Treball final de grau

GRAU DE MATEMÀTIQUES

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

TripletGAN: model for image generation

Autor: Ricardo Domínguez Mira

Director: Dr. Santi Seguí
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 18 de gener de 2019

Abstract

Generative Adversarial Networks (GANs) have been widely used for image generation. From the conception of this model in 2014 until now, there has been a increasingly interest in this model, which have lead it to be one of the most popular *Deep Learning* models nowadays. The popularity of this model has entailed a significant list of variations of it. This work aims are to study the *GAN* model, and specifically, one of its variations: *TripletGAN*.

2010 Mathematics Subject Classification. 68T05, 68T20, 68W25, 68W40

Acknowledgements

I would like to thank my director, Santi Segui, for all its time and dedication during the work. I would also like to express my thanks to my parents, my partner Julia, and my friends, because without them, I would not be who I am now.

Contents

Introduction

It is beyond all doubt that during the recent years the fields of *Machine Learning*, and specially *Deep Learning*, have undergone an unprecedented growing. This growth has been considerably more significant in what is known as *Generative* models. During the last years, *Machine Learning* important successes were usually related to *Discriminative* models which were based on *BackPropagation* and *Dropout* algorithms. It was in 2014 when Goodfellow [1] introduced the *Generative Adversarial Networks*. From then on, until current times, this model has experimented continuous modifications in order to improve it. Also, if one looks to the number of variations of this model, that have been defined over those years, it is impossible to not think of *GAN* model as one of the most popular *Deep Learning* models these days.

The aims of this work are to introduce myself in the *Deep Learning* field, and particularly, to *Deep Generative* models, in order to solve an image generation problem. One can think of this work as a three part problem. The first part is the one that sets the basis of *Machine Learning* and *Deep Learning*. This first part is structured in two chapters, in the first one we show the basics of what a *Machine Learning* algorithm is, and also some interesting methods for function estimation like *Maximum Likelihood Estimator*.

The second chapter is centered in give a brief theoretical framework for a *Multilayer Perceptron* model, which is the base model in *Deep Learning*, in this chapter we give the formal definition of a *MLP* model and the algorithm that is used for training it, we also give a proof based in [4] for the *Universal Approximation Theorem*.

The second part of the work is oriented in the specific model of *Deep Learning* that we have discussed before: *Generative Adversarial Networks*. Also in this second chapter we study the main model of this work: *TripletGAN* model. Following the same structure of part one here we also have two chapters. The fist one covers the classic *GAN* model, we give its formal definition and also a theoretical analysis of its well behaviour. In the second chapter we present the *Triplet Loss* function and also some notions about what is know as *Integral Probability Metrics*. With this and the *GAN* model defined in the previous chapter we present here a variation of the mentioned model, which name is, as we have said, *TripletGAN* model.

The last part coincides with the last chapter. This part can be seen as the experimental part. Here, we present the results of applying the *TripletGAN* model over two famous academic *datasets* (*MNIST, FASHION-mnist*). In this last part we also discuss about the technical framework on which we have programmed.

Chapter 1

Preliminaries

The purpose of this first chapter is to give a brief introduction of *Machine Learning*. We know that it is a large field, so we do not attempt to cover everything but just what we need in further chapters. The first part of this chapter aims to introduce the notation and concepts that we use in a *Machine Learning* context. The second one is oriented to introduce the estimation method of the *Maximum Likelihood Estimator*. This statistical estimation method is crucial for us to define later what we call *cost functions*.

1.1 Machine Learning Algorithm

The first apparition of th *Machine Learning* concept was in the late fifties by the hand of Samuel [11]. His definition was, however, ambiguous. He spoke explicitly of 'computers with the ability to learn' without shedding any light on what 'learn' means. In order to avoid what seems more of a metaphysical question than a mathematical one, we will rely on the following, and more formal definition, provided by Mitchel (1997) [12]:

Definition 1.1. *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

To fully understand that definition we first need to understand the three concepts that appear in it. When facing a *Machine Learning* problem we are asked to solve a particular problem: maybe we are asked to make an algorithm[1] capable of classify between cat photos or dog photos, or we could be asked to make an algorithm capable of predict the price of a house given its squared meters. Those two examples define what we name *Tasks*. The **Task** is, then, the problem we want our algorithm to solve (classify photos or predict a price in the examples just given). Our problem however, is to get the algorithm to *learn* to solve its assigned problem (*Task*).

We have now set the *Task* concept. When giving the examples we suppose that the algorithm is getting some information in order to make the prediction. This information

[1] When in the context of *Machine Learning*, we sometimes will use indistinctly the terms *algorithm* or *model*.

that the algorithm gets, is expressed as a vectorized representation of a real *observation* problem. The example of the house pricing prediction is trivial, we just have to pass to the model one scalar of its squared meters. The dog and cat images case is far more complicate. In this last scenario we face the matrix representation of a photo.This is a common practice in *Deep Learning* since it is mainly focused in images, text and sound. For the image codification we usually use multi-dimensional matrices. This objects are known as *multi-dimensional arrays* or even *Tensors* in computer science. When we face a RGB colored image we treat it as a 3D-matrix, where the third component can be understood as the volume of the matrix. In particular, for a RGB photo we have three channels, and for a gray scale image we just have one. Since we will work only with gray scale images we are not going to extend more here. Now, we introduce the definition of the object or structure that resumes all our information:

Definition 1.2. Given a *sample space* $X = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$ where each $x^{(i)}$ is considered as an *observation*, and each $x_j^{(i)}$ from $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, ..., x_m^{(i)})$ a *feature*, we can construct $\mathcal{D} \in \mathbb{R}^{n \times m}$ as a matrix[2] where each row contains one observation with all its features and the different features of each observation are confined in the matrix columns. This matrix $\mathcal{D}$ is called ***dataset*** [3].

From what have said, one can think that the observations that are defined in the *dataset* are our *Experience*. This would be, in fact, a misconception. Every *Machine Learning* algorithm needs some input, and that input is in the following categorization: **Supervised** or **Unsupervised** learning. The fact or being in one category or the other depends on what we know as **target** or **label**. This *target* represent the real value of an observation. When we get our algorithm to interact with this *label* we say that we work on a *Supervised Learning Algorithm*. When we do not pass any information about *labels* to the algorithm we are reproducing an *Unsupervised Learning Algoritm*. Almost every *Machine Learning* algorithm is classified in one of this two categories.

The only term that remains untouched is the *Performance Measure*. This *Performance Measure* is a function we chose depending on the case for a qualitative notion of how well the algorithm solves its task. The most intuitive could be the *accuracy* measure, which is used in binary classification tasks and it is as simple as the ratio between well classified observations and the total number of observations. There are a lot of *Performance Measures*, and as we said, choosing between is conditioned by our problem.

With all this information one can now define a complete *Machine Learning* model, and that is exactly what we do in the next section. We present the simplest *Machine Learning* algorithm: the *Linear Regression Model*.

[2]Note that when working in complex sample space, as the image space, the matrix would be expressed as multi-dimensional arrays or *tensors*

[3]Throughout the work we will use indistinctly $\mathcal{D}$ or X when speaking of *datasets*

Example 1: Lineal Regression

Let $x \in \mathbb{R}^n$ be one *observation* with *n features* and let $y \in \mathbb{R}$ be its corresponding **label**. Then, we define as $\hat{y}$ the approximation of y such that $\hat{y} = w^\top x$ where $w \in \mathbb{R}^n$ is known as the *weight* or *parameter* vector. We just have formally defined a *Task*. By now our algorithm is returning a prediction $\hat{y}$ but we do not know if it is a 'good' one.

In this case we are going to use the **mean squared error** as **loss function**. Every *Machine Learning* algorithm has one *loss or cost function*. This function maps an event to a real value that intuitively represents the loss associated with that event. In our scenario our events are if our predictions of y are correct or not, so it seems logical to use the following:

$$MSE = \frac{1}{m}\sum_i (\hat{y} - y)_i^2 \tag{1.1}$$

Note that the choice of this function automatically send our problem into the *Supervised* category, since for computing the error we let the algorithm to use the value of the label y. What we are doing here is to calculate the observation errors as the distance[4] between their real values of y and the predictions $\hat{y}$. This is perfectly logical as we want $\hat{y} \approx y$. When we have computed all this errors the error is then given as the mean of all of them.[5]

As we stated above, we want $\hat{y} \approx y$ to happen. This is equivalent to looking for $\nabla_w MSE = 0$ to happen. To note this, it suffices to note that $MSE \geq 0$, and that the equality only holds if the prediction is equal to the real value for every observation. Thus, minimizing MSE lead us to the more accurate prediction the model can get, given its hypothesis space. In our example our hypothesis space is $\{w^\top x\}_{w \in \mathbb{R}^n}$.

Let us express the MSE with an equivalent representation, which is considering the euclidean norm between two vectors instead of the sum of squares. We aim to solve the following equation

$$\nabla_w MSE = \nabla_w \frac{1}{m}\|\hat{y} - y\|_2^2 = 0 \tag{1.2}$$

This lead us to:

$$\begin{aligned}
\nabla_w \frac{1}{m}\|\hat{y} - y\|_2^2 = 0 \Rightarrow \frac{1}{m}\nabla_w \|Xw - y\|_2^2 = 0 \\
\Rightarrow \nabla_w (Xw - y)^\top (Xw - y) = 0 \\
\Rightarrow \nabla_w (w^\top X^\top Xw - 2w^\top X^\top y + y^\top y) = 0 \\
\Rightarrow 2X^\top Xw - 2X^\top y = 0 \\
\Rightarrow w = (X^\top X)^{-1} X^\top y
\end{aligned}$$

The last equation is a well known lineal equation system. These equations even have a name, they are called the *normal equations*. Solving them will yield to the best approximation of y.

[4] Notice that $\sum_i (\hat{y} - y)_i^2 = \|\hat{y} - y\|_2^2$

[5] This is the reason of the method being called *Mean Squared Error*

1.2 Generalization

We have just presented a basic example of a machine learning algorithm. Now, it seems legitimate to ask oneself what is the difference between a machine learning problem and an optimization one. In this last example we have just found the values of $w \in \mathbb{R}^n$ that made minimum the performance measure **MSE**. This is obviously an optimization problem, so, why are we discussing it as a machine learning one? The objective of this section is to give an answer to this last question and to present the statistical framework we are going to use along the project.

When we are trying to solve a machine learning problem our main objective is to design an algorithm that adapts well when facing new data -observations that has not experienced before-.This property of performing well with new data is called **generalization**, and is a crucial concept in the field of machine learning. In order to achieve this generalization we simulate the situation by splitting the initial dataset in two before anything else. These two parts are named **train** and **test** sets.

We can explain now the difference between proper optimization and machine learning. We are interested in getting a good result when analyzing the **performance measure** P_{test}, which is defined over the test set and we keep it as intractable. With the hope of improving P_{test} we optimize a different measure function P_{train}, which will be the **loss function** defined over the train set. Minimizing the loss function P_{train} can be solved with an optimization algorithm, and so it can be seen as an optimization problem. Nevertheless, minimizing P_{test} is what we really want, and because of that we can not say that our problem is an optimization problem. We are optimizing P_{train} but just to indirectly, and here relies the main difference, optimize P_{test}.

With relation to this we introduce the two most important problems when looking for a good *generalization*. The first one is known as **Overffiting**. We face this problem when our model has a good performance when evaluated in the training set but not in the test. The second one is produced when the model do not perform well in any of the two sets. It is named **Underfitting**. To sort these problems we define a new concept: **Capacity**. We can define the *capacity* of a model as the range of possible functions that it can fit. We can control *capacity* by choosing its *hypothesis space*. In the *Linear Regression* example we gave the hypothesis space of our model, which was $\{w^\top x\}_{w\in\mathbb{R}^n}$ for $x \in \mathbb{R}^n$. If we let the hypothesis space to have polynomials, or other kind of functions, we are augmenting its capacity. This may lead us to a complex solution that fits well the train set but not the test one. This problem is very common in *Machine Learning* and has the name of **Overfitting**. The other problem we usually have is the **Underffiting**. This problem origin is that our *hypothesis space* may not contain the function we look for, therefore, the model does not perform well in either set.

1.3 Maximum Likelihood Estimation

From now on we will consider our problem from a more probabilistic or statistic point of view. What we have considered in Example 1. as an optimization problem, finding some w^* such that the loss function was minimum, now is raised as the problem of **estimating** some function f^* over a parametric family of functions $\{f_\theta\}_{\theta\in\Theta}$ such that $f^* \approx f$, for f being the real, but unknown, objective function. In this case $\{f_\theta\}_{\theta\in\Theta}$ is our *hypothesis space*.

We consider that our observations $x \in \mathbb{R}^n$ follow a certain, but unknown, distribution function $p_r(x)$. This function plays the same role as the objective function f we considered above. We also consider a family of parametric distribution functions $\{p(x;\ \theta)\}_{\theta\in\Theta}$ that represent the hypothesis space of our algorithm. Let us introduce before anything the following definition:

Definition 1.3. *Given an observation $x \in X$, where X is considered the sample space, we can define a likelihood function as follows:*

$$\begin{aligned} L(x;\cdot) : \Theta &\longrightarrow \mathbb{R} \\ \theta &\longmapsto L(x;\theta) = f_\theta(x) \end{aligned}$$

Where f_θ is the density of the mentioned model. [6]

So we can say that if we are given a certain $x \in X$, we can make an assumption of how are the observations distributed. Once we have one assumption, we can set

$$\begin{aligned} p(x;\cdot) : \Theta &\to \mathbb{R} \\ \theta &\to p(x;\theta) = f_\theta(x) \end{aligned} \tag{1.3}$$

which is indeed a likelihood function if our assumption was right. It is important to note the importance of how well we guess the distribution. If the real distribution of the data does not belong to our *hypothesis space* for some θ then, there is nothing we can do to estimate our parameters this way. One of the most used methods to *estimate* is the *Maximum Likelihood Estimator*. This method gives us an estimate $\hat{\theta} \in \Theta$ that maximizes the likelihood function for every observation. Let us define it formally.

