POLITECNICO DI TORINO

Master degree course in Communications and computer networks engineering

Master Degree Thesis

A review of deep learning on graphs

Supervisor
prof. Enrico Magli

Candidate
Andrea Tirelli
matricola 243905

December 2018

Abstract

The topic of this thesis is Deep learning on graphs. Deep learning is the subfield of machine learning studying algorithms and models constituted by several layers of functions learning in an increasingly level of abstraction the representation of a concept. *On graphs* refers to the application of this class of methodologies to data that are inherently represented as a graph structure, requiring ad hoc strategies and theoretical machinery to perform tasks of various nature over them. In particular, this work studies, from a compilative point of view, a relatively recent model of deep learning called Generative Adversarial Network (GAN). GANs are constituted by two neural networks, the most common form of deep learning, trying to beat each other in an optimization problem, in which one network, the generator, tries to deceive the other, the discriminator, producing examples that look similar to those belonging to a dataset of interest. At the end of the training of the networks, the generator should have learned the probability distribution of the dataset, allowing the algorithm to produce new, never seen, realistic examples. The first two chapters of this thesis are devoted to the description of these concepts and to the study of the best strategies allowing a smooth learning process. In the following two chapters, various architectures are described, referring to specific domains of application of GANs, with the main focus over the problems that arise when the previously explored concepts are applied to graph structured data, studying which are the currently available solutions in the literature.

Contents

Chapter 1

GAN fundamentals

1.1 Introduction

In the machine learning framework, a generative model is a mathematical model capable of learning the probability density function generating a dataset of interest. This allows to perform any kind of statistical operation on the dataset, like: regression, classification, missing data completion and so on. In particular, Generative Adversarial Networks (GANs) are used to create previously unseen samples from the distribution. This is done through an unsupervised learning process consisting in a min-max game between two learning models: generator and discriminator. The generator is the real generative part of the algorithm, that will be able to produce new samples. Its training consists in deceiving the discriminator, producing samples which are increasingly more realistic (similar to those belonging to the dataset). The discriminator must decide whether these samples are similar enough to the real samples taken from the dataset. The feedback from the discriminator allows the generator to modify its parameters in order to accomplish its task increasingly better, until the discriminator will not be able to distinguish the samples created by the generator from those belonging to the dataset (in a certain statistical sense). At the end of this process, the generator will be ready to produce new realistic samples. In the following, it will be assumed that both generator and discriminator are neural networks, unless otherwise stated. This mechanism finds use in a variety of domains, especially in multimedia. Examples of applications ad related architectures will be addressed in the following chapters.

1.2 GAN definition

Given a n-dimensional dataset, let us define $\mathbf{x}$ as a n-dimensional sample from the dataset. The target of the GAN is to find the prior distribution p_{data} generating the dataset, which is $\mathbf{x} \sim p_{\text{data}}$. To accomplish the task, the following architecture are employed:

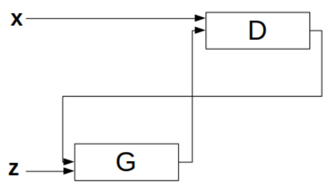

Figure 1.1: Training architecture

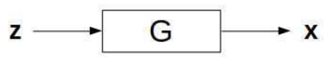

Figure 1.2: After training architecture

After the training (Figure 1.2), the Generator should be able to produce a new sample $\mathbf{x} \sim p_{\text{data}}$ receiving as an input a Gaussian random vector $\mathbf{z}$ (whose distribution is denoted by $p_{\mathbf{z}}(\mathbf{z})$). Actually, what is found is typically not exactly p_{data}, but rather an approximation, which in the following will be referred to as p_{g}. A Gaussian random variable is chosen as input because it is the distribution with maximum entropy, which means the least possible amount of information is assumed about the generating distribution, because typically one does not know anything about it. The role of the training phase (Figure 1.1) is to find the differentiable function $G(\mathbf{z}, \theta_{\text{g}})$ (where θ_{g} represents the set of parameters defining the neural network) that makes this possible. First of all, the discriminator function $D(\mathbf{x}, \theta_{\text{d}})$ (where θ_{d} represents the set of parameters defining the neural network) receives as input an n-dimensional sample, which is either from the dataset or produced by the generator; then it outputs the probability (just a simple numerical variable) that the sample belongs to p_{DATA}. As a consequence, the discriminator must be trained to minimize the output when the input comes from the generator, and to maximize the output when the input comes from the dataset. In the same way, the generator must be trained to maximize the output of the discriminator (when the discriminator receives the generator output), which gets inside the generator through the feedback mechanism (in terms of cost function). Therefore, ascending/descending the gradients of these quantities with respect to generator and discriminator parameters, will train the two neural networks to accomplish the task.

1.2.1 Formalization

Formally, this is equivalent to the min-max problem:

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})} \log D(\mathbf{x}) + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})} \log\left(1 - D(G(\mathbf{z}))\right) \tag{1.1}$$

where $\mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})} \log D(\mathbf{x})$ corresponds to the discriminator maximizing the probability of dataset samples and $\mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})} \log\left(1 - D(G(\mathbf{z}))\right)$ corresponds at the same time to the discriminator trying to minimize the probability of generator samples and to the generator trying to maximize it. Intuitively, this should led to a generator capable of producing new samples according to the distribution p_{data}, but currently there is no guarantee that the min-max problem admits as an optimum the $G(\mathbf{z}, \theta_{\text{g}})$ such data $p_{\text{g}} = p_{\text{data}}$, neither that the problem admits any optimum actually.

1.2.2 Optimal discriminator

To prove that the optimum exists and that it is the expected one, the general expression of the optimal discriminator is found, for a given $G(\mathbf{z}, \theta_{\text{g}})$. First of all, the object of the optimization can be expressed as:

$$\begin{aligned}
&\mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})} \log D(\mathbf{x}) + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})} \log\left(1 - D(G(\mathbf{z}))\right) \\
&= \int_{\mathbf{x}} p_{data}(\mathbf{x}) \log D(\mathbf{x}) \mathrm{d}\mathbf{x} + \int_{\mathbf{z}} p_{\mathbf{z}}(\mathbf{z}) \log\left(1 - D(G(\mathbf{z}))\right) d\mathbf{z} \\
&= \int_{\mathbf{x}} p_{data}(\mathbf{x}) \log D(\mathbf{x}) d\mathbf{x} + \int_{\mathbf{x}} p_g(\mathbf{x}) \log\left(1 - D(\mathbf{x})\right) d\mathbf{x} \\
&= \int_{\mathbf{x}} p_{data}(\mathbf{x}) \log D(\mathbf{x}) d\mathbf{x} + p_g(\mathbf{x}) \log\left(1 - D(\mathbf{x})\right) d\mathbf{x}
\end{aligned} \tag{1.2}$$

Referring to the previous expression as $V(D, G)$, the original problem:

$$\min_G \max_D V(D, G) \tag{1.3}$$

can be stated as:

$$\max_D V(D) \tag{1.4}$$

because $G(\mathbf{z}, \theta_{\text{g}})$ is assumed to be fixed. Using the following notation: $D(\mathbf{x}) = y$ $p_{data}(\mathbf{x}) = a$ $p_g(\mathbf{x}) = b$, the maximization can be written as:

$$\max_y a \log(y) + b \log(1 - y) \tag{1.5}$$

where the integral can be neglected because the dependence on $\mathbf{x}$ is lost. Imposing the first derivative to zero the problem becomes:

$$\frac{a(1-y) - by}{y(1-y)} = 0 \tag{1.6}$$

whose solution is:

$$y = \frac{a}{a+b} \tag{1.7}$$

or equivalently:

$$D_G^*(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})} \tag{1.8}$$

where the notation D_G^* stands for the optimal discriminator given a certain $G(\mathbf{z}, \theta_\mathrm{g})$.

1.2.3 Cost function minimization

Since the expression of the optimal discriminator has been found, the optimization problem now depends on $G(\mathbf{z}, \theta_\mathrm{g})$ only, so that it can be formulated as:

$$\min_G C(G) \tag{1.9}$$

where $C(G)$ is $V(D, G)$ with D equal to the optimal discriminator D_G^*, so that:

$$\begin{aligned} C(G) &= \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})} \log D_G^*(\mathbf{x}) + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})} \log\left(1 - D_G^*(G(\mathbf{z}))\right) \\ &= \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})} \log D_G^*(\mathbf{x}) + \mathbb{E}_{\mathbf{x}\sim p_g} \log\left(1 - D_G^*(\mathbf{x})\right) \\ &= \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})} \left[\log \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}\right] + \mathbb{E}_{\mathbf{x}\sim p_g} \left[\log \frac{p_g(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}\right] \end{aligned} \tag{1.10}$$

Now it is possible to show that the original min-max problem, which is now a minimum one, admits the optimum that was expected. To this purpose, one must first introduce two similar concepts that give a measure of the similarity between probability distributions. Let $P(x)$ and $Q(x)$ be two distribution over the same random variable x. Their Kullback-Leibler divergence is defined as:

$$KL(P\|Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right] \tag{1.11}$$

which has the property of being non negative and equal to zero if and only if $P(x) = Q(x)$, but it is not symmetric, so it can not be interpreted mathematically as a distance between distributions. Similarly, the Jensen-Shannon divergence of $P(x)$ and $Q(x)$ is defined as:

$$JSD(P\|Q) = \frac{1}{2}KL(P\|M) + \frac{1}{2}KL(Q\|M) \tag{1.12}$$

where $M = \frac{P+Q}{2}$. From the properties of Kullback-Leibler divergence, it can be showed that also the Jensen-Shannon divergence is non negative and equal to zero if and only if $P(x) = Q(x)$. A relationship between $C(G)$ and the Jensen-Shannon divergence of $p_{data}(\mathbf{x})$ and $p_g(\mathbf{x})$ can be derived now:

$$\begin{aligned}
C(G) &= \\
&= \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\log \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}\right] - \log\frac{1}{2} + \log\frac{1}{2} + \\
&\quad \mathbb{E}_{\mathbf{x}\sim p_g}\left[\log \frac{p_g(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}\right] - \log\frac{1}{2} + \log\frac{1}{2} \\
&= \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\log \frac{p_{data}(\mathbf{x})}{\frac{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}{2}}\right] + \mathbb{E}_{\mathbf{x}\sim p_g}\left[\log \frac{p_g(\mathbf{x})}{\frac{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}{2}}\right] + 2\log\frac{1}{2} \\
&= KL\left(p_{data}\|\frac{p_{data}+p_g}{2}\right) + KL\left(p_g\|\frac{p_{data}+p_g}{2}\right) - \log 4 \\
&= 2JSD(p_{data}\|p_g) - \log 4
\end{aligned} \tag{1.13}$$

This is a fundamental result, because the minimum of this expression is achieved when $JSD(p_{data}\|p_g)$ is minimum. But as said before, this happens only if the two distribution are equal, which means that the problem is solved only if $p_{data} = p_g$. This is exactly the result expected at the formulation of the min-max problem: minimizing and maximizing properly the cost functions, one gets an optimum generator that solves the problem only when it has learned a generating distribution which is equal to the generating distribution of the dataset.

