}$$

where $\mathbb{P}_m$ is the mixture $\frac{\mathbb{P}_r+\mathbb{P}_g}{2}$

- The Wasserstein distance :

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

where $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ denotes the set of all joints distributions $\gamma(x, y)$ with marginals $\mathbb{P}_r$ and $\mathbb{P}_g$

2.1.2 Why choose the Wasserstein distance?

Intuitively, if each distribution is viewed as a unit of mass of earth stacked on a metric space, the measure is the minimum cost of turning one pile into the other. This is supposed to be the product between the mass of earth that needs to be moved and the distance it has to be moved.
This is why the Wasserstein distance is also known as the Earth Mover's distance.

The motivation behind the choice of the Wasserstein distance as a loss function over the other distances introduced above has roots in the very desirable properties that (W) has :

1. Under mild assumption over g_θ introduced in 1.1.2, the Wasserstein distance has guarantees of continuity and differentiability which is not the case for the Jensen-Shannon divergence for example
2. Any sequence of probability distribution that would converge under (KL), (JS) and (TV) would also converge under (W), meaning that the Wasserstein distance induces a weaker topology i.e makes it easier for a sequence of distribution to converge

Those properties are especially useful notably in the problematic situation introduced in 1.1.1, namely when the spaces at hand are of low dimensions.
This makes the Wasserstein distance a great and sound choice in that setup.

Now that the choice is made, the question is how to optimize the Wasserstein distance given that (W) is intractable.
The authors will introduce a tractable approximation obtained through the Kantorovich-Rubinstein duality which states that :

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

where the supremum is over all the 1-Lipschitz functions f
Indeed, under the same mild assumptions mentioned earlier over g_θ, (W-KRd) admits a solution - the term becomes a $\max_{||f||_L \leq 1}$ - and :

$$\nabla_\theta W(\mathbb{P}_r, \mathbb{P}_\theta) = -\mathbb{E}_{z\sim p(z)}[\nabla_\theta f(g_\theta(z))]$$

2.2 Wasserstein GAN

2.2.1 The algorithm

Now let us recall the GAN architecture introduced in part 1.2, the GAN's field contribution of M. Arjovsky, S. Chintala and L. Bottou is to propose an alternative GAN based on an alternative way to train the generator to ultimately better approximate $\mathbb{P}_r$.
Indeed, recalling the approach introduced in 2.1, training the generator should result in minimizing a distance. That is why in the Wasserstein GAN, generator updates now centers around the use of a *critic* that outputs a quality score of a distance approximation rather than using strong discriminator-made classification.

More precisely, coming back to the remarks made at the end of 2.1.2, the focus is now on finding an approximation for the function f solving (W-KRd).
To do so, the authors train a neural network with weights w lying in a compact space $\mathcal{W}$ enforced by weights clipping - this is made to ensure that the Lipschitz condition on *critic* functions $(f_w)_{w\in\mathcal{W}}$ is met - and then backprop through $\mathbb{E}_{z\sim p(z)}[\nabla_\theta f_w(g_\theta(z))]$

Having introduce the general idea, let us now have a look at the pseudo-code of the algorithm :

Parameters to be set:

- α, the learning rate
- c, the clipping parameter
- m, the batch size
- n_{critic} , the number of iterations of the critic per generator iteration

Initialisation:

- w_0, initial critic parameters

- θ_0, initial generator's parameters

1: **while** θ has not converged **do**
2: **for** $t = 0, ..., n_{\text{critic}}$ **do**
3: Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
4: Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
5: $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
6: $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
7: $w \leftarrow \text{clip}(w, -c, c)$
8: **end for**
9: Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
10: $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
11: $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
12: **end while**

Figure 2: WGAN algorithm - as introduced earlier, the for loop from line 2 to 8 stands for the *critic*'s training, lines 9 to 11 represents the subsequent generator update

2.2.2 The practical benefits

The WGAN has two main advantages compared to regular GAN's implementations.

1. The loss of the WGAN shows properties of convergence. Indeed, the design of the WGAN - Figure 2 - implies that the loss of the algorithm is an approximation of the Wasserstein distance up to a constant meaning that the loss is directly linked to the quality of the generated samples.

