UNIVERSITÀ DEGLI STUDI DI TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

CONSTRAINING GENERATIVE ADVERSARIAL NETWORKS WITH SEMANTIC LOSS

Supervisor

Prof. Andrea Passerini

Student

Jacopo Gobbi

Co-supervisor

Dr. Paolo Morettin

Academic year 2018/2019

Thanks

I am thankful for the opportunity of working with Prof. Andrea Passerini, Dr. Paolo Morettin and Dr. Stefano Teso. Their guidance was priceless and the opportunity of being part of such a think tank has allowed me to grow as a computer scientist.
I am grateful to my parents, Renata and Fabrizio, for giving me many opportunities in life, among which we can find the chance of studying Computer Science after not having been a student for a while. Despite all the hardships, I would not trade Computer Science for anything else, and I would like my parents to know how thankful I am.
I also want to thank my family and relatives for their support, which was always positive and happily available, despite my shortcomings.
I wish my friends to know that "I don't know half of you half as well as I should like, and I like less than half of you half as well as you deserve", no matter how far they are.
Last but by no means least, I wish to express my appreciations to my girlfriend Kateryna, which I treasure. Aside from being a lovely, sweet, and patient partner in everyday life, you share my dreams and look after me with tender care.

Contents

Abstract 4

1 Introduction 5

2 Background 8

2.1 Machine Learning 8

2.2 Deep Learning 11

2.2.1 Measuring Performance 13

2.2.2 Gradient descent and the optimization of NNs 13

2.2.3 Convolutional Layers 15

2.2.4 Transposed convolution 17

2.2.5 Generative Adversarial Networks 17

2.3 Constraint Programming 20

2.4 Knowledge Compilation and Weighted Model Counting 21

2.4.1 Sentential Decision Diagrams 22

2.5 Semantic Loss 23

2.5.1 From Propositional Constraints to Semantic Loss: Pipeline 24

2.6 Related Work 24

3 CANS: Constrained Adversarial Networks 26

3.1 Motivation 26

3.2 The CAN architecture 26

3.3 Limits and workarounds 28

4 Experiments 30

4.1 Polygons and Parity Check 30

4.1.1 Dataset And Architecture 30

4.1.2 Applied Constraints 30

4.1.3 Results and Discussion 31

4.2 MNIST Still life 34

4.2.1 Applied Constraints 34

4.2.2 Dataset and Architecture 36

4.2.3 Results and Discussion 37

4.3 Molecule Generation 42

4.3.1 Chemistry Refresher 42

4.3.2 Metrics Of Interest 42

4.3.3 Architecture 43

4.3.4 Dataset 44

4.3.5 Applied Constraints 44

4.3.6 Results And Discussion 45

5 Conclusions 47

5.1 Limits 47

5.2 Future Work 47

Bibliography **49**

List of Acronyms

CA	Cellular Automata
CAN	Constrained Adversarial Network
CL	Convolutional Layer
COP	Constraint Optimization Problem
CSP	Constraint Satisfaction Problem
D	The discriminator network in a CAN or GAN
DL	Deep Learning
FC	Fully Connected
G	The generator network in a CAN or GAN
GAN	Generative Adversarial Network
GoL	Game Of Life
logP	Octanol-Water Partition Coefficient
ML	Machine Learning
QED	Quantitative Estimate of Druglikeness
SAS	Synthetic Accessibility Score
SDD	Sentential Decision Diagram
SL	Semantic Loss
SMILES	Simplified Molecular Input Line Entry system
SNCAN	Spectrally Normalized Constrained Adversarial Network
TCL	Transposed Convolutional Layer
WMC	Weighted Model Counting

Abstract

Generative adversarial networks (GANs) are seeing a great deal of usage and scrutiny, recent progress has made it so that the task of generating data characterized by being continuous or lowly structured, such as pictures of people faces, can be considered achievable with a good deal of success. On the other hand, the generation of structured objects or discrete samples that are subject to strict relationships, requirements and constraints between their variables is still something that can be considered a challenge. While these constraints should, more often than not, be easily inferable from training data, GANs still have issues in doing so, especially when said data is limited. To support GANs in this task we propose the use of a penalization factor, the semantic loss, which helps the generator network capture some aspects of training data that to us may seem obvious. The semantic loss is employed to better guide the generator by penalizing it whenever its output is a structure, or sample, that we consider invalid with respect to some arbitrary constraints. This approach makes use of knowledge compilation techniques and allows to directly instill propositional logic constraints into the generator network at training time, so that we incur in no increase of network parameters and no added computation time during inference, while being able to better approximate the original distribution. We provide an empirical analysis of different applications of this technique: the generation of polygons, the application of a subset of Convay's game of life rules to the MNIST dataset, and the sampling of molecules of pharmaceutical interest.