Definition 1.4. *We will call maximum likelihood estimator (MLE) of θ to $\hat{\theta}\colon X \to \Theta \subset \mathbb{R}^d$ such that*

$$L(x;\hat{\theta}(x)) = \sup_{\theta\in\Theta} L(x;\theta),\ \forall x \in X \tag{1.4}$$

In plane words, we can say that the MLE of a parameter θ is understood as the parameter $\hat{\theta} \in \Theta$ that maximizes the likelihood function for every observation $x \in X$. Let us retake the fact that we can set our hypothesis space $\{p(x;\ \theta)\}_{\theta\in\Theta}$ to match 1.4 once we have made an assumption about the distribution of our observations. We also point that

[6] Here f_θ is expressed as the *Radon-Nikodym derivative* of the probability distribution. This let us define a likelihood function even if the model has not an absolute continuous probability distribution

we consider our observations $x^{(i)}$ to be i.i.d. Then, as we only care for the parameters that maximize the likelihood function for every x we can express 1.4 as follows:

$$\theta_{ML} = \arg\max_{\theta \in \Theta} \prod_{i=1}^{n} p(x^{(i)}; \theta) \tag{1.5}$$

And since we aim to maximize this, and therefore we will be interested in differentiate this function, we can apply over it a log-transformation to make the differentiation part easier:

$$\theta_{ML} = \arg\max_{\theta \in \Theta} \sum_{i=1}^{n} l(x^{(i)}; \theta) \tag{1.6}$$

Where $l(x; \cdot) = \log p(x; \cdot)$. Notice that doing this transformation does not change the maximum we are looking for since the $\log()$ function is a monotonically increasing function. Now, if we divide by a constant n, without changing the maximum, we obtain the following expression:

$$\theta_{ML} = \arg\max_{\theta \in \Theta} \frac{1}{n} \sum_{i=1}^{n} l(x^{(i)}; \theta) \tag{1.7}$$

We are almost done, but before showing the last expression of 1.4 we have to present a new variable with a particular distribution:

Definition 1.5. *Let $Y = (y^{(1)}, y^{(2)}, \ldots, y^{(m)})$ be a sample of size m where $y^{(1)}, y^{(2)}, \ldots, y^{(m)}$ are the m observations from this sample. The **empirical distribution** of the sample Y is a function $F : X \to [0, 1]$ defined as:*

$$F(y) = \frac{1}{n} \sum_{i=1}^{n} 1_{\{y^{(i)} \leq y\}}$$

*Where $1_{\{y^{(i)} \leq y\}}$ is the **indicator function**.*

Taking now Definition 1.5 we have that for a fixed $x^{(i)}$ we can set $P(X = x^{(i)}) = F(x^{(i)}) - F(x^{(i)-}) = \frac{1}{n}$ with X following the empirical distribution. Thus, we can substitute this in 1.7 and we get:

$$\theta_{ML} = \arg\max_{\theta \in \Theta} \sum_{i=1}^{n} P(X = x^{(i)})\, l(x^{(i)}; \theta) \tag{1.8}$$

Which is the same as

$$\theta_{ML} = \arg\max_{\theta \in \Theta} \mathbb{E}_{X \sim \hat{p}_r} l(x^{(i)}; \theta) \tag{1.9}$$

This last equation shows the relation that exists between finding the *MLE* estimator and reducing the *cross-entropy*. This *cross-entropy* and can be expressed as:

$$H(p, q) = \mathbb{E}_q[-\log(p(x))] \tag{1.10}$$

If we take both functions and compare them, we see that maximizing the log-likelihood function $p(x;\ \theta)$ to match $f_\theta(x)$ is equivalent to minimize the *cross-entropy* between a parametric distribution function p_θ and the empirical data distribution $\hat{p}_r$. We will introduce more explicitly this *cross-entropy* function in Chapter 3, where we give the definition of the *KL-Divergence*. For now we just need to be aware of the equivalence between these two procedures since the usual practice is to set the *loss function* of a model as the *cross-entropy* and then, look for its minimum.

In order to set this clear we now introduce one of the other most common examples of *Machine Learning* model. In this case we are going to proceed by the *MLE* method to find the solution.

Example 2: Logistic Regression

The *Logistic Regression* model is very useful for binary classification problems, despite of its name. In this new scenario we aim to estimate $P(Y = 1 \mid X = x)$ and since we look for a probability, we are looking for a function $\sigma : X \to [0, 1]$. Let us introduce the hypothesis space that is used in this model as $\{\sigma_\theta(z)\}$. Where $z = \theta^\top x$. The function σ has the name of *logistic sigmoid function*.

$$\begin{aligned} \sigma : Z &\longrightarrow [0,1] \\ z &\longmapsto \frac{1}{1+e^{-z}} \end{aligned}$$

This function has some interesting properties. Its value tends to 0 when $z \to -\infty$ and tends to 1 when $z \to +\infty$. It is also a differentiable function, which is essential to minimize the cost function. Now we suppose that $P(Y = 1 \mid X = x) = \sigma_\theta(z)$. Since there are no more possibilities, $P(Y = 1 \mid X = x)$ must be $1 - \sigma_\theta(z)$. Then we can give the conditional distribution of Y given X as

$$P(Y \mid X) = \sigma_\theta(z)^y (1 - \sigma_\theta(x))^{1-y}$$

.

Now we follow the steps we followed in [*referencia]. First with the assumption of our observations to be i.i.d we can define the likelihood function as follows

$$p(\theta) = \prod_i \sigma(\theta^\top x^{(i)})^{y(i)} (1 - \sigma(\theta^\top x^{(i)}))^{1-y^{(i)}}$$

And therefore, the log-likelihood as:

$$\log(p(\theta)) = \sum_i y^{(i)} \log(\sigma(\theta^\top x^{(i)})) + (1 - y^{(1)}) \log(1 - \sigma(\theta^\top x^{(i)}))$$

This equation has not a closed solution like in the *MSE* example, in fact, this does not happen very often. In the next section we present the basis of how to approximate to the minimum of this kind of functions, and we will have finished this preliminary chapter.

1.4 Gradient Descent

The purpose of this section is to briefly give the *Gradient Descent* algorithm. This algorithm is used to find local or global minimums of functions. It is a first order iterative method, which means that it needs to compute the first derivative of the function to minimize, and also it works on an iterative way. This algorithm has an enormous importance in *Machine Learning* and it is used to find *local minimums* of the *loss functions*.

Gradient Descent

Input: $f : \mathbb{R}^n \to \mathbb{R}$ differentiable
$\mathrm{x}^{(0)}$ an initial solution
Initialize: $k \leftarrow 0$
while $\|\nabla f(x^{(k)})\| \geq 0$ **do**
  $x^{(k+1)} = x^{(k)} + \epsilon \nabla f(x^{(k)})$
  $k \leftarrow k + 1$
end
Return: $x^{(k)}$

Algorithm 1: Gradient Descent Algorithm

This algorithm is the base for the algorithms that are used in *Machine Learning*, and specially its subset: *Deep Learning*. When working in the last field we face a huge number of observations, and applying this algorithm get us a significant computational burden. In order to avoid this there have been designed some variations of *Gradient Descent*.

The most important to mention are the *Stochastic Gradient Descent (SGD)* and the *Mini-Batch Gradient Descent*. The first one computes the gradient taking one observation each iteration [7]. The *Mini-Batch*, by the other side, let us to compute the gradients for fixed sized *batches*, or sets, of observations.

[7] The *Batch Gradient Descent* represent the opposite method, it takes all the observations for each iteration

Chapter 2

Deep Learning

In this chapter we introduce a special kind of machine learning models: *Neural Networks*. We aim to show how these specific models have evolved trough time to the current days. By showing this evolution we are able to introduce briefly all the important concepts associated with *Neural Networks*. It also let us define a closed terminology to follow in a field whose concepts are sometimes interchangeable, then avoiding possible misconceptions. Also we show in this chapter that this kind of models are extremely useful for the task of function approximation by proving the *Universal Approximation Theorem*. Finally we give the algorithm that is used to making the training process possible.

2.1 The Perceptron

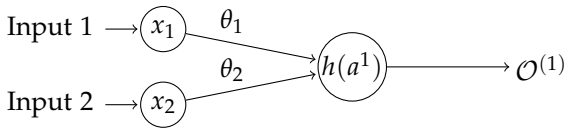
Although the name of *neural network* seems a bit pretentious it is not if we look at its first apparition. The '*Perceptron*' was introduced in [3] by Rosenblatt, who was a psychologist, in his attempt to give a mathematical model to the brain human cells: neurons. In this section we introduce a few concepts that will be required for explaining what a *Perceptron* does, and also to understand its limitations.

We are going to start by defining the *Perceptron* model. Let us take $\theta = (\theta_1, \cdots, \theta_n)$ and $x = (x_1, \cdots, x_n)$, both from $\mathbb{R}^n$. Also we need to define a function:

$$h(x) = \text{sign}(\theta^\top x), \quad \text{where sign}(z) = \begin{cases} 1 & z \geq 0 \\ -1 & z < 0 \end{cases}$$

Now we can give a visual representation of this *Perceptron* model. This kind of representations are known as *graph representations*. The cause for this to happen is that sometimes these models are considered as *directed acyclic graphs*. That means that can be represented as graphs without cycles where each node or edge represent some operations. Usually we see that the edges represent the linear transformations of the elements comming from the previous node. The nodes represent the composition of these linear transformations with a non-linear function, usually named g but in our example we named it h. Note

that a directed acyclic graph will always have at least 1 leaf node, those nodes correspond to the output nodes, in our example from below the output node is represented as $O^{(1)}$.



Where

$$a^{(1)} = \sum_{i=1}^{2} \theta_i x_i + b, \quad b \in \mathbb{R} \tag{2.1}$$

$$\mathcal{O}^{(1)} = h(a^{(1)}) = \{1, -1\} \tag{2.2}$$

Omiting the *bias* or *intercept* term for simplicity, and considering $\mathcal{O}^{(1)} = h(a^{(1)}) = h_\theta(x) = \text{sign}(\theta^\top x)$ and making use of the concepts from the previous chapter we see that what we are facing is a supervised binary classification problem. The way of training here, however, is a bit different. For updating the weights of our model what we do is just follow a simple rule every new training observation $(x^{(i)}, y^{(i)})$. That rule is the following:

$$\theta_j \leftarrow \theta_j - \frac{\epsilon}{2}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \tag{2.3}$$

Looking at this function we can see that for every time that the output is the correct one, the parameters do not change. We also note that when the output is not correct we have the following:

$$\theta_j \leftarrow \theta_j + \epsilon y^{(i)} x_j^{(i)} \tag{2.4}$$

The training process of this model is equivalent to finding a separator hyperplane between the two classes of x labeled by y as $\{-1, 1\}$ and here is were we face it most important limitation. Let us give a definition first for the following part:

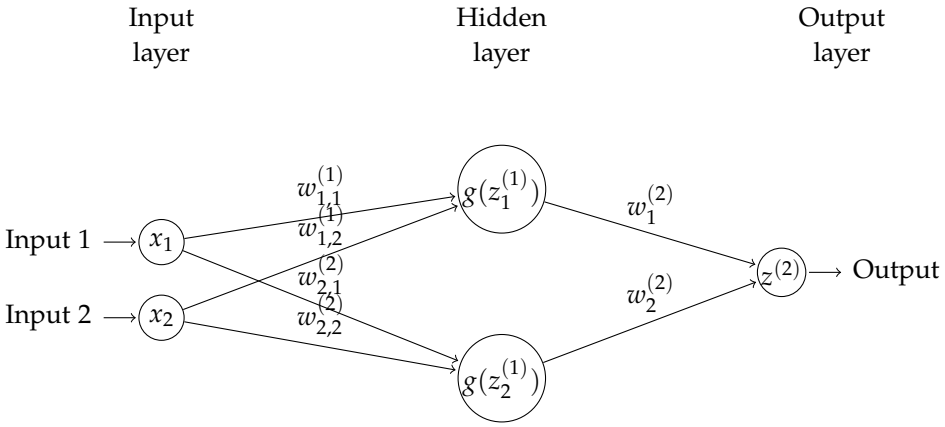
Definition 2.1. *Let X_1 and X_2 be two separated set of points in $\mathbb{R}^n$. We say that X_1 and X_2 are* ***linearly separable*** *if there exist some $w \in \mathbb{R}^n$ and $k \in \mathbb{R}$ such that*

$$\sum_i^n w^\top x < k, \ \forall x \in X_1 \ \text{ and } \ \sum_i^n w^\top x > k, \ \forall x \in X_2$$

This property of lineal separability holds for sure when facing two disjoint convex sets, but usually lead us to problems when they are not. It was showed in [2] by Minsky and Papert that the *Perceptron* were only capable of learn to classify between two classes when given a linearly separable base space. They showed in their book its limitations by remarking the failure of the *Perceptron* on learning the XOR function. In order to surpass this problem this primitive model of a *Neural Network* evolved from the single perceptron to the *Multilayer Perceptron* (MLP).