1.2.4 Training algorithm

Finally, it can be showed that, given enough capacity to generator and discriminator, the following algorithm performs the optimization in (1.1): for k times, train the discriminator sampling m times both $p_z(z)$ and $p_{data}(x)$ and ascend the stochastic gradient

$$\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^{m}[\log D(\mathbf{x}^i) + \log(1 - D(G(\mathbf{z}^i)))] \tag{1.14}$$

then train the generator sampling m times from $p_z(z)$ and descend the stochastic gradient

$$\nabla_{\theta_g}\frac{1}{m}\sum_{i=1}^{m}[\log(1 - D(G(\mathbf{z}^i)))] \tag{1.15}$$

This is the algorithm proposed in the original article [6], where k must be set empirically. The choice of k is particularly critical, as it will be shown in the following paragraph, because the discriminator must not be trained too much in order to have an effective training on the generator too.

1.2.5 -log(D) option

In the same article, it is suggested that this problem may be partially solved by substituting the cost function in (1.15) with:

$$-\log(D(G(\mathbf{z}^i))) \tag{1.16}$$

which should improve the performances in the early stage of the training, when it is easy for the discriminator to distinguish the fake samples from the real ones. Again, in the following paragraph it will be shown that this does not actually solve the problem, which requires instead a more radical modification of the cost function.

1.3 Training difficulties

Despite the success of the original GAN architecture as it is described in the previous paragraph, its training requires to pay great attention, because it turns out that a wrong choice of the parameter k leads to an unsuccessful training of the generator. This paragraph is intended to explain why this happens, even in the case in which the cost function is modified accordingly to (1.16), and to provide a first hint to get new strategies to solve the difficulties in the training phase, which will be more deeply investigated in Chapter 2. As said at the end of the previous paragraph, the problem lies in the training of discriminator. In fact, one may think that training it as much as possible should guarantee the expected approach to the optimum according to optimization formulations laid out previously. But this is not actually the case, because it turns out that the more one gets near to the optimal discriminator, the less the generator will learn. The alternative is then avoiding to reach the optimum discriminator, but in this way the min-max problem will not be solved, and as a consequence the condition $p_{data} = p_g$ will not be reached.

1.3.1 Disjoint distributions

To understand why this happens and if this is always true, some considerations have to be done on the dataset distribution p_{data} and the generator distribution p_g. A first case of interest is when they have disjoint supports, as one may approximately expect when the generator has not been trained sufficiently, so that its generating distribution is still very far from the dataset one. In this condition, it is quite reasonable that the generator may be very proficient in distinguishing the samples from the two distributions. In particular a discriminator is said to have accuracy 1, when it outputs 0 if the sample lies in the support of p_g and outputs 1 in the support of p_{data}. So essentially is a discriminator that never commits an error of classification. It can be demonstrated that in the case under exam, so distributions with disjoint supports, there always exist an optimal smooth discriminator D^* such that it has accuracy 1 and $\bigtriangledown_x D^*(x) = 0$ for any x in the union of distributions supports. This is one of the two theorems that in [8] are called "Perfect discrimination theorems".

1.3.2 Not continuous distributions

Before evaluating its consequences, another scenario of interest is taken into account, leading to a similar discrimination theorem. This is the case of p_{data} and p_g having not absolutely continuous distributions. First of all, a random variable X is said to be absolutely continuous if given a set A with measure zero, then $P(X \in A) = 0$ (in the following, the expression "continuous" will be used instead of "absolutely continuous"). It can be showed that if the support of the distribution of a random variable lies on a low dimensional manifold, the random variable can not be continuous. This is typically the case for the distributions of interest. In fact, it is widely assumed to be true that most of the data of interest in machine learning have their probability mass highly concentrated on a low dimension manifold (see for example 5.11.3 in [5]). So it is reasonable that this is the case for p_{data} too. This may not be true in general for p_g instead, being originally a Gaussian random variable, whose dimension is full in its original sample space. But it is possible if the transformation applied by the neural network of the generator concentrates the mass of the distribution in a limited region of the output space. This depends on the kind of the transformation applied, but it can be demonstrated that in the case of a composition of affine transformations and pointwise nonlinearities of the type which are typically used in neural networks (ReLus, leaky ReLus, sigmoids, etc) , the support of the output lies over a countable union of manifolds whose dimension is equal or smaller than the input space. As a consequence, if the input space dimension is smaller than the output one (as is typically the case), the support of p_g lies actually over a low dimensional manifold of its space. This means that the scenario in which the two distribution are not continuous is what one expects to have in practice, at least approximately. In order to state the discrimination theorem for this case, the concept of perfectly aligned manifolds must be defined. Simply speaking, two manifold are perfectly aligned when they overlap enough to be indistinguishable, so that it would not possible to understand whether a sample is taken from one or the other. Which would also mean that no discriminator with accuracy 1 could exist. More formally, given $\mathcal{P}$ and $\mathcal{M}$, two boundary free regular submanifold of $\mathcal{F}$, they are said to intersect transversally in x if:

$$T_x\mathcal{P} + T_x\mathcal{M} = T_x\mathcal{F} \tag{1.17}$$

where T_x stands for the tangent space of the manifold in x. $\mathcal{P}$ and $\mathcal{M}$ are said to perfectly align if there exists a point x (belonging to their union) in which they do not intersect transversally. In practice, it can be assumed that any pair of submanifolds never perfectly align, because any arbitrary small perturbation of two perfectly aligned manifolds would made them not aligned. So this can be assumed to be true in the analysis. Granted this, finally a perfect discrimination theorem can be stated also in the previous scenario. If:

- p_{data} and p_g lie on two submanifolds $\mathcal{P}$ and $\mathcal{M}$ (over which they are continuous)
- $\mathcal{P}$ and $\mathcal{M}$ do not perfectly align,

then there exists an optimal discriminator D^* with accuracy 1 such that for almost any x in $\mathcal{P}$ or $\mathcal{M}$, it is smooth in a neighborhood of x and results $\bigtriangledown_x D^*(x) = 0$.

1.3.3 Vanishing gradient explained

So in both cases examined, not continuous supports and disjoint supports, which are at least good approximations of any phase of the training, it has been proved that when the optimum discriminator is reached, no more learning is possible, because its gradient goes to zero, so that ascending or descending it will not result in any modification of the parameters of discriminator neural network. But this is reasonable, because once the discriminator has reached its optimum its parameters should not be modified any longer. The big issue, which is the reason for which the training of the standard formulation of GANs must be carefully approached, is the effect that this has on the generator. To understand it, it is useful defining a norm which allows to define the concept of similarity between the discriminator at a certain point of the training and the optimum discriminator with accuracy 1 D^*. Among the possibilities, it can be defined as:

$$\|D\| = \sup_{\mathbf{x}} |D(\mathbf{x})| + \|\nabla_{\mathbf{x}} D(\mathbf{x})\|_2 \tag{1.18}$$

Given this definition of norm and given that the two perfect discrimination theorems are valid, it can be demonstrated that:

$$\lim_{\|D-D^*\|\to 0} \nabla_{\theta_g} \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})} \left[\log\left(1 - D(G(\mathbf{z}))\right)\right] = 0 \tag{1.19}$$

This is the fundamental result that explains why if one alternates a significant amount of training of the discriminator to the training of the generator does not obtain a good result. In fact the meaning of (1.19) is that when the gradient of the cost function of the generator is descended, the more the discriminator has been trained (the closer it is to the optimum discriminator with accuracy 1) the smaller will be the gradient cost function, which means the less the parameters of the neural network of the generator will be modified to reach the optimum (the well known "vanishing gradient problem"). So one must carefully set the parameter k defining the amount of batches to be passed to the discriminator, trying to avoid to reach the optimum D^*, which would essentially stop the training of the generator. But in this way, at the same time, the min-max problem can not be optimally solved, because it foresees to obtain the optimal discriminator.

1.3.4 Vanishing gradient for -log(D)

As said in the previous paragraph, one solution that the authors of [6] were suggesting in order to improve the performances, was the modification of generator cost function according to (1.16). In this case the following result can be demonstrated:

$$\mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})} \left[-\nabla_{\theta_g} \log(D^*(G(\mathbf{z}))|_{\theta_g=\theta_0})\right] = \nabla_{\theta_g} \left[KL(p_g\|p_{data}) - 2JSD(p_g\|p_{data})\right]|_{\theta_g=\theta_0} \tag{1.20}$$

where θ_0 is any particular set of parameters θ_g. This results explains what is seen in practice. In fact the Kullback-Leibler divergence as defined in (1.11) can be rewritten according to the definition of average:

$$KL(P\|Q) = \int_{\mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right) dx \tag{1.21}$$

where $P = p_g$ and $Q = p_{data}$. From (1.21), one can observe that the Kullback-Leibler divergence assign an high cost (a high value of the integral) to a sample x such that $p_g > 0$ and $p_{data} \to 0$, and a low cost to a sample such that

$p_{data} > 0$ and $p_g \to 0$. The first situation corresponds to a fake sample, so it is unlikely that the generator will output a sample which does not belong to the original distribution. The second situation is more likely, because it has a low cost and corresponds to a sample that appears on the original distribution but that probably will not be generated by the GAN, the so called "mode dropping". This is exactly observed in reality. Notice that the Jensen-Shannon divergence in this case does not unbalance the output distribution, because it is symmetric, so it can be neglected in these considerations. But what about the training. It can be proved that the gradient:

$$\mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}\big[- \triangledown_{\theta_g} \log(D(G(\mathbf{z})))\big] \tag{1.22}$$

is a centered Cauchy distribution with infinite average and variance. The infinite variance leads to a very unstable training of generator, which makes the optimization much more difficult. But maybe more importantly, the same problem of an almost null gradient rises again, because of course the average can not be infinite but will be bounded; but at the same time being the distribution centered, the average will be zero, so that in average the generator will receive no or very small feedback from the discriminator.

1.3.5 Noise adding

In the following chapter, it will be shown that using proper metrics the performances can be improved, obtaining a training which is much more confident. One of the hints that [8] suggests, is conceptually simple: breaking the non continuity of the distributions, that it was demonstrated to be the original cause of all these phenomena. This can be done by adding noise to the samples from the dataset to be provided to the discriminator and to the output of the generator as well (to make the discriminator less vulnerable to adversarial samples). As expected, with this strategy one can see that the Jensen-Shannon divergence of the noisy version of p_{data} and p_g is minimized, contrary to what happens to their standard non continuous versions that always maximize it.

Chapter 2

Training strategies

2.1 Introduction

In this chapter, several methodologies to improve the training of GANs will be presented. Starting from the problems analyzed previously, new cost functions will be presented, solving the problems of vanishing gradient and instability that arise when the cost functions of the original paper [6] are employed. After that, some strategies that ease and speed up the process of training will be shown, dealing more with an algorithmic and conceptual point of view rather than finding new mathematical improvements of the architecture.

2.2 Wasserstein GAN

From the discussion of the previous chapter, it is clear that the problem lays on the choice of the cost function of generator and discriminator. Adding noise to both, will result in general better performances, breaking the hypothesis that make the standard cost functions fail. Another way to address the problem consists in a more radical substitution of the cause of problems: a novel way to measure the distance between the distributions of dataset p_{data} and of generator p_g, and as a consequence a radical modification of the cost function.