2. The WGAN has a more stable training and supports much more model architecture flexibility.
   Recall part 2.1.2, the Wasserstein distance has guarantees of continuity and differentiability which means that one can train the *critic* until optimality yielding dependable results on the gradient of the Wasserstein thus providing useful gradient information to update the generator and preventing rather frequent GAN's issue where generators fail to produce a reasonable variety of samples - a phenomena called mode collapse.

In their paper [1], M. Arjovsky, S. Chintala and L. Bottou uses another famous implementation of the GAN - the DCGAN - as a baseline to compare and assess the WGAN benefits. The DCGAN is now explored in further details in section 3.

3 Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks - [2]

In this section, in-depth study of the DCGAN is the opportunity to introduce a famous type of GAN with a convolutional architecture and to initiate our implementation work with an image generation task based on the MNIST dataset in part 3.2.

The idea behind the DCGAN is to bridge the gap between very popular and successful supervised computer vision CNN tasks and unsupervised tasks. However, scaling GANs with a CNN architecture is no easy task and has often yielded disappointing results.

The main contribution from A. Radford, L. Metz and S. Chintala is introducing a GAN using a precisely delimited family of CNN architectures that observes specific guidelines - part 3.1 - and gives in practical benefits studied in 3.3.

3.1 Architecture

Let us describe these guidelines.

- Discard fully connected layers on top of convolutional features. The technique used here is based on the Global Average Pooling method which is more native to the convolution structure by enforcing correspondences between feature maps and categories.

For the discriminator :

- Replacing any pooling layers with strided - step size of the kernel when traversing the image - convolutions
- Using Batch Normalization to all layers except the input layer to avoid model instability
- Using LeakyReLU activation for all layer

For the generator :

- Replacing any pooling layers with fractional-strided convolutions - also called transposed convolutions, yielding an all convolutional net
- Using Batch Normalization to all layers except the output layer to avoid model instability
- Using ReLU activation for all layers except for the output, which uses tanh

Now that we have a high level view of the architecture used for the DCGAN, let us delve into more details by implementing it in part 3.2.

3.2 DCGAN Implementation

3.2.1 Discriminator

Regarding the discriminator architecture we decided to start from the components depicted in Figure 3 and add it a twist. Namely, we added a convolutional layer and we increased the number of filters in the convolutional layers to fit with the original implementation.
What's more, we used a LeakyReLU instead of ReLU for its better performance on our use-case - part 3.2.4.

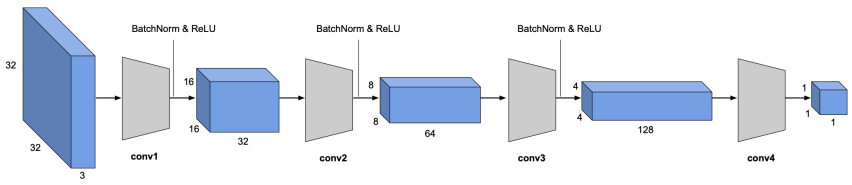


Figure 3: Example of a Discriminator architecture

3.2.2 Generator

We followed a similar procedure, starting from Figure 4, for the discriminator which basically does the opposite job.

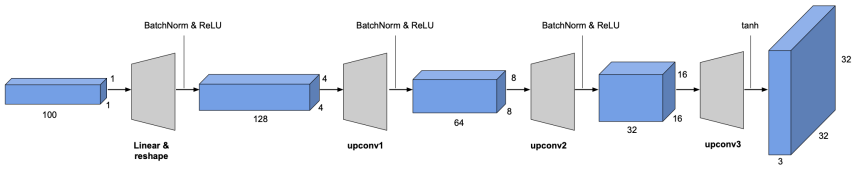


Figure 4: Example of a generator architecture

3.2.3 Training

Now that the framework is set up, we go on with the training. We trained the DCGAN with an Adam optimizer with a learning rate of 10^{-4} and $\beta = 0.5$.

