POLITECNICO DI TORINO

Master's Degree in Ingegneria Informatica (Computer Engineering)

Master's Degree Thesis

Image generation using deep adversarial generative models on graphs

Supervisors

Prof. Enrico Magli

Candidate

Michele D'Amico

July 2020

Summary

Generative Adversarial Networks (GANs) [1] are a very promising category of generative models used to approximate unknown data distributions for sampling purposes. Nevertheless, their training instability problems have hindered the possibility of experimenting with a wide variety of different GANs architectures. The introduction of Wasserstein GAN [2] and Wasserstein GAN-GP [3] overcomes such limitation providing the possibility to successfully train a broader class of architectures without instability or convergence issues. Among the possible model architectures for image generation task, convolutional neural networks (CNN) excels as for many other subfields in deep learning. Notwithstanding the nice properties of convolutional layers, which are the building blocks of CNNs, the convolution is a local operator and for this reason lacks in effectively capturing long-term dependencies, which are fundamental for reproducing plausible samples for image classes that present a well-determined structure.

To this end, this project proposes the integration of the graph convolutional layer [4] in the generator of a convolutional WGAN-GP in an attempt to remedy this limitation. The graph-convolutional layer will extract a graph representation of image data dynamically, generating a k-nearest neighbor graph. In this representation, each vertex has its vector of features taken from the activation maps and is connected to the k less distant nodes. The distance is determined through the Euclidean metric in the feature space rather than in the spatial domain as for regularly structured data. Consequently, the convolution is performed as a node aggregation function among the central node and its neighborhood of size k. Thus, this operator would result in an adaptive receptive field on the areas of the hidden layers activation maps that share some features similarities with the central node of convolution. The graph convolution will not substitute regular convolution, but instead it will extend in a complementary way its receptive field to capture also non-local dependencies. From the experiments carried out, however it emerges that this method does not provide the expected improvements. In fact, from an evaluation of the generated samples based on the inception score and on the naked-eye observation, samples generated by the networks with the graph

convolution layer are very similar to baseline samples obtained through a fully convolutional model.

Acknowledgements

Firstly, I would like to thank Professor Enrico Magli for his patience and availability, and for having proposed this thesis project in such an interesting research area. I also thank the Assistant Professors Giulia Fracastoro and Diego Valsesia for the tips and technical advice they gave me.

Secondly, I am grateful to my family and to my friends for always having supported me.

Table of Contents

List of Tables

List of Figures

Symbols

x	A scalar variable
$\boldsymbol{x}$	A vector
$\boldsymbol{A}$	A matrix
$\boldsymbol{I}$	Identity matrix
x	A scalar random variable
$\mathbf{x}$	A vector random variable
$\mathbf{A}$	A matrix random variable
$\mathbb{B}$	A set
$\mathbb{R}$	The set of real number
$\{x_1, x_2, ..., x_n\}$	The set element definition
$\mathcal{G}$	A graph
$\mathcal{V}$	A graph vertex set
$\mathcal{E}$	A graph edge set
$\lvert\boldsymbol{A}\rvert$	Determinant of matrix A
$\nabla_{\boldsymbol{x}} y$	Gradient of y w.r.t. $\boldsymbol{x}$
$\mathbf{J}_{\boldsymbol{x}} f(\boldsymbol{x})$	Jacobian matrix of $f(\boldsymbol{x})$ w.r.t. $\boldsymbol{x}$
$\mathcal{X}$	A metric space

$p(\mathrm{x})$	A probability distribution over a generic random variable x, either discrete or continuous
$p(\mathrm{x} = x)$	The probability of random variable x being in state x
$p(x\|y)$	The probability of x conditioned on y
$p(x)$	A probability density of x
$\mathrm{x} \sim p(x)$	A random variable x has distribution $p(x)$
$\mathbb{E}_{\mathrm{x}\sim p(x)}[f(x)]$	Expectation of function $f(x)$ under $p(x)$ distribution
$D_{\mathrm{KL}}(p\|q)$	Kullback-Leibler divergence between p and q
$JSD(p\|q)$	Jensen–Shannon divergence between p and q
$f(x;\theta)$	Parametrized function of x
$F_\theta(x)$	Parametrized function of x
$\sigma(x)$	Sigmoid function
$\|\boldsymbol{x}\|$	Norm of x, l2 unless otherwise stated

Acronyms

KL Kullback-Leibler

GAN Generative Adversarial Network

WGAN Wasserstein Generative Adversarial Network

WGAN-GP Wasserstein Generative Adversarial Network with Gradient Penalty

CIFAR Canadian Institute For Advanced Research

DAG Direct Acyclic Graph

ReLU Rectified Linear Units

SBN Sigmoid Belief Network

FVBN Fully Visible Belief Network

CDF Cumulative Distribution Function

VAE Variational Autoencoder

MLP Multilayer Perceptron

CNN Convolutional Neural Network

DCGAN Deep Convolutional Generative Adversarial Network

EMD Earth Mover's Distance

ECC Edge-Conditioned Convolution

SVD Singular Value Decomposition

Chapter 1

Introduction

This chapter briefly introduces the class of generative models and some basic concepts such as graphical models and neural networks for a broad understanding of how they work. In addition, different types of generative models type are described, distinguished by their architecture and their training algorithm, in order to realize which category the Generative Adversarial Networks belongs to.

1.1 Generative models

Machine learning models can be grouped into two macro-categories: discriminative model and generative model. Broadly speaking, in a classification task, the discriminative model learns from data the conditional probability distribution $p(\mathrm{y}|\mathrm{x})$ directly. Thus, given an observation x, the model can determine to which class y it belongs by calculating the probability distribution $p(\mathrm{y}|\mathrm{x} = x)$. In contrast, a generative model learns the joint distribution $p(\mathrm{x}, \mathrm{y})$, and it makes uses of Bayes rule to calculate the posterior probability, namely $p(\mathrm{y}|\mathrm{x}) = p(\mathrm{x}|\mathrm{y})p(\mathrm{y})/p(\mathrm{x})$.

Discriminative models usually perform better in classification tasks, since fitting generative models is generally more complicated. For this reason, generative models, in many cases, require some approximations that, if too strict, may lead the model to provide estimations with non-negligible errors. A notorious example is the naive Bayes classifier, which comes with the assumption of conditional independence among predictors $\mathbf{x}$ given the class variable y. However, generative models are not so-called by chance. In fact, these models by learning $p(\boldsymbol{x}, y)$ or $p(\boldsymbol{x})$ distribution from data are able to generate new samples, and not just classify new data observations.

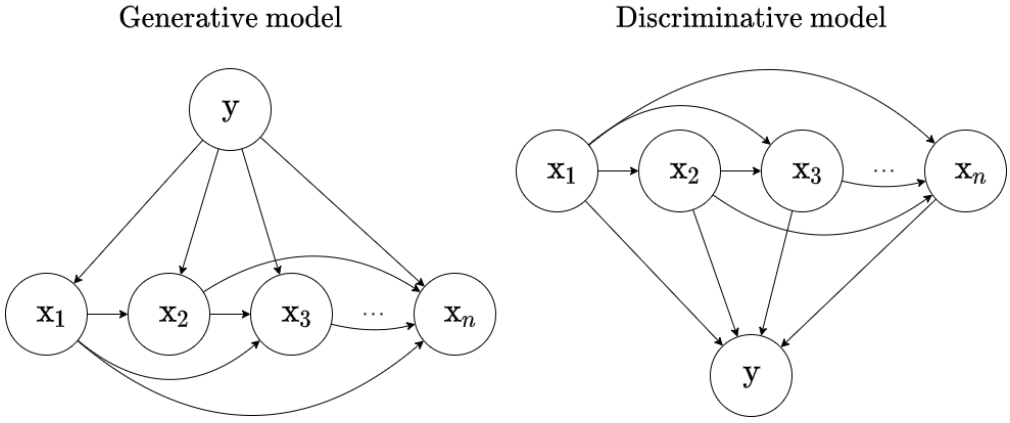


Figure 1.1: Generative vs. discriminative graphic models.

In general, given a true probability distribution $p_{\text{data}}(\boldsymbol{x})$, the purpose of a generative model is to describe a distribution $p(\boldsymbol{x}; \boldsymbol{\theta})$, which can provide a reasonable estimate of $p_{\text{data}}(\boldsymbol{x})$, given the right parameters $\boldsymbol{\theta}$. E.g., for a two dimensional Gaussian family of distributions $p(\boldsymbol{x}; \boldsymbol{\theta})$, $\boldsymbol{\theta} = [\boldsymbol{\mu}, \boldsymbol{\Sigma}]$. In a real scenario $p_{\text{data}}(\boldsymbol{x})$ is unknown except for some samples $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ drawn from it that constitute the dataset $\boldsymbol{X}$. These observations outline the empirical distribution $\hat{p}_{\text{data}}(\boldsymbol{x})$, which puts probability mass $1/m$ on each of the m data points. Now, maximum likelihood estimation is the method adopted for $\boldsymbol{\theta}$ parameters estimation :

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{1.1}$$

Then, for the properties of strictly monotonic functions, a value of θ that maximizes the log-likelihood will also maximize the likelihood function:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{1.2}$$

Rescaling the cost function by a constant factor will not alter it and consequently, the log-likelihood can be written as an expectation. This form emphasize that its maximization corresponds to the minimization of the Kullback-Leibler (KL) divergence between $\hat{p}_{\text{data}}(\boldsymbol{x})$ and $p(\mathbf{x}; \boldsymbol{\theta})$:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \log p(\mathbf{x}; \boldsymbol{\theta}). \tag{1.3}$$

The KL divergence provides a measure of how different are two distribution of the

same random variable x as:

$$D_{\mathrm{KL}}(\hat{p}_{\mathrm{data}} \| p) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\mathrm{data}}} \left[\log \frac{\hat{p}_{\mathrm{data}}(x)}{p(x)} \right] \tag{1.4}$$

$$= \mathbb{E}_{\mathrm{x} \sim \hat{p}_{\mathrm{data}}} [\log \hat{p}_{\mathrm{data}}(x) - \log p(x)]. \tag{1.5}$$

In minimizing the divergence w.r.t. the $\boldsymbol{\theta}$ parameter, the constant term $\mathbb{E}_{\mathrm{x} \sim \hat{p}_{\mathrm{data}}} [\log \hat{p}_{\mathrm{data}}(x)]$ can be ignored and the cost function assume the same form of equation (1.3).

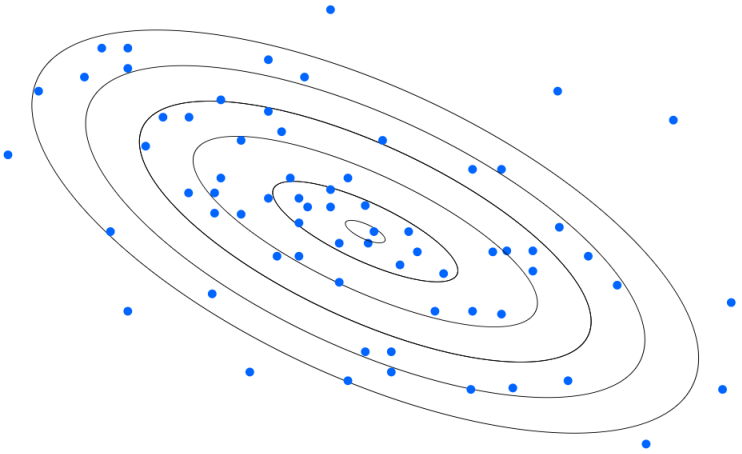
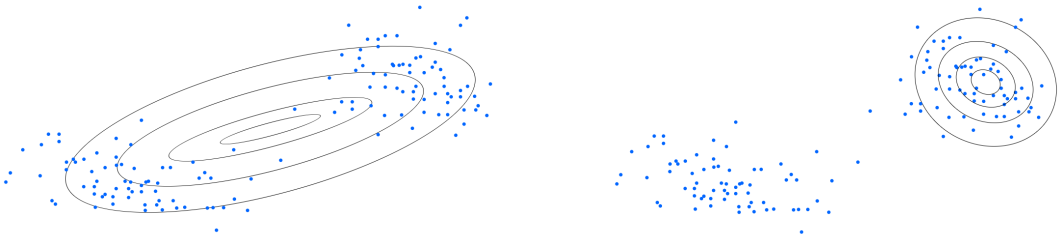


Figure 1.2: Maximum likelihood estimation of 2D Gaussian.

The previous example describes an oversimplified case because, usually, real data distribution $p_{\mathrm{data}}(\boldsymbol{x})$ has a more complex density. Plus, the maximum likelihood estimator requires that the true empirical distribution $\hat{p}_{\mathrm{data}}(\boldsymbol{x})$ lies in the model family represented by $p(\cdot\,; \boldsymbol{\theta})$, to ensure consistency property. In a real scenario, this condition usually is not met, and when dealing with model misspecification, optimizing the maximum likelihood, hence minimizing $D_{\mathrm{KL}}(\hat{p}_{\mathrm{data}} \| p_\theta)$ encourages the model to overgeneralize $p(\boldsymbol{x}; \boldsymbol{\theta})$ over $\hat{p}_{\mathrm{data}}(\boldsymbol{x})$ as can be seen in Figure 1.3a. I.e. $p(\boldsymbol{x}; \boldsymbol{\theta})$ tend to cover all the areas where $\hat{p}_{\mathrm{data}}(\boldsymbol{x}) > 0$, introducing density in areas where real data distribution has none. This occurs because of $D_{\mathrm{KL}}(\hat{p}_{\mathrm{data}} \| p_\theta)$ definition, which assumes infinite values for $p(\boldsymbol{x}; \boldsymbol{\theta}) = 0$ and $\hat{p}_{\mathrm{data}}(\boldsymbol{x}) > 0$, then forces $p_\theta(\boldsymbol{x}) > 0$ in regions that present some probability density $\hat{p}_{\mathrm{data}}(\boldsymbol{x})$.

The KL divergence is not a metric since it not satisfies symmetry property. In particular, the $D_{\mathrm{KL}}(p_\theta \| \hat{p}_{\mathrm{data}})$ minimization encourages an entirely different behavior by pushing $p(\boldsymbol{x}; \boldsymbol{\theta})$ to have a low probability density in the same region where

$\hat{p}_{\text{data}}(\boldsymbol{x})$ is low. Again, the underlying reason is that the reverse KL assumes infinite values for $\hat{p}_{\text{data}}(\boldsymbol{x}) = 0$ and $p(\boldsymbol{x}; \boldsymbol{\theta}) > 0$, then forces $p(\boldsymbol{x}; \boldsymbol{\theta}) = 0$. In model misspecification cases, $D_{\text{KL}}(p_\theta \| \hat{p}_{\text{data}})$ is minimized by distributions with low probability mass in areas where $p_{\text{data}} = 0$, leading to solutions that ignore some modes in data. Despite this drawback, the found distribution tends to undergeneralize data, and sampling from it would result in more convincing observations compared to forward KL solutions. Unfortunately, it is impossible to compute reverse KL since it requires evaluating the true probability of a generated sample. However, under some conditions minimizing the GAN objective corresponds to minimizing the JS divergence, which likewise results to be robust to overgeneralization.



(a) KL divergence **(b)** JS divergence

Figure 1.3: Illustration of the different learned distribution based on the minimized distance during training. Data drawn from a mixture of Gaussians is fitted by an isotropic Gaussians by either minimizing KL divergence or JS divergence.

1.1.1 Graphical Models

Describing a distribution of images for a generative model means providing an estimate of an unknown multivariate continuous distribution p_{data} with its support in $\mathbb{R}^{H \times W \times C}$ from which the sample observations in the dataset are drawn. For example, CIFAR10 is a widely used dataset for prototyping purposes, consisting of 60000 color images equally distributed among 10 label classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each of its 32×32 images with 3 color channels can be modeled with 3072 highly correlated observable random variables. Directed graphical models allow describing the dependencies among random variables in a structured way via conditional dependence relationships. They have a massive impact on reducing the computational complexity of inference and the spatial complexity of storing model parameters. A belief network models

the joint probability of $\mathbf{x} = (\mathrm{x}_1, ..., \mathrm{x}_\mathrm{d})$ as:

$$p(\mathrm{x1}, ..., \mathrm{x_d}) = \prod_{i=1}^{d} p(\mathrm{x}_i|\mathrm{pa}(\mathrm{x_i})), \tag{1.6}$$

where $\mathrm{pa}(\mathrm{x_i})$ symbolize the parents of $\mathrm{x_i}$ random variable. The Direct Acyclic Graph (DAG) expresses the belief network structure, in which the conditional dependencies among random variables are depicted with arrows directed from the parent node toward the child node. Making direct conditional dependencies explicit through a belief network allows a simplification of the product rule factorization of the joint probability, in which each random variable is conditioned only on its parents. This simplification follows from the assumption made with the graph structure as in Figure 1.4. In fact, a general model for a joint distribution without any assumption corresponds to a fully connected graph, where each node i is connected to the previous numbered $1, ..., i-1$ since the missing links provide most of the information in a belief network and in any case the graphical model implies an ordering among the nodes.

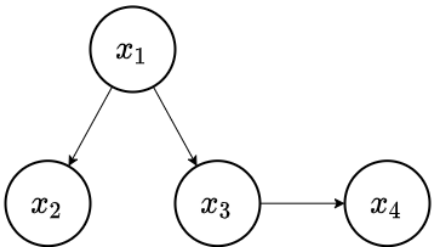


Figure 1.4: Simple graph example of a belief network where $p(x_1, x_2, x_3, x_4) = p(x_4|x_1, x_2, x_3)p(x_3|x_2, x_1)p(x_2|x_1)p(x_1) = p(x_4|x_3)p(x_3|x_1)p(x_2|x_1)p(x_1)$ for the conditional independence assumptions modeled by the network.

For these reasons a Bayesian network capable of capturing all the dependencies among the pixels in an image, through direct connections, would end up in a fully connected model with a huge number of parents per pixel and consequently, an unmanageable number of parameters, which makes its application inefficient or intractable. The introduction of the latent variables $\mathbf{z}$ is a viable solution to model such dependencies because they would be capable of capturing the interactions among visible variables $\mathbf{x}$ indirectly, provided that they have, in turn, a direct dependency with the visible variables involved. This approach leads to the definition of a joint distribution over the latent and observable variables $p(\mathbf{z}, \mathbf{x})$ [5] [6], which marginalization will describe the distribution of the observable variables:

$$p(\boldsymbol{x}) = \mathbb{E}_\mathbf{h} p(\boldsymbol{x}|\boldsymbol{z}). \tag{1.7}$$

Intuitively, discrete or continuous latent r.v $\mathbf{z}$ can describe explanatory factors of a distribution of observable r.v. $\mathbf{x}$, and technically, they allow to express complex distribution $p(\boldsymbol{x})$ in terms of more tractable joint distribution $p(\boldsymbol{x}, \boldsymbol{z})$. For instance, the Gaussian mixture distributions with K components can be easily formulated in terms of latent and visible variables since the joint distribution can be defined in term of a marginal probability and a conditional probability, $p(\boldsymbol{x}, \boldsymbol{z}) = p(\boldsymbol{z})p(\boldsymbol{x}|\boldsymbol{z})$. In this formulation the discrete latent variables $\mathbf{z} \sim p(\boldsymbol{z})$ as 1-of-K vector ($z_k = 1$ only for the element k in the vector), will encode which of the K Gaussian is responsible for that sample, whereas each conditional distribution,

$$p(\boldsymbol{x}|\boldsymbol{z}) = \prod_{k=1}^{K} \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_k} \,, \tag{1.8}$$

describes the corresponding Gaussian component. From the definition of $\mathbf{z}$ follows a categorical distribution for the marginal probability, namely:

$$p(\boldsymbol{z}) = \prod_{k=1}^{K} \phi_k^{z_k} \,, \tag{1.9}$$

in which ϕ_k is the mixing coefficient that specify the probability of seeing an element from component k.

In this example, the dependency between $\mathbf{z}$ and $\mathbf{x}$ is simple and expressed through a Gaussian distribution. However, in reality, this is often not the case, and the conditional distributions that express dependencies within the random variables in a given graphical model are very complex function, hence the need to express them through deep neural networks.

1.2 Neural Networks

The universal approximation theorem states that a multilayer perceptron, with a single hidden layer composed of a finite number of neurons, is a universal approximator of continuous functions on $\mathbb{R}^n$. In other words, given a continuous function, a neural network can provide an approximation, as good as needed, at the expense of increasing the hidden layer width to be exponentially large. However, this theorem did not mention how large the network should be to reach the desired degree of accuracy. In any case, when learning from a limited set of data, the representation capacity is not the only requirement since the training algorithm used for parameter fitting may still miss the right parameters or overfit data providing a different approximation function that does not reflect the one that underlies data.