2.2.1 Convergence

In order to find a way to get new distances/divergences which are suitable to solve GANs problems, it is useful to think in terms of distribution sequences convergence. In fact, during the training, p_g undergoes to a series of modifications that should led it to p_{data}. This corresponds to the definition of convergence of a distribution. Given:

- $\rho(\cdot,\cdot)$, a divergence or a distance between two distributions
- $(P_n)_{n\in\mathbb{N}}$, a sequence of distributions parameterized by n

$(P_n)_{n\in\mathbb{N}}$ is said to converge if and only if

$$\exists P_\infty : \rho(P_n, P_\infty) \to 0 \tag{2.1}$$

With respect to the GAN analysis, $(P_n)_{n\in\mathbb{N}}$ is the sequence of forms assumed by p_g throughout the training phase, that will be denoted by p_{g_θ}, to underline the dependence on θ. Instead, P_∞ is the target distribution p_{data}. The way the divergence $\rho(\cdot,\cdot)$ is defined, determines if the convergence is reached and how much easily it is done. In fact, the weaker is the topology induced on the compact metric set on which the space of probability measures of interest are defined, the easier will be the convergence. As a consequence, it may be useful to compare different divergences and distances to see which is the weaker (that introduces a weaker topology) one. Until this point, two divergences have been addressed: the Kullback-Leibler divergence and the Jensen-Shannon. It turns out that if there is convergence for the first, which is:

$$KL(P_n, P_\infty) \to 0 \textit{ or } KL(P_\infty, P_n) \to 0 \tag{2.2}$$

then there is convergence for the second too:

$$JSD(P_n, P_\infty) \to 0 \tag{2.3}$$

This means that the Jensen-Shannon divergence is weaker and most likely will perform better in the convergence, so that it is should be reasonable using it as a cost function instead of the Kullback-Leibler, which is actually done in the original formulation in [6]. Then, one may try to find a divergence which is even weaker than the Jensen-Shannon, which may result in better performances in terms of convergence and maybe solving the problems analyzed in the previous chapter. One fruitful chance is the Earth Mover or Wasserstein-1 distance, defined as follows:

$$W(p_{data}, p_g) = \inf_{\gamma\in\Pi(p_{data},p_g)} \mathbb{E}_{(x,y)\sim\gamma}\left[\|x-y\|\right] \tag{2.4}$$

where $\Pi(p_{data}, p_g)$ is the set of all joint distributions whose marginals are p_{data} and p_g. It can be demonstrated that (2.3) implies the convergence of Wasserstein distance, which is:

$$W(P_n, P_\infty) \to 0 \tag{2.5}$$

This means that it is the weakest distance/divergence considered until this point, which suggests that this could be a more suitable metric with respect to the Jensen-Shannon.

2.2.2 Continuity

When dealing with the convergence of a sequence of distributions, another important characteristic to be taken into account is the continuity of the correspondence between the sequence of distributions itself and the sequence of the parameters defining it. So essentially the continuity in mapping:

$$\theta_g \mapsto p_{g_\theta} \tag{2.6}$$

which means that when the set of parameters θ_g converges to a certain value θ, then p_{g_θ} converges to p_θ, the distribution defined by the parameters θ:

$$\theta_g \to \theta \Longrightarrow p_{g_\theta} \to p_\theta \tag{2.7}$$

This is particularly important, because if it is true, then the mapping between the sequence of parameters and the distance between the corresponding distributions and the target distribution:

$$\theta_g \mapsto \rho(p_{g_\theta}, p_{data}) \tag{2.8}$$

is continuous too. The weaker the distance, the most likely this continuity will be granted.

2.2.3 An example of continuity

To understand continuity's importance, consider the problem of learning $\mathbb{P}_0$, the distribution of $(0, Z) \sim \mathbb{R}^2$, with Z being a uniform random variable $Z = U[0, 1]$. Let $\mathbb{P}_\theta$ be the sequence of distributions corresponding to the functions $g_\theta(z) = (\theta, z)$, which must converge to $(0, Z)$. It results:

$$KL(\mathbb{P}_0, \mathbb{P}_\theta) = KL(\mathbb{P}_\theta, \mathbb{P}_0) \to +\infty \; for \; \theta \neq 0 \tag{2.9}$$

$$JSD(\mathbb{P}_0, \mathbb{P}_\theta) = \log 2 \; for \; \theta \neq 0 \tag{2.10}$$

$$W(\mathbb{P}_0, \mathbb{P}_\theta) = |\theta| \tag{2.11}$$

This means that trying to use the Kullback-Leibler or Jensen-Shannon divergence to optimize $g_\theta(z)$ will not work. In fact they are not continuous in the sense defined in the previous paragraph, because as θ varies, the value of the divergence does not change (Jensen-Shannon) or remains infinite (Kullback-Leibler). This means that taking their derivative results in either a vanishing gradient (Jensen-Shannon) or does not make sense at all (Kullback-Leibler), so that they are completely useless in an optimization process. On the contrary, Wasserstein distance is continuous with respect to θ, so that while θ is made vary following its gradient, the distance between $\mathbb{P}_\theta$ and $\mathbb{P}_0$ will smoothly decrease until to the optimum $\theta = 0$ is reached. As a consequence the Wasserstein metric is the only that provides convergence for this example. This means that in terms of cost function for a neural network trying to learn $\mathbb{P}_0$, it is the only working option. But the point is whether this is always true, and in what terms, or if it is just a lucky chance.

2.2.4 Continuity of Wasserstein

What has been seen in the previous example is actually valid in an wide variety of cases. Its demonstration derives from the following theorem, proved in [9]. Given:

- $\mathbb{P}_r$, a fixed distribution over a metric space $\mathcal{X}$
- Z, a random variable over another space $\mathcal{Z}$
- $g : \mathcal{Z} \times \mathbb{R}^d \to \mathcal{X}$, a function denoted as $g_\theta(z)$
- $\mathbb{P}_\theta$, the distribution of $g_\theta(z)$

then:

1. if g continuous in $\theta \implies W(\mathbb{P}_r, \mathbb{P}_\theta)$ continuous in θ
2. if g is locally Lipschitz and satisfies 1. $\implies W(\mathbb{P}_r, \mathbb{P}_\theta)$ is continuous everywhere and differentiable almost everywhere
3. 1. and 2. are false for JSD and KL

This means that the previous example is actually quite the general case, given that $g_\theta(z)$ is continuous. So it is definitely convenient to use it as a cost function in the GAN architecture, thus substituting the Jensen-Shannon divergence. Notice that the assumptions of this theorems allows to directly apply it to any feedforward neural network composed by any common affine transformations and pointwise nonlinearities.

2.2.5 The critic

In order to come up with an algorithmic strategy to train the GAN minimizing the Wasserstein metric between p_{data} and p_{g_θ}, one must first solve the problem of practically exploring all the space of possibilities of the term $\Pi(p_{data}, p_g)$ in (2.4), representing the set of all joint distributions whose marginals are p_{data} and p_g. The first step to overcome this difficulty is to apply the Kantorovich-Rubinstein duality to the infimum of the formula, so that (2.4) becomes:

$$W(p_{data}, p_{g_\theta}) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{\mathbf{x}\sim p_{data}}\left[f(\mathbf{x})\right] - \mathbb{E}_{\mathbf{x}\sim p_{g_\theta}}\left[f(\mathbf{x})\right] \tag{2.12}$$

where $\|f\|_L \leq 1$ is the set of all 1-Lipschitz functions over the compact metric space $\mathcal{X}$ on which the probability distributions are defined. It is useful to notice that if $\|f\|_L \leq 1$ is substituted by $\|f\|_L \leq K$, then all K-Lipschitz functions are found, and with respect to the case $K = 1$, the function $K\ W(p_{data}, p_{g_\theta})$ is obtained. Being K a constant, it can be neglected in the maximization problem, so that (2.12) can be written as:

$$\max_{w\in\mathcal{W}} \mathbb{E}_{\mathbf{x}\sim p_{data}}\left[f_w(\mathbf{x})\right] - \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}\left[f(\mathbf{x})\right] \tag{2.13}$$

where f_w is any K-Lipschitz (for some K) function parameterized by a certain set of parameters $w \in \mathcal{W}$. This function, that will be referred to as *the critic* according to the notation of [9], corresponds to the discriminator function, which is now slightly different from the original definition, because it needs no longer to output a probability. The only characteristic that must be guaranteed is that f_w remains K-Lipschitz. To this purpose, one viable way is to maintain the weights w, representing the parameters of discriminator neural network θ_d, inside a space $\mathcal{W}$ which is compact, throughout all the training phase. A practical way to do this is to clip w inside a symmetric interval after each gradient update, normalizing the weights of the neural network after each update. It turns out that even if this strategy works, it must be addressed carefully, because too small weights lead to the widespread phenomenon of vanishing gradient, whereas too large weights takes longer to be optimized. In the following paragraph, an effective regularization will be added to the cost function, so that almost no hyperparameter tuning is necessary.

2.2.6 Algorithm

So finally, the algorithmic implementation of what has been said until this point can be presented. It is similar to the standard one, but with some important differences. Given $[-c, c]$ the clipping interval, for k cycles, sample m times from $p_{\mathbf{z}}(\mathbf{z})$ and $p_{data}(\mathbf{x})$, then ascend the stochastic gradient of the discriminator:

$$\nabla_w \frac{1}{m} \sum_{i=1}^{m} \left[f_w(\mathbf{x}^i) - f_w(G\left(\mathbf{z}^i\right)) \right] \tag{2.14}$$

where notice that the parameter w could be written as θ_d and f_w as $D()$, according to the notation of previous chapter. After that, clip the resulting weights w to the interval $[-c, c]$. Then sample m times from $p_{\mathbf{z}}(\mathbf{z})$ and descend the stochastic gradient of the generator:

$$-\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} f_w(G(\mathbf{z}^i)) \tag{2.15}$$

There are two big differences with respect to the algorithm in chapter 1, other than the clipping and a slightly change of notation for the discriminator. First, logarithms are missing, because the Wasserstein does not require them. Secondly, and most importantly, the parameter k, which defines how many times the discriminator undergoes to a phase of training, is completely free. In fact there is no limitation to how much the discriminator can be trained before passing to the generator, as the Wasserstein guarantees a useful differentiability almost everywhere, which means that the gradient will not saturate, unless the parameters will be close enough to the optimum and there will be anything else to learn. So in practice this means that k can be as high as preferred, but in particular it is a good idea to bring the discriminator close to the optimal as possible, because it will provide the generator with the most reliable gradient possible. So essentially the merit of the Wasserstein GAN is that the intuitive idea of alternating the full optimization of the two neural networks is not broken any longer, as it was in the standard GAN version.

2.3 Improved Wasserstein GAN

Wasserstein GAN provides a new effective approach to the minimization of the distance between p_{data} and p_g, but as anticipated before the way the Lipschitzianity of the critic (alias the discriminator) is assured through the weights clipping, may lead sometimes to training difficulties. The problems of this techniques will be laid out and a regularization technique present in [7] will be shown, allowing to solve in a robust way the difficulties caused by the clipping.