Here is the pseudo-code we used:

DCGAN Training Loop

1: Draw m training examples $\left\{x^{(1)}, \ldots, x^{(m)}\right\}$ from the data distribution p_{data}
2: Draw m noise samples $\left\{z^{(1)}, \ldots, z^{(m)}\right\}$ from the noise distribution p_z
3: Generate fake images from the noise: $G\left(z^{(i)}\right)$ for $i \in \{1, \ldots m\}$
4: Compute the discriminator loss:

$$J^{(D)} = \frac{1}{2m}\sum_{i=1}^{m}\left[\log D\left(x^{(i)}\right)\right] + \frac{1}{2m}\sum_{i=1}^{m}\left[\log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right]$$

5: Generate fake images from the noise: $G\left(z^{(i)}\right)$ for $i \in \{1, \ldots m\}$
6: Compute the generator loss:

$$J^{(G)} = \frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(G\left(z^{(i)}\right)\right)\right]$$

Now let us have a look at the results we got from our implementation on our MNIST image generation use-case.

3.2.4 Results

Figure 5 page 14 shows how the DCGAN generator is learning to generate the MNIST images.

Indeed, nothing precise can be derived from the first epoch but epoch 10 is much more convincing and the generator is able to produce what could be real written numbers.
In the meantime the losses of the discriminator and the generator are decreasing which means the DCGAN is learning properly.

Not only this framework architecture has given great result for our experiment but also great practical benefits in general. We are now reviewing them in part 3.3.

3.3 The practical benefits

Three benefits are usually taken from the implementation of the DCGAN

- The DCGAN is more stable to train in most settings and does not require heavy parameter tuning
- The DCGAN showed very competitive performance with other unsupervised algorithms

- Latent space has some valuable local linear properties allowing easy semantic quality manipulations of the generated sample - this is very similar to NLP vector as in Word2Vec for instance

Next section will be dedicated to another use-case leveraging the GAN machinery, namely Image-to-Image Translation in an unsupervised setting.

4 Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks - [3]

It is very interesting to note that the CycleGAN, as the DCGAN - all things being equal - is trying to bridge the gap between a very successful supervised computer vision tasks and the same unsupervised task.
The motivation behind the need for an unsupervised setting in image translation lies in the fact that paired images are very scarce resources.

The main contribution from Jun-Yan Zhu, Taesung Park, Phillip Isola and Alexei A. Efros is introducing a method - the CycleGAN - that captures the special characteristics of one image collection and translate them to the other image collection without ever requiring paired images. Part 4.1 will introduce the modelling needed to perform such a task and we will delve into details of the implementation in part 4.2.

4.1 CycleGAN modelling

The task that the CycleGAN is tackling is translating an image from a data collection X to a target data collection Y in the absence of paired examples.
Thus, the model aims at learning a mapping functions $G : X \mapsto Y$ which output $\hat{y} = G(x)$ perfectly mimics an image from Y.

Again, as we have seen multiply times, being in an unsupervised setting, this amounts to learning a distribution.
Now, the issue is that mappings G inducing the same probability distribution than Y are not unique which means that an input x might not be matched with a meaningful corresponding y.

A workaround this problem is found using regularization through transitivity, more precisely using cycle consistency.
The concept is illustrated in paper [3]: if a sentence in English is translated into French, the procedure to translate it back into English should produce the exact same initial English sentence.

Introducing a second mapping $F : Y \mapsto X$, the concept is enforced through checking that F and G are indeed inverse functions.

That said we now introduce the corresponding losses that we will then use in our implementation in part 4.2.

4.1.1 Adversarial losses

Recalling, part 1.2 and the introduction on GANs, here we have two traditional GAN losses.

One of them is thus given by :

$$\mathcal{L}_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{data}(y)}\big[\log D_Y(y)\big] + \mathbb{E}_{x \sim p_{data}(x)}\big[\log(1 - D_Y(G(x)))\big]$$

where D_Y is the discriminator for the generator G

4.1.2 Cycle Consistency loss

The Cycle Consistency loss is given by :

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_{x \sim p_{data}(x)}\left[||F(G(x) - x||_1\right] + \mathbb{E}_{y \sim p_{data}(y)}\left[||G(F(y) - y||_1\right]$$

The full objective is the sum of those three losses controlled with a parameter λ.‘

Now that we have a deeper knowledge of the CycleGAN framework, let us delve into more details by implementing it in part 4.2

4.2 CycleGAN Implementation

4.2.1 Blocks and Generators

The two generators of our CycleGAN are made of different blocks:

- First the `downsample` block that will *encode* the input using a convolutional layer ;
- Then the `residual` block that transforms the input but keeps the same output size ;
- Finally the `upsample` that does the opposite job and *decodes* the input with a transpose convolutional layer.