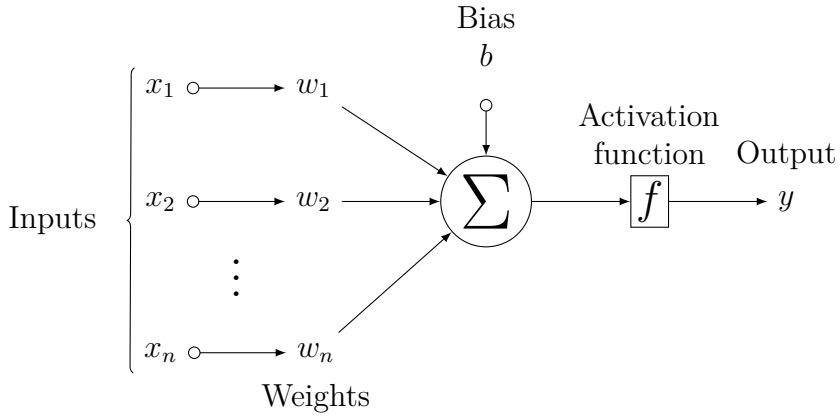


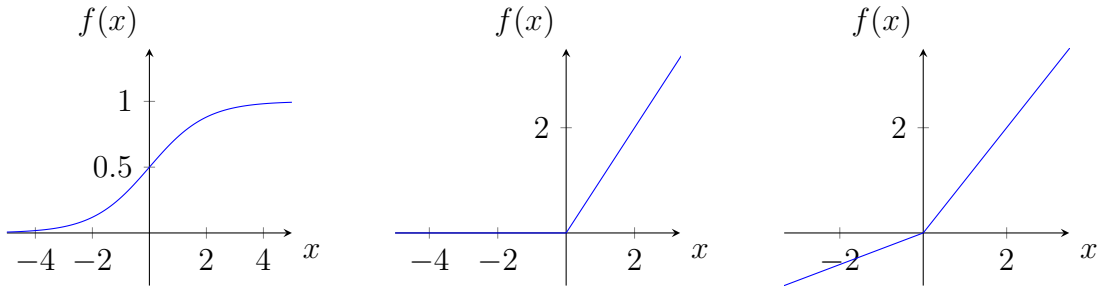
Figure 1.5: Mathematical neuron model

The basic building block of a neural network is the neuron. In nature, a neuron is a particular type of cell capable of communicating with other neurons through electric impulses that generate chemical messenger, neurotransmitters. Consequently, each neuron cell can receive the input signals from the others on dendrites and can produce a spike on its axon if a certain electric potential threshold is reached. Inspired by biology, the mathematical neuron model presents some analogies with it. Specifically, in its formulation, given the connection weights $w_1, ...w_i$ and a bias term b, it will receive multiple incoming signals $x_1, ..., x_i$ and produce the output as:

$$y = f\left(\sum_i x_i w_i + b\right), \tag{1.10}$$

where f is the activation function that emulates the potential threshold behavior. Historically, the sigmoid function $\sigma(x) = 1/(1+e^{-x})$ was then main used activation function, whereas nowadays, ReLU (Rectified Linear Units) and its variant Leaky ReLU became the most frequent ones in modern architectures. Empirical evidence shows that ReLU $f = \max(0, x)$ accelerates the network convergence during training because of its shape without any gradient saturating zone. The major drawback of ReLU is its susceptibility to high learning rates, which for large weight updates, drive more than necessary neurons in its $x < 0$ zone, where they irreversibly dies. For this reason, Leaky ReLU tries to mitigate this problem by introducing a small slope in the negative region.

Feedforward neural networks are arranged in multiple layers, each of these comprehends several neurons which work in parallel and provide the activations for such layer. The connections are defined between neurons that belong to adjacent layers only. Thus, each neuron at layer l receives signals from layer $l-1$ neurons,



(a) Sigmoid **(b)** ReLU **(c)** Leaky ReLU

Figure 1.6: Comparison between three of the most used activation functions

which in turn is weighted and propagated to all neurons at layer $l+1$, after passing through the activation function. Consequently, neural networks learn from a family of functions that involves the composition of as many functions as the layers defined in the architecture: $y = f^l(...f^2(f^1(x)))$. The last layer is generally called the output layer, whereas the first one is the input layer, and the layers in between are called hidden layers because they usually are not directly inspected. In fact, generally in evaluating the network performances, only the last layer output is assessed.

Designing a neural network requires the definition of the architecture of the constituent layers and the choice of a cost function. Suppose that f^* is the function that the model wants to approximate with a specific neural network architecture, capable of describing a family of function parameterized by $\boldsymbol{\theta}$. Since one is interested only in the best-approximating function a cost function is defined in order to evaluate what is the error between the true function f^* and $f(\boldsymbol{x};\boldsymbol{\theta})$ for some parameters $\boldsymbol{\theta}$. In a real scenario, where f^* is unknown, a dataset consisting of the observed noisy samples from f^* is nevertheless provided. Consequently, the cost function will measure the error of the model in fitting those observed samples. Regardless of the problem that the network needs to tackle, the training procedure aims to find the most suitable parameters to better approximate the target function f^*, which is typically a conditional distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$. This is achieved through maximum likelihood estimation, or equivalently from the minimization of the negative log-likelihood cost function:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p(\boldsymbol{y}|\boldsymbol{x}) \tag{1.11}$$

A further specification of $p(\boldsymbol{y}|\boldsymbol{x})$ completely defines J. For example, in the binary classification task, plugging in a Bernoulli distribution as $p(\boldsymbol{y}|\boldsymbol{x})$ would lead to the binary cross-entropy loss function, and conversely, mean squared error loss function derives from constraining $p(\boldsymbol{y}|\boldsymbol{x})$ distribution to be Gaussian with fixed variance.

The extensive use of cost function in machine learning is motivated by their convex shape that is easy to optimize. Unfortunately, this assumption does not hold for neural networks because they present multiple local minima in the cost function hyper-surface. In practice, it turns out that neural networks still achieve very good results in many cases, even though they converge to a local minimum.

To get a complete picture, it worth mentioning some aspects about the training algorithm, namely stochastic gradient descent, used for learning through weight parameters update, whose effect may minimize the cost function. The training process involves different steps, and the following is a brief overview of them. The first step requires only the sampling of a batch of data $\mathbb{B} = \{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$ from the entire dataset $\mathbb{X}$. In the second step, called the forward propagation, for each input $\boldsymbol{x}$ in the batch, the information propagates all through the network enabling the computation of the cost function as the expectation of the error over the observation in the mini-batch. Then, in the third step, the partial derivatives are computed for each connection weight. The partial derivatives of the cost function w.r.t. to each weight $w_{i,j}$ reveal how much the cost function changes for a slight change in $w_{i,j}$. The calculation of the derivatives is done applying the chain rule, a technique to compute the composition of functions derivatives. Let x be a real number and f and g two differentiable functions such that $z = f(g(x))$, then the chain rule states:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{1.12}$$

where $y = g(x)$. This concept is easily extended to vectors in computing gradients. Given a generic neuron j at layer l and an incoming weighted connection from neuron i, at previous layer $l-1$, backpropagation allows to calculate partial derivatives for each weight $\partial J / \partial w_{i,j}^{(l)}$ efficiently. The gradients of the cost function w.r.t. $w_{ij}^{(l)}$ are computed starting from the last layer in the network and passing the intermediate gradients to downstream layers, as in the forward propagation, but in reverse order. In a feedforward neural network, at each layer l the input signal for a neuron comes from the activations of the neurons at the previous layer $a_i^{(l-1)}$, and the weighted sum of the input signals is accordingly in the form $z_j^l = \sum_i w_{i,j}^l a_i^{l-1} + b_j^l$. Thus, given a neuron i at a generic hidden layer l, the partial derivative of the loss w.r.t. z_j^l can be defined as:

$$\delta_j^l = \frac{\partial J}{\partial z_j} = \sum_k \frac{\partial J}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = f'(z_j^l) \sum_k w_{j,k}^{l+1} \delta_k^{l+1} \tag{1.13}$$

The application of chain rule makes explicit the upstream gradient term δ_k^{l+1}, which is backpropagated from the next layer to avoid recomputing it. For a given weight $w_{i,j}^{(l)}$, on the connection between neuron j at layer l and neuron i at layer $l-1$, the

partial derivative of the cost function w.r.t. is defined as:

$$\frac{\partial J}{\partial w_{i,j}^{l}} = a_i^{l-1}\delta_j^l. \tag{1.14}$$

Likewise, the partial derivative for bias term b_j^l can be obtained as:

$$\frac{\partial J}{\partial b_j^l} = \delta_j^l. \tag{1.15}$$

In the fourth and last step of training, each weight and bias is updated moving toward the nearest minimum according to the estimate provided by the corresponding partial derivative:

$$w_{i,j}^{l} = w_{i,j}^{l} - \epsilon\frac{\partial J}{\partial w_{i,j}^{l}} \tag{1.16}$$

$$b_{i,j}^{l} = b_{i,j}^{l} - \epsilon\frac{\partial J}{\partial b_j^l}, \tag{1.17}$$

where ϵ is the learning rate, a scalar hyperparameter on which depends the magnitude of the update. This whole four step training procedure is performed iteratively until a stopping criteria is met.

1.3 Deep Generative Models

Deep generative models use deep computational graphs to define the conditional distributions or in general the interactions among the random variables in the model. It is crucial to distinguish the computational graph, responsible for describing the sequence operation that the neural network will perform, from the graphical model describing interactions among random variables. It is, then, not surprising that deep generative models without a graphical model and latent variables may exist.

Latent and visible random variables in deep generative graphical models are organized in layers, and a dense number of connections links adjacent layers encouraging a sparse representation of lower layer variables. As a result, also visible units x_i are connected with multiple hidden units h_j, which therefore provide a distributed representation of x_i. Besides, variables interactions are learned from data, so in general latent variable learns to represent concepts poorly interpretable by a human.

According to [7] the deep generative models could be distinguished based on

whether they rely either on a directed or an undirected graphical model, whether they define deterministic or stochastic layers, and whether they describe an explicit or an implicit distribution over the observable random variables $\mathbf{x}$.

1.3.1 Stochastic models

Stochastic generative models consist of none, one or more connected layers of hidden random variables in addition to the layer of observable ones. The linking edges between units can be directed or undirected, and the type of this connection will determines the model categorization.

Directed stochastic model

Consider a given set $\mathbb{S} = \{s_1, s_2, ..., s_N\}$ of binary or real-valued stochastic variables, Bayesian networks can intuitively describe the existing causal dependencies among variables. A deep stochastic model generates new samples via ancestral sampling based on the underlying graphical model, namely considering only the direct dependency between the random variable and its ancestors. Conditional probability for s_i is defined as:

$$p(s_i|s_1, s_2, s_{i-1}) = p(s_i|\text{pa}(s_i)). \tag{1.18}$$

Sigmoid belief networks [8] belong to this category, and as its name may suggest, in this model stochastic latent variables are designed to be binary: $s_i \in \{0, 1\}$. It follows that the probability for the activation of hidden binary units is:

$$p(s_i = 1|\text{pa}(s_i)) = \sigma(\sum_j W_{ji} s_j + b_i), \tag{1.19}$$

where $\boldsymbol{W}$ and $\boldsymbol{b}$ are learning parameters of the network, and variables in graph are ordered such that $W_{ji} \neq 0$ for $j \geq i$. SBN learning is based on maximum likelihood estimation. Hence, it aims to find the weight values for the network that maximize the likelihood of the observable units for training data. The learning rule for parameter update that derives from it is $\Delta W_{ji} = \eta\ s_j\ (s_i - p(s_i = 1|\text{p}a(s_i)))$, and requires the calculation of posterior distribution to obtain the parent states s_j given the observed state s_i. The posterior distribution is not factorial because of the explaining-away phenomenon that occurs in the presence of two or more hidden variables with a causal link to a third common random variable, named collider. When conditioning on it, as in calculating the posterior distribution, it would create a sort of association with the connected upstream variables. The model's stochastic variables can be partitioned in two groups, namely $\boldsymbol{x}$ and $\boldsymbol{z}$ in order to distinguish the ones which are directly observed from data. Consequently,

the inferred posterior probability defined as $p(\boldsymbol{z}|\boldsymbol{x}) = p(\boldsymbol{x}, \boldsymbol{z})/p(\boldsymbol{x})$ is intractable to compute because marginalizing over the visible variables would require to sum over all possible configurations of hidden units in the upper layers, exponential in the number of units, since posterior is not factorial as mentioned. Nonetheless, approximate methods, as the wake-sleep algorithm [9], were proposed to train SBN by computing gradient approximation that somehow makes the model still capable of learning.

Fully visible belief networks (FVBNs) [10], also called auto-regressive networks, on the other hand, represent an extreme case in which no conditional independence assumptions is made. These models are characterized by the extensive use of observable stochastic variables only with no latent units. They model the joint probability density in a tractable form using the chain rule of probability, thus expressing it as the product of conditional probability distributions. A neural network would approximate each of these conditional probabilities:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|x_1, ..., x_{i-1}). \tag{1.20}$$

From a different perspective, these models can be viewed as a generalization of classification methods, used for estimating a conditional probability. However, here, instead of predicting class label y at each step, x_i is inferred. Scalability is the biggest limit of FVBN since generating a new sample requires a sequential computation of $P(x_i|x_1, ..., x_i)$ for each step $i = 1, ..., n$, and therefore has a linear cost $O(n)$. This computation cannot be parallelized and can even require some minutes for a single sample.

Undirected models

The greater difficulty of training undirected models has hindered their spread and progress. Below is a rough view of some of the issues arising in training these models.

Undirected graphical models offer an alternative way of describing dependencies among stochastic variables. As opposed to directed models, an undirected link is responsible for grouping variables with affinities, rather than specifying a directional dependency. Thus, the interactions among stochastic variables are measured by a factor $\phi(\mathcal{C}) > 0$ for each clique $\mathcal{C}$, which is a subset of random variables that form a complete graph. The undirected graph $\mathcal{G}$ illustrates for a given model its underlying structure and allows easy identification of cliques that collectively define through their factors the unnormalized probability distribution $\tilde{p}(\boldsymbol{x}) = \prod_{\mathcal{C}\in\mathcal{G}} \phi(\mathcal{C})$. Then, a partition function $Z(\theta) = \int \tilde{p}(\boldsymbol{x})d\boldsymbol{x}$ is introduced to normalize the unnormalized

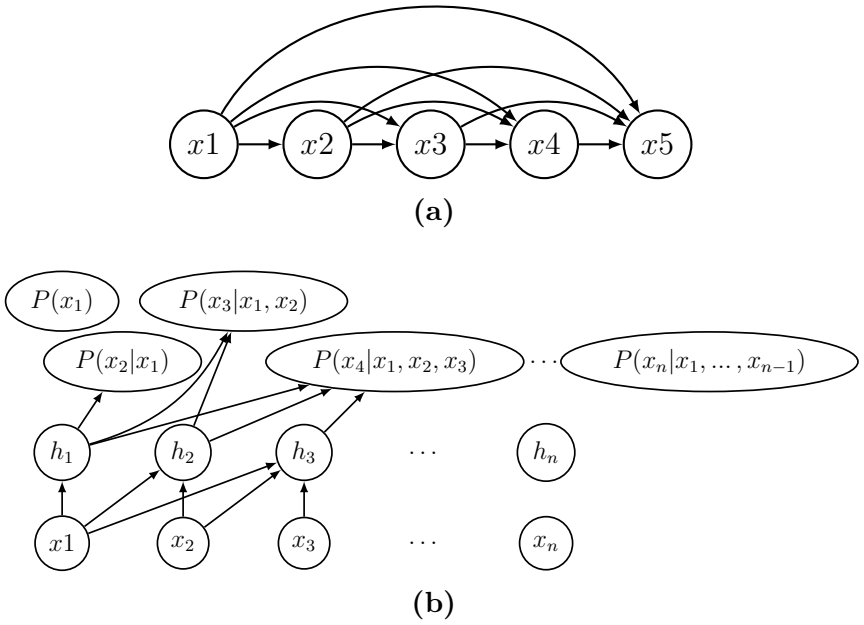


Figure 1.7: A fully visible belief network models the joint probability through the chain rule. As a result, given a particular order, each variable x_i depends from the previous $i-1$ ones. (a) The graphical model for an FVBN (b) Computational Graph for a neural auto-regressive model with a hidden layer used to predict x_i from variables $x_1, ..., x_{i-1}$. This illustration was inspired from [11]

probability distribution, which typically is in the form $\tilde{p}(\boldsymbol{x}) = \exp(-E(\boldsymbol{x};\theta))$ to ensure positive potential factors:

$$p(\boldsymbol{x};\theta) = \frac{\tilde{p}(\boldsymbol{x};\theta)}{Z(\theta)}. \tag{1.21}$$

Similar to directed models, variables are partitioned in hidden and observable, and for deeper models, they are further organized in different layers. A notorious basic model, the restricted Boltzmann machine, consists of a single hidden layer where interactions are limited only to units located on adjacent layers. This aspect leads to some nice properties:

$$p(\boldsymbol{z}|\boldsymbol{x}) = \prod_i p(z_i|\boldsymbol{x}), \tag{1.22}$$

$$p(\boldsymbol{x}|\boldsymbol{z}) = \prod_i p(x_i|\boldsymbol{z}). \tag{1.23}$$

Since the conditional and the posterior distributions are factorial no explaining

away phenomenon can occur during learning, but still, some limitation arises from $Z(\theta)$ considering that it is intractable to compute it. From the gradient $\nabla_{\boldsymbol{\theta}} - \log(Z(\boldsymbol{\theta}))$ results an expectation term under the model distribution, which computation involves the integration over all possible configuration of hidden and visible variables. The whole gradient equation become:

$$\nabla_\theta \frac{1}{N} \log p(\boldsymbol{X}) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}} \nabla_\theta \log \tilde{p}(\mathbf{x};\boldsymbol{\theta}) - \mathbb{E}_{\mathbf{x}\sim p(x;\theta)} \nabla_\theta \log p(\mathbf{x};\boldsymbol{\theta}). \tag{1.24}$$

Gibbs sampling represents a naive method to obtain this gradient approximation, since it is not possible to compute the exact gradient. This learning approach is

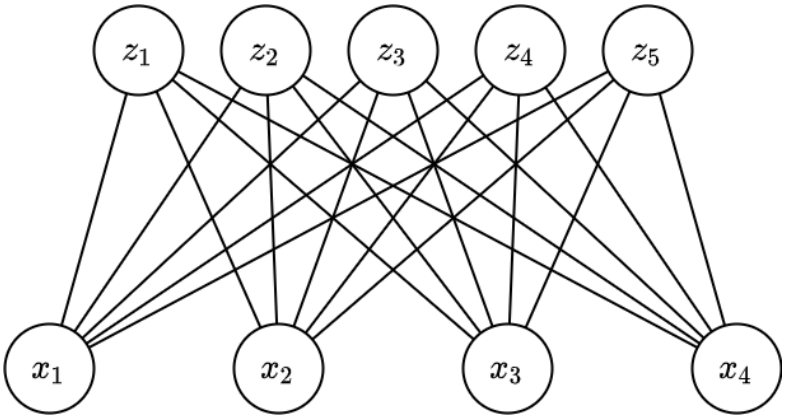


Figure 1.8: The Restricted Boltzmann machine is an undirected graphical model composed of two layers: one with observable units and the other with hidden units. The connections are defined only between units of adjacent layers.

conceptually divided into two phases: the positive phase where $\tilde{p}$ is maximized for observation drawn from data according to the first term on the right-hand side, and the negative phase during which $\tilde{p}(x)$ is minimized for samples drawn from the model distribution. Although they were proposed more efficient algorithms, they are still based on Markov chain Monte Carlo, and over the years, the research focus have shifted away to models fully trainable with backpropagation, which will be introduced in the next section.

1.3.2 Differentiable generator models

Formally, differentiable generator models are shallow directed stochastic models, but nevertheless, this class stands out for the use of differentiable deterministic layers to transform a sample from latent variables $\boldsymbol{z}$ to a sample $\boldsymbol{x}$ over $p(\boldsymbol{x})$. These models demand a generator network able to map samples drawn from $p(\boldsymbol{z})$ to $\boldsymbol{x}$

through a differentiable function $\boldsymbol{x} = g(\boldsymbol{z}; \boldsymbol{\theta}_g)$ implemented by a neural network [11]. The network architecture will thus outline the family of distributions from which the most suitable is picked via parameter optimization. In a restricted number of cases, this mapping is achieved analytically with the inverse transform method. Let $F(x)$ $\mathbb{R} \mapsto [0,1]$ indicate the cumulative distribution function, which is continuous and monotone, thus invertible. Let $F^{-1}(z), z \in [0,1]$ be the inverse function, it can be used to obtain a random sample x by drawing a random value $\mathrm{z} \sim U(0,1)$, and then $\mathrm{x} = F^{-1}(\mathrm{z})$, hence x will be distributed as F, i.e. $P(\mathrm{x} \leq x) = F(x)$. For example, inverting the exponential CDF, $F(x) = 1 - e^{-\lambda x}$ leads to $F^{-1}(z) = -(1/\lambda)\ln(1-z)$, which allows drawing a new sample $\mathrm{x} = -(1/\lambda)\ln(1-\mathrm{z})$ with $\mathrm{z} \sim U(0,1)$ or equivalently $1 - \mathrm{z} \sim U(0,1)$. For a generic distribution $p(\boldsymbol{x})$, this method of sampling requires the calculation and the inversion of the indefinite integral of its density function, which would be feasible only for a limited set of distributions.

Differentiable generator models, for an arbitrary distribution $p(\boldsymbol{x})$, will define a more general non-linear function $g(\boldsymbol{z})$, that is not the inverse CDF, but similarly to $F^{-1}(\boldsymbol{z})$ transforms the $p(\boldsymbol{z})$ distribution into $p(\boldsymbol{x})$ with a change of variable:

$$p_x(\boldsymbol{x}) = p_z(g^{-1}(\boldsymbol{x}))\left|\nabla g^{-1}\right| \tag{1.25}$$

The use of these models has been encouraged by the success of backpropagation, with feedforward neural networks, in classification tasks. Differently from the classification tasks, the observations $\boldsymbol{X}$ drawn from $p_{\text{data}}(\boldsymbol{x})$ are generally the only data provided to a generative model, thus learning implies determining a mapping from latent space to the sample space, rather than to class labels. Variational Autoencoders [12] and Generative adversarial Networks [13] belong to this category of models, although they are very different in their way of learning, they both generate new samples through a differentiable generator network. The implicit or explicit definition of the density function $p(\boldsymbol{X}; \boldsymbol{\theta})$ is the main characteristic feature of these models: whereas the VAE model directly maximizes the variational lower bound $\mathcal{L}$, the GAN model interact with $p(\boldsymbol{X}; \boldsymbol{\theta})$ only by sampling from it. In particular, the GAN loss function is a minimax game between discriminator and generator networks that indirectly pushes the generator to approximate the distribution of underlying data. Before going into GANs details, some further notions about VAE would be precious for the overview.