2.2 Multilayer Perceptron

As we said at the end of the last section, the *Multilayer Perceptron*, or as we are going to name it along the project: **MLP**, was the solution for skipping the linearly separation limitations. We can see this model as an evolution of the previous one. What we do now is to stack different perceptrons in one or more layers. Here we show graphically the simplest *MLP* that exists:



The elements of this graph are the following: $x = (x_1, x_2)$ a single observation vector, $W^{(1)} = (w_1^{(1)}, w_2^{(1)})$ a matrix, whose rows are the weight vectors of each neuron in the hidden layer, i.e, $w_i^{(1)} = (w_{i,1}^{(1)}, w_{i,2}^{(1)})$ for $i = 1, 2$. We have a second linear transformation over the outputs of the hidden layer with $w^{(2)} = (w_1^{(2)}, w_2^{(2)})$ being its weight vector. Lastly, if we let $g(\cdot)$ be a non-linear function, lets say a *sigmoidal* one, we can express the elements $z^{(i)}$ as follows

$$z_i^{(1)} = \sum_{j=1}^{2} w_{i,j}^{(1)} x_j + b_i^{(1)}, \quad i = 1, 2$$

$$z^{(2)} = \sum_{j=1}^{2} w_j^{(2)} g(z_j^{(1)}) + b^{(2)}$$

This simple *MLP* model is proved to solve the *XOR* approximation task. It surpass the problem of its predecessor by composing non-linear functions with lineal ones. Making this enables the model to learn not only lineal functions [1], but a huge space of different functions as we will show in Section 3. This non-lineal functions that are used on the *hidden layer*, which name comes from the fact of it nor being the input nor the output one, but an intermediate layer, are named **activation functions**. Even there are some activation functions that are of common use, now the *ReLU* activation function [2] could be the most popular, but taking the decision of choosing one over the others is still an active area of

[1] Note that if we let $g(\cdot)$ to be linear, the total composition of functions will give a linear function, and we can not approximate a non-linear function with a linear one.

[2] *ReLU* stands for *Rectifier Linear Unit*, and we can represent it as: $\max[x, 0]$ for a given x

research. On the graph we showed before we chose g to be a *sigmoidal* activation function, this particular one, and the tahn$(\cdot)$ function are commonly used for classification tasks, and were the two most popular until the apparition of *ReLU*. Every one of them has its own benefits and limitations, we only need for them to be *differentiable* in order to get the gradients during the training process as we will see later.

Let us introduce finally a formal generalization of a *MLP*. As we understand this model now, the first step would be a linear transformation of some input vector $x \in \mathbb{R}^n$.

$$\begin{aligned} A^{(1)} \ : \ \mathbb{R}^n &\longrightarrow \mathbb{R}^m \\ (x_1, \cdots, x_n) &\longmapsto (z_1, \cdots, z_m) \end{aligned} \tag{2.5}$$

Where $A^{(1)}(x) = W^{(1)}x + b^{(1)} = z$ for W being a $m \times n$ matrix, with m denoting the number of hidden units that has the hidden layer 1, and b is a m-dimensional vector for the *bias* term. Now if we retake equation 2.5 and compose it with a non-lineal function applied element wise we get

$$\begin{aligned} g^{(1)} \ : \ \mathbb{R}^m &\longrightarrow \mathbb{R}^m \\ (z_1, \cdots, z_m) &\longmapsto (h_1, \cdots, h_m) \end{aligned} \tag{2.6}$$

With $h = g^{(1)}(z) = (g^{(1)} \circ A^{(1)})(x)$. We can see these compositions as the functions that resume the process of passing from one layer to the next. The important point to note here is that we can express an *MLP* model just as a differentiable function f. This function must be, of course, a composition of functions such as $F^{(i)} = (g^{(i)} \circ A^{(i)})$. With this we can define a differentiable $f = (F^n \circ \cdots F_2 \circ F_1)$ that resumes our *MLP* model. Now we give a more general and formal definition of a neural network by defining its components. We will use this notation the following sections.

Definition 2.2. *Given $x \in \mathbb{R}^n$ a vectorized observation. Let be $(W^{(l)}, b^{(l)})$ the pair consisting in the weight matrix and the vector intercept for the layer l where $1 \leq l < L$. Then we can define a MLP with L layers by defining its components as follows:*

For $1 \leq l < L$

- *We define the **hidden layers** as $h^{(l)} = g(z^{(l)})$. Where $z^{(l)} = W^{(l)}h^{(l-1)} + b^{(l)}$ and $g(\cdot)$ is usually a non-linear real valued function.*

And for $l = 0$ and $l = L$

- *The first layer is defined as $h^{(0)} = x$. This is why it is usually also called **input layer**.*
- *The last layer is defined as $h^{(L)} = F(h^{(L-1)})$. Where F is called **output function** and that is the reason for knowing this last layer as **output layer**.*

As we have seen what a *MLP* is, let us show how this parametric family of functions performs in the task of being used as approximator functions.

2.3 Universal Approximation Theorem

In this section we aim to proof the *Universal Approximation Theorem*. This theorem gives a theoretical proof of how *MLP* are capable of approximate a wide variety of functions. We also want to make clear that our work is not centered in giving a deep analysis of this theorem, and then, we are going to give only an intuition behind the theorems used during the proof, instead of proving them too. It is worth pointing out that we are going to follow the version of the proof that was given in [4]. This proof is based on *sigmoid* activation functions, a more general proof can be obtained in [5]. Let us present the two theorems that we will use later.

Theorem 2.3. *Hahn-Banach Theorem. Let X be a real vector space, p a subliner functional on X, $M \subset X$ and f a linear functional on M such that $f(x) \leq p(x)$, $\forall x \in M$. Then there exists a linear functional F on X such that $F(x) \leq p(x)$ for all $x \in X$ and $F_{|M} = f$.*

This theorem is used as a generalization of the *hyperplane separator theorem* to topological vector spaces and it is an important tool in *functional analysis*. We do not aim to proof it but to give its intuition in our problem scenario. It is shown in [14] that given the conditions of X being a normed space and $d(x, M) > 0$ [3] there exists a functional F defined over X with $F \neq 0$ but $F_{|M} = 0$. This is essentially what is used during the proof, and since we will proof the existence of this functional, we will represent it as the following theorem states.

Theorem 2.4. *Riesz-Markov-Kakutani Representation Theorem. Let X be a locally compact Hausdorff space and $\mathcal{C}_c(X)$ the space of compact support functions defined on X. For any positive linear functional I on $\mathcal{C}_c(X)$, there is a unique regular Borel measure μ on X such that*

$$I(f) = \int_X f(x) d\mu(x)$$

We do not attempt to give any details here as the theory that underlie this theorem is far beyond of the purpose of our work.

Definition 2.5. *We say that $\sigma \in \mathcal{C}(I_m)$ is **discriminatory** relative to a signd Borel measure μ if*

$$\int \sigma(y^\top x + \theta) d\mu(x) = 0$$

$\forall y \in \mathcal{I}_{\Updownarrow}$ and $\theta \in \mathbb{R}$, implies that $\mu = 0$.

The point about this definition is to note σ as a non-destructive function. The basic idea is to understand that if we pass some x to σ, there would exist some y and θ such that the linear transformation of $y^\top x + \theta$ could be send to a non-zero measure set, if not, it is the measure μ itself which is 0. This help us to prevent losing information by applying this kind of functions when designing a *MLP*. In annex 7 we have given the proof of every bounded *sigmoid* function being discriminatory. We have moved it to the Annex because

[3] Here d represents the uniform norm

we have not notions of some mathematical concepts that involve those proves (*Fourier Transform*).

Theorem 2.6. *Let σ be a bounded, discriminatory sigmoid function. Let $\mathcal{C}(I_m)$ be the space of real-valued continuous functions on $I_m = [0,1]^m$. Then given $\epsilon > 0$ and any function $f \in \mathcal{C}(I_m)$, there exist $N \in \mathbb{N}$, $\beta_i, c_i \in \mathbb{R}$, $w_i \in \mathbb{R}^m$ for $i = 1, \cdots, N$ with which we may define:*

$$F(x) = \sum_{i=1}^{N} \beta_i \sigma(w_i^\top x + c_i) \tag{2.7}$$

such that

$$| F(x) - f(x) | < \epsilon, \ \ \forall x \in C(I_m). \tag{2.8}$$

In other words, functions of the form $F(x)$ are dense in $\mathcal{C}(I_m)$.

Proof. We start by defining $\mathcal{N} \subset \mathcal{C}(I_m)$ as the space of functions of the form 2.7. It is easy to see that $\mathcal{N}$ is a linear subspace of $\mathcal{C}(I_m)$. We aim to prove that $\mathcal{N}$ is dense in $\mathcal{C}(I_m)$, i.e $\overline{\mathcal{N}} = \mathcal{C}(I_m)$. In order to prove that we are going to suppose the opposite.

Let suppose then $\overline{\mathcal{N}} \neq \mathcal{C}(I_m)$. We have that the closure of $\mathcal{N}$ is a closed, proper subspace of $\mathcal{C}(I_m)$. Here we make use of Theorem 2.3 *(Hahn-Bach Theorem)* to assert the existence of a bounded linear functional L on $\mathcal{C}(I_m)$ such that $L_{|\overline{\mathcal{N}}} = 0$ but with $L \neq 0$. It is obvious that $L_{|\overline{\mathcal{N}}} = 0$ implies $L_{|\mathcal{N}} = 0$. Now, using Theorem 2.4 we know that this functional can be represented as

$$L(h) = \int_{I_m} h(x) d\mu(x) \tag{2.9}$$

for some $\mu \in \mathcal{M}(I_m)$ and $\forall h \in \mathcal{C}(I_m)$. In particular, as $\forall F \in \mathcal{N} \subset \mathcal{C}_c(X)$ we have $L_{|\mathcal{N}} = 0$, and then, we can say the following

$$L(F) = \int_{I_m} F(x) d\mu(x) = \int_{I_m} \sigma(y^\top x + \theta) d\mu(x) = 0 \tag{2.10}$$

for all y and θ.

We started the proof following the assumption that σ was a *discriminatory* function. Thus, we must have now $\mu = 0$. But this follows a contradiction with the fact of $L \neq 0$ since given $\mu = 0$ we have

$$\int_{I_m} h(x) d\mu(x) = 0, \ \ \forall h \in \mathcal{C}(I_m). \tag{2.11}$$

Hence, $\mathcal{N}$ must be dense in $\mathcal{C}(I_m)$ □

This Theorem we have just proved implies that a *MLP* big enough will be able to represent almost every function we are interested in learning. The Theorem, however, do not guaranty that the algorithm will be to *learn* that function. Maybe because the algorithm used to find the optimal parameters fails or maybe because we face an *Overfitting* problem.

This Theorem also do not say anything about the structure of a *MLP*. It assures that a *large enough MLP* can reproduce any continuous function over a compact set, but do not say anything about the number of units or layers [4].

2.4 Backpropagation Algorithm

In this section we present the algorithm that makes possible the training process of a *MLP* model: the *Backpropagation Algorithm*. Let us remember that we will work on a *Machine Learning* problem, hence our goal is to make our algorithm to *learn*. In Chapter 1 we gave the definition of a *cost function*, when working with *Neural Networks* we usually chose the *cross-entropy* as our cost function. This scenario is now not as simple as when we tried to find the optimal of a *lineal regression*, we are facing now very complex functions, defined over huge parameter spaces, and obviously not convex. Although there are a lot of gradient based methods *(SGD, MiniBatch Gradient Descencent...)* to find local optimums of the cost function the problem here is to compute all the derivatives with respect to thousands of parameters, and *Backpropagation* is the solution.

Let us first give the intuition of the algorithm and then we will proof its functionality. If we imagine the first step of our process to be the input going trough the *MLP* until it is transformed to the output as goign *forward*, then it will be easy to understand what to do when going *backwards*. Once we have computed the output, the backpropagation algorithm does the inverse to compute the error of each layer from the output one until the input layer [5].

Let us define $\delta_j^{(l)} = \partial h_j^{(l)} / \partial z_j^{(l)}$ as the error of neuron j in layer l. We are going to show that by using this definition of the error it is possible to construct an iterative method of computing the error of each neuron of the network. Later on we will use this errors to compute our main objective which are the partial derivatives with respect to the parameters of the network.