We decided to start from the components depicted in Figure 6 and added 2 `downsample` blocks, 9 `residual` blocks and 2 `upsample` blocks.

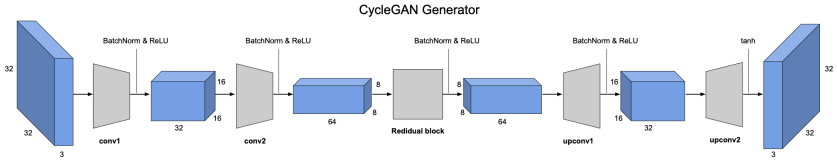


Figure 6: Example of a generator architecture for a cycleGAN

4.2.2 Discriminators

Regarding the discriminators we followed the same procedure as for the DCGAN.

4.2.3 Training

Regarding training we used batch sizes of 1. Instead of the Binary Cross Entropy Loss we decided to use the Mean Squared Error who showed much better results.
The cycle loss was implemented with the Mean Absolute Error.

Recalling part 4.1, here is the pseudo-code we used :

CycleGAN Training Loop

1: Draw m training examples $\left\{x^{(1)}, \ldots, x^{(m)}\right\}$ from domain X
2: Draw m training examples $\left\{y^{(1)}, \ldots, y^{(m)}\right\}$ from domain Y
3: Compute the D_X discriminators loss:

$$J^{(D_X)} = \frac{1}{m} \sum_{i=1}^{m} \left(D_X\left(x^{(i)}\right) - 1\right)^2 + \frac{1}{n} \sum_{j=1}^{n} \left(D_X\left(G_{Y \to X}\left(y^{(j)}\right)\right)\right)^2$$

4: Compute the D_Y discriminators loss:

$$J^{(D_Y)} = \frac{1}{m} \sum_{i=1}^{m} \left(D_Y\left(x^{(i)}\right) - 1\right)^2 + \frac{1}{n} \sum_{j=1}^{n} \left(D_Y\left(G_{X \to Y}\left(y^{(j)}\right)\right)\right)^2$$

6: Update the discriminators
7: Compute the $Y \to X$ generator loss:

$$\mathcal{J}^{(G_{Y \to X})} = \frac{1}{n} \sum_{j=1}^{n} \left(D_X\left(G_{Y \to X}\left(y^{(j)}\right)\right) - 1\right)^2 + \lambda_{\text{cycle}}\, \mathcal{J}_{\text{cycle}}^{(Y \to X \to Y)}$$

8: Compute the $X \to Y$ generator loss:

$$\mathcal{J}^{(G_{X \to Y})} = \frac{1}{m} \sum_{i=1}^{m} \left(D_Y\left(G_{X \to Y}\left(x^{(i)}\right)\right) - 1\right)^2 + \lambda_{\text{cycle}}\, \mathcal{J}_{\text{cycle}}^{(X \to Y \to X)}$$

9: Update the generators

4.3 Results

Figure 7 page 14 shows the CycleGAN learning evolution through epochs.

The main challenge to translate images from USPS to MNIST lies in the "font" of the numbers.
The MNIST dataset numbers are rather thin while they are larger for the USPS dataset. Also, USPS number are a slightly blurrier than the MNIST ones.

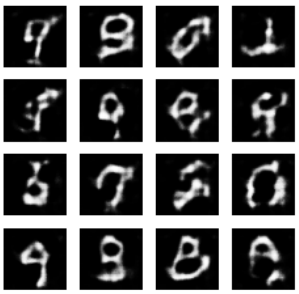
The CycleGAN has clearly adapted to these differences. During epoch 12 it took the MNIST images and enlarged the lines while for the USPS image it reduced the line and made it clearer.

References

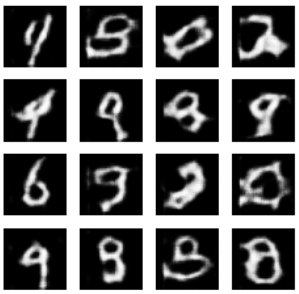
[1] Martin Arjovsky, Soumith Chintala and Léon Bottou. Wasserstein GAN. In *International conference on machine learning*, 2017.