1 Introduction

During the process of creation, be it an essay or a wooden toy, we produce something that is characterized by a set of features that, from our perspective, allows us to match what we have done with what we wanted to achieve.
An adult will spontaneously avoid extravagantly long sentences knowing they have a negative impact, without the need for an external supervisor to explicitly tell him the maximum allowed length for each sentence, whereas to a child that just started learning a sentence could be whatever ends up with a certain character, no matter the length or if it actually makes sense.
Indeed, some features guide us along the way while going from a rough sketch to the final outcome, however, some of them are simply too subtle or maybe too ingrained in our ways of doing to the point that they happen to escape our attention and the attention of who has to judge our work.
Without trying to delve into philosophy, these features can be seen as constraints that we have to fulfill in order to consider our output correct, the measure of how much a constraint is complied with could be continuous, or it could be either strictly respected (true) or not (false).
Constraints optimization is a major topic of Computer Science, and for a good reason: it is often synonym with defining what we want first, and automatically obtaining it later, or at least searching for a good approximation to the solution we are looking for.
As for many - if not all - subsets of Computer Science, constraints optimization is everything but a solved problem. This is related to computability, while some constraints and their combination can be solved in polynomial time, a good chunk of them ends up being computationally unfeasible and NP-complete.
If optimizing constraints can be a challenge, we can say - with a certain degree of safety - that discovering what constraints or features characterize a dataset is probably not so simple as well. Indeed, intelligence tests such as the ones from MENSA often involve some forms of pattern recognition [1].
The moral of the story is that, if we are to use an algorithm to imitate the process of creation, we should take into account that a program lacks the context and information provided by a lifetime of learning, and that the dataset to which we apply such algorithm might lack the qualities or the quantities to allow our algorithm to generate something from the dataset, "but new", instead of having it copying and reproducing the input data.
The capacity to instill knowledge in an algorithm can be a priceless tool to obtain better results in terms of computation required and overall quality, if we have the capacity of giving our constraints a proper definition.
In machine learning, some algorithms or models are "discriminative", given an input, they must output some kind of classification with respect to it, like deciding if a photo represents a cat or a dog. Some others are called "generative", and are trained on a dataset in the hope that they learn to imitate its distribution, allowing for the sampling of new data points. We should highlight that there are also generative classifiers, considered generative due to the fact that they learn the data distribution in an explicit way.
Of course, different generative models exist, and the task of constrained generation is not an unexplored area, possible approaches would be Markov Chains [79], or multi-objective bio-inspired algorithms [83]. In the GAN architecture we have a generator network which has the task of defeating the discriminator network by synthesizing new data points which the discriminator network cannot distinguish from real data samples, oppositely, the discriminator job is to be able to separate the real from the fake.
GANs have been producing impressive results on different tasks, their effectiveness has been exemplified, for example, in works by NVIDIA, where GANs are trained to produce pictures of people faces while being able to control aspects of the generated person [47], in the semantic segmentation of an image [46], or in [14], which can now be considered the state of the art in terms of high fidelity artificial images.
There are, however, areas where results are significantly different, this is the case where the goal is to generate credible, structured and/or discrete objects, such as molecules, that are correct given some hard constraints, like chemical validity. One of the main reasons is that the network may struggle to learn what are and are not the hard constraints in the data, by simply looking at training examples.

Figure 1.1: High fidelity pictures generated by a machine learning system, BigGAN [14].