Variational Autoencoder

Variational Autoencoder (VAE) [12] receives its name from the variational inference, a process of approximate inference used for training the model. A VAE model includes a decoder network and an encoder network. The decoder is responsible for

generating new samples $\hat{\mathbf{X}}$, which resemble training data $\mathbf{X}$. Specifically for each new synthetic data point $\hat{\boldsymbol{x}}$, the model draws a sample $\boldsymbol{z}$ from a defined probability density $p(\boldsymbol{z})$ that will run through the decoder network producing $\hat{\boldsymbol{x}}$. Thus, the decoder network $g(\boldsymbol{z};\boldsymbol{\theta})$, defined as a parametrized family of deterministic function, models the conditional probability $p(\boldsymbol{x}|\boldsymbol{z};\boldsymbol{\theta})$, which can be assumed to be a multivariate Gaussian distribution $p(\boldsymbol{x}|\boldsymbol{z};\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{x};\boldsymbol{\mu}(\boldsymbol{z};\boldsymbol{\theta}),\boldsymbol{\sigma}(\boldsymbol{z};\boldsymbol{\theta})^2\boldsymbol{I})$, by determining the parameters $\boldsymbol{\mu}(\boldsymbol{z};\boldsymbol{\theta})$, and $\boldsymbol{\sigma}(\boldsymbol{z};\boldsymbol{\theta})$. Regarding the prior distribution, it is defined as a simple multivariate Gaussian $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}, 0, \boldsymbol{I})$ under the assumption that any distribution in the same dimension can be generated through a complicated function. In this case, the decoder neural network is responsible for the mapping.

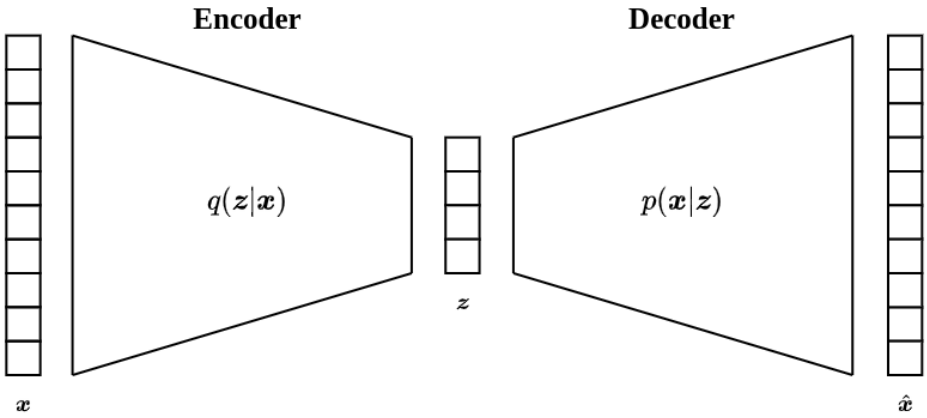


Figure 1.9: A naive illustration of Variational Autoencoder that not include reparametrization trick

Now, fitting the model distribution to data means maximizing the marginal likelihood:

$$p(\boldsymbol{X};\boldsymbol{\theta}) = \int p(\boldsymbol{X}|\boldsymbol{Z};\boldsymbol{\theta})p(\boldsymbol{Z})d\boldsymbol{Z}. \tag{1.26}$$

The above integral is intractable, especially in high dimensional latent space. Specifically, the complexity of the marginal log-likelihood evaluation is due to the presence of the integral, which prevents from applying the logarithm on the joint distribution and expressing it as a summation:

$$ln\, p(\boldsymbol{X};\boldsymbol{\theta}) = \ln\left(\int p(\boldsymbol{X}|\boldsymbol{Z};\boldsymbol{\theta})p(\boldsymbol{Z})d\boldsymbol{Z}\right). \tag{1.27}$$

By consider for simplicity a single data observation $\boldsymbol{x}^{(i)}$ from $\boldsymbol{X} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$, it can be shown that, for any distribution $q(\boldsymbol{z})$, the marginal log-likelihood can be expressed in an alternative form:

$$\log\ p(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) = \mathcal{L}(q,\boldsymbol{\theta}) + D_{\mathrm{KL}}(q(\boldsymbol{z})\|p(\boldsymbol{z}|\boldsymbol{x}^{(i)})) \tag{1.28}$$

$$= \int q(\boldsymbol{z})\ \log\left(\frac{p(\boldsymbol{x}^{(i)},\boldsymbol{z};\boldsymbol{\theta})}{q(\boldsymbol{z})}\right)d\boldsymbol{z} + D_{\mathrm{KL}}(q(\boldsymbol{z})\|p(\boldsymbol{z}|\boldsymbol{x}^{(i)}). \tag{1.29}$$

Since $\boldsymbol{x}^{(i)}$ is given, and q can be any distribution, it appears that the marginal log-likelihood is equal to the sum of $\mathcal{L}$ and D_{KL}, which respectively are a non-negative quantity, and the variational lower bound. This suggests that during the $\mathcal{L}$ maximization, $ln\ p(\boldsymbol{x}^{(i)};\boldsymbol{\theta})$ could be indirectly pushed upwards, and at the same as $D_{\mathrm{KL}}(q(\boldsymbol{z})\|p(\boldsymbol{z}|\boldsymbol{x}^{(i)}))$ decreases the lower bound gets tighter approaching to the exact marginal log-likelihood. It thus makes sense to define q distribution to be conditioned as $q(\boldsymbol{z}|\boldsymbol{x}^{(i)})$ in order to produce a closer posterior approximation. At this point, it is possible to rearrange the lower-bound as:

$$\mathcal{L}(q,\boldsymbol{\theta},\boldsymbol{\phi}) = \int q(\boldsymbol{z})\ \log\left(\frac{p(\boldsymbol{x}^{(i)},\boldsymbol{z};\boldsymbol{\theta})}{q(\boldsymbol{z})}\right)d\boldsymbol{z} \tag{1.30}$$

$$= \int q(\boldsymbol{z})\log\ p(\boldsymbol{x}^{(i)}|\boldsymbol{z})d\boldsymbol{z} + \int q(\boldsymbol{z})\log\ \frac{p(\boldsymbol{z})}{q(\boldsymbol{z}|\boldsymbol{x}^{(i)})}d\boldsymbol{z} = \tag{1.31}$$

$$= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}|\boldsymbol{x}^{(i)})}[\log\ p(\boldsymbol{x}^{(i)}|\boldsymbol{z}))] - D_{\mathrm{KL}}(q(\boldsymbol{z}|\boldsymbol{x}^{(i)})\|p(\boldsymbol{z})). \tag{1.32}$$

As for $p(\boldsymbol{x}^{(i)}|\boldsymbol{z})$, a neural network parameterizes the distribution $q(\boldsymbol{z}|\boldsymbol{x}^{(i)}) = \mathcal{N}(\boldsymbol{z};\boldsymbol{\mu}(\boldsymbol{x}^{(i)};\boldsymbol{\phi}),\boldsymbol{\sigma}(\boldsymbol{x}^{(i)};\boldsymbol{\phi})^2\boldsymbol{I}))$, which is also assumed to be Gaussian for simplicity. Then, the encoder network is in charge of producing $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ vectors for the mean and variance. In the last right-hand side equation, the first term $\mathbb{E}_{\mathbf{z}\sim q(\boldsymbol{z}|\boldsymbol{x}^{(i)})}[\log\ p(\boldsymbol{x}^{(i)}|\boldsymbol{z}))]$, called reconstruction error, may appear familiar since it is frequently present in autoencoders loss functions. For Gaussian distributions, it corresponds to the mean squared error between the original observation $\boldsymbol{x}^{(i)}$ and the one reconstructed from $\boldsymbol{z}$. The second term, $D_{\mathrm{KL}}(q(\boldsymbol{z}|\boldsymbol{x}^{(i)})\|p(\boldsymbol{z}))$, acts as a regularizer by pulling the chosen distribution $q(\boldsymbol{z}|\boldsymbol{x}^{(i)})$ to $p(\boldsymbol{z})$, preventing the model from brutally overfitting data by assigning all the probability mass to training observations only.

Since it is not possible to propagate gradient through a stochastic operation such as sampling, a slight change is done to the network by moving sampling operation $\boldsymbol{z}\sim q(\boldsymbol{z}|\boldsymbol{x}^{(i)})$ to the network input, which is not traversed by the gradient. This technique is called the reparameterization trick [12] (Figure 1.10). The distribution $q(\boldsymbol{z}|\boldsymbol{x}^{(i)})$ is thus expressed as two steps generative process, during which, first, a noise variable is sampled from a naive distribution $p(\boldsymbol{\epsilon}) = \mathcal{N}(\boldsymbol{\epsilon},0,\boldsymbol{I})$ and then it is

run through a deterministic transformation g, such that $\boldsymbol{z} = g(\boldsymbol{\epsilon}, \boldsymbol{x}^{(i)}; \boldsymbol{\theta})$ will be distributed as $q(\boldsymbol{z}|\boldsymbol{x}^{(i)})$. For Gaussian distributed variables this is obtained defining $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\epsilon}\, \boldsymbol{\sigma}^2 \boldsymbol{I}$, for $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$, rather than $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2 \boldsymbol{I})$.

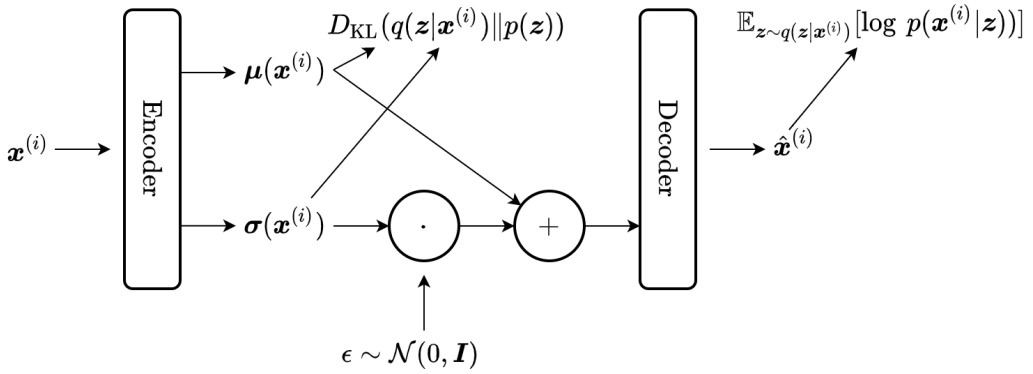


Figure 1.10: The image illustrates a variational autoencoder after the reparametrization trick.

In practice, when training a VAE, during the forward pass the encoder is fed with a mini-batch of observations $\boldsymbol{x}^{(i)}$ sampled from data $\boldsymbol{X}$, and for each of these, the network provides the posterior approximation $q(\boldsymbol{z}|\boldsymbol{x}^{(i)})$, by computing $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ from which $\mathbf{z}$ is sampled using the reparameterization trick. Then, the reconstruction error is estimated computing the expectation of $\log p(\boldsymbol{x}^{(i)}|\mathbf{z})$ over the samples drawn from $q(\boldsymbol{z}|\boldsymbol{x}^{(i)})$, whereas the KL divergence term can be computed analytically in a closed form in the cases where both $p(\boldsymbol{z})$ and $q(\boldsymbol{z}|\boldsymbol{x})$ are assumed Gaussian. The gradient required to update the parameters is evaluated on the expectation of $\mathcal{L}$ over the observations $\boldsymbol{x}^{(i)}$ in the mini-batch. In this setup, the encoder and decoder network can be trained until the convergence of parameters $\boldsymbol{\theta}, \boldsymbol{\phi}$ by iteratively calculating the cost function and updating them using the gradients on mini-batches $\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi})$. At test time, only the decoder network is retained, and new samples are generated by running through the decoder the samples $\mathbf{z} \sim \mathcal{N}(\boldsymbol{z}; 0, \boldsymbol{I})$.

Variational methods, although they are biased because of the error between $\mathcal{L}$ and $p(\boldsymbol{X})$, they actually produce satisfactory parameter estimates. Unfortunately, a critical downside affects VAE, which on image generative models is visible as a blurry effect in the produced samples. According to Goodfellow [11], this is probably due to the log-likelihood maximization, which as $D_{\mathrm{KL}}(\hat{p}_{\mathrm{data}}\|p_\theta)$ minimization, encourages overgeneralization.

Chapter 2

Generative Adversarial Network

This chapter will cover starting from the original idea of GANs, the several issues that arise in training these models, and the related improvements that followed over the years.

Generative adversarial networks [13][1] depict a scenario similar to a two-player minimax game in which the generator model G and the discriminator model D compete against each other. The generator G aims to reproduce the data distribution, whereas D estimates the probability that a sample originates from training data or G. In particular samples are generated through $G(\boldsymbol{z}; \boldsymbol{\theta}_g)$, that symbolize a parametrized differentiable function mapping a latent prior $p_z(\boldsymbol{z})$ over the metric space $\mathcal{Z}$ to $p_g(\boldsymbol{x})$ over $\mathcal{X}$, typically implemented by a multilayer perceptron. The discriminator D, also implemented by a multilayer perceptron $D(\boldsymbol{x}; \boldsymbol{\theta_d})$, is responsible for estimating the probability $D(\boldsymbol{x})$ that given a sample $\boldsymbol{x}$ it either belongs to real data p_{data} or it was generated from $p_g(\boldsymbol{x})$. Then, the game solution can be found by simultaneously optimizing for some D and G the value function:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \tag{2.1}$$

In this framework the two player, respectively G and D face each other in minimizing or maximizing the same value function $V(G, D)$, based on the roles played, by acting only on their own parameters. Hence, the game consists of two turn where on the first one, the discriminator tries to push $D(\boldsymbol{x})$ as close as possible to 1 and $D(G(\boldsymbol{z}))$ to be near 0 to maximize its cost function. It should be noted

that the discriminator cost function is not different from any other cross-entropy function for binary classifiers, although it has the peculiarity of having the batch split in two halves, namely the real data batch and the generated sample batch. By contrast, the generator G tries to minimize the same cost function by acting on the second term to minimize the expectation for values of $D(G(\boldsymbol{z}))$ approaching 1, which happens for generated samples misclassified as real observations. As a result, in evaluating the cost function each one of the players is intrinsically influenced by the effect of the counterpart. A solution that optimizes the value function would is thus denoted by a Nash equilibrium tuple $(\boldsymbol{\theta}_d, \boldsymbol{\theta}_g)$, namely a point in solution space that is simultaneously a local minimum for both $J^{(D)}(\boldsymbol{\theta}_d)$ and $J^{(G)}(\boldsymbol{\theta}_g)$. In ideal conditions a GAN may converge to a solution so that the generated samples will be indistinguishable by the discriminator network, which will assign 0.5 probability to both real and generated samples. Unfortunately, in general, the attainment of an equilibrium solution is not guaranteed for a minimax game.

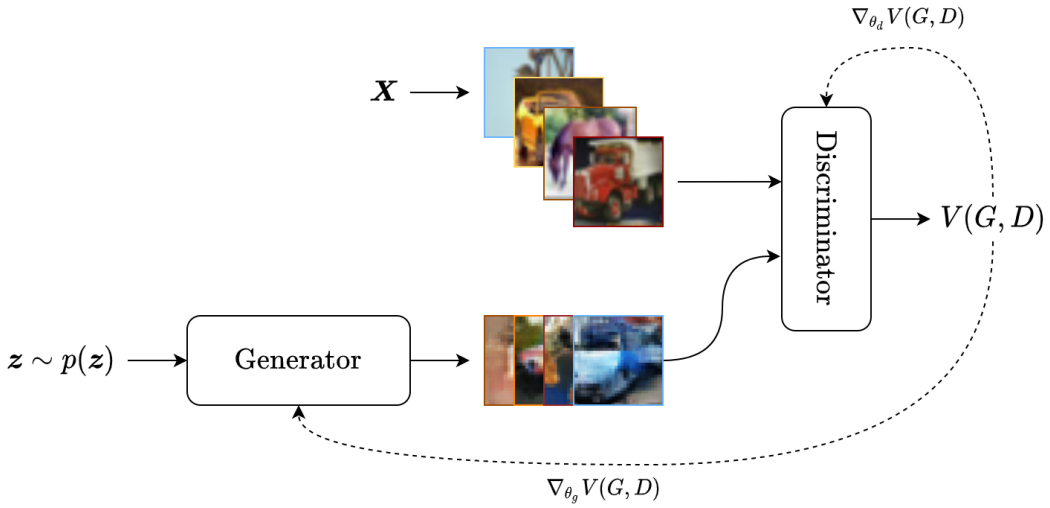


Figure 2.1: GAN architecture.

2.1 GAN convergence theory

It is possible to study the algorithm convergence from a theoretical perspective [13]. This is achievable only through the abstraction from capacity constraint, training time, and finite dataset limitation and just considering a model with infinite capacity and hence the whole space of probability density functions. From

the value function:

$$\begin{aligned} V(G,D) &= \int_{\boldsymbol{x}} p_{data}(\boldsymbol{x}) \log D(\boldsymbol{x})\, d\boldsymbol{x} + \int_{\boldsymbol{z}} p(\boldsymbol{z}) \log(1 - D(G(\boldsymbol{z})))\, d\boldsymbol{z} \\ &= \int_{\boldsymbol{x}} p_{data}(\boldsymbol{x}) \log D(\boldsymbol{x}) + p_g(\boldsymbol{x}) \log(1 - D(\boldsymbol{x}))\, \boldsymbol{x}, \end{aligned} \tag{2.2}$$

where the second equation is derived from $\mathbb{E}_{\mathbf{z}\sim p_{\boldsymbol{z}}}[\log(1-D(G(\boldsymbol{z}))] = \mathbb{E}_{\mathbf{x}\sim p_G(\boldsymbol{x})}[\log(1-D(\boldsymbol{x}))]$ it can be obtained the optimal discriminator D^* for a given generator G as:

$$D^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}. \tag{2.3}$$

Plugging in the optimal discriminator D^* into value function, it becomes:

$$\begin{aligned} C(G) &= \max_D V(G,D) \\ &= \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}} \left[\log D^*(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}} \left[\log\left(1 - D^*(G(\boldsymbol{z}))\right)\right] \\ &= \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}} \left[\log D^*(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{x}\sim p_g} \left[\log\left(1 - D^*(\boldsymbol{x})\right)\right] \\ &= \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}\left(\boldsymbol{x}\right)}{P_{\text{data}}\left(\boldsymbol{x}\right) + p_g(\boldsymbol{x})}\right] + \mathbb{E}_{\boldsymbol{x}\sim p_g} \left[\log \frac{p_g(\boldsymbol{x})}{p_{\text{data}}\left(\boldsymbol{x}\right) + p_g(\boldsymbol{x})}\right] \end{aligned} \tag{2.4}$$

Now, $C(G)$ is called the virtual training criterion and it must be proven that its minimum is attained if and only if $p_g = p_{\text{data}}$. For the optimal G with $p_g = p_{\text{data}}$, D^* will be unable to discriminate and assign $\frac{1}{2}$ probability by chance to p_g and p_{data} samples:

$$C^* = \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x}) \log \frac{1}{2} d\boldsymbol{x} + \int_{\boldsymbol{x}} p_g(\boldsymbol{x}) \log\left(1 - \frac{1}{2}\right) d\boldsymbol{x} \tag{2.5}$$

$$= -\log 2 \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x})\, d\boldsymbol{x} - \log 2 \int_{\boldsymbol{x}} p_g(\boldsymbol{x})\, d\boldsymbol{x} = -\log 4. \tag{2.6}$$

Subtracting both left and right-hand side of equation 2.6 from $C(G)$ results:

$$\begin{aligned} C(G) = &\int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x}) \log\left(\frac{p_{\text{data}}(\boldsymbol{x})}{p_G(\boldsymbol{x}) + p_{\text{data}}(\boldsymbol{x})} + \log 2\right) d\boldsymbol{x} + \\ &\int_{\boldsymbol{x}} p_g(\boldsymbol{x}) \log\left(\frac{p_G(\boldsymbol{x})}{p_G(\boldsymbol{x}) + p_{\text{data}}(\boldsymbol{x})} + \log 2\right) d\boldsymbol{x} - \log(4), \end{aligned} \tag{2.7}$$

which can be rewritten in the form:

$$\begin{aligned} C(G) &= -\log 4 + D_{\mathrm{KL}}\left(p_{data}\middle\|\frac{p_{data}+p_g}{2}\right) + D_{\mathrm{KL}}\left(p_g\middle\|\frac{p_{data}+p_g}{2}\right) \\ &= -\log 4 + 2 \cdot JSD(p_{data}\|p_g), \end{aligned} \tag{2.8}$$

where $JSD(p_{data}\|p_g)$ represent the Jensen-Shannon divergence between the data and the model distributions. Since such divergence is defined to be positive and zero only for equal distributions the global minimum of $C^* = -\log 4$ is achieved only for $p_{\mathrm{data}} = p_g$.

2.2 Training the adversarial net

The assumptions made for the convergence theory are not met, in practice, when G and D are implemented through MLPs. Consequently, the previous section results do not apply because they rely on convexity property, which is no more guarantee. In a real scenario, with both the players modeled by a neural network, the entire system is simultaneously trained with stochastic gradient descent and backpropagation algorithm. Besides, since training the discriminator until convergence is computationally intractable and can lead to overfitting, one can alternatively train the discriminator for k steps and the generator for one step. This should provide a good enough estimate of the optimal D to allow the training of G.

The training algorithm reported in Algorithm 1 describes in detail the training process, in which, at each iteration two mini-batches are provided to D, one drawn from the dataset $\boldsymbol{X}$ and the other generated through G from prior $p_{\boldsymbol{z}}$. At this point, k optimization steps are done for D and one for G, by computing for each step their gradients and updating their weights. In training D and G using stochastic gradient descent, each player will strive to optimize its own cost function that depends on the parameters of both networks. This means that at each discriminator training step, D is going towards reducing $J^{(D)}$ by acting on $\boldsymbol{\theta}_d$, without caring if it may also increase $J^{(G)}$ undoing the opponent's progress and vice versa. In the extreme cases, if this happens at every iteration step, the equilibrium would never be reached, and the value function would end up in an endless orbit. In practice, even when oscillating, GANs are nevertheless capable of producing quality samples without finding an equilibrium solution.