Proposition 2.7. *Let L be the cost function (or loss function) for our MLP network and let $\delta_j^{(l)}$ be the error of neuron j in layer l as we defined it before. Then, following the notation introduced in Definition 2.2, we can compute the error for each neuron j in every layer l by using the following iterative method:*

For the last layer ($l = L$)

$$\delta_j^{(L)} = \frac{\partial L}{\partial h_j^{(L)}} \sigma'(z_j^{(L)}) \tag{2.12}$$

And for $1 \leq l < L$

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} W_{k,j}^{(l+1)} \sigma'(z_j^{(l)}) \tag{2.13}$$

[4] The proof given by [5] is not reduced only to one layer like the one of [4]

[5] Which is not being considered as the input does not have any error to compute

Proof. We are going to proof equation 2.12 in first place. Following the definition of $\delta_j^{(l)}$ and making use of the *Chain Rule of Calculus* it is immediate that:

$$\delta_j^{(L)} = \frac{\partial L}{\partial z_j^{(L)}} = \frac{\partial L}{\partial h_j^{(L)}} \frac{\partial h^{(L)}}{\partial z_j^{(L)}} = \frac{\partial L}{\partial h_j^{(L)}} \sigma'(z_j^{(L)}) \tag{2.14}$$

The only thing we needed here was to remember the definition of $h^{(L)}$, and that is $h^{(L)} = \sigma(z^{(L)})$. Now, given that we know how to compute the error for the last layer we can compute the rest in terms of their next. Using again the Chain Rule of Calculus we get

$$\delta_j^{(l)} = \frac{\partial L}{\partial z_j^{(l)}} = \sum_k \frac{\partial L}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \tag{2.15}$$

Now we just have to study the two terms separately. We know by definition that $\partial L / \partial z_k^{(l+1)} = \delta_k^{(l+1)}$. Retaking the definition of $z_k^{(l+1)}$ we can see its relation with $z^{(l)}$ as follows

$$z_k^{(l+1)} = W_k^{(l+1)} \sigma(z^{(l)}) + b^{(l+1)} = \sum_j W_{k,j}^{(l+1)} \sigma(z_j^{(l)}) + b^{(l+1)} \tag{2.16}$$

From this we see that its partial derivative with respect to $z_j^{(l)}$ will be 0 except for j-th element of $\sigma(z_j^{(l)})$. Thus

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k W_{k,j}^{(l+1)} \sigma'(z_j^{(l)}) \tag{2.17}$$

Lastly, we just have to bring equations 2.16 and 2.17 together to conclude the proof:

$$\delta_j^{(l)} = \sum_k \frac{\partial L}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \delta_k^{(l+1)} W_{k,j}^{(l+1)} \sigma'(z_j^{(l)}). \tag{2.18}$$

□

Once we have shown the process to compute all the error terms of our *MLP* it is time to use them to compute the partial derivatives with respect to the parameters.

Proposition 2.8. *Given L the cost function (or loss function) for a MLP network on which we hace calculated the error terms $\delta_j^{(l)}$ for all l and j, we can compute the partial derivatives of L with respect to its parameters as follows:*

$$\frac{\partial L}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} h_j^{(l-1)} \tag{2.19}$$

$$\frac{\partial L}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{2.20}$$

Proof. We will proof the first and we will show that prove the second is equivalent to it. First we want to retake the definition of $z_k^{(l)}$

$$z_k^{(l)} = \sum_j W_{k,j}^{(l)} \sigma(z_j^{(l-1)}) + b_k^{(l)}. \tag{2.21}$$

It is easy to see from 2.21 that $\partial z_k^{(l)} / \partial w_{i,j}^{(l)} = 0$ for every $k \neq i$. Now, using again the Chain Rule of Calculus we get

$$\frac{\partial L}{\partial w_{i,j}^{(l)}} = \sum_k \frac{\partial L}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} \tag{2.22}$$

From what we said in the above paragraph we have $\partial z_k^{(l)} / \partial w_{i,j}^{(l)} = 0$ for every $k \neq i$ and $\partial z_k^{(l)} / \partial w_{i,j}^{(l)} = \sigma(z_j^{(l-1)}) = h_j^{(l-1)}$ if $k = i$. Then, the expression 2.22 can be rewritten as:

$$\frac{\partial L}{\partial w_{i,j}^{(l)}} = \frac{\partial L}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}} = \delta_i^{(l)} h_j^{(l-1)} \tag{2.23}$$

The intercept (or bias) part is analogous, we just have to compute the partial derivative with respect to $b_j^{(l)}$ but from the definition of $z_j^{(l)} = W_j^{(l)} \sigma(z^{(l-1)}) + b_j^{(l)}$ we know that $\partial z_k^{(l)} / \partial b_j^{(l)} = 1$ for all l and j when $k = j$ and it would be 0 if not. Then:

$$\frac{\partial L}{\partial b_j^{(l)}} = \sum_k \frac{\partial L}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \frac{\partial L}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{2.24}$$

□

With this last proof we have everything we need to use a *MLP* model in order of solving a *machine learning* problem. Essentially we have shown that *MLP*'s are a family of differentiable functions that are extremely useful for function approximation. Also we have seen that given a cost function L, such as *cross-entropy*, we can compute its partial derivatives to adjust its parameters with a gradient descent method. In the following chapter we will discuss a special kind of *neural network* model which is based in *MLP* functions.

Chapter 3

Generative Adversarial Models

Let us introduce this chapter with a little metaphor of what are we going to reproduce. Imagine you have a two player non-cooperative game between a cop and a criminal, whose crime is to counterfeit currency. The cop job is to become good on distinguishing fake coins over the legal ones. By the other side, the criminal is trying to make coins that are difficult to distinguish in order to pass them as legal ones. This is the exactly philosophy of this model. First in this chapter we will present two ways of calculating the divergence between two distribution probability functions OJO NAME. Then, we will give a theoretical analysis of the Generative Adversarial Model (*GAN*). It is worth pointing out that we refer to this analysis as theoretical because we are not going to consider the model as a parametric one, but we are going to show a convergence analysis in the probability density functions space.

3.1 Theoretical Framework

In this section we present two methods to quantify the similarity between two probability distribution functions. In fact, we have spoken of one of them in Chapter 1. These methods will be used later to understand what are we doing and why, when training the Generative Adversarial Model.

Definition 3.1. *Given p and q two probability distributions of a continuous random variable we define the* ***Kullback-Leibler Divergence*** *as follows:*

$$D_{KL}(p \mid\mid q) = \int_{-\infty}^{+\infty} p(x) \frac{\log p(x)}{\log q(x)} dx. \tag{3.1}$$

If we remember the definition of *cross-entropy* from Chapter 1 we can see now that it comes from here:

$$D_{KL}(p \mid\mid q) = \int_{-\infty}^{+\infty} p(x) \frac{\log p(x)}{\log q(x)} dx = \int_{-\infty}^{+\infty} p(x) \log p(x) - \int_{-\infty}^{+\infty} p(x) \log q(x)$$

If we consider this equation as a sum, the right component can be rewritten as:

$$\mathbb{E}_p[-\log(q)]$$

Which is, indeed, the *cross-entropy* definition. That means that minimizing the *cross-entropy* lead us to minimize the *KullbackLeibler Divergence*, i.e., the dissimilarity between p and q.

Proposition 3.2. *D_{KL} is always non-negative and $D_{KL} = 0$ if and only if $p = q$ almost everywhere.*

Proof. From the definition of D_{KL} we can rewrite equation 3.1 as:

$$D_{KL}(p \,||\, q) = \int_{-\infty}^{+\infty} p(x) \frac{\log p(x)}{\log q(x)} dx = \mathbb{E}_{p(x)}\Big[\log \frac{p(x)}{q(x)}\Big]. \tag{3.2}$$

Now let $A = \{x \mid p(x) > 0\}$. Thus:

$$D_{KL}(p \,||\, q) = \int_A p(x)\big(-\log \frac{q(x)}{p(x)}\big)dx \;\geq -\log \int_A p(x)\frac{q(x)}{p(x)}dx \;\geq -\log \int_X q(x)dx. \tag{3.3}$$

Here we have used the *Jensen's Inequality* on the first inequality, which stands that $g(\mathbb{E}[X]) \leq \mathbb{E}[g(X)]$ for any convex function g. Now, since q is a probability distribution function we have:

$$D_{KL}(p \,||\, q) \geq -\log \int_X q(x)dx = -\log 1 = 0. \tag{3.4}$$

With this we have proved $D_{KL} \geq 0$. Since the function $-\log()$ is strictly convex we can see from equation 3.2 that for the first inequality, the equality only holds when $\frac{q(x)}{p(x)} = c$, $\forall x$. Therefore:

$$\int_A q(x)dx = c \int_A p(x)dx = c. \tag{3.5}$$

Also the last inequality from the same equation 3.2 turns equality when:

$$\int_X q(x)dx = c \int_A p(x)dx = c. \tag{3.6}$$

Therefore, $c = 1$ and we have that $D_{KL} = 0 \Leftrightarrow p = q$ almost everywhere.

□

We have seen a very interesting property of the *KL Divergence*. However, it is important to notice that it is asymmetric. We can think of it when $p(x)$ is close to 0 but $q(x)$ is not. To fix this symmetry problem we present a different way of measuring similarities between distributions:

Definition 3.3. *Given p and q two probability distributions and let $D_{KL}(p \,||\, q)$ be their KL Divergence we can define the **Jensen-Shannon Divergence** as:*

$$D_{JS}(p \,||\, q) = \frac{1}{2} D_{KL}(p \,||\, \frac{p+q}{2}) + \frac{1}{2} D_{KL}(q \,||\, \frac{p+q}{2})$$

Observation 3.4. *It is worth pointing out that $D_{JS}(p \mid\mid q) \geq 0$ from its definition. Even more, we can say that $D_{JS}(p \mid\mid q) = 0$ if and only if $p = q$ almost everywhere. We have just to look at the D_{KL} divergences, they are 0 only if and only if*

$$p = \frac{p+q}{2} \Longrightarrow 2p = p + q \Longrightarrow p = q \ \text{ almost everywhere.}$$

3.2 Generative Adversarial Networks

This is the main section of this chapter. Its purpose is to formulate a Generative Adversarial Network (*GAN*) model and to give a theoretical analysis of what it does. Its first apparition was in [1] by the hand of Goodfellow (2014). We will follow this paper in order to show its theoretical behaviour.

The main objective of a *GAN* model is, as its name says, to generate an specific output. When we face a generative problem what we are saying is that we aim to estimate a join distribution function of the form $P(X, Y)$, where X is the dataset and Y are the targets, or labels. The examples we have seen until now were all examples of discriminative models. A discriminative model can be seen as a model whose objective is to estimate a conditional distribution $P(Y \mid X)$. This kind of models have been extremely useful for image generation, and it may be the model on which the most significant advances this recent yeas. In order to just show the potential of this models let us show the following figure.

Figure 3.1: Hyperrealistic fake images generated by NVIDIA, from [17]

Now, retaking the metaphor we gave in the introduction of this chapter, we will define a model with two MLP networks: one will play the role of the cop, and the other as the criminal. Given that we aim to *generate* fake inputs it is this last one on which we are interested. Let us introduce them as follows:

Discriminador

Let $D(x, \theta_d)$ be a differentiable function expressed as a MLP network with parameters θ_d. Usually when working with *GAN* models we are going to treat with images, hence x will be a vectorized image such that $x \in \mathbb{R}^n$ for some n. We will name this function **Discriminator**. Its output will be a scalar that lies on $[0, 1]$. This scalar can be seen as the probability of x coming from the real distribution $\mathbb{P}_r$ rather than from the distribution that we are going to generate.

Generator

Let $G(z, \theta_g)$ be a differentiable function expressed as a MLP network with parameters θ_g. Here, the input z will be a noise variable on which we will define some prior $p_z(z)$ (we will always consider this z to follow a normal distribution, so $z \sim \mathbb{N}$). This function is known as **Generator**. The objective of this function G is to reproduce an estimation $\mathbb{P}_g = \hat{\mathbb{P}}_r$ over the real data distribution $\mathbb{P}_r$. In simpler words, we aim to get our generator to *learn* a probability distribution function $\mathbb{P}_g$ such that $\mathbb{P}_g \approx \mathbb{P}_r$.

Now that we have presented the two agents of our model it is time to explain how they interact, or specifically, how they compete. The acronym of a *GAN* model includes the word *Adversarial*, that is why we say compete, and that is what we are going to show. On one side we have said that our discriminator function D returns the probability of x coming from $\mathbb{P}_r$ instead of $\mathbb{P}_r$. So, given $x \sim p_r(x)$, where $p_r(x)$ denotes here the distribution function of the real data, we will be interested in our discriminator to maximize $\mathbb{E}_{x \sim p_r(x)}[\log D(x)]$. By the other side, given what we will call a *fake* observation, $G(z)$ with $z \sim p_z(z)$, our discriminator is supposed to output a low probability for it being real, what is the same to say that our discriminator will like to maximize $\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$.

If we look now to the generator, we will see that its objective is the opposite to the discriminator for the second part. Our generator function is interested in making the discriminator function D to output a close to 1 value. Saying this is equivalent to say that our generator will try to minimize $\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$.

Wrapping all this together we arrive to a *non-cooperative two player zero-sum game* in which the **Generator** is trying to fool the **Discriminator**. Making use of the min-max formulation for a zero sum non-cooperative two players game we formalize this as:

$$\min_{\theta_g} \max_{\theta_d} L(G, D) = \mathbb{E}_{x \sim p_r(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{3.7}$$

Or what is the same:

$$\min_{\theta_g} \max_{\theta_d} L(G, D) = \mathbb{E}_{x \sim p_r(x)}[\log D(x)] + \mathbb{E}_{x \sim p_g(x)}[\log(1 - D(x))] \tag{3.8}$$

Essentially, what we are going to show is that effectively the generator G ends up defining a distribution function p_g over the fake observations $G(z)$ obtained with $z \sim p_z$. Now we will give a theoretical analysis of this. We want to make clear that this is, indeed, a theoretical analysis, and it means that we are not going to consider this model

to be parametric but we are going to give a convergence analysis in a density probability functions space. We are going to start by giving the following proposition:

Proposition 3.5. *Given a fixed generator G, the optimal discriminator D is:*

$$D_G^*(x) = \frac{p_r(x)}{p_r(x) + p_g(x)} \tag{3.9}$$

Where p_r denotes the real distribution of our data and p_g the estimation from G.