2.3.1 Clipping effects

The first category of effects have been already explained previously: if the weights are clipped to an interval $[-c, c]$ with a c which is too little will lead to the vanishing gradient phenomenon. In the opposite case, when c is large, it takes in general longer to bring the weights to their optimum and this can even lead to too big updates when the gradient is descended, so that the phenomenon of the exploding gradients arises. But the clipping may also have another harmful effect over the use of the model capacity. In fact experimental evidences in [7] show that the critic learns an over-simplistic function, so that generalization problems can arise. This is essentially because the effect of the clipping makes the large majority of weights approaching the extremes of the clipping interval, making it harder for the generator to catch complex distributions.

2.3.2 Gradient penalty

One possible solution to the problem is a form of regularization called *Gradient penalty*. The idea behind it derives from a property that the optimal critic is demonstrated to have under the condition of Lipschitzianity previously described: the norm of its gradient is 1 almost everywhere under p_{data} and p_g. If this condition is enforced throughout the training phase, the weights will converge to a solution that respect it, guaranteeing as a consequence the Lipschitzianity. A way to accomplish this task consists in introducing a penalty over the solutions that do not respect it, adding the following penalty term to the cost function in (2.13):

$$\lambda\, \mathbb{E}_{\hat{\mathbf{x}} \sim p_{\hat{\mathbf{x}}}} \left(\|\nabla_{\hat{\mathbf{x}}} f_w(\hat{\mathbf{x}})\|_2 - 1 \right)^2 \tag{2.16}$$

where $p_{\hat{\mathbf{x}}}$ is the distribution of the set of samples:

$$\hat{\mathbf{x}} = \epsilon \mathbf{x} + (1 - \epsilon)\mathbf{y} \tag{2.17}$$

with $\mathbf{x} \sim p_{data}$, $\mathbf{y} \sim p_g$ and $\epsilon \sim Uniform[0, 1]$. In particular the distribution $p_{\hat{\mathbf{x}}}$ is defining a random variable which is uniformly distributed over points lying on straight lines between pair of points from p_{data} and p_g, because the optimal critic contains those kind of lines, having in particular gradient norm equal to 1. Thus, what changes in the algorithm is that the discriminator must ascend:

$$\nabla_w \frac{1}{m} \sum_{i=1}^{m} \left[f_w(\mathbf{x}^i) - f_w(G\left(\mathbf{z}^i\right)) + \lambda\ \left(\|\nabla_{\hat{\mathbf{x}}} f_w(\hat{\mathbf{x}}^i)\|_2 - 1 \right)^2 \right] \tag{2.18}$$

and of course the clipping is not present anymore. Empirical results show that avoiding the clipping in flavor of gradient penalty cancel out the problems described previously, in particular referring to the weights concentrating on the extremes of clipping intervals. Another notable advantage is that the tuning of hyperparameters is very easy, so reducing the training time. So overall the gradient penalty results to perform better than the standard approach to Wasserstein GAN.

2.4 Progressive growing

Until this point, the analysis over GANs methodologies has been focused on the mathematical properties of the cost functions and how they relate to the effectiveness on training. Another approach to improve the performances of the model is focusing on how the structure of the layers composing the neural networks of generator and discriminator can be modified. This is done in [14], which introduces a technique called *Progessive growing.* This idea, which is actually applicable to a wide range of domains, has come out to address datasets composed of high quality images, which exhibit many levels of scales structures, so that it is difficult, and as a consequence time consuming, to learn at the same time all the details at all scales. If this is true for any machine learning algorithm applied to the image generation, in particular the training for GAN results particularly prone to find difficulties in starting creating a generator which is able to deceive the discriminator. In fact, given the high level of detail, the distribution will be particularly complex, so that it is very easy for the discriminator to understand whether a sample is fake or not in the initial stage, thus amplifying gradient problems. Another problem that arises, this time for any kind of algorithm, with an high dimensional dataset like this, stands in the fact that the batches of samples for the stochastic gradient descent can not be too large, because of memory constraints.

2.4.1 Solution

A possible solution for the problem is training progressively the neural networks. This means that instead of having generator and discriminator the respectively outputs and gets as input a full resolution sample, they handle it at increasing scale. So at the beginning the structure of the neural network is simple, maybe starting with just one single layer. The samples produced and taken as input are downsampled versions of the original one, thus the training in this stage is very fast. And in this phase the high scale details are learned. Then, more layer are smoothly added to the networks, so that the details at higher resolution are considered and the samples are nearer and nearer to the final format. In the meanwhile, generator and discriminator learn at each stage of the growth just a limited amount of information, which just deals with single new resolution scale of details, because the previous scales have been previously learned. Overall, as demonstrated in [14], a lot of time is saved, because most of the training is done at lower resolution, in which the amount of information to learn is limited, and at the beginning the training results more stable.

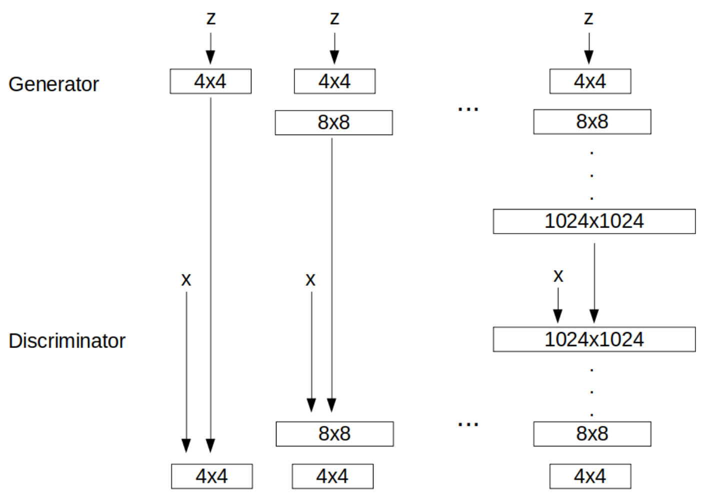

Figure 2.1: Progressive growing: z represents the noise taken as input from the generator, x are the images from dataset. In the first stage just a 4x4 layer is present, which means that the discriminator will receive images down-sampled to that resolution either from the dataset or from the ouput of the generator. Then, in the following stages, additional layer are added, until the full resolution 1024x1024 is reached

2.5 Conditional GAN

To conclude this chapter about training, the concept of conditional GAN is explained, which allows to use GANs to produce samples with label. In fact, there may be datasets composed by the samples plus some kind of extra information about the meaning of that sample. This happens for example for images, in which it may be useful knowing what kind of objects is present in the generated sample. The extension of the GAN model to this case is, in its basic form, quite straightforward: the discriminator receives as an input the sample plus the label; the generator receives again the noise but with a label too, so that it will be trained to find not only the distribution of the samples from the datset, but also how a certain label is associated to part of that distribution, so that through the feedback of the discriminator the generator will shape the output to contain a label too. Referring to the original cost function, the problem can be now formalized as:

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})} \log D(\mathbf{x}|\mathbf{y}) + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})} \log\left(1 - D(G(\mathbf{z}|\mathbf{y}))\right) \tag{2.19}$$

where $\mathbf{y}$ represents the label. This formalization may find application in a variety of occasions, for example [11] shows how this can be applied to generate images with label describing in human language the content, starting from a dateset which is labelled with words describing the content of the images.

Chapter 3

Signals over graphs

3.1 Introduction

Signals over graphs, or Graph signal processing, aims at extending signal processing techniques to data which can be conveniently represented as graphs. This finds use in a wide variety of domains, like networks of sensors, social networks, biological networks and many other. This chapter will present some fundamental concepts of the theory, in order to provide a useful background for using deep learning over dataset that can be represented as graphs. In particular, the central issue is providing a definition of the convolution operator on graphs which is suitable to be employed in a convolutional neural network.

3.2 Spectral methods for convolution

The idea behind Signals over graphs is to represent the samples of a signal as nodes of a graph, whose edges formalize some kind of relationship among the samples. Formally, given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, W)$ where:

- $\mathcal{V}$, is the set of vertices of cardinality n
- $\mathcal{E}$, is the set of edges of cardinality m
- $W \in \mathbb{R}^{n \times n}$, is the adjacency matrix

a signal over the graph $x : \mathcal{V} \to \mathbb{R}^n$ can be regarded as a vector $x \in \mathbb{R}^n$ whose i-th element is the value of x at the i-th node.

3.2.1 Graph Fourier transform

To define the Fourier transform in this setting, it is necessary to express the idea of frequency domain as it is done in the common signal processing. To this purpose, the graph Laplacian is defined as:

$$L = D - W \tag{3.1}$$

with $D \in \mathbb{R}^{n \times n}$ being the diagonal degree matrix defined as:

$$D_{ii} = \sum_j W_{ij} \tag{3.2}$$

It is then possible to define a normalized version of the Laplacian:

$$L = I_n - D^{-\frac{1}{2}} W D^{-\frac{1}{2}} \tag{3.3}$$

which, being a real symmetric positive semidefinite matrix, admits n eigenvectors $\{u\}_{l=0}^{n-1}$ and the corresponding ordered nonnegative eigenvalues $\{\lambda\}_{l=0}^{n-1}$, which can be interpreted as the frequency support of the graph. Now, using the notation of [12], given $U = [u_0, ..., n_{n-1}]$ and $\Lambda = diag([\lambda_0, ..., \lambda_{n-1}])$, through which is possible diagonalizing the Laplacian as $L = U \Lambda U^T$, the Graph Fourier transform of the signal x is defined as:

$$\hat{x} = U^T x \tag{3.4}$$

with its inverse being $x = U\hat{x}$.

3.2.2 A first definition of convolution

It is now possible to define a convolution operation in an analogous way to the standard signal processing: the inverse transform of the product between transforms. So, if x and y are two signals defined on the graph $\mathcal{G}$, their graph convolution is defined as:

$$x *_{\mathcal{G}} y = U(\hat{x} \odot \hat{y}) \tag{3.5}$$

where $\odot$ is the the element-wise Hadamard product between two vectors. The purpose of this explanation relies on the fact that this techniques are necessary to define those operations that the convolutional layers of convolutional neural networks applies to images, so to grid of samples to be filtered, which are in this case graph instead. It is then useful to see the previous definition in terms of a sample x that undergoes to a transformation acted by a filter g_θ. So the result of filtering can be written as:

$$\begin{aligned} y &= g_\theta(L)\, x \\ &= g_\theta(U\Lambda U^T)\, x \\ &= U\, g_\theta(\Lambda)\; U^T x \end{aligned} \tag{3.6}$$

where the last equivalence of (3.6) can be interpreted as a convolution. In fact $U^T x$ is the graph Fourier transform of x and when g_θ is a non-parametric filter, as is typically the case, it is defined as a diagonal matrix whose parameters are free:

$$g_\theta(\Lambda) = diag(\theta) \tag{3.7}$$

with $\theta \in \mathbb{R}^n$. This means that the product $g_\theta(\Lambda)\; U^T x$ can be actually interpreted as an Hadamard product, because the resulting vector has on the i-th row the multiplication of the element on the i-th rows of the two terms. Thus $g_\theta(\Lambda)$ is the spectral form of the filter. But applying the filtering in this way is not easy, for two reasons. The learning complexity is in the dimension of data $\mathcal{O}(n)$, because all the n elements of $\theta \in \mathbb{R}^n$ must be learned. Also, the filter is not localized in space, because the whole graph is taken into account in the spectral domain.