In situations where D easily recognizes generated samples with high confidence, the value function (2.1) results not very suitable for training since the term $(1 - D(G(\boldsymbol{z}))$ starts to saturate, and as a result the generator gradient vanishes. Goodfellow et al. proposed a heuristically motivated solution that mitigates this

problem with the introduction of a new generator cost function formulation in which the generator is trained to maximize $J^{(G)} = -\log(D(G(\boldsymbol{z}))$. This corresponds to maximize the log probability that the discriminator is mistaken rather than minimizing the probability of being correct. The new cost function then makes the GAN lose further theoretical guarantees of convergence, but on the other hand, the generator gradient will not vanish in cases where the discriminator easily rejects its samples. Besides, the solution can no longer be described as an equilibrium point since the previous value function is no more representative after this adjustment.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter.

1: **for** number of training iterations **do**
2: **for** k steps **do**
3: Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, ..., \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
4: Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{data}(\boldsymbol{x})$.
5: Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\boldsymbol{\theta}_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]. \tag{2.9}$$

6: **end for**
7: Sample minibatch of m noise samples$\{\boldsymbol{z}^{(1)}, ..., \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
8: Update the generator by descending its stochastic gradient:

$$\nabla_{\boldsymbol{\theta}_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right). \tag{2.10}$$

9: **end for**
10: ▷ The gradient-based updates can use any standard gradient-based learning rule.

2.3 GAN Issues

Together with vanishing gradient, training a GAN model is further complicated from the occurrence of mode collapse and from its uninformative loss, which makes it possible neither to monitor the training process nor to evaluate the generated samples quality basing on it.

Vanishing Gradient

Arjoski and Bottou in [14] provided some theoretical explanation for the vanishing gradient problem that affects the GAN training algorithm, and they carry out a theoretical analysis of the heuristic introduced in the previous section, which somehow seems to solve the problem. The vanishing gradient represents the greatest contradiction in GAN training. Precisely, the closer is D to D^*, the more valuable will be gradient for the generator since it will better approximate the gradient of Jensen Shannon divergence, but simultaneously the more its norm tends to decrease. This obliges GAN practitioners to seek for a trade-off for k hyperparameter that certainly makes unfriendly the training procedure. According to Arjoski, the causes of this behavior must be searched behind the measure used and the distributions characteristics.

Based on strong and theoretical evidence, the authors assumed that in general p_{data} lies in a low-dimensional manifold. In addition, given the prior $\boldsymbol{z} \sim p(\boldsymbol{z})$, defined on a metric space $\mathcal{Z} < \mathcal{X}$, the distribution p_g, generated through $G_\theta : \mathcal{Z} \mapsto \mathcal{X}$ is not continuous and is contained in a countable union of low-dimensional manifolds whose dimensions are at most as big as the ones of $\mathcal{Z}$. When p_g and p_{data} have their support contained on two disjoint compact subsets a perfect discriminator $D^* : \mathcal{X} \to [0, 1]$ exists, and it is characterized by perfect accuracy and $\nabla_{\boldsymbol{x}} D^*(\boldsymbol{x}) = 0$ for all $\boldsymbol{x}$ in both p_g and p_{data} supports. On the countrary for two manifolds that match perfectly on a big portion of the space, there is no discriminator that can perfectly separate them. However from a broader perspective, they prove that for two submanifolds of $\mathbb{R}^d$ that do not have the full dimension, the probability that they do not perfectly align after small perturbations is 1. With this in mind, a perfect discriminator still exists for p_g and p_{data} distributions with their support contained in two manifold that neither perfectly align nor have full dimensions. As a result, in general D^* is not able to provide any information with its gradient, and besides, there is no way to measure similarities between non perfectly aligned manifolds since both KL divergences will be infinite and JSD will be constant. Thus, let D be a non optimal discriminator such that $\|D - D^*\| < \epsilon$ and given $\mathbb{E}_{\mathbf{z} \sim p(\boldsymbol{z})}[\|\mathbf{J}_{\boldsymbol{\theta}} G_{\boldsymbol{\theta}}(\boldsymbol{z})\|_2^2] \leq M^2$, then the gradient norm is bounded as:

$$\left\| \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{z} \sim p(\boldsymbol{z})} \left[\log \left(1 - D \left(G_{\boldsymbol{\theta}}(\boldsymbol{z}) \right) \right) \right] \right\|_2 < M \frac{\epsilon}{1 - \epsilon}, \tag{2.11}$$

where as D approaches to D^*, ϵ get closer to 0 and consequently the gradient vanishes.

Regarding the use of the heuristic based cost function $J^{(G)} = -\frac{1}{2} \mathbb{E}_{\mathbf{z}} \log D(G(\boldsymbol{z}))$, they show that it does not effectively solve the vanishing gradient problem since from the theoretical analysis it has emerged that the gradient provided, although

it does not vanish, causes unstable updates. For the optimal discriminator D^*, the generator gradient become:

$$\mathbb{E}_{\mathbf{z}\sim p(\boldsymbol{z})}\left[-\nabla_{\boldsymbol{\theta}} \log D^*\left(G_{\boldsymbol{\theta}}(\boldsymbol{z})\right)\right] = \nabla_{\boldsymbol{\theta}}\left[D_{\mathrm{KL}}\left(p_g \| p_{\mathrm{data}}\right) - 2JSD\left(p_g \| p_{\mathrm{data}}\right)\right] \tag{2.12}$$

This equation shows a first term, that is the reversed KL divergence, which encourage the generator to produce real looking samples by assigning high-cost penalty to samples that not resemble real data, consequently promoting mode dropping effect. As for the second term, the the negative JSD has the effect of pushing the two distribution to diverge. In cases where D is not the optimal discriminator, under some strong assumption it results that $\mathbb{E}_{\mathbf{z}\sim p(\boldsymbol{z})}\left[-\nabla_{\theta} \log D\left(G_{\theta}(z)\right)\right]$ is a zero centered Cauchy distribution with infinite expectation and variance. Since the distribution is zero-centered, the expected update for bounded updates will be zero and will not provide any information to the gradient. The authors also got a practical confirmation of this cost function behavior by training the discriminator with a fixed generator. They observed an increase in generator gradient norm as D approaches to D^* and a gradual increase of variance, identifiable as noise in gradients, which give rise to instability in the whole training process.

One of the solutions proposed to the vanishing gradient issue requires to break the assumption of not perfect alignment between the support manifolds by introducing continuous noise to the inputs of the discriminator. For instance let $\epsilon, \epsilon' \sim \mathcal{N}\left(0, \sigma^2 I\right)$ be the random noise and $\tilde{G}_{\theta}(\boldsymbol{z}) = G_{\theta}(\boldsymbol{z}) + \epsilon'$ the resulting generated samples, the gradient provided by the optimal discriminator D^* will no longer be 0, indeed:

$$\mathbb{E}_{\mathbf{z}\sim p(\boldsymbol{z}),\epsilon'}\left[\nabla_{\boldsymbol{\theta}} \log\left(1 - D^*\left(\tilde{G}_{\boldsymbol{\theta}}(\boldsymbol{z})\right)\right)\right] = 2\nabla_{\boldsymbol{\theta}} JSD\left(p_{r+\epsilon} \| p_{g+\epsilon}\right). \tag{2.13}$$

The alternative proposed solution would be the use of a different measure from JSD able to capture the similarity between two disjoint manifolds, such as Wasserstein distance that will be cover in the section 2.5.

Mode Collapse

Mode collapse represents an unwanted scenario in which many values from $p(\boldsymbol{z})$ are mapped on a restricted subset of samples p_g with the consequence of producing somewhat similar outputs from the generator, which result indistinguishable from real data to the discriminator. Since the discriminator processes each sample and computes its gradient independently from the others samples, the gradient can push the generator to map many different points in $\mathcal{Z}$ space to a single-mode in $\mathcal{X}$, which D consider real with high confidence. If the collapse happens, there is no way that the discriminator could fix this harmful mapping using gradient descent for the same reason that it could not prevent it. As a result, further

training the discriminator to recognize the fake samples will drive the generator to find a new map to a different subset of samples in $\mathcal{X}$ capable of fooling the discriminator rather than providing additional entropy for the generator outputs. Mode collapse may happen when training is not correctly balanced between D and G, for example when the generator is trained over multiple batches without updating the discriminator.

Uninformative loss

The discriminator loss trend is not so informative by itself for deciding whether to continue or stop training since it does not have a well-defined behavior. When generated samples get better, it may happen that rather than decreasing, the JS estimate increases or stays constant.

Without a proper metric it will not be easy to compare different architectures and models. Salisman et al. proposed the Inception Score [15], an automatic evaluation method for image samples capable of emulating human evaluation criteria. Inception score metric is defined as:

$$\exp(\, \mathbb{E}_{\mathbf{x}} D_{\mathrm{KL}}(p(y|\boldsymbol{x}) \| p(y))\,) \tag{2.14}$$

Its name comes from the Inception model [16], which is the classifier used to compute the conditional label distributions. This formulation as the divergence between conditional and marginal label distribution comes from the need to find a measure correlated with human criteria. In particular, generated images are required to provide conditional label distributions $p(y|\boldsymbol{x})$ with low entropy, which correspond to sharper images whose class of belonging is predictable with high confidence by the Inception model. In addition, the marginal class distribution, $\int p(y|\boldsymbol{x}) = G(\boldsymbol{z}))dz$, is desired to have high entropy, i.e. the metric should penalize generators with some bias towards producing samples from a subset of the classes. If the two mentioned characteristics are satisfied, then D_{KL} will result in a high Inception score.

2.4 Convolutional Generative Adversarial Networks

Before delving into the description of a new loss function that aims to solve vanishing gradient related issues with a theoretical foundation, deep convolutional GAN (DCGAN) architectures and their building blocks need to be introduced since this class of models is very effective for image generation tasks.

Convolutional Neural Networks are widely known for their extensive use in discriminative models on image classification tasks. Technically, convolutional generative adversarial networks class include all the GANs that make use of the convolution operation, which name originates from the homonymous mathematical operation. Next, the convolution and related operations will be described in order to introduce one of the first successful experiments of convolutional GAN in literature.

2.4.1 Convolution operation

Convolution in deep learning field typically mean a slightly different operation from the mathematical one. Given a two dimensional input I, e.g. an image, and a kernel K, the result of the convolution is:

$$C(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{2.15}$$

where both the operands are described by a matrix. In practice, the kernel is spatially slid over the image for different values of i and j, and at each step is computed the summation among the products of the elements in the sliding window centered in $I(i,j)$ and K. People generally refer to kernel also with the name filter, whereas the output produced is generally called feature map or activation map. This operation in convolutional neural networks is implemented through the convolutional layer, which is characterize by the following interesting properties.

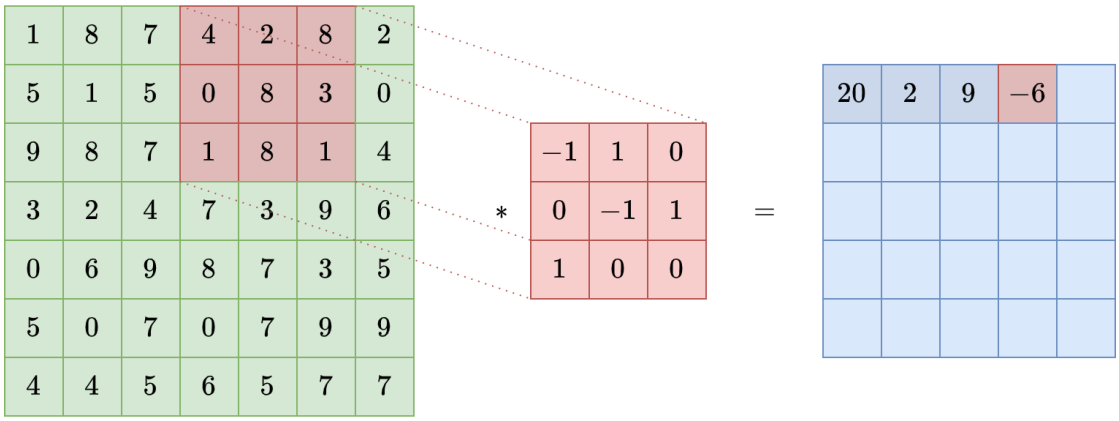


Figure 2.2: An example of 2D convolution of a 7×7 activation map with a 3×3 kernel with stride 1.

Local Connectivity Neurons in a convolutional layer are arranged as a matrix rather than a vector, and each neuron is connected only to a local area of the input

for that layer. For this property, each hidden neuron in the hidden layer receives signal only from a subset of the incoming inputs from the previous layer, unlike fully connected neural networks, in which every neuron is connected to all neurons in the previous layer. This characteristic is achieved by using a kernel size smaller than the input for such layer and proves to be fundamental in image processing, where meaningful features can be discovered by looking only at a pixel and its local neighborhood. This spatial extent of connectivity is called receptive field, and it is considered as a hyperparameter that strictly depends on kernel size. It must be mentioned that in the presence of the depth dimension in images, although spatially located, the connection extends through all the channels. Besides, since each connection is represented by a weight, the reduction in the number of connections between adjacent layers neurons leads to a huge decrease in model parameters size.

Parameter Sharing This technique is used to limit the out of control growth of parameter size in a model when enhancing its learning capacity. In convolutional layers it is implemented by using the same kernel weights multiple times at each step of the convolution during the computation of the output feature map. Importantly, sharing weights rather than learning different weight parameters for each location leads to a massive reduction in the number of needed parameters because of its strong regularization action. During backpropagation, given that each weight affects multiple output pixels, all the locations of the activation maps produced will affect flowing gradients.

Translational Equivariance The convolution operation is equivariant to translation as a direct consequence of using shared filters. This means that given a function f that translates the input I the order in which f and convolution are applied does not matter as they lead to the same result. For example, this property proved to be very effective when some local function is useful in multiple locations like an edge or blob detector in images.

The introduction of the convolutional layer was a breakthrough in image processing since in this field is fundamental to recognize objects independently of their position using a translational equivariant operation, and to capture the hierarchical organization of patterns by considering gradually more abstract features, via stacking several convolutional layers.

2.4.2 Pooling

The pooling operation computes summaries over small local areas for each activation map, introducing some additional properties. There are different pooling functions in literature such as max pooling, average pooling, or L^2 norm of neighboring pixels. All these variants provide invariance property for small translation due to the summarization operation and the overlapping areas of the sliding window among the different steps. A network could benefit from such invariance, especially if the main focus is searching for some features that maximize the activation function rather than detecting their precise location. However, the pooling layer is mainly used for downsampling the feature map trying not to lose meaningful information extracted by the previous layer of the network. In practice, the most used downsampling layer is the max-pooling layer with a 2×2 filter, which for each feature map preserves only 25% of the activations. Precisely, during the max-pooling operation the window is sled all trough the input image for each channel, and at each step only the maximum value in the filter is retained while the others are discarded.

2.4.3 Padding and stride

The padding was mainly introduced to counter the border effects, which consist in a decrease in width and height sizes of activation maps after every convolution operation. E.g. applying a 3×3 convolution on a 32×32 image will result in a 30×30 feature map. In this situation, if it is needed to preserve the image resolution, one should consider applying padding by merely adding around the image a number of frames of zeros that depend on kernel size. In the example provided, one frame is sufficient to keep dimensions unchanged.

The strided convolution represents an alternative method of downsampling. Specifically, the stride indicates how much the kernel window must slide between each step of convolution, hence for a regular convolution the default stride is 1. However for value $s > 1$ the convolution will also result in a shrinking of the input size. By including the slide, the convolution formulation become:

$$C(K, I, s)_{i,j,k} = \sum_{l,m,n} \left[I_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n} \right] \tag{2.16}$$

Springenberg et al. studied in [17] the impact of removing max-pooling layers from a reference CNN architecture and delegating the downsampling to convolutions with stride $s > 1$. Two approaches have been evaluated: the removal of each pooling layer by replacing them with an increased stride in the convolutional layer

that precedes it, the and replacement of the pooling layers with newly inserted convolutional layers with stride greater than one.

Then, the experiments that took place on CIFAR10 have shown that for the first method a little degradation of overall classification performances occured, and they supposed it could be due to the reduction of the overlapping regions of the strided convolutions. Surprisingly in the second method, the additional convolutional layers are effective and provide some improvements over the baseline model with pooling. These tests have been done by making sure the increase of parameters was not the main cause of the lower classification error, but the authors nevertheless emphasize that it cannot be ruled out that the convolutional layers used as a replacement have just learned the pooling function.

2.4.4 Transposed convolution operation

Transposed convolution is a transformation that goes in the opposite direction of convolution and it can, therefore, be used for upsampling, thus projecting a feature map to a higher-dimensional space. In a feedforward neural network an affine transformation at some layer l, which maps a d^{l-1} dimensional input to an hidden d^l dimensional output, is obtained through the dot product with a weight matrix $\boldsymbol{W} \in \mathbb{R}^{d^{l-1} \times d^l}$. Likewise, a transformation that goes in the opposite direction, i.e. from d^l dimensional representation to d^{l-1} dimensional one, has to learn a weight matrix with transposed shape $\boldsymbol{W'} \in \mathbb{R}^{d^l \times d^{l-1}}$. If a regular convolution is designed as a normal feedforward connection characterized by weights hardly constrained to zero in areas outside the sliding window, an upsampling function can be applied using a similarly structured matrix with transposed dimensions. That is where its name originates. Interestingly, this operation learns a function whose forward pass exactly corresponds to the backward pass of a regular convolution, and the same goes for its backward pass. In this manner, a mapping is obtained between lower and higher dimensional feature maps.

Given a transposed convolution, one can show that the same result can be obtained with a standard convolution whose specific settings depend on some existent relationship between the two operation [18]. For instance, a transposed convolution with a stride $s > 1$ is equivalent to a convolution with zeros inserted between the input units and is called fractionally-strided convolution.

2.4.5 Nearest-Neighbor Interpolation

Nearest-neighbor interpolation is the simplest interpolation method used for upsampling. Let $\boldsymbol{A}^{n \times n}$ be a pixel matrix, it can be scaled to a greater dimension

by inserting new pixels in between the original ones. This method consists in determining the value of the newly inserted pixels from their nearest neighboring pixels, thus assuming the intensity values of them.

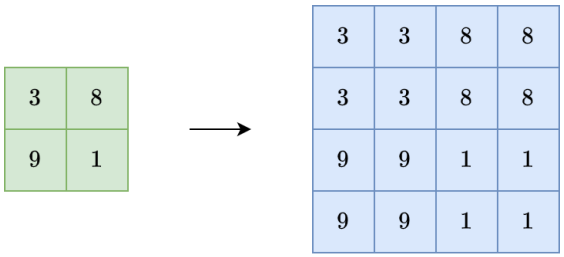


Figure 2.3: An example of nearest-neighbor interpolation upsampling from a 2×2 matrix to a 4×4 one.

The combination of nearest-neighbour interpolation followed by a padded convolution that presere the new upscaled dimensions represent an alternative method to transposed convolution for upsampling.

2.4.6 Deep Convolutional Generative Adversarial Network

Radford et al. introduced in [19] a GAN convolutional architecture with the intention to reuse the intermediate layers extracted from the trained discriminator in an unsupervised learning fashion. This model represents a successful attempt to introduce the convolution in the GAN framework even though it was not the first time that convolution was used in this research area. Next, will follow the key architectural points for a successful implementation of the DCGAN.

Firstly, the generator network used in this work was composed only by one fully connected layer required to map the random noise vector to a tensor with height, width, and depth dimension. On top of this layer, multiple transposed convolutional layers were stacked to upsample the image signal until the pixel space dimension is reached. ReLU activation functions have proved to work well in the generator hidden layers, whereas, for the output layer, the Tanh activation was used. Secondly, the discriminator network, inspired by the more recent image classification models, where the fully connected layers on top of the convolutional features are substituted with a global average pooling, is composed only by different convolutional layers followed by leaky ReLUs with $\alpha = 0.2$. Moreover, since the global average pooling resulted in slowing down convergence, the last convolutional layer was directly flattened and fed through a sigmoid function. The pooling layers have also been removed, to encourage the network to learn the downsampling

operation directly from data similarly to the upsampling operation in the generator. Lastly, the batch normalization was extensively used on both networks except for the last layer in the generator. It allowed a more stable training by forcing the activation inputs to have zero mean and unit variance, and in the author's opinion, such a measure can help the model prevent mode collapse situations.

Interesting experiments about the generator latent space have also been carried out in their work. They have shown that by inspecting the interpolation between a series of 9 random points in $\mathcal{Z}$, it can diagnose the generator learning. In particular, if the distribution learned by the generator gives rise to smooth semantic transitions in the images produced, it is the case of a correct learning process. On the contrary, if the learned distribution exhibits sharp changes, they prove that overfitting has occurred. Moreover, it results that the latent space allows vector arithmetic for creating new points with handcrafted semantic features.

2.5 Wasserstein GAN

The introduction of Wasserstein GAN [2] and its novel distance measure was fundamental in solving the intrinsic training difficulties of GANs in a theoretical motivated way. The GAN optimization can be formalized as a problem where it is required for a sequence of probability distributions $p_{g;t}$, $t \in \mathbb{N}$, to converges to p_{data}, specifically when $\rho(p_{g;t}, p_{\text{data}}) \to 0$. Since the optimization acts directly on θ_t, the definition of a generator model characterized by continuous mapping $\theta \mapsto p_g$ will guarantee for θ_t values that converge to θ^* the convergence of $p_{g;t}$ distribution to p_g. Likewise, if the used distance ρ is defined to be continuous it can be used as a loss function $\theta \mapsto \rho(p_g, p_{\text{data}})$, so that the distribution convergence will strictly depends on it since for $\theta_t \to \theta^*$, a continuous distance measure will tend to 0, $\rho(p_g, p_{\text{data}}) \to 0$. This means that minimizing ρ w.r.t. θ will lead p_g towards p_{data}.