[2] Alec Radford, Luke Metz and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In *arXiv 1511.06434*, 2016.

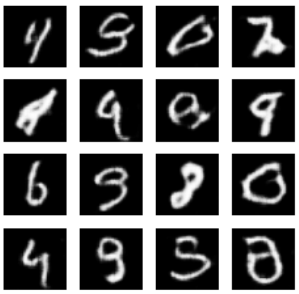
[3] Jun-Yan Zhu, Taesung Park, Phillip Isola and Alexei A. Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In *Proceedings of the IEEE international conference on computer vision*, 2017.



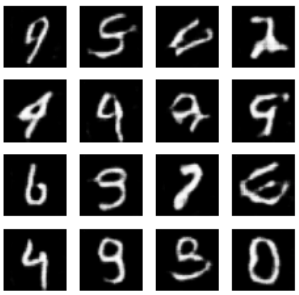
(a) Epoch 1



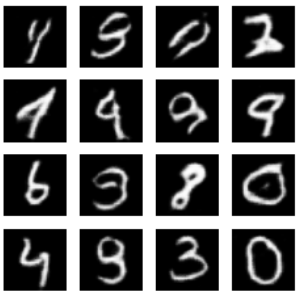
(b) Epoch 2



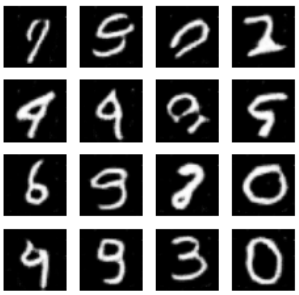
(c) Epoch 3



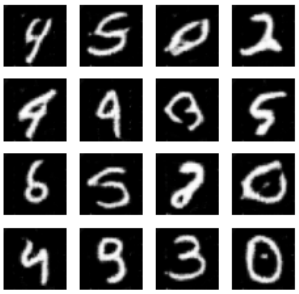
(d) Epoch 4



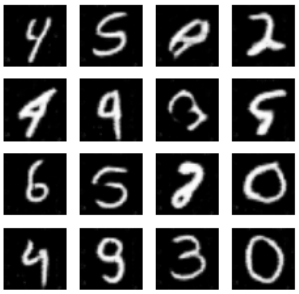
(e) Epoch 5



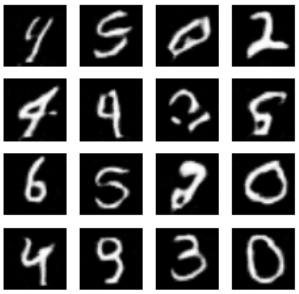
(f) Epoch 6



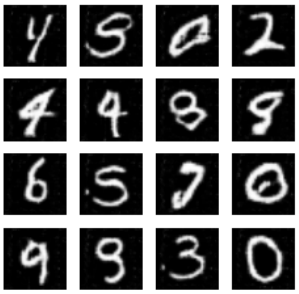
(g) Epoch 7



(h) Epoch 8

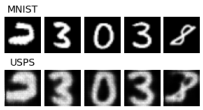


(i) Epoch 9

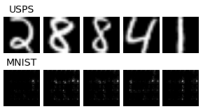


(j) Epoch 10

Figure 5: Evolution of the DCGAN through training



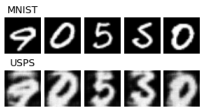
(a) Epoch 1: MNIST to USPS



(b) Epoch 1: USPS to MNIST

(c) Epoch 4: MNIST to USPS

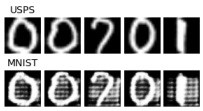
(d) Epoch 4: USPS to MNIST



(e) Epoch 7: MNIST to USPS

(f) Epoch 7: USPS to MNIST

(g) Epoch 8: MNIST to USPS



(h) Epoch 8: USPS to MNIST

(i) Epoch 11: MNIST to USPS

(j) Epoch 11: USPS to MNIST

(k) Epoch 12: MNIST to USPS

(l) Epoch 12: USPS to MNIST

Figure 7: Evolution of the cycleGAN through training