3.2.3 Polynomial filters

To overcome these difficulties, a polynomial filter can be used:

$$g_\theta(\Lambda) = \sum_{k=0}^{K-1} \theta_k \Lambda^k \tag{3.8}$$

It solves the problem of the learning complexity because now only the K parameters of $\theta \in \mathbb{R}^K$ must be learned. Also, it can be demonstrated that such a filter is localized, around the node which is processed, in a neighborhood whose elements are at distance up to K. So this means that the filter is localized in space. So this kind of filter is suitable to be learned by a convolutional neural network. But actually other two problems arise. The cost in terms of operations to be done is still high,because the expression in (3.6) has computational complexity $\mathbb{O}(n^2)$. This can be solved by a recursive reformulation of the polynomial filter, using the Chebyshev expansion, so that the complexity is highly reduced. But a more important difficulty comes out when approaching the definition of convolution using the spectrum of the graph: the graph structure must remain the same, only the values of the signal can change. This makes it hard to find a way to apply machine learning techniques to datasets in which this hypothesis is not true, because in this case also the structure itself of the graph must be parametrized and learned. This means that another way to define the convolution must be found in this kind of domains.

3.3 Edge-Conditioned Convolution

To address non constant graph structures, [10] proposes a novel more suitable way to define the convolution on graphs, which does not take into account the Laplacian, so that also the edges of the graph can be learned by a convolutional network. To this purpose, to each vertex of the graph will be assigned not just a single real value but in general a vector of values, also referred to as *labels*. The same is true for edges. This approach is specifically designed to be applied with convolutional networks, so it is assumed that the signal over the graphs undergoes to a series of transformation defined by a feedforward neural network whose layers are indexed by $l \in \{0, ..., l_{max}\}$. The function that describes how the original vector of labels for each vertex is modified through the network is denoted as:

$$X^l : \mathcal{V} \mapsto \mathbb{R}^{d_l} \tag{3.9}$$

which means that the i-th vertex has, at layer l, a vector of labels $X^l(i)$ of dimension d_l, where d_l means that the number of labels varies depending on the particular layer l. So essentially it is a matrix in the space $\mathbb{R}^{n \times d_l}$. For example X^0 denotes the initial set of labels assigned to the indices, so the input. Then it is useful to define a function $L : \mathcal{E} \mapsto \mathbb{R}^s$ which denotes the labels of the edges of the graph (L in this case has nothing to deal with the Laplacian). Again, L is essentially a matrix in $\mathbb{R}^{m \times s}$. The notation $N(i)$ denotes all the neighbors of the vertex i-th plus the vertex i-th itself, or formally $N(i) = \{j \in (v) : (j, i) \in \mathcal{E}\} \cup \{i\}$. So for directed graphs it takes into account only the nodes with edges directed to node i-th.

3.3.1 Convolution definition

Now it is finally possible to define the convolution and understand in which way this relates to the parameters learned by the feedforward neural network. The idea is that the value of the labels of a vertex, at a given layer, is a weighted sum of the corresponding labels of the neighbors of the vertex. The advantage is twofold: from one side the commutativity grants that the way the the vertices are indexed is completely transparent, so that no ordering is needed; on the other side there is no problem on choosing which vertices should be filtered when elaborating a certain vertex, because only the neighbors are taken into account, independently by their number. The disadvantage is that using just the neighbors, part of the structural information is lost. To recover it, the filter takes into account also the label of the edges, conditioning in this way the weights of the filter. In this analysis, the filter is the feedforward neural network and the weights are the neural network learnable weights and bias. So the network can be represented as a layer specific function:

$$F^l \ : \ \mathbb{R}^s \mapsto \mathbb{R}^{d_l \times d_{l-1}} \tag{3.10}$$

which given an edge $L(j,i)$ outputs the matrix weights $\Theta^l_{ji} \in \mathbb{R}^{d_l \times d_{l-1}}$ defining how labels of vertex j must be used in the weighted sum to obtain the labels of vertex i. Thus the Edged-Conditioned Convolution defines how to obtain the labels of a certain vertex i at layer l:

$$X^l(i) = \frac{1}{|N(i)|} \sum_{j \in N(i)} F^l(L\,(j,i)\,; w^l) X^{l-1}(j) + b^l \tag{3.11}$$

or equivalently with the output weights matrix:

$$X^l(i) = \frac{1}{|N(i)|} \sum_{j \in N(i)} \Theta^l_{ji} X^{l-1}(j) + b^l \tag{3.12}$$

This is the way labels are evaluated through the neural network, then applying the common learning technique like gradient descent is possible to find the weights w^l and bias b^l that optimize the cost function.

3.4 Pooling and coarsening

Until this point, the main topic of this chapter has been the convolution operator on a graph and its use within a convolutional network. But another fundamental operation in this setting is the pooling. In convolutional neural network, pooling layers are inserted among convolutional layers to reduce the amount of parameters to learn, so reducing the risk of overfitting, trying to summarize sets of adjacent features by typically applying some kind of downsampling. But if this operation is straightforward in image domain, where the features are collected over grids, this is not easy as well when dealing with graphs, because in this case the concept of downsamping must be properly defined. It is called Graph Coarsening and in the setting laid out previously, it consists in a non trivial sequence of operations. First of all, a new setting of vertices must be created, either by removing some of them or merging groups into single ones. After that a new edge structure must be created among the new nodes, defining in a proper way new edge label. Finally, it is necessary to map the features of the nodes before the coarsening to the nodes after the coarsening, by establishing a proper mapping between the two groups and keeping into account also the new edges. This is the general idea behind the pooling in the graph environment, but ad hoc solutions must be considered depending on the type of data and network structure in use.

Chapter 4

GAN specialised architectures

4.1 Introduction

This chapter is devoted to the application of the concepts studied until this point to special domains of interest, with a special focus on some graphs-related topics. In general the analysis will be concentrated on a variety of architectures proposed in relatively recent research papers, studying their peculiarities and results.

4.2 Learning generative models for 3D point clouds

A point cloud is a set of unordered points, typically in three dimensions, that represents a discretization of an object, acquired for example with laser scans. The fact of having a dataset composed of unordered sets poses major challenges, because any permutation of a set has the same semantic meaning, so it must be associated at the same output. As a consequence, the operations performed over the data must be overall permutation-independent. Also, the distribution of points is very irregular. Within deep learning framework, several solutions were proposed to solve this issues. Among them, the voxelization of the point cloud (representing the point cloud over a regular three dimensional grid) and special deep architectures like PointNet, whose solution is processing singularly each point of the cloud, then applying global permutation-independent operations. Another approach consists in using the Edge-Conditioned Convolution described in paragraph 3.3, which has the great advantage of being by definition permutation-independent, because of the commutativity of the operations to calculate the convolution. It also allows to catch the very irregular nature of point clouds exploiting the local aggregation of features among neighbor nodes (in this case points). But at the same time, the implementation of this strategy for a generative model based on a GAN is challenging, because the structure of the graph is not known in advance to the generator which simply receives a random prior, so understanding how to handle this local operation is not straightforward.

4.2.1 ECC application

In [2], all of these problems are addressed, proposing a new architecture. The fundamental operation is the Edge-Conditioned convolution, as said previously. The idea, referring to (3.11), is to use a fully connected network F^l which is a function of the difference between the features of each pair of nodes, so that (3.11) becomes:

$$X^l(i) = \frac{1}{|N(i)|} \sum_{j \in N(i)} F^l\left(X^{l-1}(j) - X^{l-1}(i)\right) X^{l-1}(j) + b^l \tag{4.1}$$

or similarly using the notation of [2]:

$$h_i^{l+1} = \sigma\left(\sum_{j \in \mathcal{N}_i^l} \frac{F_{w^l}^l\left(h_j^l - h_i^l\right) h_j^l}{|\mathcal{N}_i^l|} + h_i^l W^l + b^l\right) \tag{4.2}$$

where:

- h_i^l stands for $X^l(i)$
- $\sigma()$ is a non linearity
- $\mathcal{N}_i^l$ does not include the node i any longer
- W is a linear transformation of the nodes themselves

But one problem to solve is deciding which points belong to the neighborhood of the node under consideration. In fact, being the point cloud the output of the generator, the generator does not know which is the associated graph to be used to define the neighborhood during the processing operations. The proposed solution is to exploit the pairwise distance between nodes features to find which are the nearest points of the node under consideration. So, if k is the hyperparameter defining how many points must form a neighborhood, then the k nodes with smallest $|h_j^l - h_i^l|$ will be those considered in $\mathcal{N}_i^l$ of equation (4.2). This means that the building block of the generator is a combination of two operations: for each point of the cloud, its k nearest neighbors are found, building the corresponding neighborhood graph; then for each node the ECC is applied, exploiting the features of nodes composing the neighborhood. So the result of each block is a new labeling of the nodes, with the network learning weights just as a function of pairwise features difference. As a consequence, after the block a new representation in vectorial form of the graph structure is obtained, procedure known as graph embedding. Putting several blocks one after the other, creates an hierarchy of graph embeddings, representing increasingly better the graph structure as one gets farther from the input layer.

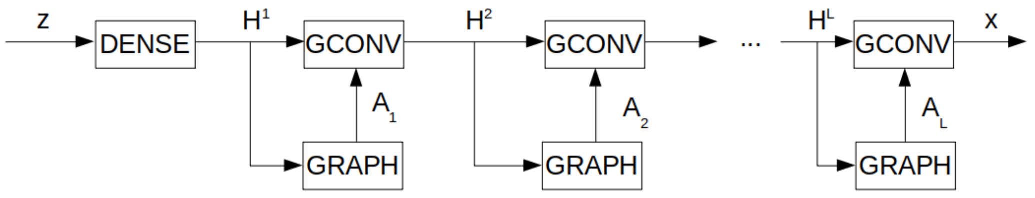

Figure 4.1: The architecture of the generator. After a dense layer, a succession of convolutional layer, composed of a convolutional block and of a nearest neighbor graph generating block, forms a hierarchy of graph embeddings representing the cloud structure

4.2.2 Upsampling

Many typologies of data have a multiresolution representation, that can be exploit to ease the extraction of information. For example in 2D images, low frequencies can be properly processed and upsampled to predict higher resolution. This is exaclty what is done by convolutional neural network applied to images, through convolutional and upsampling layers. If the convolutional side of the processing has been previously analyzed, it seems reasonable to try to find a way to represent and exploit the multiresolution properties also in the case of the point clouds. So what has to be done is finding a way to implement the generator in such a way that it is able to represent and process the point cloud at different resolutions, thus a way to provide an upsampling operation must be defined. The solution proposed in [2] is very similar to the logic behind the convolution. First of all, to each convolutional layer l, follows an upsampling layer. In particular, if the convolutional layer has N^l features vectors, corresponding to N points, grouped in a matrix $H^l \in \mathbb{R}^{N^l \times d^l}$, then the upsampling layer creates N^l new features grouped in the matrix $\tilde{H}^l \in \mathbb{R}^{N^l \times d^l}$, corresponding to N new points. Concatenating the two matrices H^l and $\tilde{H}^l$, a new matrix $H^{l,up} \in \mathbb{R}^{2N^l \times d^l}$ is obtained, corresponding to $2N$ nodes. So the upsampling layer provide an upsampling factor of 2. As said, the way new features are found is similar to the ECC. In fact, a new vector of feature, corresponding to a certain new node i-th created by the upsamplig layer, is given by:

$$\tilde{h}_i^l = \sigma\left(\sum_{j \in \mathcal{N}_i^l} \frac{diag(U^l_{\tilde{w}^l}\left(h_j^l - h_i^l\right))h_j^l}{|\mathcal{N}_i^l|} + h_i^l \tilde{W}^l + b^l\right) \tag{4.3}$$

which can be interpreted exactly as (4.2) apart from the term $diag(U^l_{\tilde{w}^l})$, where $U^l_{\tilde{w}^l})$ is again a fully connected network, that differently with respect to the term $F^l_{w^l}$ of (4.2), now does not produce a dense matrix but a diagonal matrix, which means that the product with h_i^l will be row by row, meaning that each feature is handled independently by the other. This also reduces the number of parameters to be learned.