Proof. Given any generator G the goal of the discriminator D is to maximize $L(G, D)$.

$$L(G, D) = \int_x p_r(x) \log(D(x)) + p_g(x) \log(1 - D(x))dx \tag{3.10}$$

Sine the integral is going trough every value of x we can omit it to find the maximum. Also we are interested in $D(x)$, so we propose the following:

Let $x^* = D(x)$, $A = p_r(x)$, $B = p_g(x)$. Therefore, and considering log as the logarithm in base e we have:

$$\begin{aligned} f(x^*) &= A \log x^* + B \log(1 - x*) \\ \frac{\partial f(x^*)}{\partial x^*} &= A\frac{1}{x^*} - B\frac{1}{1 - x^*}. \end{aligned} \tag{3.11}$$

Now we just have to set $\frac{\partial f(x^*)}{\partial x^*} = 0$ and undoing the change of names we did before we obtain:

$$D(x) = x^* = \frac{A}{A + B} = \frac{p_r(x)}{p_r(x) + p_g(x)}. \tag{3.12}$$

□

Notice that in 3.11 we are safely differentiating with respect to $D(x)$ because our discriminator do not need to be defined outside the union of the supports of p_r or p_x. Also we have to remember that D outputs a probability and this probability never takes the value of 0 but values that are close to it.

Now, considering this optimal discriminator D^* we define what we call the *virtual training criterion* for G as follows:

$$\begin{aligned} C(G) &= \max_D \mathbb{E}_{x \sim p_r(x)}[\log D^*(x)] + \mathbb{E}_{x \sim p_g(x)}[\log(1 - D^*(x))] \\ &= \max_D \mathbb{E}_{x \sim p_r(x)}\big[\log \frac{p_r(x)}{p_r(x) + p_g(x)}\big] + \mathbb{E}_{x \sim p_g(x)}\big[\log \frac{p_g(x)}{p_r(x) + p_g(x)}\big]. \end{aligned} \tag{3.13}$$

Theorem 3.6. *The global minimum of the virtual training criterion C(G) is achieved if and only if $p_g = p_r$. At that point, C(G) takes the value $-\log 4$.*

Proof. To begin with this proof we retake definition 3.3 from section 3.1.

$$
\begin{aligned}
D_{JS}(p_r \,||\, p_g) &= \frac{1}{2}D_{KL}(p_r \,||\, \frac{p_r + p_g}{2}) + \frac{1}{2}D_{KL}(p_g \,||\, \frac{p_r + p_g}{2}) \qquad (3.14)\\
&= \frac{1}{2}\Big(\int_{-\infty}^{+\infty} p_r(x)\log\big(\frac{2p_r(x)}{p_r(x)+p_g(x)}\big)dx + \int_{-\infty}^{+\infty} p_g(x)\log\big(\frac{2p_g(x)}{p_r(x)+p_g(x)}\big)dx\Big)\\
&= \frac{1}{2}\Big(2\log 2 + \int_{-\infty}^{+\infty} p_r(x)\log\frac{p_r(x)}{p_r(x)+p_g(x)}dx + \int_{-\infty}^{+\infty} p_g(x)\log\frac{p_g(x)}{p_r(x)+p_g(x)}dx\Big)
\end{aligned}
$$

The two integrals from the equation above can be seen as:

$$\int_{-\infty}^{+\infty} p_r(x)\log\frac{p_r(x)}{p_r(x)+p_g(x)}dx = \mathbb{E}_{x\sim p_r(x)}\big[\log\frac{p_r(x)}{p_r(x)+p_g(x)}\big] \qquad (3.15)$$

$$\int_{-\infty}^{+\infty} p_g(x)\log\frac{p_g(x)}{p_r(x)+p_g(x)}dx = \mathbb{E}_{x\sim p_g(x)}\big[\log\frac{p_g(x)}{p_r(x)+p_g(x)}\big] \qquad (3.16)$$

Thus, taking equations 3.14 and 3.15, 3.16 we obtain:

$$D_{JS}(p_r \,||\, p_g) = \frac{1}{2}\big(2\log 2 + C(G)\big). \qquad (3.17)$$

And finally:

$$C(G) = 2D_{JS}(p_r \,||\, p_g) - 2\log 2. \qquad (3.18)$$

Now we have expressed the virtual training criterion $C(G)$ in relation with the Jensen-Shannon Divergence. The minimum of $C(G)$ is achieved when the minimum of D_{JS} is achieved. Since we have observed in observation 3.4 D_{JS} reaches its minimum if and only if $p_r = p_g$ almost everywhere, then we have the same for $C(G)$. Also we know that this minimum for D_{JS} is 0 and therefore, looking at the equation 3.18 we obtain that the minimum for $C(G)$ takes the value $-2\log 2 = -\log 4$

□

With this last theorem we have shown that the training process of a *GAN* model, and in particular, trying to minimize the generator, is equivalent to minimize the *JS-Divergence* between the real data distribution function and our estimate distribution function. This lead us to a generator that can be represented as a distribution probability function, hence, given some input it will map it to an output that will follow the same distribution as the real data on which we have trained our model. We give the algorithm of this model in annex 7.2.

3.3 Problems with GANs

Even though GAN models have been a really powerful tool in order to solve the task of generate fake images than look very close to real ones, they have a very complicated way of training. We have defined the training of a GAN model as a simultaneous training between two models which compete with each other in a zero sum game. This will lead us to some of the problems that will be discussed below.

3.3.1 Gradient Descent techniques and Nash Equilibrium Problem

The idea of training a GAN model to the optimal model can be though to be the same as finding a *Nash equilibrium* for a zero sum non-cooperative game. The discriminator goal is to minimize the cost function $J^D(\theta^D, \theta^G)$ meanwhile, the generator is interested on minimize $J^G(\theta^D, \theta^G)$. The solution for this game shoud be some point $(\theta^{D*}, \theta^{G*})$. At this point either of the two cost functions J^D, J^G are minimum with respect to its associated parameters θ^D, θ^G respectively.

In [6] the authors discussed about how the modifications over the parameters trough the training process could affect the relations between the two cost functions J^D and J^G. For doing so they present the following example that shows the non-convergence of a zero-sum game in a specific scenario with $J^D = f_1$ and $J^G = f_2$:

Example 3.7. Let $f_1(x) = xy$ and $f_2(y) = -xy$. Suppose that f_1 is the cost function for the first player which is trying to minimize with respect to x in a two zero sum non-cooperative game. Suppose that f_2 is the analogous cost function for the second player that would be interested in minimizing it with respect to y. Then:

$$\frac{\delta f_1}{\delta x} = y \quad \frac{\delta f_2}{\delta y} = -x$$

If we now follow the method of Gradient Descent with a learning rate ϵ we have to update $x = x - \epsilon$ and $y = y + \epsilon$ in one iteration. The paper states that at some point the gradient descent algorithm lead us to a stable orbit rather than converging to the equilibrium point $x = y = 0$.

This example shows that finding a *Nash equilibrium* with gradient descent is not assured. Therefore, and keeping in mind that when we work with GANs we are working with complex cost functions that are not convex, with continuous parameters and high-dimensional parameter spaces, make this problem even bigger.

3.3.2 Perfect Discriminator Problem

Before presenting this problem we have to make clear that we are going to introduce some mathematical concepts without going to delve into them. Also we are going to give an informal definition for one of them. We will do so in order to not exceed something that is nothing more than an example.

Definition 3.8. *Given a function f whe define the **support** of f as:*

$$Supp(f) = \{x \mid f(x) \neq 0\}$$

Definition 3.9. *Let $D : X \to [0, 1]$ be a differentiable function expressed as a MLP and two different probability distribution functions $\mathbb{P}_r$ and $\mathbb{P}_g$ with disjoint supports. We say that D is a perfect discriminator if:*

$$\mathbb{P}_r[D(x) = 1] = 1 \text{ and } \mathbb{P}_g[D(x) = 0] = 1$$

i.e. the discriminator always take the value 1 when evaluated on a set containing $Supp(\mathbb{P}_r)$ and 0 if evaluated on a set containing $Supp(\mathbb{P}_g)$.

Now we have defined what a *perfect discriminator* is we have to give the intuition of why is it a problem. If during the training process we encounter an optimal discriminator which is perfect (also it is said *discriminator whith accuracy 1*) the gradients generated during back-propagation will be 0 almost everywhere, and therefore, there will not be changes on the performance for the generator.

In [8] the authors discuss about the perfect discriminator problem in a context of manifold learning. We are going to show what happens when the supports of our two $\mathbb{P}_r$ and $\mathbb{P}_g$ are disjoint. In order to do so we first have to define what a **manifold** is. Since we are looking for showing this as an example we are going to introduce the concept in an intuitive way to avoid the formal construction of what a manifold is.

Definition 3.10. *(Intuitive definition) Given a topological space X that can be locally identified with a Euclidian space with dimension m we say that X is a n-**manifold**.*

Now we can present and prove the following theorem: Ojo

Theorem 3.11. *Given two probability distributions $\mathbb{P}_r$ and $\mathbb{P}_g$ such that their supports are contained on two disjoint compact subsets M and Q respectively, there exists a smooth optimal discriminator $D^* : X \to [0,1]$ that is perfect and $\nabla_x D^*(x) = 0,\ \forall x \in M \cup Q$.*

Proof. Since M and Q are two disjoint compact sets we can set ϵ to be de distance between them such that $0 < \epsilon = d(M,Q)$. Now we can define:

$$\hat{M} = \{x \mid d(x,\, M) \leq \epsilon/3\}$$
$$\hat{Q} = \{x \mid d(x,\, Q) \leq \epsilon/3\}$$

Now, because of the definition of ϵ we have that $\hat{M}$ and $\hat{Q}$ are two disjoint compact sets. Therefore, by *Urysohn's Lemma* there exists a smooth function $D^* : X \to [0,1]$ such that $D^*(x) = 1\ \forall x \in M$ and $D^*(x) = 0\ \forall x \in Q$. If we now look at the equation that the discriminator D is trying to maximize:

$$\mathbb{E}_{x \sim p_r(x)}[\log D(x)] + \mathbb{E}_{x \sim p_g(x)}[\log(1 - D(x))]$$

We can observe that for the function D^* we notice that is an example of a *perfect discriminator* and therefore the above function always take the value 0.

Furthermore, if we take $x \in M \cup Q$ and we assume that $x \in M$ -the proof is analogous if we assume $x \in Q$- we know that there exist a neighbourhood B of x such that $B = B(x, \epsilon/3)$ on which $D^*(x)$ is constant $\forall x \in B$. Thus, $\nabla_x D^*(x) = 0,\ \forall x \in M \cup Q$ concluding the proof.

□

3.3.3 Mode Collapse Problem

This problem is the last we are going to discuss and also is the problem we are interested to fix in this work. The problem of Mode Collapse is originated when a *GAN*

model is asked to generate different classes of objects, and takes place during the training process of the Generator. The problem consist in that the generator tends to emit high probability samples but just for a few classes, and then it reaches its objective of fooling the discriminator without the necessity of generate what we can name *difficult classes*. To make this concept clear we show a simple example:

Imagine we are interested in generating only 2 classes of images A and B. Assume that the images from class A have a simple representation meanwhile the elements from B are harder to replicate. The generator, trough its process of training, will tend to produce much more elements from the class A than from B and therefore the results will not be the desired.

In the following chapter we aim to propose a variation of the standard *GAN* model that fixes this problem. We will see that during the model construction we will face too the *Perfect Discriminator* problem and we will give a solution for it too.

Chapter 4

TripletGAN

This chapter will be exclusively focused on the model we have worked on. Here we aim to formally introduce the variation of the standard *GAN* model we have chosen to reproduce in the practical part. This evolved *GAN* model takes the name of *TripletGAN* model. It was introduced in [7] and it can be thought as a simple *GAN* model on which we change the *discriminator* function [1] for a function with the name of *Triplet Loss* function. As we mentioned in the final part of the last chapter this variant aims to fix the *Mode Collapse* problem, and we will see how later on. Following the logic path we are going to give first the definition of this new cost function and our motivation to use it. Then, we will enter in detail on the *TripletGAN* model itself.

4.1 Triplet Loss function

The idea of this cost function, *Triplet Loss*, has its origin in [10]. In that paper the authors discussed about a *nearest-neighbour classification* problem, which is a special kind of unsupervised learning problem. Even though they did not use it in a Neural Network context, there are several authors [2] that use this function, and its variations, as cost functions for an **embedding**-designed neural networks.

Definition 4.1. *We say that $f : X \to \mathbb{R}^d$ is an embedding function that maps an input x into a d-dimensional Euclidean space.*

The motivation of an *embedding* is to map our input space X to a proper d-dimensional Euclidean Space in which $f(x) \in \mathbb{R}^d$ are *well represented*. This notion of *well represented* depend on each problem. In *natural language processing* embeddings are very popular. In those scenarios the usual practice is to get a vectorized representation of words in order to study the structure in an Euclidean space. We would expect words like *London* and *England* to be close in that space, for example.

[1] The *loss function* of the original discriminator was the *cross-entropy* between the empirical distribution and our estimated probability function

[2] Highlighting the work of [9]

Going back to the *Triplet Loss* function we want to point out the reason of the *Triplet* part in the *Triplet Loss* function name. When working with *Triplet Loss* we construct a specific kind of *triplets*. Those triplets are defined as follows.