Now, the Earth-Mover distance (EMD) or Wasserstein-1 is defined as:

$$W(p_{\text{data}}, p_g) = \inf_{\gamma \in \Pi(p_{\text{data}}, p_g)} \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\gamma} \left[\|\boldsymbol{x} - \boldsymbol{y}\| \right], \tag{2.17}$$

where $\Pi(p_{\text{data}}, p_g)$ is the set of all joint distributions $\gamma(\boldsymbol{x}, \boldsymbol{y})$, whose marginal are p_{data} and p_g. The intuition behind EMD is that each probability distribution is thought in terms of mass put on each point or interval, and the aim is to move mass from probability distribution p_g to p_{data}. The term $\gamma(\boldsymbol{x}, \boldsymbol{y})$ in the equation specify the transport plan to transform the first distribution into the second, and among all these plans the EMD indicates the infimum γ in term of costs.

It can be proven that for a continuous generator g, e.g. a feedforward neural

network $G_{\boldsymbol{\theta}}$, $W(p_{\text{data}}, p_g)$ is continuous. Moreover, if g is locally Lipschitz and $\mathbb{E}_{\mathbf{z}\sim p}[L(\boldsymbol{\theta}, \boldsymbol{z})] < +\infty$ is true for local constants $L(\boldsymbol{\theta}, \boldsymbol{z})$, then $W(p_{\text{data}}, p_g)$ is continuous everywhere and differentiable almost everywhere. As before this means that when $W(p_{\text{data}}, p_g) \to 0$ also $p_g \to p_{\text{data}}$. This is typically not valid for JS and KL divergences since they can be non-continuous for a given continuous g. As a result, EMD is a weaker metric than them, in the sense that every sequence that converges under the mentioned divergences also converge under EMD but not the opposite. From a comparison among all the aforementioned metrics in measuring the distance between two low dimensional manifold distribution as the ones discussed in section 2.3, emerges that only the EMD is continuous and capable of driving p_g towards convergence when those distributions have disjoint support or their intersection is a set of measure zero.

By introducing the EM distance as the generator cost function in GAN framework, training would consist in minimizing it with stochastic gradient descent, leading p_g to converge on p_{data}. In practice, calculating Wasserstein distance is intractable, but nonetheless it can be approximated through Kantorovich-Rubinstein duality[20], which states:

$$W(p_{\text{data}}, p_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}(\boldsymbol{x})}}[f(\boldsymbol{x})] - \mathbb{E}_{\mathbf{x}\sim p_g(\boldsymbol{x})}[f(\boldsymbol{x})], \tag{2.18}$$

and as specified in the formulation, the supremum must be searched all over the 1-Lipschitz functions. For GAN related applications the 1-Lipschitz functions constraint can be relaxed to K-Lipschitz functions since searching for a supremum over K-Lipschitz functions will lead to $K \cdot W(p_{\text{data}}, p_g)$, namely the previous supremum scaled by a constant K. Thus, given a parametrized family $\{f_{\boldsymbol{w}}\}_{\boldsymbol{w}\in\mathcal{W}}$ of all K-Lipschitz functions for some K, the objective function to optimize is:

$$\max_{\boldsymbol{w}\in\mathcal{W}} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[f_{\boldsymbol{w}}(\boldsymbol{x})] - \mathbb{E}_{\mathbf{z}\sim p(\boldsymbol{z})}[f_{\boldsymbol{w}}(G_{\boldsymbol{\theta}}(\boldsymbol{z})]. \tag{2.19}$$

If the solution found corresponds to the supremum for a certain $\boldsymbol{w} \in \mathcal{W}$ it will be equal to $K \cdot W(p_{\text{data}}, p_g)$ and the resulting gradient for the generator will be:

$$\nabla_{\boldsymbol{\theta}} \left[K \cdot W(p_{\text{data}}, p_g)\right] = -\mathbb{E}_{\mathbf{z}\sim p(\boldsymbol{z})} \left[\nabla_{\boldsymbol{\theta}} D_{\boldsymbol{w}} \left(G_{\boldsymbol{\theta}}(\boldsymbol{z})\right)\right]. \tag{2.20}$$

The $D_{\boldsymbol{w}}$ term called critic is an approximation of the K-Lipschitz function f, and it is the neural network implementation of the parametrized family $f_{\boldsymbol{w}}$. Then, as with GANs, to provide reliable gradient for the generator, discriminator optimization steps alternate with those for the generator.

Lastly, in order to enforce K-Lipschitz continuity for some K that depends on $\mathcal{W}$ the network, weights w are defined in a compact space $\mathcal{W}$. For this reason,

the authors proposed to naively fulfil this condition by clamping the weight to a fixed box after each gradient update. Unfortunately, some complications can arise depending on the clipping hyperparameter c since if it is too large, training the critic will require more time in order to allow for all the weights to reach their limit. On the contrary, for small c vanishing gradients issues can arise, especially for deeper networks or architectures without batch normalization layers.

Algorithm 2 WGAN proposed algorithm for default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$

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

Require: $\boldsymbol{w}_0$, initial critic parameters . $\boldsymbol{\theta}_0$, initial generator parameters.

while $\boldsymbol{\theta}$ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{\boldsymbol{x}^{(i)}\}_{i=1}^{m} \sim p_{\text{data}}$ a batch from the real data.
    Sample $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$ a batch of prior samples.
    $\boldsymbol{g}_{\boldsymbol{w}} \leftarrow \nabla_{\boldsymbol{w}} \left[\frac{1}{m} \sum_{i=1}^{m} D_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}) - \frac{1}{m} \sum_{i=1}^{m} D_{\boldsymbol{w}}(G_\theta(\boldsymbol{z}^{(i)})) \right]$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \cdot \text{RMSProp}(\boldsymbol{w}, \boldsymbol{g}_{\boldsymbol{w}})$
    $\boldsymbol{w} \leftarrow \text{clip}(\boldsymbol{w}, -c, c)$
  end for
  Sample $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$ a batch of prior samples.
  $\boldsymbol{g}_{\boldsymbol{\theta}} \leftarrow -\nabla_{\boldsymbol{\theta}} \frac{1}{m} \sum_{i=1}^{m} D_{\boldsymbol{w}}(G_\theta(\boldsymbol{z}^{(i)}))$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \cdot \text{RMSProp}(\boldsymbol{\theta}, \boldsymbol{g}_{\boldsymbol{\theta}})$
end while

The introduction of EM distance allowed overcoming some unpleasant problems of GANs. First of all, the critic differently from discriminator could be potentially trained to optimality without incurring in vanishing gradient and also avoiding, in this way, the unpleasant situation deriving from mode collapse. In addition, empirical evidence suggests a correlation between the loss and the quality of generated samples. Although the loss will not represent a new metric, it still can help determine the training behavior for a given architecture because comparing different models with different critics requires to compute the constant scaling factor of each one that is intractable.

2.5.1 Gradient Penalty Regularization

Gulrajani et al. in [3] illustrated some issues arising from the weight clipping regularization method used to guarantee Lipschitz constraint on the WGAN critic.

They argue that constraining the critic to be defined within the box $[-c, c]$ is a too strict condition, and consequently the resulting set of functions satisfying such condition is a subset of K-Lipschitz function. Hence, such limitation causes the critic to learn simpler functions that ignore higher moments of the data distribution with the risk of missing the optimal critic f^*.

The regularization method proposed in WGAN-GP is motivated by the theoretical property for which it will exist a 1-Lipschitz function that is the optimal critic f^* and has the gradient norm almost 1 everywhere under $p_{\text{data}}$and p_g. Their experiments on WGAN show that during training, the gradient norm of the critic, which depend on the threshold value c, may either explode or vanish as it is propagated deeper in the network. It has also been observed that the WGAN critic gradient chases the maximum gradient norm k instead of getting close to 1, the gradient norm that the optimal critic would have. Hence, the solution proposed is to penalize the critic whenever its gradient norm is different from 1 as a form of regularization, but still there is no way to penalize gradient everywhere. With this in mind, a tractable and empirically promising approximation for this method, called the soft version, consists of applying the penalty only for random samples $\hat{\mathrm{x}} \sim p_{\hat{x}}$, where $p_{\hat{\boldsymbol{x}}}$ distribution represents all the points in $\mathcal{X}$ located on the straight lines joining pair of points x and y respectively sampled from p_{data} and p_g. Since the optimal critic f^* has gradient norm 1 on $\hat{\mathbf{x}}$ points, the penalty would encourage the same behaviour also for the trained critic. By adding the regularization term, the WGAN loss becomes:

$$L = \mathbb{E}_{\tilde{\mathbf{x}} \sim p_g}[D(\tilde{\boldsymbol{x}})] - \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[D(\boldsymbol{x})] + \lambda \mathbb{E}_{\hat{\mathbf{x}} \sim p_{\hat{\boldsymbol{x}}}}[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2], \tag{2.21}$$

where λ is a multiplicative constant typically set to 10.

It should be mentioned that with the introduction of the gradient penalty, the use of batch normalization in the critic network is discouraged since it would introduce correlations between images. This, as a result, will invalidate the penalty effectiveness as it requires to be applied to each sample independently. The use of layer normalization could be a valid alternative when normalization is necessary. Next, it is shown the full training algorithm.

Algorithm 3 WGAN with gradient penalty given the default values of $\lambda = 10$, $n_{\text{critic}} = 5$ $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$

Require: The gradient penalty coefficient λ, the number of critic iterations per generator item n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2
Require: $\boldsymbol{w}_0$, initial critic parameters . $\boldsymbol{\theta}_0$, initial generator parameters.
while $\boldsymbol{\theta}$ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    for $i = 1, ..., m$ **do**
      Sample $\{\boldsymbol{x}^{(i)}\}_{i=1}^{m} \sim p_{\text{data}(x)}$ from the real data.
      Sample $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$ a prior sample.
      Sample $\epsilon \sim U[0,1]$ a random number
      $\tilde{\boldsymbol{x}} \leftarrow G_{\boldsymbol{\theta}}(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} =\leftarrow D_{\boldsymbol{w}}(\tilde{\boldsymbol{x}}) - D_{\boldsymbol{w}}(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_{\boldsymbol{w}}(\hat{x})\|_2 - 1)^2$
    end for
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \cdot \text{Adam}(\nabla_{\boldsymbol{w}} \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, \boldsymbol{w}, \alpha, \beta_1, \beta_2)$
  end for
  Sample $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$ a batch of prior samples.
  $\theta \leftarrow \text{Adam}(\nabla_{\boldsymbol{\theta}} \frac{1}{m} \sum_{i=1}^{m} -D_{\boldsymbol{w}}(G_{\boldsymbol{\theta}}(\boldsymbol{z})), \boldsymbol{\theta}, \alpha, \beta_1, \beta_2)$
end while

Chapter 3

Graph Convolution

Graphs with their inherent structure allow a flexible representation of data in the non-Euclidean domain, and this has motivated their adoption in a variety of fields, including image processing. From a technical perspective, a graph is defined as an ordered pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with a non-empty set of nodes or vertices $\mathcal{V}$ and a set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, each of which connects a pair of vertices. A graph could be either directed or undirected based on the edge definition. In the first case, an edge indicates the direction of the connection, whereas in the second case, an edge represents a bidirectional connection between two vertices. In addition, in weighted graphs the connections are also characterized by weights, which express the strength of the relationships described by the edges. Then for $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{W})$, a weighted graph, it can be introduced $\boldsymbol{W} \in \mathbb{R}^{N \times N}$ a weight matrix in which each connection between vertices v_i and v_j is described by a w_{ij} entry representing the weight on such edge e_{ij}, or its absence if $w_{ij} = 0$.

In the same way that an image signal is represented as a discrete signal defined on each position of a 2D grid structure, a graph signal can be seen as a set of samples defined on each node of the graph structure. The most straightforward signal, consisting of only one sample per node, can be defined through $f : \mathcal{V} \mapsto \mathbb{R}$, a function that maps each of the N vertices to a real value, or equivalently by a vector representation $\boldsymbol{f} \in \mathbb{R}^N$ where the i-th element is the value that the function assumes at vertex v_i. Therefore, assuming that the 2D image grid is a special graph structure where each node is connected with its spatial adjacent pixels, the convolution on images can be viewed as a particular case of graph convolution on a grid-structured graph. However, although it might be possible to process graph signal with traditional methods by naively treating the graph vertices as an ordered sequence, it should be considered that in general, a graph lies in non-Euclidean space. Extending in this way to graph signals the concepts and the transforms used

on regular structured data, defined on Euclidean spaces, may lead to the definition of operations that will miss meaningful information embedded in the graph irregular structure. As a consequence, it is not straightforward to generalize even basic operators such as translation, which is one of the building blocks in performing the convolution operation between the graph signal $f(i)$ and the impulse response of a filter $h(i)$. This led to the definition of new approaches capable of extending the convolution operation to graph structured data. All these approaches can be classified in spectral-based methods and spatial-based methods.

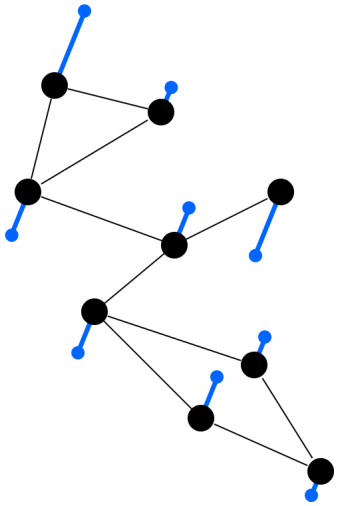


Figure 3.1: A graph signal.

3.1 Spectral-based Methods

Spectral-based methods for convolution [21] include all such methods that calculate the convolution of graph signals in the spectral domain. Given a weighted undirected graph $\mathcal{G}$ with N vertices, the graph Laplacian $\boldsymbol{L} \in \mathbb{R}^{N \times N}$ is a matrix describing the graph structure as:

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

where $\boldsymbol{D}$ is the degree matrix, a diagonal matrix in which the non zero element are $d_{ii} = \sum_j w_{ij}$. From a different perspective, the graph Laplacian describe a linear operator of difference $\mathcal{L}$ in the space of graph signals:

$$\mathcal{L}f(i) = \sum_{j \in \mathcal{N}_i} w_{ij} \left(f(i) - f(j) \right), \tag{3.2}$$

where the set $\mathcal{N}_i = \{v_j \ ; \ e_{ji} \in \mathcal{E}\}$ includes all the vertices v_j directly connected to node v_i by an edge.

For a generic finite length P digital signal and length Q filter, the linear convolution is calculated performing a circular convolution with n defined only in $[0, P+Q-2]$ interval, and it results in a new sequence with maximum length $R = P + Q - 1$:

$$(f * h)(n) = \sum_{m=0}^{P-1} f(m)h((n-m) \bmod R). \tag{3.3}$$

The convolution theorem states that the convolution of two sequences in the time domain is equal to the pointwise product of the signals in the frequency domain:

$$(f * h)(n) = \text{iDFT}\{\text{DFT}\{f\} \cdot \text{DFT}\{h\}\}, \tag{3.4}$$

where:

$$\text{DFT}\{f(n)\} = \sum_{n=0}^{N-1} f(n)e^{-2\pi nk/N} \tag{3.5}$$

$$\text{iDFT}\{F(k)\} = \frac{1}{N} \sum_{k=0}^{N-1} F(k)e^{2\pi nk/N}, \tag{3.6}$$

Similarly to digital signals DFT, it is possible to define a transform for graph signals called graph Fourier transform [22]. The main motivations behind calculating convolution in the spectral domain rather than in the spatial one is the lack of a defined translation operator for graph signals because the change of variable technique, used for regular signals, cannot be easily generalized for graphs.

The graph Laplacian matrix $\boldsymbol{L}$, which is real and symmetric can be decomposed via eigendecomposition in $\boldsymbol{L} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T$, where $\boldsymbol{U}$ is an orthonormal matrix composed of column eigenvectors $\boldsymbol{u}_l$ and $\boldsymbol{\Lambda}$ is a diagonal matrix with the corresponding λ_l eigenvalues for $l = 0, ..., N-1$. Assuming that the eigenvalues are sorted in ascending order $\lambda_0 \leq \lambda 1 \leq ... \leq \lambda_{N-1}$, and $\boldsymbol{u}_l$ with them, graph Fourier transform is defined as:

$$\mathcal{GF}\{f(i)\} = \hat{f}(\lambda_l) = \langle f, u_l \rangle = \sum_{i=1}^{N} f(i)u_l^*(i), \tag{3.7}$$

and its inverse transform as:

$$\mathcal{IGF}\{f(\lambda_l)\} = f(i) = \sum_{l=0}^{N-1} \hat{f}(\lambda_l)u_l(i). \tag{3.8}$$

The eigenbasis coordinates of signal vector $\boldsymbol{f}$ may have a similar interpretation to the energy of each frequency for regular structured signals. Indeed, each basis vector $\boldsymbol{u}_l$, indicates a different graph Fourier mode, which turns out to be smoother for lower eigenvalues and oscillate faster for the larger ones. As a result, the eigenvector $\boldsymbol{u}_0$ associated with λ_0, since it identifies the non-oscillating component, it is constant and assumes $1/\sqrt{N}$ value for each element of the basis vector. The $\mathcal{GFT}$ therefore shows a way of decomposing a generic graph signal into a linear combination of $\hat{f}(\lambda_l)$ modes. In the graph spectral domain, the convolution is calculated as:

$$\begin{aligned}(f * h)(n) &= \mathcal{IGF}\{\mathcal{GF}\{f\} \cdot \mathcal{GF}\{h\}\} \\ &= \sum_{l=0}^{N-1} \hat{f}(\lambda_l)\hat{h}(\lambda_l)u_l(i). \end{aligned} \tag{3.9}$$

Different limitations emerge from such a definition of convolution: the eigendecomposition has $O(n^3)$ computational complexity, and in particular, filters are domain-dependent so that learning them in the spectral-domain for a specific graph is not extensible to a general graph with a different structure. As a consequence, such methods require to constrain all the graphs in the dataset to have the same structure. Recent works [23][24] address these problems.

3.2 Spatial-based Methods

The spatial-based methods efficiently calculate convolution as information propagation over the graph nodes. Although these methods are more prone to scale to large graph applications, they do not have a strong theoretical foundation as the spectral counterpart. As described in [25], the spatial construction is derived by assuming that the usual 2D grid image structure is replaced with a generic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{W})$, where $\boldsymbol{W} \in \mathbb{R}^{N \times N}$ is the symmetric non-negative matrix of the weighted edges. For a given threshold δ and node v_j, the convolution is then performed as a local operation through the aggregation of the connected neighborhood of v_j:

$$\mathcal{N}_j = \{v_i \in \mathcal{V} : W_{ij} > \delta\}. \tag{3.10}$$

Given a deep graph neural network with multiple graph convolutional layers indexed by l, with $1 \leq l \leq l_{\max}$, and let $\mathcal{V}^0$ be the initial graph structure $\mathcal{V}$, then for each layer l of the network, the input graph represented by $\mathcal{V}^{l-1}$ is partitioned into d^l number of clusters. Every cluster will determine a vertex in $\mathcal{V}^l$ and corresponds to neighborhood in $\mathcal{V}^{l-1}$, which groups all the vertex in the same defined partition. In each layer, the graph signal is processed by calculating the convolution within

each neighborhood. As a result, the incoming F^{l-1}-dimensional graph signal over the graph $\mathcal{V}^{l-1}$ is transformed in an F^l-dimensional signal over $\mathcal{V}^l$. As the signal propagates during the inference, through all the network, the graph structure will progressively become coarser, while the graph signal dimensionality will increase, as happens in CNNs.

At layer l, the signal $\boldsymbol{x}_j^l$ is obtained as:

$$x_j^l = P^l \sigma \left(\sum_{i=1}^{F^{l-1}} \boldsymbol{K}_{ij}^l \boldsymbol{x}_i^{l-1} \right) \ (j, = 1, ..., F^l) \tag{3.11}$$

where F^l indicates the number of filters, $\boldsymbol{x}_i^{l-1} \in \mathbb{R}^{d^{l-1} \times F^{l-1}}$ represents the input graph signal and $\boldsymbol{K}_{ij}^l \in \mathbb{R}^{d^{l-1} \times d^{l-1}}$ is the filter matrix that is nonzero only for the element in $\mathcal{N}^l = \{\mathcal{N}_i^l \ ; \ i = 1, ..., d^{l-1}\}$, the set of neighborhoods around each vertex in $\mathcal{V}^{l-1}$. Lastly $\sigma(\cdot)$ stands for a non-linear activation function and P^l for a pooling operation over each cluster for $\mathcal{V}^l$ nodes. This simple method, unfortunately is not capable of sharing filter weight over the nodes of the graph.

3.2.1 Edge-Conditioned Convolution with Dynamic Filters

In defining Edge-Conditioned Convolution (ECC) [26], inspired by Dynamic filter networks [27], Simonovsky et al. proposed a viable method of weights sharing for the spatial-based graph convolution. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed or undirected graph with N vertices and M edges. For each layer l two essential function are introduced, namely a vertex labeling function $X^l : \mathcal{V} \mapsto \mathbb{R}^{F^l}$ that assigns a graph signal to each vertex and an edge labeling function $L : \mathcal{E} \mapsto \mathbb{R}^S$ that likewise assigns edge features. ECC, as other spatial methods, computes the convolution in an information propagation fashion, then the filtered signal on every vertex is obtained from a weighted sum of the neighboring nodes signals. The filter weights involved into this aggregation are conditioned on the edge label. This is where Dynamic filter networks [27] are employed, providing edge-specific filter matrices generated from the edge labels L with a neural network $\mathcal{F}^l : \mathbb{R}^S \mapsto \mathbb{R}^{F^l \times F^{l-1}}$. Inspired by the regular convolution on the grid structure, the filters are shared along the layer l using the same network $\mathcal{F}^l$ to generate them and the aggregation method operates locally in the neighborhood $\mathcal{N}_i = \{v_j \ ; \ e_{j,i} \in \mathcal{E}\} \cup \{v_i\}$. The resulting equation for

the Edge Conditioned Convolution is:

$$\begin{aligned} X^l(i) =& \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} \mathcal{F}^l(L(j,i); \boldsymbol{W}^l) X^{l-1}(j) + \boldsymbol{b}^l \\ =& \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} \boldsymbol{\Theta}^l_{ji} X^{l-1}(j) + \boldsymbol{b}^l \end{aligned} \tag{3.12}$$

where $\boldsymbol{\Theta}^l_{ji}$ filters are the dynamically generated from $\mathcal{F}^l$ network, and $\boldsymbol{W}^l$ and $\boldsymbol{b}^l$ are learnable weights .