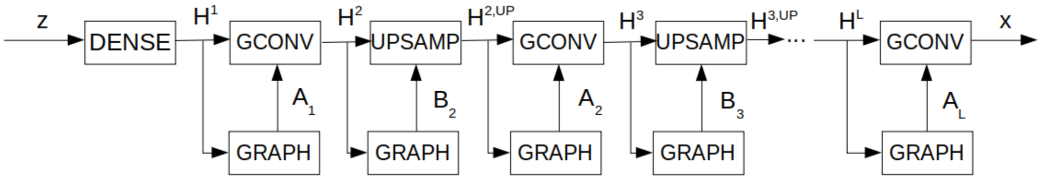

Figure 4.2: The generator with the adding of the upsampling layers, composed of the convolutional-like upsampling block and the nearest neighbor graph generating block

4.2.3 Conclusions

Even if the upsampling layers are not strictly required to obtain the hierarchy of graph embeddings throughout the generator network, it can be experimentally verified that the performances of the architecture when upsampling is used outperforms the case without it and also other GAN architectures specialized in point cloud handling, but that do not use the concept of localized representation. The reason behind the usefulness of the upsampling is that without it the convolutional layers need to learn a very large amount of parameters, working since the first layers on the full dimension of the point cloud. Applying upsampling, allows to work on smaller dimensional point clouds, that only in the last stage reach the full dimension of the dataset. Another interesting phenomena described in [2] is that when new points are generated, they are typically far from the neighborhood from which they were predicted, but the distribution of the new neighbors is the expected one. This is caused by how the network learn to generate new points, essentially consisting in learning a transformed version of the neighborhood structure and copying it in another area, so that the structure is preserved, but new points are generated far from those used for the prediction.

4.3 NetGAN

Many complex systems can be represented as graphs which exhibit characteristics common to a wide number of systems, independently from the domain of application. Examples of these properties are the small average distance among nodes, the high clustering coefficient (representing communities in social networks), an heavy tail distribution of node degrees and many others. Explicit generative models can model analytically some of these peculiar behaviors, like the Barabasi-Albert and the Watts-Strogatz model, but catching all of them with just a single model is still a difficult problem. But having a realistic generative model available is important because it is needed fo widespread operations like data augmentation, anomaly detection and recommendation. It is then useful trying to find an implicit (non explicit) generative model that can be used to catch the structure of the graph for this kind of operations. But as shown in previous paragraphs, dealing with discrete objects like graphs is particularly challenging, as their processing must be done so that some properties are guaranteed (like being invariant to nodes permutation). In particular, dealing with graphs representing complex systems represents a peculiar difficulty: typically large datasets coming from the same exactly distribution are not available, so the model must learn from a single graph. In [1] a new architecture is proposed to face the problems, called NetGAN. It is a GAN which is trained using random walks sampled from the graph. Once trained, the generator is able to produce random walks whose nodes are distributed accordingly to the structure of the graph. Properly weighting the number of appearances of an edge into a set of generated random walks, it is possible to construct the adjacency matrix of a graph, which has the same topological features characterizing the original one but which is not the graph itself. Also, it turns out that this architecture is very effective at link prediction, despite not being specifically trained for this purpose.

4.3.1 LSTM

Up to this point only feedforward neural networks have been considered, where the neurons' activation flows only in one direction: from input to output. But the neural networks composing generator and discriminator of NetGAN exploit another typology of neural network, based on the so called Long Short-Term Memory (LSTM) cell, which is a form of Recurrent Neural Network (RNN). RNNs are an extension of feedforward networks because they allow connections pointing backwards. A very simple example is a network composed of recurrent neurons. The neuron of any feedforward network is the basic unit performing the processing operations, which consists in weighting the input vector $\mathbf{x}$ and then applying a point-wise non linearity, so that the output $\mathbf{y}$ is:

$$\mathbf{y} = \sigma(\mathbf{x}\, W_{\mathbf{x}} + b) \tag{4.4}$$

where $W_{\mathbf{x}}$ is the weights matrix and b the bias. A recurrent neuron, in addition, also exploit the output of the previous time step (the previous sample) to evaluate the current time step output, so that, given the input at time t denoted as $\mathbf{x}(t)$, the output at time t is:

$$\mathbf{y}(t) = \sigma(\mathbf{x}(t)\, W_{\mathbf{x}} + \mathbf{y}(t-1)\, W_{\mathbf{y}} + b) \tag{4.5}$$

where $W_{\mathbf{y}}$ defines how to weight the previous time output as an input.

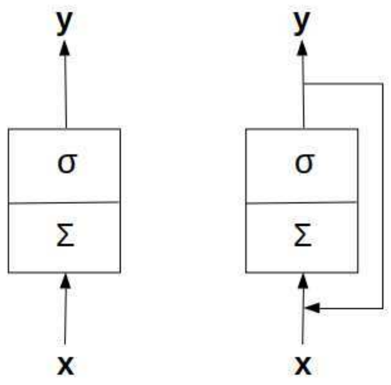

Figure 4.3: On the left, the neuron of a feedforward network. On the right, a recurrent neuron

This is useful when in the data is present some kind of temporal relationship, as happens when the input is a time series, so that it is possible to perform a prediction exploiting the dependence of a sample from the previous, which are fed to the network as an input. For example NetGAN exploit this principle to predict the following node of a random walk given the previous ones. So the difference with respect to feeforward network is that neurons own a kind of memory of the past, so that they can be thought as a memory cell characterized by a certain state. The output is then dependent on the current input and the previous time step state of the cell. There are many ways to structure the memory cell, one of the most effective, and the one used in NetGAN, is the LSTM cell. There are actually various implementations of LSTM, anyway the structure in NetGAN is in particular the original idea proposed in [13]. It is shown in the figure below.

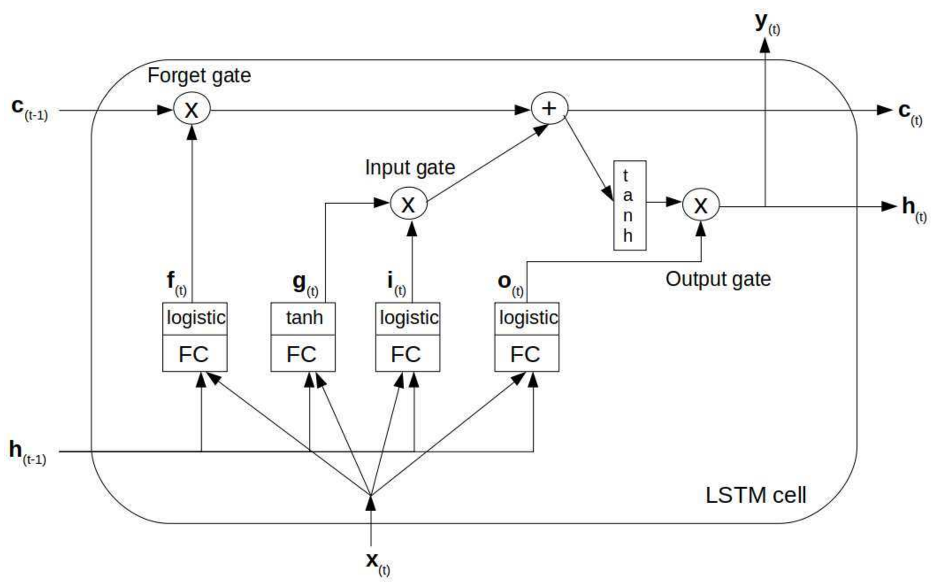

Figure 4.4: LSTM cell. FC stands for fully connected networks.

The state of the cell is split in two parts: one keeping into account the long term state, denoted as $\mathbf{c}_{(t)}$, and the other the short term, denoted as $\mathbf{h}_{(t)}$. The network can decide what to store in the long term state using the forget gate, which drops memory, and with the addition operation which follows on the right, adding memory. So at each time step some long term memory is added and some is dropped. After that, $\mathbf{c}_{(t)}$ is copied and passed through the tanh function, and then the result is filtered by the output gate. This produces the short term state $\mathbf{h}_{(t)}$ (which also corresponds to the current step output $\mathbf{y}_{(t)}$). The gates that allow these operations are regulated by four fully connected networks, which are fed with $\mathbf{x}_{(t)}$ and $\mathbf{h}_{(t-1)}$. The main layer is the one that outputs $\mathbf{g}_{(t)}$, which has the role of analyzing the current inputs $\mathbf{x}_{(t)}$ and the previous short term state $\mathbf{h}_{(t-1)}$. Its output is then partially stored in the long term state. The other three fully connected layers directly control the gates, since they use a logistic function whose output is directed to an element-wise multiplication, so that if the ouput is zero the gate is closed, if it is one it is opened. In particular, $\mathbf{f}_{(t)}$ controls which part of the long term state must be deleted. $\mathbf{i}_{(t)}$ controls which part of $\mathbf{g}_{(t)}$ must be added to the long term state. $\mathbf{o}_{(t)}$ controls which part of the long term state should be read and output at this time step. So essentially the ability of the LSTM cell is to learn to recognize an important input, store it in the long term state and learn to preserve or remove it when needed.

4.3.2 The architecture

Consider a graph with N nodes described by a binary adjacency matrix $A \in \{0,1\}^{N \times N}$, a set of random walks of length T is sampled from it. They will constitute the dataset used to train the GAN, which is in particular a Wasserstein GAN applying gradient penalty. The choice or random walks guarantees that the architecture is invariant with respect to permutation of nodes. Also, it guarantees the exploitation of the sparsity of matrix, because only its non zero entries are used. Now the structure of the generator is addressed. As said, its neural network is a RNN with LSTM cells. So the product of the generator is a "time series" $(\mathbf{v}_1, ..., \mathbf{v}_T)$, in which the element $\mathbf{v}_t$ is a one-hot vector indicating which is the node at the t-th step of the random walk. If f_θ denotes the neural network with parameters θ, then at each time step t, two quantities are produced:

- $\mathbf{m}_t$, the memory state of the model, which is split in the LSTM cells in the short term state $\mathbf{h}_t$ and long term state $\mathbf{c}_t$, as described in the previous paragraph
- $\mathbf{p}_t$, which is a vector of logits which parametrize the probability distribution of the next node to be sampled (being represent as a one-hot vector)

In particular, the exact distribution from which $\mathbf{v}_t$ is sampled is the categorical distribution $Cat(\sigma(\mathbf{p}_t))$ with σ being the softmax function. For time step t+1, the input are $\mathbf{v}_t$ and $\mathbf{m}_t$, with the initial state $\mathbf{m}_0$ being obtained by using as an input a Gaussian random variable $\mathbf{z}$ (which remains the input of the generator as in any GAN), passing through a parametric function $\mathbf{g}_{\theta'}$.