Definition 4.2. *Given x_a, x_p and x_n tree points from X we can create a triplet $\mathcal{T}$ in the following way:*

$$\mathcal{T} = (x_a, x_p, x_n)$$

*Where x_a is called **anchor**, and x_p, x_n are called **positive** and **negative** respectively.*

Here we suppose that each element have a class. This *loss function* is specially useful when there are a lot of classes with few examples for class. The intuition here is to build those triplets such that the *anchor* x_a and the *positive* x_p will be the same class and the *negative* x_n will be from another class. One can see now why this is useful when having a lot of classes. The main objective here is to put together the same class points and moving far away the rest. Doing this defines a space with separated sets of points, each set for one class if the algorithm has done well its job.

Figure 4.1: Triplet Loss function intuition, from [10]

Hence, this is the scenario we are looking for:

$$|| f(x^a) - f(x^p) ||_2^2 + \alpha < || f(x^a) - f(x^n) ||_2^2$$

$\forall (f(x^a), f(x^p), f(x^n)) \in \mathcal{T}$.

Where f is an embeddig between X and $\mathbb{R}^d$ for some appropriate d, $|| \cdot ||_2$ is the Euclidean norm, $\mathcal{T}$ is the set of all possible triplets in the training set. The element $\alpha \in \mathbb{R}$ is a margin that is imposed between two pairs in order to be less strict with the *positive* pair distance.

Thus, the loss fuction is defined as:

$$L = \sum_i^n \Big[|| f(x_i^a) - f(x_i^p) ||_2^2 - || f(x_i^a) - f(x_i^n) ||_2^2 + \alpha \Big]_+ .$$

Where $\big[\cdot \big]_+$ is the same as $\max(\cdot, 0)$.

One thing more to point out of this model is one of its most important problems. This problem is that, sometimes, when given random triplets, the model just consider the easiest *positives* and the easiest *negatives*. That is, the losses may become 0 very soon and

that prevent the gradients to update the parameters. This is usually solved by considering only what is known as *hard-triplets*. Those are, given an *anchor*, we take as *positive* the most different element of its class and for *negative* the most similar from a different one.

Integral Probability Metrics

This metric between two probability distributions is introduced in [16]. However, the general definition of a IPM that we are going to introduce is not a metric but a pseudo-metric, that is, the property of $d(P, Q) = 0$ if and only if $P = Q$ is not assured. Our goal in this subsection is to prove that in the case we are going to study the IPM is, indeed, a metric.

Definition 4.3. *Given $(X, \mathcal{A})$ a probability space. Let $\mathcal{F}$ be the function set of measurable functions defined on X, and P and Q two arbitrary distributions in $(X, \mathcal{A})$, the **IPM** distance [3] between them is defined as:*

$$d_{\mathcal{F}}(P, Q) = \sup_{f \in \mathcal{F}} \left| \int f dP - \int f dQ \right|$$

4.2 TripletGAN Model

Now we have defined everything we need to set the theoretical framework for our model to do exactly what we want it to do, but before doing so, we are going to give the intuition behind combining this two models. As we have seen in Section X. this model aims to solve the *Mode Collapse* problem, and we will see why when explaining our combined *loss function*. We will start showing what will be the structure of our model. For doing so we will specify the structure that our generator and discriminator will have and how they interact with each other in this specific scenario.

Discriminator
Our discriminator will be still a differentiable function expressed as a MLP with parameter space Θ_f. In this specific case, this MLP will have the structure of an embedding. For getting this structure it suffices to set the *Triplet Loss* function as the new cost function of our previous discriminator. Also, in order to respect the notation that is introduced in the paper we are going to call the discriminator f.

Generator
Our generator takes the same structure that it has in the standard GAN model. That means that it will be a differentiable function expressed as a MLP with parameter space Θ_g that takes some noise input $z \sim \mathbb{N}$ and returns an element from the image space X. We will name it G.

Unlike the *Triplet Loss* case we will have now only two classes of interest. These classes will be the *true* class and the *fake* class. Following the paper the fake classes will be

[3] The *IPM* can be considered a distance just if we have considered before a proper set $\mathcal{F}$, if not, we can not say for sure that it is a *distance*

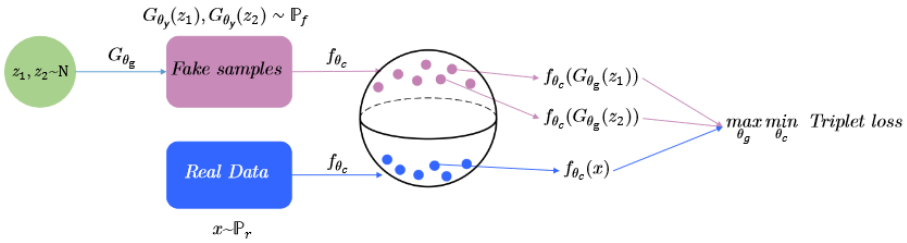


Figure 4.2: TripletGAN visual representation, from [7]

chosen to be the **anchor** and **positive** classes of our *triplets*, the true classes then will be the **negative** ones in those triplets. With all this information we are able now to formulate min-max problem similar to the one we defined for a classic GAN.

$$\min_{\Theta_g} \max_{\Theta_f} \mathcal{L}_t$$

Where:

$$\mathcal{L}_t = \mathbb{E}_{\substack{y \sim \mathbb{P}_r \\ z_1 \sim \mathbb{N}}} \mid\mid f(y) - f(G(z_1)) \mid\mid - \mathbb{E}_{z_1, z_2 \sim \mathbb{N}} \mid\mid f(G(z_1)) - f(G(z_2)) \mid\mid \tag{4.1}$$

Now we can see what we said about the *Mode Collapse* problem at the beginning of this section. If we look at equation 4.1 and specially focus on the *generator* objective, which is making $\mathcal{L}_t$ minimum. We see that under this scenario the generator not only needs to make $\|f(y) - f(G(z_1))\|$ small, just as in the standard *GAN* model, now we have also $\|f(G(z_1)) - f(G(z_2))\|$ and the generator is interested in making this large. For doing so it will try to explore the other classes options to give the most different output. Following this, it is logical to think that the generator will end up generating elements from a bigger variety than the standard *GAN*. The reason is that in order to minimize the *loss function*, the model is forced not only to make the outputs similar to the real ones but also because it is forced to generate very different outputs for the given one, by exploring the other classes.

If we now denote $S^n = \{x \in \mathbb{R}^{n+1} :\mid\mid x \mid\mid = 1\}$ as the n-dimensional sphere, $\mathcal{F} = \{f \mid f : R \to S^n, f measurable\}$ and $\mathcal{T} = \{\mid\mid f(x) - f(y) \mid\mid \mid f \in \mathcal{F}\}$ and finally we retake the definition of **IPM** on $(X, \mathcal{A})$ between two arbitrary distributions $\mathbb{P}$ and $\mathbb{Q}$ with respect to $\mathcal{T}$ we have:

$$\begin{aligned} d_{\mathcal{T}}(\mathbb{P}, \mathbb{Q}) &= \sup_{g \in \mathcal{T}} \left\{ \mathbb{E}_{(x,y) \sim \mathbb{P}} g(x,y) - \mathbb{E}_{(x,y) \sim \mathbb{Q}} g(x,y) \right\} \\ &= \sup_{f \in \mathcal{F}} \left\{ \mathbb{E}_{(x,y) \sim \mathbb{P}} \mid\mid f(x) - f(y) \mid\mid - \mathbb{E}_{(x,y) \sim \mathbb{Q}} \mid\mid f(x) - f(y) \mid\mid \right\} \end{aligned} \tag{4.2}$$

Let $\mathbb{P}$ be the independent join distribution of $\mathbb{P}_r$ and $\mathbb{P}_g$ where these two are the real distribution of our data and the distribution that our generator is estimating respectively,

and let $\mathbb{Q}$ be the independent join distribution of $\mathbb{P}_g$ and $\mathbb{P}_g$. Now it can be seen that minimizing the loss function 4.2 is equivalent to minimize the **IPM** between these two independent join distributions.

Before we can say that minimizing this last loss function implies making $\mathbb{P}_g$ to become equal to $\mathbb{P}_r$, we first have to prove that $d_{\mathcal{T}}(\mathbb{P}, \mathbb{Q})$ implies $\mathbb{P} = \mathbb{Q}$. As we stated at the introduction of this section, this property is not assured for a general **IPM**. We aim now to prove this implication in order to show that minimizing the Triplet Loss function for a *GAN* network yields the same result we had for a standard *GAN* model, which is making $\mathbb{P}_g \approx \mathbb{P}_r$.

Lemma 4.4. *Given S a measurable set, with $m(S) < \infty$ and $m(S) > 0$ we can represent it as the union of two disjoint measurable sets with positive measure. Here m denotes the Lebesgue measure.*

Proof. We are going to follow the proof given in [7], in which the author only consider the case $m = 1$ for it can be generalized. Once we have set $m = 1$ we consider

$$f : \mathbb{R} \to \mathbb{R}^+, \ f(x) = m(S \cap [-x, x]) \tag{4.3}$$

It is easy to see that f is continuous and $lim_{x\to\infty} f(x) = m(S)$. If now we consider some ϵ such that $0 < \epsilon < m(S)$ we can say that there exist some c such that $m(S \cap [-c, c]) = \epsilon$ using the fact of f being continuous. Then, $S \cap [-c, c]$ and $S \cap (\mathbb{R} \setminus [-c, c])$ are the two disjoint sets we wanted.

□

Theorem 4.5. *Suppose $\mathbb{P}$, $\mathbb{Q}$ are probability distribution functions over $\mathbb{R}^n$. Let*

- $\mathcal{F} = \{f \mid f : \mathbb{R}^n \to S^n, \ f \text{ measurable}\}$.
- $d_{\mathcal{T}}(\mathbb{P}, \mathbb{Q}) = \sup_{f\in\mathcal{F}} \{\mathbb{E}_{y\sim\mathbb{P}x\sim\mathbb{Q}} \ || f(x) - f(y) || - \mathbb{E}_{(x,y)\sim\mathbb{Q}} \ || f(x) - f(y) || \}$.

Assume that $p(x)$ and $q(x)$ are the densities of $\mathbb{P}$ and $\mathbb{Q}$ respectively, then:

$$d_{\mathcal{T}}(\mathbb{P}, \mathbb{Q}) = 0 \longleftrightarrow \mathbb{P} = \mathbb{Q}$$

Proof. The left to right implication is obvious so we are going to give the proof for the right to left one. We suppose that $\mathbb{P} \neq \mathbb{Q}$, therefore we just have to find one $f_0 \in \mathcal{F}$ such that the distance is not equal to 0.

We start by looking at $d_{\mathcal{T}}(\mathbb{P}, \mathbb{Q})$:

$$d_{\mathcal{T}}(\mathbb{P}, \mathbb{Q}) = \sup_{f\in\mathcal{F}} \left\{ \int_{\mathbb{R}\times\mathbb{R}} || f(x) - f(y) || q(x)p(y) \, dx \, dy - \int_{\mathbb{R}\times\mathbb{R}} || f(x) - f(y) || q(x)q(y) \, dx \, dy \right\} \tag{4.4}$$

From this equation we can derive to:

$$d_{\mathcal{T}}(\mathbb{P},\mathbb{Q}) = \sup_{f\in\mathcal{F}} \left\{ \int_{\mathbb{R}\times\mathbb{R}} || f(x) - f(y) || q(x)(p(y) - q(y))dx\, dy \right\} \tag{4.5}$$

Now, looking to this last equation and knowing that, since $p(x)$ and $q(x)$ are measurable, we can define $S_1 = \{p(y) - q(y) > 0\}$ and $S_2 = \{p(x) > 0\}$ knowing that both are measurable too. Now we are interested in those sets for reasons we will explain later:

First, since $p(x)$ and $q(x)$ are density functions we can say the following:

$$\int_{\mathbb{R}^2} p(x) - q(x)dx = \int_{p(y)-q(y)>0} p(x) - q(x)dx + \int_{p(y)-q(y)<0} p(x) - q(x)dx = 0 \tag{4.6}$$

That means that if $m(S_1) = 0$ we will have that $m(p(y) - q(y) < 0) = 0$ too, and then, we will be contradicting the fact that $\mathbb{P} \neq \mathbb{Q}$. The results for S_2 are analogous.