The method flexibility is emphasized in [26] describing the approach adopted in a point in cloud classification problem, in which application the graph is constructed by creating $\mathcal{V}$ vertices from each point $p \in \mathbb{P}$ and assigning on each one its own signal $X^0(i) = X_p(p)$. Direct edges $e_{i,j}$, which connect every vertex v_i to all elements v_j in its defined neighborhood $\mathcal{N}_i$, are labeled by L, a function that computes spatial distance between the edge nodes in Cartesian and spherical coordinates $L(v_j, v_i) = (\delta_x,\ \delta_y,\ \delta_z,\ \|\delta\|,\ \arccos \delta_z/\|\delta\|,\ \arctan \delta_y/\delta_x)$, where δ represent the difference w.r.t. the subscript coordinate. Nevertheless, nothing prevents from using in the possible applications of this method other labeling functions based on different distance metrics.

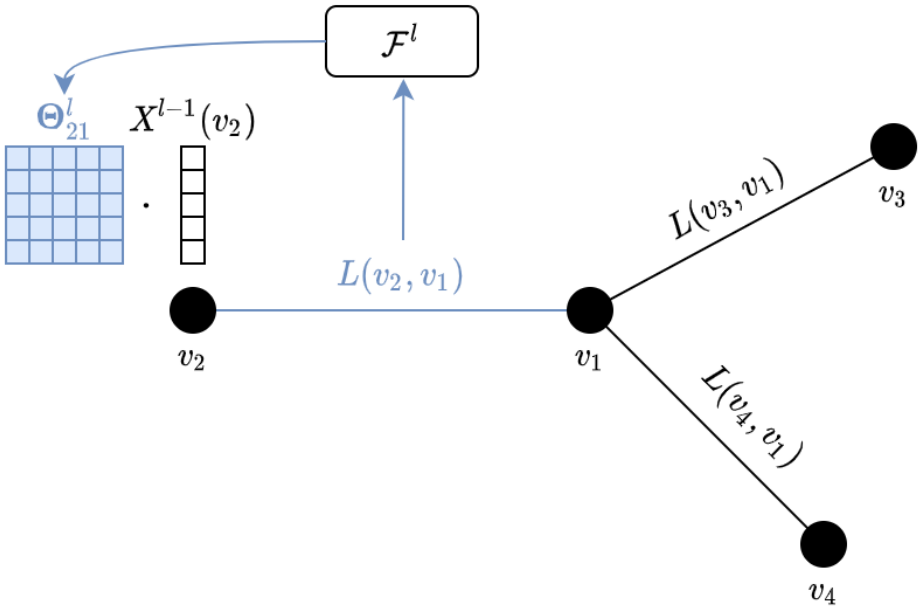


Figure 3.2: The edge specific weight matrix $\boldsymbol{\Theta}_{21}$ is generated by $\mathcal{F}^l$ for computing the neighbor v_2 contribution $\boldsymbol{\Theta}^l_{21} X^{l-1}(2)$ to the convolution over $\mathcal{N}_1$ centered on v_1.

3.2.2 Graph convolutional layer

In [4], Valsesia et al. introduce the graph convolutional layer in which they use a graph convolution operation, in a novel way, to capture self-similarities on images on distant spatial locations with features similar to the center pixel of convolution. This method has been used for image denoising since it was inspired by previous classic approaches based on harnessing self-similarities for such a task.

Given an image with N pixels and a F^l dimensional signal defined on each of them, the graph convolution is used to aggregate the feature vectors in $\boldsymbol{H}^l \in \mathbb{R}^{F^l \times N}$ with their own neighbors, represented by the closest nodes in terms of similarity distance in the hidden space of layer l. Specifically, two convolution are computed in the graph convolutional layer, respectively one inspecting spatial local position and the other inspecting non-local areas. The resulting signals are aggregated as follows:

$$\boldsymbol{h}_i^{l+1} = \frac{\boldsymbol{h}_i^{l+1,\mathrm{NL}} + \boldsymbol{h}_i^{l+1,\mathrm{L}}}{2} + \boldsymbol{b}^l, \tag{3.13}$$

where $\boldsymbol{h}_i^{l+1,\mathrm{L}} \in \mathbb{R}^{F^l}$ represents the output of a regular 3×3 convolution, whereas $\mathbf{h}_i^{l+1,\mathrm{NL}}$ is the outcome of a non-local convolution. This method results in an operation with an adaptive receptive field, which will also include the image signal's areas where the feature most similar to the center node of convolution are located.

For each graph convolutional layer a graph is dynamically built by connecting each pixel i to its k most similar neighbors in the feature space $\mathbb{R}^{F^l}$ based on the Euclidean distance of their feature vectors $\|d^{l,j\to i}\| = \|\boldsymbol{h}_j^l - \boldsymbol{h}_i^l\|_2$. It must be pointed out that in selecting the neighbors the pixels already considered by local convolution are excluded from the graph construction. Then, the node aggregation for non-local convolution here is performed through the ECC method [26] applied on each node of the k-regular graph $\mathcal{G}^l = (\mathcal{V}, \mathcal{E}^l)$, with $|\mathcal{V}| = N$ and $\mathcal{E}^l \subseteq \mathcal{V} \times \mathcal{V}$. Notably, the generated filter weights $\boldsymbol{\Theta}^{l,j\to i}$ are conditioned on the edge label $L(i,j) = \boldsymbol{h}_j^l - \boldsymbol{h}_i^l$, which is also based on the distance in $\mathbb{R}^{F^l}$.

Starting from (3.12) Valsesia et al. have introduced the graph convolution, equipped with an edge-attention term $\gamma^{j\to i}$, as follows :

$$\begin{aligned} \boldsymbol{H}_i^{l+1,NL} &= \sum_{j \in \mathcal{N}_i^l} \gamma^{l,j\to i} \frac{\mathcal{F}_{\boldsymbol{w}^l}^l(d^{l,j\to i})\boldsymbol{H}_j^l}{|\mathcal{N}_i^l|} \\ &= \sum_{j \in \mathcal{N}_i^l} \gamma^{l,j\to i} \frac{\boldsymbol{\Theta}^{l,j\to i}\boldsymbol{H}_i^l}{|\mathcal{N}_i^l|}, \end{aligned} \tag{3.14}$$

where $\mathcal{N}_i^l$ is the neighborhood of v_i, and $\boldsymbol{\Theta}^{l,j\to i} \in \mathbb{R}^{F^l \times F^{l+1}}$ is the dynamic filters

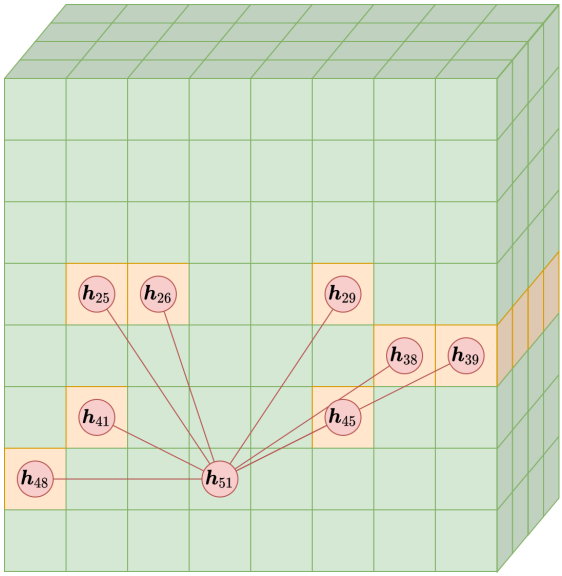


Figure 3.3: The k most similar neighbors are selected for pixel at position 51 from the features vectors in $\boldsymbol{H}^{\boldsymbol{l}}$

matrix generated by $\mathcal{F}^l_{\boldsymbol{w}^l}$, a fully connected neural network parameterized by $\boldsymbol{W}^l$ weight matrix. The term $\gamma^{j\to i}$ makes learning more stable by shrinking the signals coming from the furthermost features in hidden space by a factor:

$$\gamma^{l,j\to i} = \exp(-\|\boldsymbol{d}^{j\to i}\|_2^2/\delta). \tag{3.15}$$

Thus, $\gamma^{l,j\to i}$ only depends on δ hyper-parameter and on edge label as for $\boldsymbol{\Theta}^{l,j\to i}$, so it is likewise shared in the network layer.

The use of ECC, in this context, should provide adaptive filters that can further generalize convolution for spatial distant location, at the cost of adding more complexity and making learning harder.

Lightweight ECC

In [4] a lightweight version of ECC is introduced with the aim of mitigating its shortcomings, namely the space complexity and the over-parameterization, with two techniques: low-rank node aggregation and circulant approximation.

Low-rank node aggregation This method enables a significant decrease in memory requirement for the ECC computation. A considerable amount of memory is used even for the simplest dynamic filter network $\mathcal{F}$, consisting of a feedforward

neural network with a single hidden layer. In fact, $\mathcal{F}$ has to generate a $\boldsymbol{\Theta}^{l,j\to i}$ weight matrix for each neighbour v_j in $\mathcal{N}_i$, for each pixel i and image in the batch.

The low-rank approximation adopts the substitution of $\boldsymbol{\Theta}^{l,j\to i}$ with a rank $r < \text{rank}(\boldsymbol{\Theta}^{l,j\to i})$ approximation matrix $\tilde{\boldsymbol{\Theta}}^{l,j\to i}$. This method is based on the singular value decomposition, which for a generic real matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ is defined as the factorization into the product of three matrices:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{3.16}$$

where both $\boldsymbol{U} \in \mathbb{R}^{m\times m}$, $\boldsymbol{V} \in \mathbb{R}^{n\times n}$ are orthonormal matrices, and $\boldsymbol{\Sigma} \in \mathbb{R}^{m\times n}$ is a diagonal matrix containing all the singular values of $\boldsymbol{A}$ on its diagonal. Importantly, the decomposition can be performed in such a way that arranges the singular values in a descending order over the $\boldsymbol{\Sigma}$ diagonal. Now, the Eckart Young Mirsky theorem:

$$\begin{aligned} \min_{\tilde{\mathbf{A}}} \quad & \|\mathbf{A} - \tilde{\mathbf{A}}\|_2 = \sigma_{k+1} \\ \text{s.t.} \quad & \text{rank}(\tilde{\mathbf{A}}) = r, \end{aligned} \tag{3.17}$$

states that the optimal rank $r < \text{rank}(\mathbf{A})$ approximation to $\boldsymbol{A}$ is achievable with the truncated SVD method in which only the r largest singular value and relative singular vector are computed:

$$\begin{aligned} \tilde{\boldsymbol{A}} &= \boldsymbol{U}_r \tilde{\boldsymbol{\Sigma}}_r \boldsymbol{V}_r^T \\ &= \sum_{i=1}^{r} \boldsymbol{\sigma}_i \boldsymbol{u}_i \boldsymbol{v}_i^T . \end{aligned} \tag{3.18}$$

With this in mind, each generated matrix $\boldsymbol{\Theta}^{l,j\to i}$ can be likewise approximated by the sum of rank one matrices as:

$$\tilde{\boldsymbol{\Theta}}^{l,j\to i} = \sum_{s=1}^{r} \kappa_s^{j\to i} \boldsymbol{\theta}_s^{j\to i,L} \boldsymbol{\theta}_s^{j\to i,R^T} \tag{3.19}$$

with $\boldsymbol{\theta}_s^{j\to i,L} \in \mathbb{R}^{F^l}$, $\boldsymbol{\theta}_s^{j\to i,R^T} \in \mathbb{R}^{F^{l+1}}$ and $\kappa_s^{j\to i} \in \mathbb{R}$ and $1 \leq r \leq F^l$. However, since $\boldsymbol{\theta}_s^{j\to i,L}$ and $\boldsymbol{\theta}_s^{j\to i,R^T}$ are not necessarily orthogonal r represents the maximum possible rank attained by the approximation matrix. The use of $\tilde{\boldsymbol{\Theta}}^{l,j\to i}$ in the output layer of $\mathcal{F}$ will thus mitigate the memory occupation issues by reducing learned parameters from $F^{\text{hidden}} \cdot F^l \cdot F^{l+1}$ to $F^{\text{hidden}} \cdot r(F^l + F^{l+1} + 1)$, not including the bias weights. In addition, this method avoids the explicit computation of the entire matrices $\boldsymbol{\Theta}^{l,j\to i}$, thus preventing to fully load it in memory, and instead, it just obtains the edge filters by multiplying the edge labels to one factor at a time. In conclusion low-rank node aggregation will benefits from the smaller spatial requirements for filter storage and from lower computational complexity, which

become $O(r(F^l + F^{l+1} + 1)$. Yet, this method also has a considerable disadvantage since it requires a more accurate weight initialization to avoid different order of magnitude between graph convolutional layer input and output, and between incoming and outcoming gradients during backpropagation because if they are on a different scale they could lead to exploding or vanishing gradients.

Circulant approximation of dense layer In [4] a further approximation is used on $\mathcal{F}$ still on the second affine layer that generates $\boldsymbol{\theta}_s^{j\to i,L}$ and $\boldsymbol{\theta}_s^{j\to i,R^T}$. The use of a circulant matrix structure prevents the rise of issues related to over-parameterization, which can lead to vanishing gradients. The approximate weight matrix used is composed of multiple stacked partial circulant matrices. In each partial matrix there are the same learning parameter but with a different circular shift. Given a weight matrix $\boldsymbol{W} \in \mathbb{R}^{n\times m}$ which defines a linear map $\mathbb{R}^n \mapsto \mathbb{R}^m$, its approximation is composed by p replicas $\tilde{W}^1, ..., \tilde{W}^p \in \mathbb{R}^{n\times m/p}$ where each of these represents the same matrix whose elements are shifted by m/p as if it were a flatten row-major vector. As a result the aforementioned methods cuts down $\boldsymbol{W}^L$ dimension from $F^l F^l r$ to $F^l \cdot \frac{F^l}{m} r$ and analogously $\boldsymbol{W}^R$ to $F^l \frac{F^{l+1}}{m} r$.

3.2.3 Edge Convolution

The Edge Convolution (EdgeConv) from [28] is another spatial-based method for graph convolution that was used in a similar context to [4], namely on a dynamically built k-nearest neighbors graph $\mathcal{G}^l = (\mathcal{V}, \mathcal{E}^l)$. Given $\mathcal{G}^l$ EdgeConv computes edge features $\boldsymbol{e}_{ij}$ for every edge e_{ij} connected to a node v_i through the function $h_{\boldsymbol{\phi}} : \mathbb{R}^{F^l} \times \mathbb{R}^{F^l} \mapsto \mathbb{R}^{F^{l+1}}$ implemented by a MLP with parameters $\boldsymbol{\Phi}$. Subsequently, the edge features are aggregated with a sum or max operation over all the neighborhood. Precisely, the method proposed by Wang et al. calculate edge features as:

$$\boldsymbol{e}_{ij} = h_\Phi(\boldsymbol{h}_i^l \,;\, \boldsymbol{h}_j^l - \boldsymbol{h}_i^l) \tag{3.20}$$

in the attempt to capture both the global structure from $\boldsymbol{h}_i$ features and the relative structure with the neighbors' distances in the feature space. The h function can be implemented also by 1×1 convolution with $\boldsymbol{W}_{\boldsymbol{\psi}}^l \in \mathbb{R}^{F^l \times F^{l+1}}$ and $\boldsymbol{W}_{\boldsymbol{\phi}}^l \in \mathbb{R}^{F^l \times F^{l+1}}$ weight matrices, where $m = 1, ..., F^{l+1}$ represent the index of resulting output channel. Thus, for the sum aggregating function it results:

$$\boldsymbol{h}_{im}^{l+1} = \sum_{j\in\mathcal{N}_i^l} \mathrm{ReLU}(\boldsymbol{W}_{m;\boldsymbol{\phi}}^l \cdot (\boldsymbol{h}_j^l - \boldsymbol{h}_i^l) + \boldsymbol{W}_{m;\boldsymbol{\psi}}^l \cdot \boldsymbol{h}_i^l). \tag{3.21}$$

Notably, as the authors argue, this definition may recall the standard convolution on the 2D grid where the h function would provide the weighted contribution for

each input location in the sliding window. Moreover, although it shares a similar aggregation function, this operation differs from ECC[26] since its MLP network directly computes the contribution of each neighbor to the convolution operation rather than providing a projection matrix conditioned on each edge pair.

Chapter 4

Modeling Non-Local Dependencies for Image Generation

Image synthesis is a rather complex task to which Deep Convolutional GANs have given a significant contribution [19], proving the convolutional layer's effectiveness for the generation of reasonably good quality image samples. It is well known that convolution and strided convolution successfully captures and reproduces local patterns in the image that fall into their receptive field. This means that spatially distant dependencies in an image are modeled by using multiple stacked convolutional layers, which result in a greater indirect receptive field as the network goes deep. Consequently, the only way the generator could learn the long distance dependencies is as a composition of convolution operations, which in practice in most of the cases this does not happen properly since the generator miss the overall image structure. In fact, as Zhang et al. pointed out in [29], by analyzing the sample produced from deep convolutional GANs, one could notice that the network is usually more prone to learn successfully textures and spatially local structures rather than the structural image patterns. To this end, the use of bigger kernels to enlarge the convolutional layer receptive field would not be a realistic solution since it would make the network lose the efficiency benefits deriving from small kernels, hence resulting in an unfeasible method for very deep networks.

4.1 Self-Attention and Graph Convolutional Layer

Self-Attention The self-attention mechanism on image defined in [30],[31], and adapted to the GAN framework in [29] has proven its effectiveness in overcoming the limitations in learning structural patterns. In self-attention GAN (SAGAN) this method is used in both the generator for modeling dependencies from distant locations in the generated images, and in the discriminator to control and enforce the structural constraints more accurately. Let $\boldsymbol{X} \in \mathbb{R}^{N \times F}$ be the activation maps for a generic layer, the attention mechanism is defined as:

$$\boldsymbol{y}_i = \frac{1}{\mathcal{C}(\boldsymbol{X})} \sum_{\forall j} d\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) h\left(\boldsymbol{x}_j\right). \tag{4.1}$$

This equation describe a non-local operator in which the output features $\boldsymbol{y}$ are computed by considering for each position i in the activation maps the similarity distance between all possible j positions with $d(\boldsymbol{x}_i, \boldsymbol{x}_j)$. The other terms h and $\mathcal{C}$ represent respectively a function that maps $\boldsymbol{x}_j$ to a new representation, and a normalization factor required to normalize over the N positions. Precisely, the formulation used in [29] computes the similarity distance with a Gaussian kernel in an affine space, so that equation (4.1) becomes:

$$\boldsymbol{o}_i = \boldsymbol{W}_v \sum_{j=1}^{N} \left[\text{softmax}(\boldsymbol{W}_f \boldsymbol{x_i})^T \boldsymbol{W}_g \boldsymbol{x}_j)\right] \boldsymbol{W}_h \boldsymbol{x_j} \tag{4.2}$$

where $\boldsymbol{W}_f, \boldsymbol{W}_g, \boldsymbol{W}_h, \boldsymbol{W}_v^T \in \mathbb{R}^{F \times \bar{F}}$ are learnable weight matrices used for 1×1 convolution and $\bar{F}$ is the depth dimension in the affine space. The output of the attention block:

$$\boldsymbol{y}_i = \gamma \boldsymbol{o}_i + \boldsymbol{x}_i, \tag{4.3}$$

is weighted by a scalar γ initialized as 0 to gradually introduce its contribution during training by assigning as much emphasis as needed on non-local dependencies as the iterations proceed.

Graph Convolutional Layer This method, in comparison, pursues a different approach in which the convolutional layer's receptive field is directly enlarged to also include self-similarities as non-local dependencies, in performing the convolution. The graph convolutional layer [4] during the convolution operation, indeed, considers for each neuron $\boldsymbol{x}_i$ not only neurons inside the surrounding local receptive field, but also the neurons $\boldsymbol{x}_j$ outside that share some similarity with $\boldsymbol{x_i}$. This can be

achieved using a graph convolution operation on a dynamically generated k-nearest neighbor graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, defined by the set of vertices $\mathcal{V}$, one for each position i in $\boldsymbol{X}$ and the set of edges $\mathcal{E}$ that connects each $\boldsymbol{x}_i$ with its most similar neighbors $\boldsymbol{x}_j$. In particular, the graph representation as a generalization of the 2D regular grid allows the use of an arbitrary metric d different from the spatial distance between two nodes. Hence the definition of d as the Euclidean distance in feature space, $d(\boldsymbol{x}_i, \boldsymbol{x}_j) = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2$ [4] [28]. The graph built is then exploited in computing the non-local output features for each position i as the aggregation of the features $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, for $j \in \mathcal{N}_i$, by a spatial-based graph convolution operation.

As a result, the graph convolutional layer is capable of modeling dependencies among spatially distant pixels in the image in an efficient way with sparse connectivity and shared weights property as for regular convolution. Consistently with the method proposed in the graph convolutional layer [4] the outcome of such operation will be aggregated with the output of a regular convolution as defined in the equation (3.13) to simultaneously consider the contribution of both local and non-local receptive field in generating the new activation maps. It should be emphasized that the neurons that already fall in the local receptive field are excluded from the non-local receptive field to avoid consider them twice.