Considering that in a real graph the number of nodes N con be quite large, using a $\mathbf{f}_\theta$ which produces a logistic vector $\mathbf{p}_t \in \mathbb{R}^N$ can be computationally demanding. So, before passing it to the LSTM, the vector is projected to a space of dimension $H \ll N$, by using a proper transformation defined by $W_{down} \in \mathbb{R}^{N \times H}$. After the elaboration, the result is then brought back to the

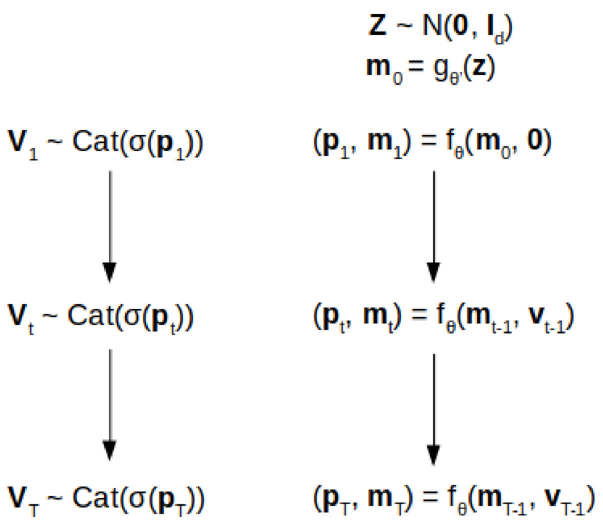

Figure 4.5: The scheme describes how the parameters are passed from one time step to the following through the LSTM architecture

original space using the inverse transformation. Another type of problems derives from the sampling of $\mathbf{v}_t$. In fact it is sampled from a categorical distribution. But sampling from a categorical distribution is a non differentiable operation, causing issues with the backpropagation. The solution proposed in [1] is to use the Straight-Through Gumbel estimator, which consists in obtaining a transformed version of $\mathbf{v}_t$:

$$\mathbf{v}_t^* = \sigma\left(\frac{\mathbf{p}_t + \mathbf{g}}{\tau}\right) \tag{4.6}$$

with $\mathbf{g}$ being sampled from a Gumbel distribution and τ a temperature parameter. $\mathbf{v}_t = onehot(\arg\max \mathbf{v}_t^*)$ is used to be passed to the next time step, while the gradient will flow through $\mathbf{v}_t^*$ during the backpropagation, being differentiable.

To conclude, also the discriminator owns an LSTM structure. At every time step t a vector $\mathbf{v}_t$ is processed, until all the T vectors of the random walk received as input are analyzed. At the end, the probability of the random walked to be real is produced.

4.3.3 Adjacency matrix and early stopping

Once the generator has learned the probability distribution of a random walk over the graph of interest, it can be used to produce the adjacency matrix of a new similar one. To this purpose, the first step is to make the generator produce a large amount of samples (in the paper it is suggested some hundreds thousands) which are exploit to build a score matrix S counting how many times an edge appears in any random walk. S is then symmetrized imposing $s_{ij} = s_{ji} = \max\{s_{ij}, s_{ji}\}$. To obtain the adjacency matrix, S must be made binary. Because of some possible unbalances in the sampling of nodes, applying strategies like thresholding or similar, may leave to singletons or many low degree nodes to be excluded. It is instead applied the following strategy:

1. for every node i, a neighbor j is sampled with probability $p_{ij} = \frac{s_{ij}}{\sum_v s_{iv}}$ This guarantees that no singletons are present. If and edge has been already selected, the procedure is repeated.

2. If the number of selected edges at previous point is not equal to the number of edges of the original graph, new edges are added by sampling them with probability $p_{ij} = \frac{s_{ij}}{\sum_{u,v} s_{uv}}$, until the desired number is reached

As said this procedure makes the sampling of edges more balanced, but does not guarantee connectivity. There is another issue regarding generalization capability of the NetGAN. In fact if the solution would converge to the very same graphs from which the random walked are sampled, which is a legitimate outcoming of the algorithm, it would be absolutely useless. Then some strategy should be used to guarantee that the model does not converge to it. One viable way is to adopt an early stopping strategy which controls how close the generated graph is to the original one. The authors propose two ways. The first consists in keeping a sliding window of random walks while training the GAN, using it to build the matrix S, which is enough for link prediction purposes. When the ROC curves that allow to evaluate the link prediction performances satisfy a certain requested precision, the training is stopped. The other way consists in stopping when the generated graphs overlaps with a certain percentage to the original graph: depending on the percentage, the generalization capability is chosen, depending on how much the generated graph is requested to be similar to the original or not.

4.3.4 Conclusions

[1] provides the results of NetGAN in the generation of new realistic graphs, showing that its performances are comparable or even better than other state-of-art algorithm dealing with same datasets that have been tested. A strong advantage is that no assumption over the data is done for the training

of this architecture, differently from the other explicit models that catch well only the characteristics that they model, while struggling with the other that are not taken into account. As said previously, another interesting property of NetGAN is that it performs well also on link prediction, even if it not trained specifically for this purpose, again obtaining comparable results to other state-of-art models. A final remark is on the hyperparamter tuning. It turns out that most of them are not critical, if chosen within reasonable intervals, for the performances (evaluated on link prediction). But the length of the random walk T is instead quite relevant, showing that the greater it is the better, even if the improvements saturates after a certain point.

4.4 GraphGAN

In the previous paragraphs of this chapter, two different kinds of graph input have been addressed. One was a set of point clouds, processed so that they could be associated to a set of graphs. The second was a single graph, whose structure had to be learned by NetGAN. In this paragraph, another architecture proposed in [4] is described, dealing with the second case of study: GraphGAN. But if the typology of problem studied is similar to the one of NetGAN, the idea over which GraphGAN relies is much more similar to the one presented for the point clouds: graph embeddings. In fact, it belong to the class of model referred to as Graph representation learning, whose objective is to represent each vertex of a graph as a low dimensional vector, to facilitate processing tasks over the graph. In particular the GAN mechanism is exploited to use not only the generator, but also the discriminator network. In fact GraphGAN is conceived to produce both a generative model, capable of learning the connectivity distribution $p_{true}(v|v_c)$ between each node of the graph v_c and all the remaining nodes, and a discriminative model capable of performing prediction over the link existence, outputting the probability of the edge presence between two nodes. So the generator $G(v, v_c; \theta_G)$, where θ_G are the parameters to be learned, produces a neighbor v of a certain input vertex v_c respecting the true distribution $p_{true}(v|v_c)$, whereas the discriminator $D(v, v_c; \theta_D)$, where θ_D are the parameters to be learned, outputs the probability that the edge between the nodes v and v_c exists. They play the well known min-max game described in the first chapter. In particular, if the graph posses V vertices than the min-max optimization becomes:

$$\min_{\theta_G} \max_{\theta_D} V(G, D) \tag{4.7}$$

with the objective function being:

$$V(G,D) = \sum_{c=1}^{V} \left(\mathbb{E}_{v\sim p_{true}(\cdot|v_c)}\left[\log D(v,v_c;\theta_D)\right] + \mathbb{E}_{v\sim G(\cdot|v_c;\theta_G)}\left[\log\left(1 - D(v,v_c,\theta_D)\right)\right]\right) \tag{4.8}$$

4.4.1 Discriminator optimization

The role of the discriminator is to correctly recognize pair of nodes which are connected by an edge or not by maximizing the log probability of assigning 1 to pair extracted from $p_{true}(v|v_c)$ an zero to those produced by the generator. In particular, if $\mathbf{d}_v$ and $\mathbf{d}_{v_c}$, belonging to a k-dimensional vector space, are the representation of v and v_c for the discriminator, then:

$$D(v,v_c) = \sigma(\mathbf{d}_v^T \mathbf{d}_{v_c}) = \frac{1}{1 + e^{\mathbf{d}_v^T \mathbf{d}_{v_c}}} \tag{4.9}$$

Properly ascending the gradient of $\log D(v,v_c)$ when the input comes from the true distribution and the gradient of $1 - \log D(v,v_c)$ when the input comes from the generator, trains the discriminator.

4.4.2 Generator optimization

On the contrary the generator must maximize the probability assigned to the pairs of nodes that it produces. To this purpose, the following gradient must be descended:

$$\bigtriangledown_{\theta_G} V(G,C) \tag{4.10}$$

or equivalently:

$$\sum_{c=1}^{V} \mathbb{E}_{v\sim G(\cdot|v_c)}\left[\bigtriangledown_{\theta_G} \log G(v|v_c)\ \log\left(1 - D(v,v_c)\right)\right] \tag{4.11}$$

But the exact form of $G(v,v_c)$ has still to be specified. One straightforward solution, consists in apply a softmax function to all vertices:

$$G(v,v_c) = \frac{e^{\mathbf{g}_v^T \mathbf{g}_{v_c}}}{\sum_{v\neq v_c} e^{\mathbf{g}_v^T \mathbf{g}_{v_c}}} \tag{4.12}$$

where $\mathbf{g}_v$ and $\mathbf{g}_{v_c}$, belonging to a k-dimensional vector space, are the representation of v and v_c for the generator. But this solution presents two major problems. The first is that the softmax must consider all the vertices to be sure to output a probability. And this operation must be repeated for each sample considered in the gradient descent. This is extremely expensive in terms of computational cost, considering that typical real graphs have millions of nodes. The other issue is that locally the graph presents important structural information, as described in the paragraph regarding point clouds, but it is not exploited, because all vertices are considered each time and all of them are treated in the same way. To overcome these difficulties, [4] proposes a new way to evaluate the softmax function, called Graph softmax. If v_c is the node under consideration, the first step is to perform a Breadth First Search on the graph staring from v_c. The result is a tree T_c rooted in v_c. Now, given any node v, $\mathcal{N}_c(v)$ is the set of the neighbors of v in T_c. Then, given another node $v_i \in \mathcal{N}_c(v)$, its relevance probability give v_c is:

$$p_c(v_i|v) = \frac{e^{\mathbf{g}_{v_i}^T \mathbf{g}_v}}{\sum_{v_j \in \mathcal{N}_c(v)} e^{\mathbf{g}_{v_j}^T \mathbf{g}_v}} \tag{4.13}$$

which is a softmax function over $\mathcal{N}_c(v)$. Each vertex v is reachable through a unique path from v_c belonging to T_c. Denoting this path as $P_{v_c \longrightarrow v} = (v_{r_0}, ..., v_{r_m})$ the Graph softmax for vertex v_c is defined as:

$$G(v, v_c) = \left(\Pi_{j=1}^{m} p_c(v_{r_j}|v_{r_{j-1}})\right)\ p_c(v_{r_{m-1}}|v_{r_m}) \tag{4.14}$$

Other than being normalized, it can be demonstrated that the Graph softmax decreases exponentially with the increase of the distance between v and v_c, thus providing the locality which was previously missing. Also, it can be demonstrated that its calculation has complexity $\mathcal{O}(d \log V)$, with d being the average node degree, which is typically small, but being the dependence on V just logarithmic, the complexity is greatly reduced. To conclude, the final aspect to be defined is the generating(sampling) strategy of the generator. It can be done either by evaluating the Graph softmax for all vertices $v \neq v_c$ and performing random sampling proportionally to their approximated connectivity probabilities, or a more effective strategy is using a special online generating method, which greatly reduces the complexity with respect to the offline method.