Now if $m(S_1 \cap S_2) = 0$ we can choose f_0 such that $f_0(x) = z_0$ for all $x \in S_1$ and $f_0(x) = -z_0$ for all $x \in S_2$, where $z_0 \in S^n$. For the rest of points we set $f_0(x) = 0$. This way, and retaking equation 4.5 we can set:

$$\begin{aligned}
&\int_{\mathbb{R}\times\mathbb{R}} || f(x) - f(y) || q(x)(p(y) - q(y))dx\, dy \\
=&\int_{S_1\times S_2} || f(x) - f(y) || q(x)(p(y) - q(y))dx\, dy \\
&\int_{S_2\times S_1} || f(x) - f(y) || q(x)(p(y) - q(y))dx\, dy \\
=&\int_{S_1\times S_2} 2q(x)(p(y) - q(y))dx\, dy \\
&\int_{S_2\times S_1} 2q(x)(p(y) - q(y))dx\, dy > 0
\end{aligned} \tag{4.7}$$

Note that $\|z_0\| = 1$. Equation 4.7 lead us to $d_{\mathcal{T}}(\mathbb{P},\mathbb{Q}) > 0$ which is a contradiction with our hypothesis. If we look now at the case where $m(S_1 \cap S_2) > 0$ we first need to use lemma 4.4 to guarantee the existence of two sets S_1^* and S_2^* such that $S_1 \cap S_2 = S_1^* \cup S_2^*$. If we now proceed just like before, and we set $f_0(x) = z_0$, $\forall x \in S_1^*$ and $f_0(x) = -z_0$, $\forall x \in S_s^*$ and $f_0(x) = 0$ for the rest of $x \in \mathbb{R}^n$, this will yield a similar result with the previous scenario. □

Before finishing we have to face a common problem when training GAN models with an *IPM*. This problem is known as the *perfect discriminator problem* and we have discussed about it in Chapter 3. In [7] the author claims that the supports of P_r and P_g are disjoint except for a 0 measurable set. Saying this is the same as say that the discriminator f is able to separate the samples of both distributions in base space without failing. If we now retake the loss function of our discriminator:

$$\mathcal{L}_f = \mathbb{E}_{\substack{y\sim\mathbb{P}_r \\ z_1\sim\mathbb{N}}} || f(y) - f(G(z_1)) || - \mathbb{E}_{(z_1,z_2)\sim\mathbb{N}} || f(G(z_1)) - f(G(z_2)) || \tag{4.8}$$

We can see that maximizing this is equivalent to the following: For the first term, and since our values of f lie on S^n, the function f has to map any point x_0 to its antipodal point in the n-sphere and for the second term would be maximized only if every point coming from P_g is send to a fixed point in S^n. That will lead us to the following optimal discriminator:

$$f(x) = \begin{cases} z_0 & x \sim \mathbb{P}_r \\ -z_0 & x \sim \mathbb{P}_g \end{cases} \tag{4.9}$$

In this situation we will have no gradients to update our generator G. The way of not letting this to happen is to do the same thing we did in Section 4.1. We have to change a bit our loss function to the following one:

$$\mathcal{L}_f = \mathbb{E}_{\substack{y \sim \mathbb{P}_r \\ x_1, x_2 \sim \mathbb{P}_f}} \Big[- \mid\mid f(y) - f(x_1) \mid\mid + \mid\mid f(x_1) - f(x_2) \mid\mid + c \Big]_+ \tag{4.10}$$

Now we have change the loss function we are ready to see if the problem remains or not. When minimizing the new loss function we now expect that the distance between the real embedding and the false embedding would be less than the distance between fake embeddings by a margin of c. Since we are in the n-sphere we can say that if c is less than π our discriminator would stop being a perfect discriminator but a mapping between two clusters in S^2, by the other side if c is greater than π this new loss function will perform just as the old one. Hence, setting a threshold $c < \pi$ solves the problem of the *perfect discriminator*.

One of the main differences between the classic GAN model and ours is that when training, if some triplets stop to pass gradient there may be other triplets that will do it. In classic GAN model if the loss exceeds the threshold it will stop to pass gradient to every sample coming from the same minibatch [4].

[4]Supposing that we are using Mini-Batch Gradient Descent, which is the usual

Chapter 5

Experiments

This last chapter is centered in the application of the *TripletGAN* model. As we have said in the previous chapter, we look for our algorithm to *generate* images, and this lead us to the following poblem: The way of quantify the success of a generator algorithm is usually complicated. For this reason we have decided for a qualitative evaluation. The first part of this chapter is focused in giving a brief technical resume of the computational framework we have worked on. In the other part we give the results of our models and we also present the data we have chosen to generate and why. The codes for this experimental part can be found in `https://github.com/ricdomi/TripletGAN`

5.1 Experimental Framework

For the experimental part we have chosen *Python* as the programming language. Despite of the fact that *R* is the most popular programming language in applied statistics we can say that *Python* ha gained a huge popularity inside the *Deep Learning* community, and one of the reasons may be the release of the *TensorFlow* [19] in 2015. This open source library is specialized in representing computations as *data flow* graphs and it introduces an special type of *object* which is the *Tensor* object. Also *TensorFlow* can run over multiple CPU's or GPU's. It is precisely this structure which enables *TensorFlow* to automatically compute gradients during the training process in parallel using the *Automatic Differentiation* method, which is extremely useful for us, and it reduces the computing time in a significant way.

Tensorflow

Although nowadays there are libraries that are capable of running on top of *TensorFlow*, such as *Keras* [18], which is more user-friendly, we have chosen to work with *TensorFlow* directly. This decision has been made following a personal preference.

Google Colab

As we said in Section 5.1 one of the advantages of *TensorFlow* is to be able to run using GPU's. As we do not have a GPU-based hardward we have chosen *Google Colab* as the

cloud environment to work on. The *Colab* name stands for *Google Colaboratory*. This cloud environment is *Python* supported, lets the user to freely use a GPU accelerator and it has the main libraries installed so the user only needs to import them.

5.2 Arquitecture

In this section we introduce our model in a practical way. Here we show with details the final structure of our networks (number of *layers*, *nodes* in each *layer*, etc.). It is worth to point that our model consist in two neural networks, so we will give the structure for both of them. We also present here the data we are going to work with. As it will be seen later, we have chosen two simple *datasets* that are commonly used for academic purposes. We will present first the *MNIST* images database, which can be though as the *'Hello, world!'* in the image recognition field. We will also use a similar dataset but with much complex images in order to deal with a more realistic problem: *FASHION-mnist*. Lastly, we will present the results of our application of a *TripletGAN* model when generating images from these two mentioned datasets.

Discriminator

The discriminator D of the model has been defined as a *MLP* model with 256, 512 and 1024 nodes for the first, second and third layer respectively. It gets as input a vectorized gray scale image of size 28×28 pixels, which is the same as saying that it gets a vector of dimension 784. The output of this model is an *embedding* of an image to a 4-dimensional space ($\mathbb{R}^4$). Hence, a 4-dimensional vector. We have used the *leaky ReLU* activation function for each hidden layer and the simple linear activation for the output layer.

Generator

The generator G has been defined as a *MLP* model with the discriminator inverse structure. It has also 3 hidden layers, but now with 1024, 512 and 256 nodes for the firs, second and third layer respectively. The input here is a *noise* vector z with dimension 128, and with $z \sim \mathcal{N}(0, 1)$. Lastly as we want to generate similar images as the real, the output of the generator must be the same size the real images are. Hence, the output of our generator should be a 784-dimensional vector[*]. The activation function also has been the *leaky ReLU* function except for the output layer, where we have used the $\tanh(\cdot)$ activation function. Here we have also applied a *batch normalization* on the third hidden layer.

The training process consists in 100.000 training steps considering each time a batch of size 128. The *loss functions* are the same as the loss functions $\mathcal{L}_f$ and $\mathcal{L}_G$ showed in Algorithm X. The optimizer we have used is the *Adam* optimizer, with the same learning rate and β_1 parameter for both, $lr = 5e - 4$ and $\beta_1 = 0.5$. Lastly, the *leaky ReLU* α value is set at 0.2.

5.3 Datasets and Results

MNIST

He have just said some properties of this *dataset* above. Let us resume them here:

Total number of observations: 70.000, splited in 60.000 for training and 10.000 for testing. Every element x is consider as a gray scale image with size 28×28 pixels. Every image x represents a hand-written digit between 0 and 9. Every label y is represented as a 10-dimensional vector where all the components are 0 except one, $y = [0, 1, 0, \cdots, 0]$ is the label for the '1' image, $y = [1, 0, 0, \cdots, 0]$ for the '0' digit an so on.

Now we present our generated images in comparison with the real images from *MNIST dataset*.

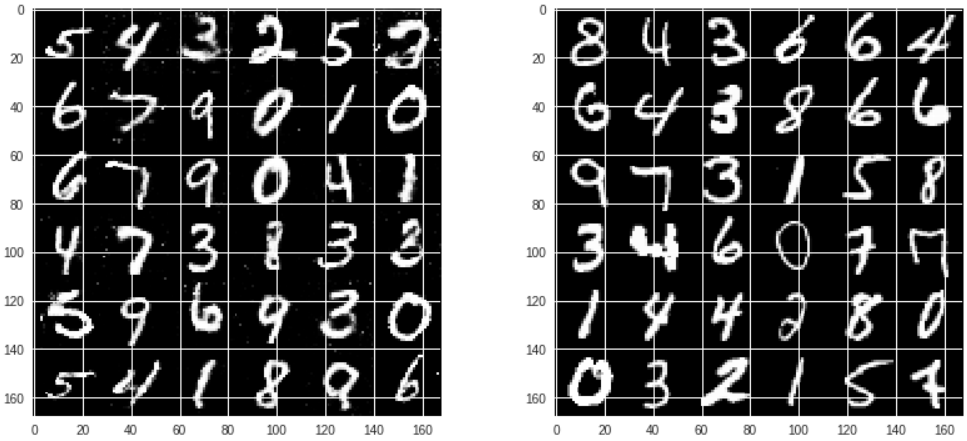


Figure 5.1: Fake images (*Left*) vs Real images (*Right*)

The following figure can corroborate in an intuitive way that our generator has actually well defined a distribution of our real images.

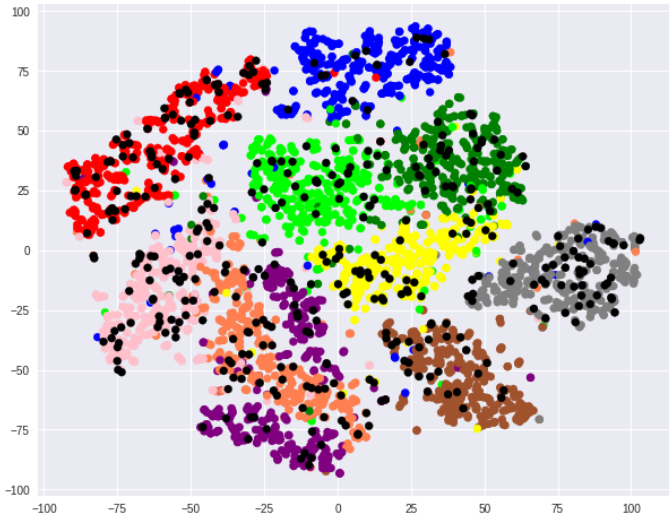


Figure 5.2: t-SNE visualization of real images (colorised) and fake images (black)

This figure is a visual representation of the output of a *t-SNE* algorithm over our generated images and real images. This algorithm is a popular *embedding* algorithm in *Machine Learning* community. It is used for visual representations in 2 or 3 dimensions of high dimensional data. We do not attempt to explain what this algorithm do right now but for those who want to know more [20] could be a good reference.

In Figure 5.2 we can see the different digits clusters separated by colors. The *black* dots are our generated images, and since they are generated, they are not labeled. From the image we see that our *black* dots are equally distributed along the different color clusters so we know that there is no *Mode Collapse* problem. We also can see the nonexistence of far, isolated dots. That means that our generator is correctly mapping its inputs with some outputs that follow the real data distribution.

FASHION-mnist

In order to avoid showing only the hand-written generated digits we have used a second, and a bit more complex, *dataset*. We have chosen the *FASHION-mnist* because it preserves the properties of our original*MNIST*, but now, instead of numbers, we have 10 different categories of clothing. We will resume its properties just like we did for *MNIST*:

The total number of observations is 70.000, splitted in 60.000 for training and 10.000 for testing. Every element x is consider as a gray scale image with size 28×28 pixels. Every image x represents a different clothing category: Shoes, shirts, etc. Every label

y is represented as a 10-dimensional vector where all the components are 0 except one, $y = [0, 1, 0, \cdots, 0]$ is the label for, for example, '*shirts*', $y = [1, 0, 0, \cdots, 0]$ for the '*shoes*' and so on.

Here we show a comparison between a set of real images and our set of generated images. We see that the generated ones are often more blurry . That could had been solved by augmenting the complexity of our model functions but that would have implied more training time and the definition of a different model. Since our intention was to show that the model worked not only on *MNIST* but in other datasets, we have decided to not improve it to do better in *FASION-mnist*.

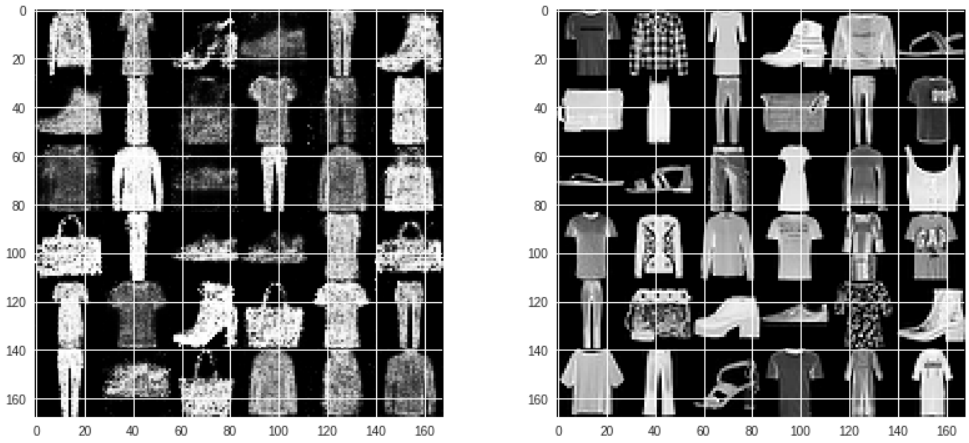


Figure 5.3: Fake images (*left*) vs Real images (*Right*)

Chapter 6

Conclusions

The main objective of this work was to analyze the *Generative Adversarial Model*, and particularly, one of its variants: *TripletGANs*. In order to achieve that objective we have introduced the topic from its basis. We started at general *Machine Learning* concepts to finally end on *TripletGAN*.