4.2 Proposed Architectures

For the sake of this study, the introduction of graph convolutional layers in the WGAN framework was considered strictly necessary only for the generator network to model long-distance dependencies in a more direct way. Consequently, in this work, a single convolutional architecture was tested for the critic network and three different architectures for the generator. Respectively: a convolutional baseline model (Figure 4.2), a second model that implements the graph convolutional layer [4] through EdgeConv [28] (Figure 4.3) and a third one that implements the same operation through ECC [26] (Figure 4.4). Initially, lightweight ECC [4] was a candidate method too, but unfortunately its use has caused severe instability problems during training, thus making it impractical.

As shown in Figure 4.1, the critic is composed of three 5×5 convolutional layers with stride 2 to downsample the image inputs as they pass through the network up to the last layer, whose output is used to compute the estimate of the Wasserstein distance that separates the sample distribution from the generated one. It must be noted that batch normalization was just omitted as suggested in [3] because it prevents the independent calculation of the gradient penalty for each sample.

The generator baseline shown in Figure 4.2 consists of a CNN equipped with

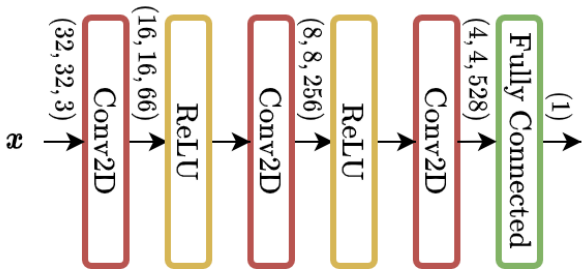


Figure 4.1: Critic architecture

3×3 kernel size convolutions. The starting noise vector $\boldsymbol{z}$ of size 132 is mapped to a $4 \times 4 \times 528$ tensor by a fully connected layer (every feature size used is designed to be multiple of 33 and 11 to be compatible with the Lightweight ECC rank and circulant approximations parameters). As the signal goes through the generator network, it is upsampled using the nearest-neighbor interpolation followed by a convolution that halves the number of features preserving spatial complexity. Then, a further 3×3 convolution is performed, leaving the signal depth unchanged. The upsampling plus two convolutions block is replicated until the signal reaches $32 \times 32 \times 66$ size. Finally, the activation maps depth 66 is transformed to 3, the RGB channels, by a 1×1 convolution.

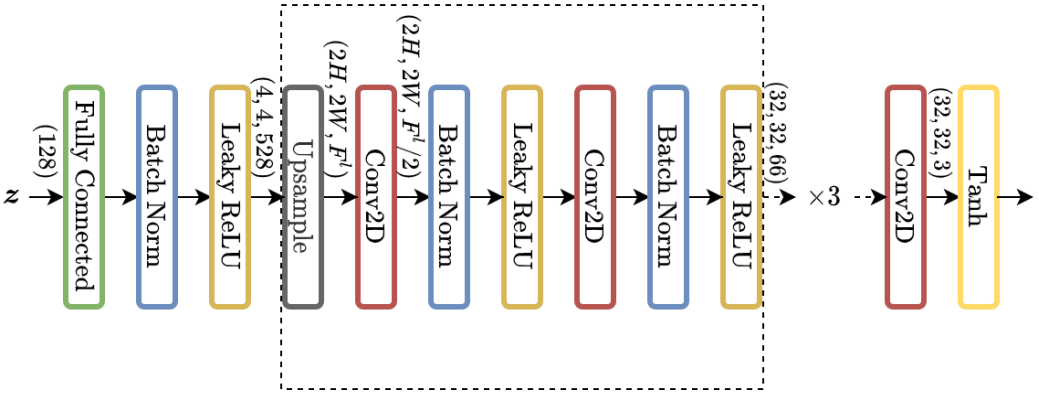


Figure 4.2: Baseline generator architecture

The architectures featuring graph convolutional layers are obtained from the baseline model by replacing the second convolution after the upsampling with a graph convolutional layers, except for the first block. This choice was motivated by the high number of features, namely 264, which makes ECC impractical since the dimensionality of $\boldsymbol{\Theta}_{ij}^{l}$ depends cubically on F^l. Moreover, a graph convolution on a small size 8×8 activation map would not result much different from a regular

convolution.

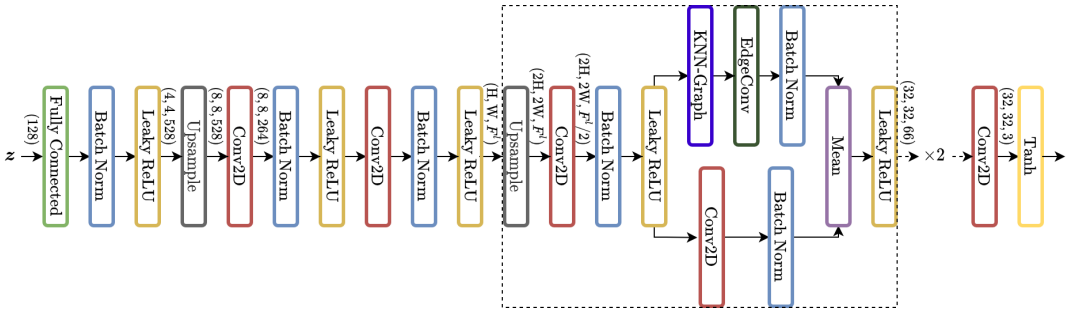


Figure 4.3: EdgeConv implementation of graph convolutional generator architecture

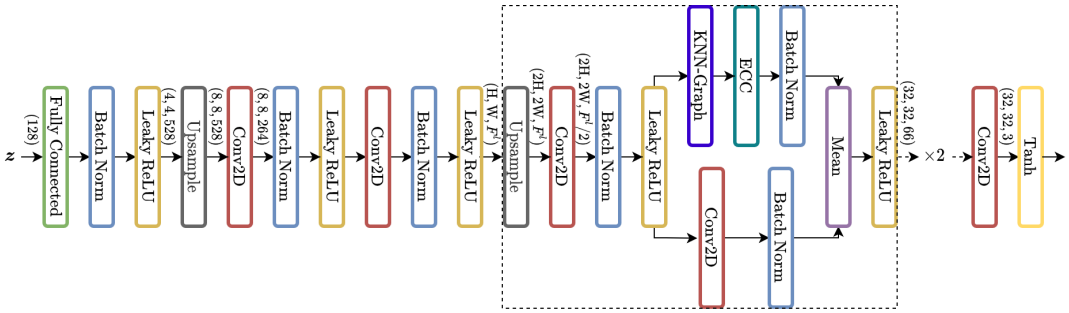


Figure 4.4: ECC implementation of graph convolutional generator architecture

In each graph convolutional layer the k-nearest neighbor graph $\mathcal{G}^l$ is constructed accordingly to [4] and [28] by selecting the nearest nodes to $\boldsymbol{h}_i^l$ in $\mathbb{R}^{F^l}$, the feature space learned by the previous layer. Subsequently, the Cartesian coordinates of row and column are concatenated to the existing features $\boldsymbol{h}^l$ to add a spatial context. Then, the exact method defined by each operator was respected during the calculation of non-local features, hence in ECC the $\boldsymbol{\Theta}_{ij}^l$ matrices are derived from the label function $L(i,j) = \boldsymbol{h}_j^l - \boldsymbol{h}_i^l$, whereas in EdgeConv the edge features $\boldsymbol{e}_{ij}$ are computed from $(\boldsymbol{h}_i \; ; \; \boldsymbol{h}_j - \boldsymbol{h}_i)$ since this method directly involves the feature vector of the pixel at position i.

4.3 Experiments

The proposed experiments are focused on the potential advantages or disadvantages stemming from the use of graph convolutional layers and not to compare the presented models to alternative methods such as SAGAN [29]. For this purpose, a

comparison based on the Inception Score (IS) was made between the baseline and graph convolutional models, therefore the results must be interpreted with this in mind. In all the experiments the WGAN-GP was trained in an unsupervised approach on the CIFAR-10 training dataset, consisting of 50000 samples, with a $5:1$ iteration ratio between the critic and the generator for 10^5 generator update step. All the networks were trained with the RMSprop optimizer and a batch size of 32, or 16 and 8 when required from GPU memory capacity limits. Respectively the starting learning rates used for each batch size were 1×10^{-4}, 7×10^{-5} and 5×10^{-5} decayed linearly to 0 after 10^5 generator iterations. Several non-local receptive field sizes were evaluated for each graph convolutional network, parametrized by the number of selected neighbors k, to analyze the contribution of modeling the non-local dependencies between the neuron in the center of convolution and a more extensive area represented by its neighborhood $\mathcal{N}_i$, of size k, in the dynamically generated graph $\mathcal{G}^l$.

Table 4.1 shows the inception scores obtained at the end of training for the different architectures and neighborhood sizes, whereas Figure 4.5 shows the IS progress over the generator training iterations.

Generator Architecture	batch size	learning rate	k	Inception Score
Baseline	32	1×10^{-4}	0	6.442 ± 0.511
EdgeConv	32	1×10^{-4}	8	6.154 ± 0.421
EdgeConv	32	1×10^{-4}	16	5.889 ± 0.345
EdgeConv	32	1×10^{-4}	32	5.942 ± 0.492
ECC	32	1×10^{-4}	8	6.204 ± 0.356
ECC	16	7×10^{-5}	16	5.802 ± 0.293
ECC	8	5×10^{-5}	32	5.649 ± 0.217

Table 4.1: Inception scores obtained on CIFAR-10 after 1×10^5 generator iterations

Figure 4.5: Inception score progress over the generator training iteration

The collected results show that there is no clear evidence of whether this method may help or not the generator to capture spatially distant dependencies. The generators that include graph convolutional layer do not outperform the baseline, but instead, their inceptions scores are very close to it. Specifically, the slightly worse results obtained from ECC networks with $k = 16$ or $k = 32$ are probably due to the smaller batch size and learning rate. Moreover, by visually inspecting the samples generated by the graph convolutional architectures, it can be seen that in some limited cases they have a somewhat meaningful structure, and their belonging class is recognizable. Nevertheless, this happens also for baseline samples since some structures such as car shapes, horses and deer side views appear easier to model. Still, the remaining majority of samples have a blob shape without any defined structure as happens for the baseline architecture.

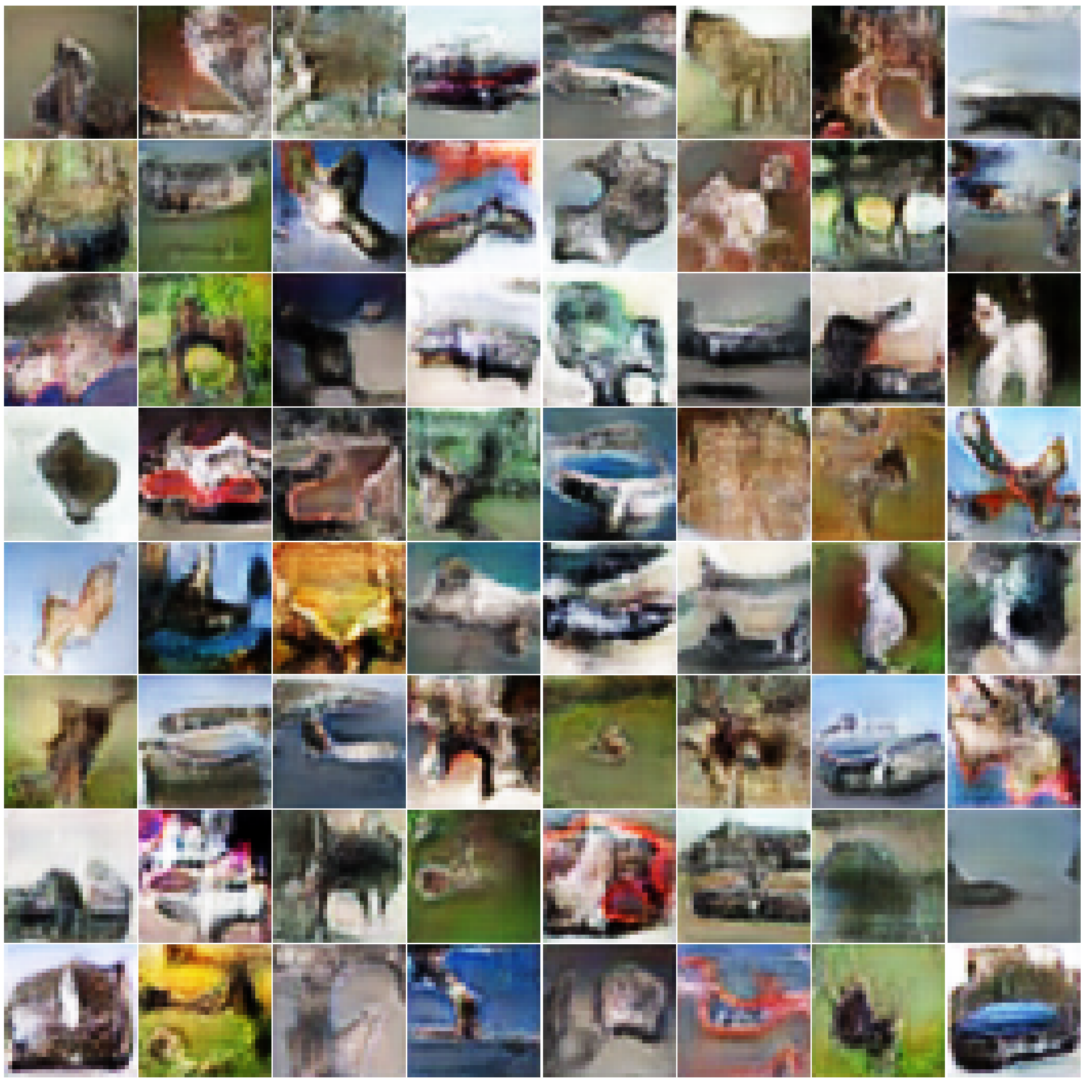


Figure 4.6: Samples generated from baseline model at iteration 100000

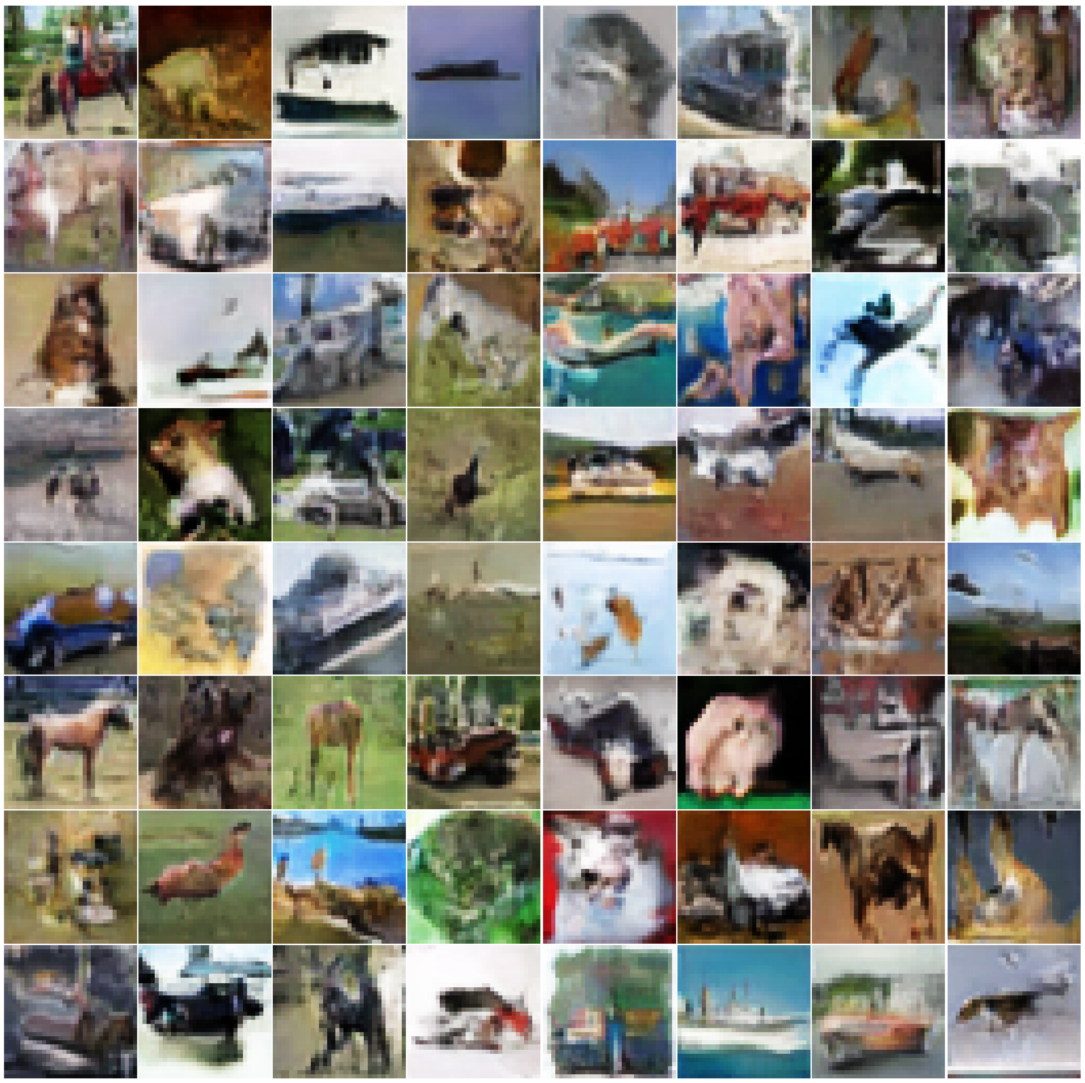


Figure 4.7: Samples generated from EdgeConv model with $k = 8$ at iteration 100000

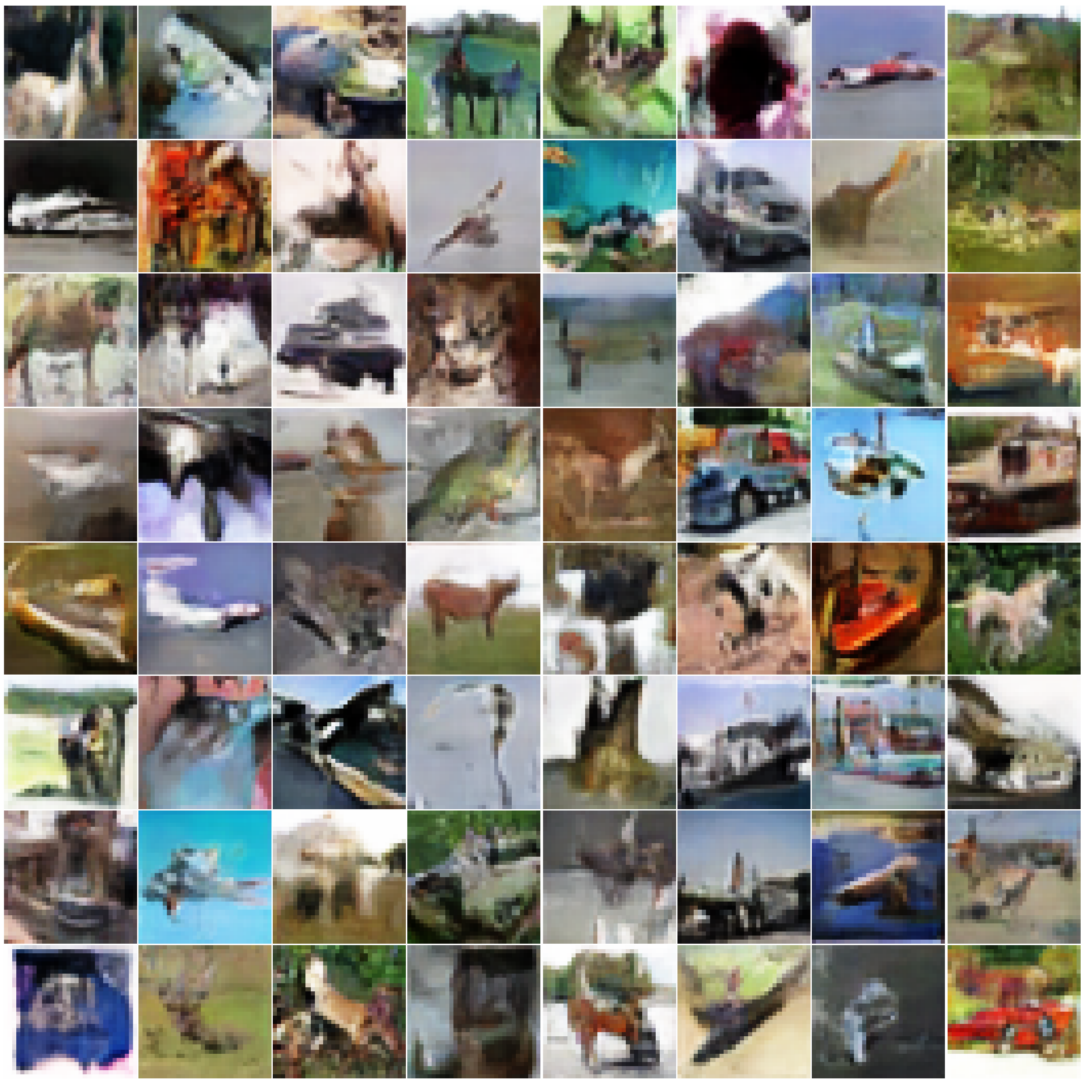


Figure 4.8: Samples generated from EdgeConv model with $k = 16$ at iteration 100000