4.4.3 Conclusions

Properly alternating the training of discriminator and generator, according to the corresponding previous paragraph, results in the training of GraphGAN. [4] tests the GraphGAN architecture in three tasks: link prediction, node classification and recommendation. In all the cases, GraphGAN outperforms all the other learning techniques which are specifically trained as either generative or discriminative models. This is because of the adversarial training of the two networks, which allows the GAN based architectures to outperform other techniques in many tasks.

4.5 StackGAN

In this paragraph the attention is shifted to an architecture dealing with images generation. GANs proved to be very successful in generating realistic images, using a variety of strategies like the progressive growing presented in the second chapter. But while it is relatively easy to impose some kind of constraint on the content of the image, using for example some labeling as presented in paragraph 2.5 with conditional GAN, it is not easy to provide an high level interface that allows to constrain the image content based on an informal relatively complex text description. This would be particularly beneficial to computer graphics, where the passage from the high level concept to the pixel level details still requires a lot of human effort. In [3] a new architecture called StackGAN is proposed, with the target to contribute to solve this issue. The idea is to split the problem in two stages: in the first one, given the text description, only the corresponding primitive shape and colors are caught; the second, based on the output of the first stage and again on the text content, is able to capture text information that were previously omitted, thus producing higher resolution and photo-realistic details. These two stages are modeled as two distinct GAN, in which the first stage GAN, called Stage-I GAN, is used as input for the second, referred to as Stage-II GAN.

4.5.1 Conditioning Augmentation

Before passing to the aforementioned architectures, [3] proposes a special condition technique on the generator that helps improving the performances. In fact, introducing a text content in the training of GAN is equivalent to exploit the Conditional GAN architecture of chapter 2. So that if generator and discriminator of a standard GAN are the function G(z) and D(x), where z is the Gaussian variable and x the data input (in this case an image), then conditioning of the two to the text is equivalent to write them as G(z, c) and D(x, c), with c being the text content. Of course the text cannot be passed as it is, but rather it must be embedded in some kind of vectorial representation (denoted as t), which is assumed to be done by an external encoder φ_t, which is pretrained independently from the GAN. In other models for image generation, the text embedding is nonlinearly transformed to generate conditioning latent variables to be fed to the generator. But being the dimension of the embedding space quite large, this causes typically a discontinuity in the latent space that as explained in the first chapter is not desiderable. The Conditioning Augmentation technique aims at solving this issue. This is done using φ_t not as the conditioning variable itself, but as a set of parameters through which the variable is produced. In particular, instead of having a c which is the result of φ_t, it used a variable $\hat{c}$ which is an independent Gaussian $\mathcal{N}(\mu(\varphi_t), \Sigma(\varphi_t))$. But because the text embedding is used just to set the parameters of the Gaussian, the number of samples that can be taken from the distribution can be greater than the embedding itself, thus producing a major number of pairs (z, $\hat{c}$) to be passed to the generator instead of (z, c). Also, to further improve the smoothness of the conditioning distribution and to reduce the overfitting, a regularization term is added to the generator cost function:

$$D_{KL}\left(\mathcal{N}\left(\mu(\varphi_t), \Sigma(\varphi_t)\right) \| \mathcal{N}(0, I)\right) \tag{4.15}$$

with D_{KL} being the Kullback-Leibler divergence.

4.5.2 Stage-I GAN

As previously said, the first GAN has the role to produce low resolution images, just focusing on the rough shape and colors of objects. Starting from the embedding of the text description φ_t, which corresponds to a certain image I_0, the Conditioning augmentation is applied, so that a set of $\hat{c}_o$ variables are sampled from $\mathcal{N}(\mu_0(\varphi_t), \Sigma_0(\varphi_t))$. Then the discriminator, denoted as D_0, and the generator, denoted as G_0, are alternatively trained in the standard way ascending and descending respectively the cost functions:

$$\mathcal{L}_{D_0} = \mathbb{E}_{(I_0,t)\sim p_{data}}\left[\log D_0\left(I_0, \varphi_t\right)\right] + \mathbb{E}_{z\sim p_z, t\sim p_{data}}\left[\log\left(1 - D_0\left(G_0\left(z, \hat{c}_0\right), \varphi_t\right)\right)\right] \tag{4.16}$$

$$\mathcal{L}_{G_0} = \mathbb{E}_{z\sim p_z, t\sim p_{data}}\left[\log\left(1 - D_0\left(G_0\left(z, \hat{c}_0\right), \varphi_t\right)\right)\right] + \lambda\, D_{KL}\left(\mathcal{N}\left(\mu_0(\varphi_t), \Sigma_0(\varphi_t)\right) \| \mathcal{N}(0, I)\right) \tag{4.17}$$

which are the traditional GAN objective functions but in the conditional version, with $\hat{c}_0$ being the conditioning variable and with the regularization term for the generator, as described in the previous paragraph. Going deeper in the implementation detail, the functions μ_0 and Σ_0 are implemented as two fully connected networks receiving as input φ_t. In particular, of Σ_0 is obtained the diagonal σ_0. Then $\hat{c}_0$ is obtained as $\mu_0 + \sigma_0 \odot \epsilon$ with $\odot$ being the element-wise Hadamard product and $\epsilon \sim \mathcal{N}(0, I)$. Then it is concatenated with a N_z dimensional noise vector to generate a $W_0 \times H_0$ image using a series of upsampling blocks. Speaking of the discriminator, φ_t is compressed in N_d dimension and then spatially replicated to form a $M_d \times M_d \times N_d$ tensor. Instead the image is reduced to dimension $M_d \times M_d$ using downsampling blocks and it is concatenated with the previous tensor and the result is fed to a 1×1 convolutional layer. Finally, a fully connected layeer with one neuron output the decision score.

4.5.3 Stage-II GAN

In the previous stage, low resolution images are produced, missing very probably also part of text content information. Conditioning Stage-II GAN on this result and again on the text embedding it is possible to improve the generated samples up to have them similar to realistic images. So in this case the input to the generator is not the Gaussian variable z but the result of the generator $s_0 = G_0(z, \hat{c}_0)$ and another latent variable $\hat{c}$ which derives from the same text encoder of $\hat{c}_0$ but whose average and variance parameters are obtained with a different fully connected layer. So the equations of the objective-functions becomes:

$$\mathcal{L}_D = \mathbb{E}_{(I,t)\sim p_{data}}\left[\log D\left(I, \varphi_t\right)\right] + \mathbb{E}_{s_0\sim p_{G_0}, t\sim p_{data}}\left[\log\left(1 - D\left(G\left(s_0, \hat{c}\right), \varphi_t\right)\right)\right] \tag{4.18}$$

$$\mathcal{L}_G = \mathbb{E}_{s_0\sim p_{G_0}, t\sim p_{data}}\left[\log\left(1 - D\left(G\left(s_0, \hat{c}\right), \varphi_t\right)\right)\right] + \lambda\, D_{KL}\left(\mathcal{N}\left(\mu_0(\varphi_t), \Sigma_0(\varphi_t)\right) \| \mathcal{N}\left(0, I\right)\right) \tag{4.19}$$

where:

- G is the generator of Stage-II GAN
- D is the discriminator of Stage-II GAN
- (I, t) are taken from the real data distribution

Again alternatively optimizing them with gradient descent results in the training of generator and discriminator. Speaking of the implementation details of the generator, φ_t is used again to generate the N_g dimensional text embedding $\hat{c}$ which is spatially replicated as a tensor in $M_g \times M_g \times N_g$. At the same time, s_0 is passed to downsampling blocks until it falls into a $M_g \times M_g$ space. Once the two tensors have been coupled, they undergo to a series of residual blocks which learn a multimodal representation of the features in the text and image domains. At the end, upsamplig blocks brings them into a $W \times H$ resolution. The discriminator has a similar structure to the one in Stage-I, but it takes as fake examples both synthetic images with their text embedding and also real images but with mismatched text embeddings.

4.5.4 Conclusions

The strategy of splitting the learning is similar to the one of progressive growing analyzed in chapter 2 (which follows as publication date Stake-GAN), even if here the domain of application is slightly different, because in this case the image synthesis is related to the extraction of information from a text. But again the idea is very successful, producing state of art performances on a number of dataset, competing with or outperforming other strategies challenging the same problem.

Bibliography

[1] Daniel Zügner Stephan Günnemann Aleksandar Bojchevski, Oleksandr Shchur. Netgan: Generating graphs via random walks, 1 June 2018.

[2] Diego Valsesia Giulia Fracastoro. Learning localized generative models for 3d point clouds via graph convolution, 2018.

[3] Hongsheng Li Shaoting Zhang Xiaogang Wang Xiaolei Huang-Dimitris Metaxas Han Zhang, Tao Xu. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks, 5 August 2017.

[4] Jialin Wang Miao Zhao Weinan Zhang Fuzheng Zhang-Xing Xie Minyi Guo Hongwei Wang, Jia Wang. Graphgan: Graph representation learning with generative adversarial nets, 22 November 2017.

[5] Aaron Courville Ian Goodfellow, Yoshua Bengio. Deep learning, 18 November 2016.

[6] Mehdi Mirza Bing Xu David Warde-Farley Sherjil Ozair Aaron Courville Yoshua Bengio Ian J. Goodfellow, Jean Pouget-Abadie. Generative adversarial nets, 10 June 2014.

[7] Martin Arjovsky Vincent Dumoulin Aaron Courville Ishaan Gulrajani, Faruk Ahmed. Improved training of wasserstein gans, 25 December 2017.

[8] Léon Bottou Martin Arjovsky. Towards principled methods for training generative adversarial networks, 17 January 2017.

[9] Soumith Chintala Martin Arjovsky and Léon Bottou. Wasserstein gan, 9 March 2017.

[10] Nikos Komodakis Martin Simonovsky. Dynamic edge-conditioned filters in convolutional neural networks on graphs, 8 August 2017.

[11] Simon Osindero Mehdi Mirza. Conditional generative adversarial nets, 6 November 2014.

[12] Pierre Vandergheynst Michaël Defferrard, Xavier Bresson. Convolutional neural networks on graphs with fast localized spectral filtering, 5 February 2017.

[13] Jurgen Schmidhuber Sepp Hochreiter. Long short-term memory, 1997.

[14] Samuli Laine Jaakko Lehtinen Tero Karras, Timo Aila. Progressive growing of gans for improved quality, stability, and variation, 26 February 2018.