Along the work we have found some difficult situations. The most common of them was the lack formalities and strict definitions. We have noticed that the field of *Deep Learning* is closer to *engineering* than it is to *pure mathematics*. Concretely, there are a lot of publications that do not follow any rigorous methodology when proving their statements. Although it has not been the usual case when searching for the most relevant publications, it has been a problem when looking for more humble papers. The other main difficulty has been the fact of learning a new programming language such as *Python*, even though, learning it has been a great personal success.

As we have showed in the experimental part, we have achieved our goal of generating fake images. This goal, however, has been conditioned to work just with *gray scale* images. We were not fully satisfied with this, since our intention was to generate more realistic images (colourised objects, faces, etc.). However, we are now aware of the training times of more complex models, and that is why this problem is not so relevant for us now. This work has been the perfect intersection between working with a complex *Deep Learning* model thanks to do it with low resolution images.

This work can be extended in two ways. One of them is to continue the study of the *Triplet Loss* function with other *metrics*, not just the euclidean. There are also variants of the *TripletLoss* function that would be interesting to test. The other part focus the study not in the loss function, but in the *GAN* model itself. As we said through the work there exist wide variety of *GAN*-based models. Some of them are really interesting such as Deep Convolutional Generative Adversarial Networks (DCGANs), which differs from the usual *GAN* in the architecture of the networks, changing the *MLP* networks with *Convolutional Networks*. Also WGANs are a pretty interesting variation of the *GAN* model, and follow the same intuition that our variation: using an *IPM* distance known as *Wasserstein Distance*.

Chapter 7

Annex

7.1 Lemma and Proposition from Chapter 3

Theorem 7.1. *Lebesgue's Dominated Convergence Theorem. Let $\{f_n\}$ be a sequence of real-valued measurable functions on a measure space $(S, \mathcal{A}, \mu)$. Suppose that the sequence converges pointwise to a function f such that $\mid f_n(x) \mid \leq \mid g(x) \mid$ with g being an integrable function $\forall n$ and $\forall x \in S$. Then f is integrable and:*

$$\lim_{n\to\infty} \int_S \mid f_n - f \mid d\mu = 0$$

Lemma 7.2. *Any bounded, measurable sigmoidal function σ is discriminatory. In particular, any continuous sigmoidal function is discriminatory.*

Proof. **Let** $\sigma \in \mathcal{C}(I_m)$ be a function such that

$$\int_{I_m} \sigma(w^\top x + \theta) d\mu(x) = 0, \ \forall w \in \mathcal{C}(I_m) \text{ and } \forall \theta \in \mathbb{R}$$

Now, for $\lambda, \phi \in \mathbb{R}$ we define the function σ_λ as follows

$$\sigma_\lambda(\lambda(w^\top x + \theta) + \phi) \begin{cases} 1 & \text{for } \ w^\top x + \theta > 0 \text{ as } \ \lambda \to \infty, \\ 0 & \text{for } \ w^\top x + \theta < 0 \text{ as } \ \lambda \to \infty, \\ \sigma_\lambda(\phi) & \text{for } \ w^\top x + \theta = 0 \text{ for all } \ \lambda \end{cases}$$

We have to notice here that this family of functions has the same property that σ has, hence:

$$\int_{I_m} \sigma_{\lambda,\phi}((w^\top x + \theta) + \phi) d\mu(x) = 0, \ \forall w \in \mathcal{C}(I_m) \text{ and } \forall \theta \in \mathbb{R}$$

Observe that the family of functions $\{\sigma_{\lambda,\phi} \mid \lambda \in \mathbb{R}_{\geq 0}\}$ converges pointwise to $\Sigma : I_m \to \mathbb{R}$

$$\Sigma(x) \begin{cases} 1 & \text{for } w^\top x + \theta > 0, \\ 0 & \text{for } w^\top x + \theta < 0, \\ \sigma(\phi) & \text{for } w^\top x + \theta = 0 \end{cases}$$

when $\lambda \to \infty$.

We can see that the mentioned family not just converges pointwise to $\Sigma(x)$ but also is *dominated* by it. Given a family of real-valued functions $\{f_n\}$ on a measure space, what we understand for a function that is dominated by other integrable function g is that $\mid f_n(x) \mid \leq g(x)$ for all x.

We now let $H_{w,\theta} = \{x \mid w^\top x + \theta = 0\}$ be an affine hyperplane and also we let $HS_{w,\theta} = \{x \mid w^\top x + \theta > 0\}$ be an open half-space. If we define $g(x) = \max(1, \sigma(\phi))$ and since $\mid \sigma_\lambda(x) \mid \leq \mid g(x) \mid$ we can apply the Theorem 7.1 to get:

$$\begin{aligned} 0 = \lim_{\lambda \to \infty} \int_{I_m} \sigma_\lambda(x) d\mu(x) &= \int_{I_m} \lim_{\lambda \to \infty} \sigma_\lambda(x) d\mu(x) \\ &= \int_{I_m} \Sigma(x) d\mu(x) = \sigma(\phi)\mu(H_{w,\theta}) + \mu(HS_{w,\theta}) \end{aligned} \tag{7.1}$$

On the last equality we just have to remember the form of $\Sigma(x)$.

Now we define J_y as a compact interval containing $\{y^\top x \mid x \in I_m\}$ for a fixed $y \in \mathbb{R}^m$. Given that μ is a finite signed measure we can define F to be a bounded functional on $L^\infty(J_y)$. The intuition behind this last statement is that if we integrate over a compact with respect to a finite measure, we are not able to get an infinite result. This functional F has the following form

$$F(h) = \int_{I_m} h(y^\top x) d\mu(x)$$

If we consider now $h = \mathbb{1}_{[\theta,\infty)}$ the indicator function for the set $[\theta, \infty)$ we end up with:

$$F(h) = \int_{I_m} h(y^\top x) d\mu(x) = \sigma(\phi)\mu(H_{y,\theta}) + \mu(HS_{y,\theta}) = 0$$

.

That is because what we saw in 7.1 and also because the fixed y is a particularity of one $w \in \mathbb{R}^m$. In a similar way we will have also $F(h) = 0$ for $h = \mathbb{1}_{(\theta,\infty)}$. By linearity we can extend $F(h) = 0$ to any indicator function of any interval, therefore for any simple function[1] Given that we know that simple functions are dense in $L^\infty(J_y)$ we have $F = 0$.

In the original paper the author gives a particular pair of bounded measurable functions $s(u) = \sin(\alpha^\top x)$ and $c(u) = \cos((\alpha^\top x)$ such that

$$F(s + ic) = \int_{I_m} (\cos((\alpha^\top x) + i \sin((\alpha^\top x)) d\mu(x) = \int_{I_m} e^{i(\alpha^\top x)} d\mu(x) = 0$$

[1] A simple function f is a function of the form $\sum_i a_i X_{A_i}$, where $\mathcal{X}_{A_i}$ is known as *indicator function*

for all α. This is a specific choice of h using the Fourier transform of μ. This being 0 implies that $\mu = 0$ and, therefore σ is discriminatory. We are not going to detail this last part but we wanted to show that there exists a particular function h that implied $\mu = 0$, even though the theory for constructing this Fourier transform is beyond the scope of this project.

□

7.2 Algorithms

Require: Generator G_{θ_g}; noise z; Discriminator f_{θ_f}; dataset $\mathcal{S} = \{X_i\}$; learning rate ϵ; batch size N; threshold c.
Initialize: θ_g, θ_f;
while *TripletGAN has not converged* **do**
  Sample a minibatch from S, yield $x_i,\ i = 1 \cdots N$;
  Sample a minibatch from z, yield $z_i,\ i = 1 \cdots N$;
  Sample triplets from these two minibatchs, yield:
  $\mathcal{T} = \{\ \mathrm{t}_{i,j} \mid t_{i,j} = (G_{\theta_g}(z_i), G_{\theta_g}(z_j), x_i)\},\ i, j = 1 \cdots N$
  For simplicity we define:
  $\mathrm{d}_i = \|f_{\theta_f}(x_i) - f_{\theta_f}(G_{\theta_g}(z_i))\|$
  $\mathrm{d}_{i,j} = \|f_{\theta_f}(G_{\theta_g}(z_i)) - f_{\theta_f}(G_{\theta_g}(z_j))\|$
  And then:
  $\mathcal{L}_f(\theta_g, \theta_f) \leftarrow \frac{1}{N(N-1)} \sum_{i=1}^{N} \sum_{j=1}^{N} \min(d_i - d_{i,j},\ c)$
  $\theta_f \leftarrow \theta_f + \epsilon \nabla_{\theta_f} \mathcal{L}_f(\theta_g, \theta_f)$
  $\mathcal{L}_g(\theta_g, \theta_f) \leftarrow \frac{1}{N(N-1)} \sum_{i=1}^{N} \sum_{j=1}^{N} (d_i - d_{i,j})$
  $\theta_g \leftarrow \theta_g - \epsilon \nabla_{\theta_g} \mathcal{L}_g(\theta_g, \theta_f)$
end

Algorithm 2: TripletGAN Algorithm

for *number of training iterations* **do**
  for *k steps* **do**
    Sample a minibatch of m noise samples $\{z^{(1)}, \cdots, z^{(m)}\}$ from noise prior $p_g(z)$;
    Sample a minibatch of m examples $\{x^{(1)}, \cdots, x^{(m)}\}$ from data generating distribution $p_r(x)$;
    Update the discriminator by ascending its stochastic gradient:
    $\theta_d \leftarrow \theta_d + \epsilon \nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))\right]$
  end
  Sample a minibatch of m noise samples $\{z^{(1)}, \cdots, z^{(m)}\}$ from noise prior $p_g(z)$;
  Update the generator by descending its stochastic gradient:
  $\theta_d \leftarrow \theta_d - \epsilon \nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log(1 - D(G(z^{(i)})))\right]$
end

Algorithm 3: GAN Algorithm

Nota para parte '4.1 DEFINICION': https://math.stackexchange.com/questions/918509/more-general-definition-of-expected-value

Bibliography

[1] Goodfellow, Ian, et al. Generative adversarial nets. In *Advances in neural information processing systems*. 2014. p. 2672-2680. https://arxiv.org/abs/1406.2661

[2] Minsky, Marvin; Papert, Seymour A. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[3] Rosenblatt, Frank. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 1958, vol. 65, no 6, p. 386.

[4] Cybenko, George. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 1989, vol. 2, no 4, p. 303-314. https://pdfs.semanticscholar.org/05ce/b32839c26c8d2cb38d5529cf7720a68c3fab.pdf

[5] Hornik, Kurt; Stinchcombe, Maxwell; White, Halbert. Multilayer feedforward networks are universal approximators. *Neural networks*, 1989, vol. 2, no 5, p. 359-366.

[6] Salimans, Tim, et al. Improved techniques for training gans. En *Advances in Neural Information Processing Systems*. 2016. p. 2234-2242. http://papers.nips.cc/paper/6125-improved-techniques-for-training-gans.pdf

[7] Cao, Gongze, et al. TripletGAN: Training Generative Model with Triplet Loss. *arXiv preprint arXiv:1711.05084*, 2017. https://arxiv.org/abs/1711.05084

[8] Arjovsky , Martin; Bottou, Léon. "Towards principled methods for training generative adversarial networks". *arXiv preprint arXiv:1701.04862*, 2017. https://arxiv.org/abs/1701.04862

[9] Schroff, Florian; Kalenichenko, Dmitry; Philbin, James. Facenet: A unified embedding for face recognition and clustering. En *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015. p. 815-823. https://arxiv.org/abs/1503.03832

[10] Weinberger, Kilian Q.; Blitzer, John; Saul, Lawrence K. Distance metric learning for large margin nearest neighbor classification. *Journal of Machine Learning Research*, 2009, vol. 10, no Feb, p. 207-244. http://www.jmlr.org/papers/v10/weinberger09a.html

[11] Samuel, Arthur L. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 1959, vol. 3, no 3, p. 210-229.

[12] Mitchell, Tom M., et al. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 1997, vol. 45, no 37, p. 870-877.

[13] Goodfellow, Ian, et al. *Deep learning*. Cambridge: MIT press, 2016.

[14] Sokal, Alan. *Handout: 6, The Hahn-Banach Theorem and the Duality of Banach Spaces*, Functional Analysis. University College London, London [on line] `http://www.ucl.ac.uk/~ucahad0/`

[15] Simovici, Dan. *Mathematical Analysis for Machine Learning and Data Mining*. World Scientific Publishing Co., Inc., 2018, 895–898.

[16] Müller, Alfred. Integral probability metrics and their generating classes of functions. *Advances in Applied Probability*, 1997, vol. 29, no 2, p. 429-443.

[17] Karras, Tero, et al. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017. `https://arxiv.org/abs/1710.10196`

[18] Chollet, Francois. *Deep learning with python*. Manning Publications Co., 2017. `http://faculty.neu.edu.cn/yury/AAI/Textbook/Deep%20Learning%20with%20Python.pdf`

[19] Abadi, Martín, et al. Tensorflow: a system for large-scale machine learning. En *OSDI*. 2016. p. 265-283. `https://www.tensorflow.org/`

[20] Maaten, Laurens van der; Hinton, Geoffrey. Visualizing data using t-SNE. *Journal of machine learning research*, 2008, vol. 9, no Nov, p. 2579-2605.`http://www.jmlr.org/papers/v9/vandermaaten08a.html`