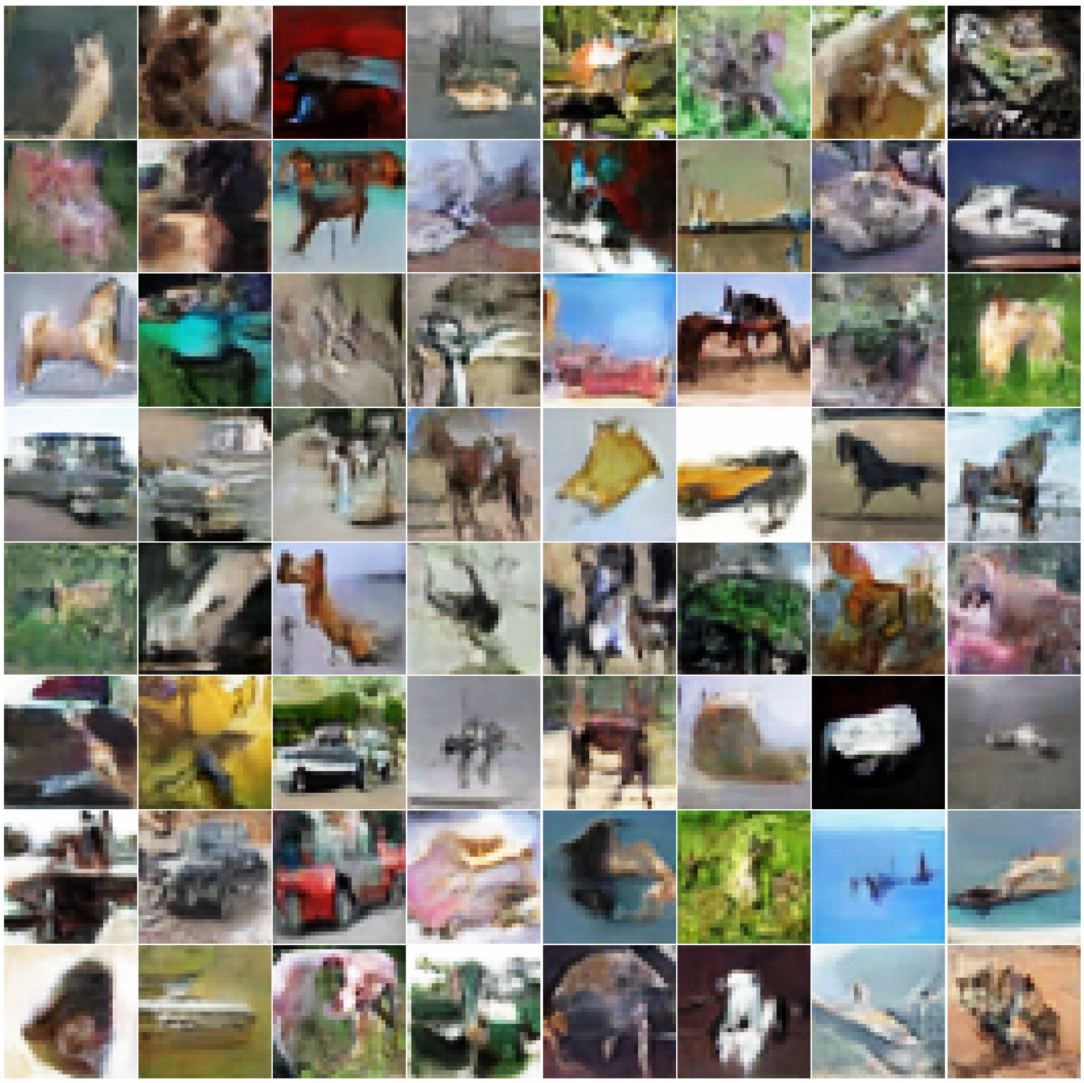


Figure 4.9: Samples generated from EdgeConv model with $k = 32$ at iteration 100000

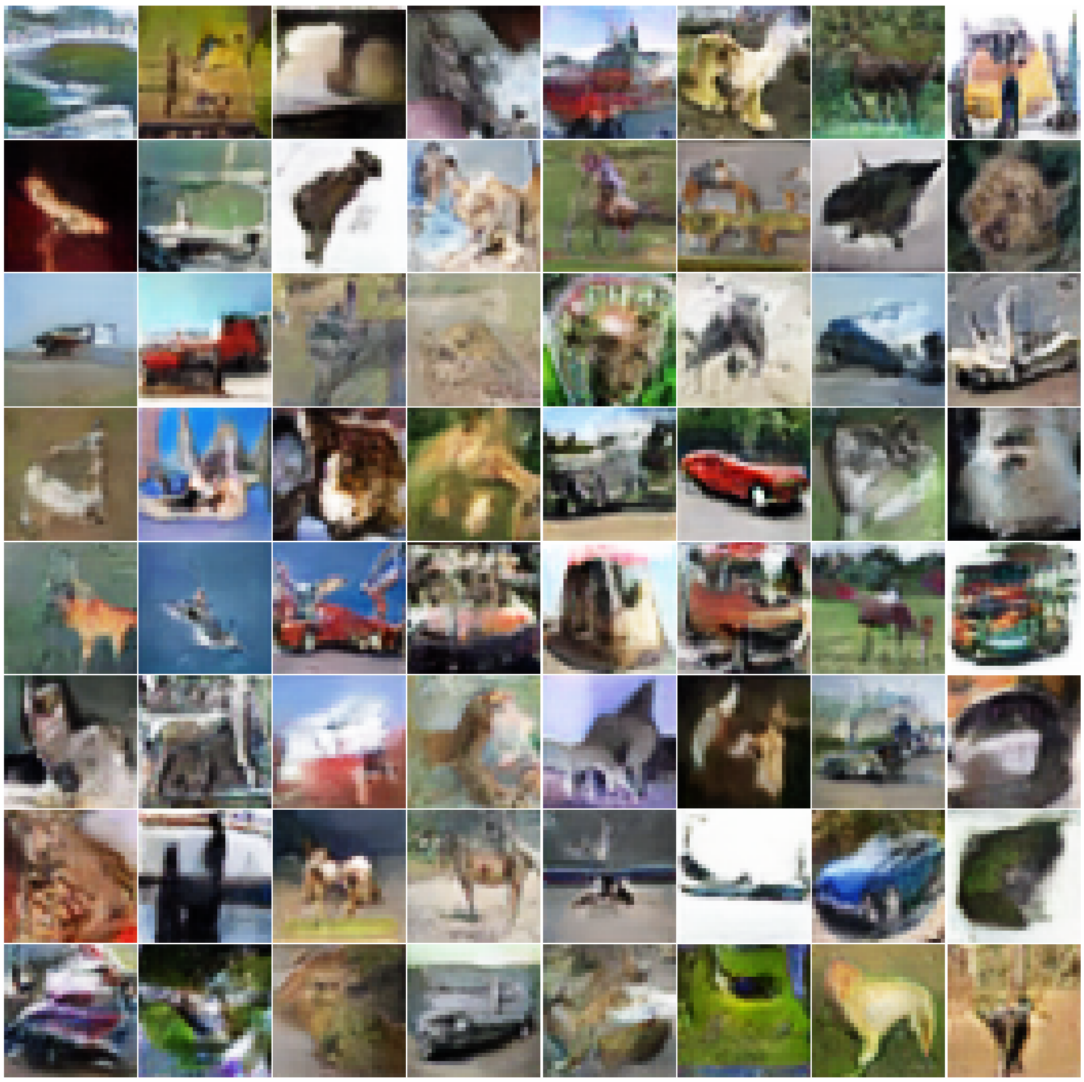


Figure 4.10: Samples generated from ECC model with $k = 8$ at iteration 100000

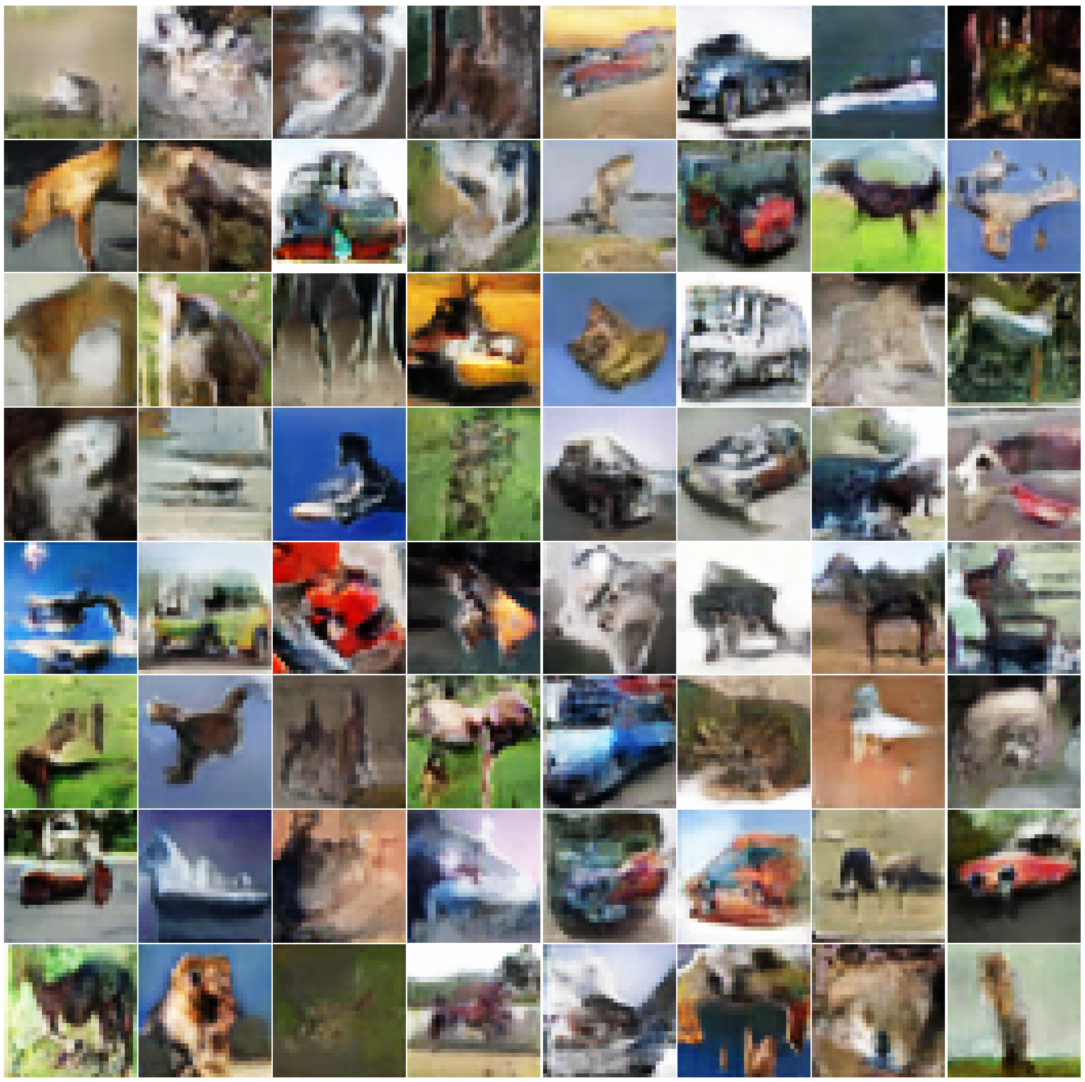


Figure 4.11: Samples generated from ECC model with $k = 16$ at iteration 100000

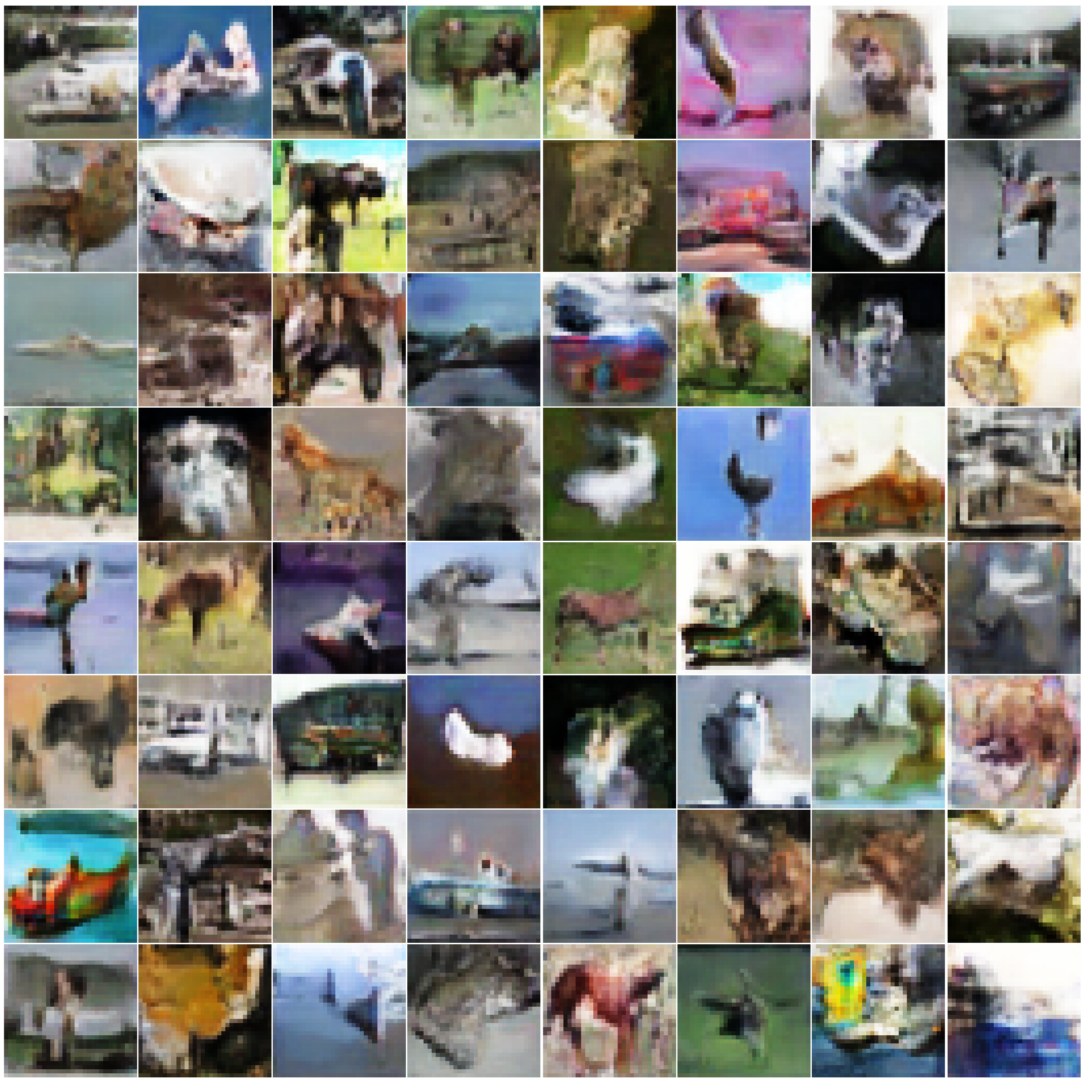


Figure 4.12: Samples generated from ECC model with $k = 32$ at iteration 100000

Chapter 5

Conclusion

In this project, the introduction of the graph convolutional layer in the generator architecture has not actually solved the lack of a global structure in generated images, and from the analysis done, it did not provide evidence of any improvement over the baseline. However, it cannot be claimed that this method might not be beneficial in the GAN framework in any case. On the contrary, it would be crucial to deepen the analysis. Since both the two methods of graph convolution tested have provided results similar to the baseline model in any neighborhood size condition, perhaps some common problems hinder the generators from learning non-local patterns. In light of these evidence, some hypotheses can be proposed. These results are perhaps mainly dependent on the used critic architecture, and possibly, contrary to the initial assumptions, the graph convolutional layer is strictly required also in the critic network to train the generator networks successfully. This could allow the critic to capture long-range dependencies between distant pixels, thus providing a better estimate of the Wasserstein metric and, consequently, push the generator towards learning structural patterns. Alternatively, it might be possible that the networks learn to ignore the graph convolution, hence the non-local receptive field, by exploiting only the local component in graph convolutional layer, thus behaving like a traditional CNN and providing a score close to the baseline model. Otherwise, it might also be possible that the graph convolutional layer requires to be refined in some detail. Tests could be done using a different similarity metric, a different aggregation function between the receptive fields, or even a different graph building method. However, GANs are probably not the most straightforward framework for testing these redefinitions on the graph convolutional layer isolating it from other possible sources of problems. For instance, it may be useful to detect non-local patterns for image classification tasks where a more interpretable loss can be advantageous in detecting anomalies during the graph convolutional network

training. In this manner, the different components of the graph convolutional layer can be incrementally integrated or changed while monitoring whether improvements in the considered loss occur.

Bibliography

[1] Ian J. Goodfellow. «NIPS 2016 Tutorial: Generative Adversarial Networks». In: *ArXiv* abs/1701.00160 (2016) (cit. on pp. ii, 19).

[2] Martín Arjovsky, Soumith Chintala, and Léon Bottou. «Wasserstein GAN». In: *ArXiv* abs/1701.07875 (2017) (cit. on pp. ii, 32).

[3] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. «Improved Training of Wasserstein GANs». In: *NIPS*. 2017 (cit. on pp. ii, 34, 51).

[4] Diego Valsesia, Giulia Fracastoro, and Enrico Magli. «Deep Graph-Convolutional Image Denoising». In: *ArXiv* abs/1907.08448 (2019) (cit. on pp. ii, 43, 44, 46, 50, 51, 53).

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 365–366, 430–432. ISBN: 0387310738 (cit. on p. 5).

[6] Christopher Bishop. «Latent Variable Models». In: *Learning in Graphical Models*. MIT Press, Jan. 1999, pp. 371–403. URL: `https://www.microsoft.com/en-us/research/publication/latent-variable-models/` (cit. on p. 5).

[7] Zhijian Ou. «A Review of Learning with Deep Generative Models from perspective of graphical modeling». In: *ArXiv* abs/1808.01630 (2018) (cit. on p. 10).

[8] Lawrence K Saul, Tommi Jaakkola, and Michael I Jordan. «Mean field theory for sigmoid belief networks». In: *Journal of artificial intelligence research* 4 (1996), pp. 61–76 (cit. on p. 11).

[9] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. «The" wake-sleep" algorithm for unsupervised neural networks». In: *Science* 268.5214 (1995), pp. 1158–1161 (cit. on p. 12).

[10] Yoshua Bengio and Samy Bengio. «Modeling high-dimensional discrete data with multi-layer neural networks». In: *Advances in Neural Information Processing Systems*. 2000, pp. 400–406 (cit. on p. 12).

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016, pp. 131–133, 168–169, 177–181, 198–201, 330–358, 563–571, 585–588, 694–695 (cit. on pp. 13, 15, 18).

[12] Diederik P. Kingma and Max Welling. «Auto-Encoding Variational Bayes». In: *CoRR* abs/1312.6114 (2013) (cit. on pp. 15, 17).

[13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. «Generative Adversarial Nets». In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., 2014, pp. 2672–2680. URL: `http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf` (cit. on pp. 15, 19, 20).

[14] Martín Arjovsky and Léon Bottou. «Towards Principled Methods for Training Generative Adversarial Networks». In: *ArXiv* abs/1701.04862 (2017) (cit. on p. 24).

[15] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen, and Xi Chen. «Improved Techniques for Training GANs». In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett. Curran Associates, Inc., 2016, pp. 2234–2242. URL: `http://papers.nips.cc/paper/6125-improved-techniques-for-training-gans.pdf` (cit. on p. 26).

[16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. «Rethinking the Inception Architecture for Computer Vision». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), pp. 2818–2826 (cit. on p. 26).

[17] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. «Striving for Simplicity: The All Convolutional Net». In: *CoRR* abs/1412.6806 (2014) (cit. on p. 29).

[18] Vincent Dumoulin and Francesco Visin. «A guide to convolution arithmetic for deep learning». In: *ArXiv* abs/1603.07285 (2016) (cit. on p. 30).

[19] Alec Radford, Luke Metz, and Soumith Chintala. «Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks». In: *CoRR* abs/1511.06434 (2015) (cit. on pp. 31, 49).

[20] C. Villani. *Optimal Transport: Old and New*. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 2008. ISBN: 9783540710509. URL: https://books.google.it/books?id=hV8o5R7%5C_5tkC (cit. on p. 33).

[21] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. «The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains». In: *IEEE Signal Processing Magazine* 30.3 (2013), pp. 83–98 (cit. on p. 38).

[22] D. I. Shuman, B. Ricaud, and P. Vandergheynst. «A windowed graph Fourier transform». In: *2012 IEEE Statistical Signal Processing Workshop (SSP)*. 2012, pp. 133–136 (cit. on p. 39).

[23] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. *Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering*. 2016. arXiv: 1606.09375 [cs.LG] (cit. on p. 40).

[24] Thomas N Kipf and Max Welling. «Semi-Supervised Classification with Graph Convolutional Networks». In: *arXiv preprint arXiv:1609.02907* (2016) (cit. on p. 40).

[25] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. *Spectral Networks and Locally Connected Networks on Graphs*. 2013. arXiv: 1312.6203 [cs.LG] (cit. on p. 40).

[26] Martin Simonovsky and Nikos Komodakis. «Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs». In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 29–38 (cit. on pp. 41–43, 47, 51).

[27] Xu Jia, Bert De Brabandere, Tinne Tuytelaars, and Luc Van Gool. «Dynamic Filter Networks». In: *NIPS*. 2016 (cit. on p. 41).

[28] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. «Dynamic Graph CNN for Learning on Point Clouds». In: *ACM Trans. Graph.* 38 (2019), 146:1–146:12 (cit. on pp. 46, 51, 53).

[29] Han Zhang, Ian J. Goodfellow, Dimitris N. Metaxas, and Augustus Odena. «Self-Attention Generative Adversarial Networks». In: *ArXiv* abs/1805.08318 (2019) (cit. on pp. 49, 50, 53).

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention is All you Need». In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 5998–6008. URL: http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf (cit. on p. 50).

[31] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. *Non-local Neural Networks*. 2017. arXiv: `1711.07971 [cs.CV]` (cit. on p. 50).