The goal of this work is to provide a novel framework for constrained generation by applying to GANs a loss that was previously used only for discriminative tasks, and with a limited scope, on simpler neural architectures. This loss, named **Semantic Loss**, allows the instillation of prior knowledge in the form of propositional logic by providing an arithmetic circuit (and from that, a penalty factor) that can be applied to the output of the generator network to direct it towards the generation of samples that better respect our constraints.
The circuit is obtained by exploiting existing knowledge compilation techniques to find a compact representation of the constraints.
The loss deriving from this circuit provides a direct gradient to the generator network, in contrast with other techniques such as reinforcement learning; the circuit essentially measures the probability mass given to valid and invalid objects by the generator.
We build onto different tools to provide a pipeline that, starting from propositional logic constraints, allows plugging in the aforementioned loss. The motivation behind this work is that GANs and constraints optimization can be complimentary of each other, GANs are able to capture aesthetic or inherently hard to define features that may otherwise be impossible to write down, whereas introducing a constraints optimization loss might amend the apparent lack of capability of inferring hard constraints from samples which seems to be present in GANs, at least when data is limited.
Moreover, training in this way allows the constraints to be learned directly by the generator, without the need for more parameters or networks. This means that sampling new (valid) objects after training is computationally identical to performing the usual forward pass on the generator network, given that the circuit is of no use after training.
This document is organized by first providing a background to machine learning, neural networks of both discriminative and generative type, formalizing constraints, knowledge bases and the SL. A chapter is then dedicated to formalizing the proposed framework where GANs and SL are combined, this is then followed by chapter 4, where we enumerate the different experiments and results, while chapter 5 serves as a conclusion for this work.

Contributions

Aside from simply applying the so called semantic loss to the GAN architecture and comparing results, we try to solve the scalability issues that are bundled with writing constraints in propositional logic and with the used knowledge compilation method that produces the resulting arithmetic circuit, which easily becomes hard to manage.
Moreover, we propose a method which makes use of the semantic loss to improve the uniqueness of the generated samples, novel in the fact that constrained generation usually tends to lead to less unique samples as a collateral effect of optimizing for valid ones.

We provide insights on the GAN tendencies when they are subject to different form of supervision, in our case, when one of those supervisions is related to Boolean constraints. Showcasing an empirical analysis in the generation of structures of different complexities that maintain a style consistent with the training data.

2 Background

This section serves as an introduction for all the different pieces needed to understand the whole picture. Starting from a broad view of machine learning, we later focus on explaining the various types of learning, to later delve into what is really of interest for us: deep learning.
Deep learning is presented from the ground up building from concepts of linear algebra and mathematical analysis, we cover the general training of a neural network, we then move onto generative adversarial networks, our architecture of choice.
Last but not least, we have a brief coverage of constraints, knowledge bases, sentential decision diagrams and the SL.

2.1 Machine Learning

We could say that the concept of machine learning (ML), or the broader concept of intelligence that is artificial, crafted by men (or gods), is a dream, or perhaps a challenge, that has been standing for a long time.
It can, at least, be traced back to ancient Greece through Greek mythology: the bronze giant Talos and Pandora can be seen as artificial life forms.
Despite the lack of a consensus on the definition of intelligence, ML and other areas of Computer Science related to artificial intelligence went from solving very simple problems to tackling complex tasks in a way that 30 years ago could be considered magical or miraculous.
In its first steps, this area of research solved problems almost programmatically, making use of formalities and mathematics to automate tasks that were difficult or tedious to humans, but reasonably simple to implement once the formula was known.
Naturally, it then started shifting towards more complex problems that often involve features or concepts that are harder to formally ground, ironically, these are what humans excel at, like drawing, or understanding what is happening (or what is going to happen) given a situation, speech or face recognition, etc. These tasks often require substantial amount of knowledge and background information that humans have "for free", and that instead needs to be encoded by the machine in some way.
Nowadays, we could say that ML is a mature field with a flourishing research area that is ever-growing, with a lot of interest being captured by those algorithms that are able to provide solution to problems that are hard to formalize, and which are instead what we could define as "intuitive".
A big selling point is related to automation, the aim is in fact not to program those algorithms (even if we properly could), but to train them so that the knowledge we provide is summarized into something that can function as it is.
The transition between formal and intuitive has going on for many years, with many wrong turns. As previously mentioned, contextual information is a key component of these less formal problems.
The knowledge base approach is one possible solution to this, it usually involved human operators manually adding formal statements to a database in order to build a base of (formal) knowledge through which a program could later infer and reason.
Sadly, this line of work based on hard-coded knowledge has not been much successful, for different reasons: the computation required to infer, the difficulty and length of correctly encoding a lot of information in a formal way, which also poses the problem of identifying what and what not constitutes important information.
From their difficulties we stumble on a possible definition of ML, [37] defines ML as the ability of machines "to acquire their own knowledge, by extracting patterns from raw data", and identify the lack of this key element as the cause of the failure of the aforementioned systems.
Given the theme of this thesis, in which we propose a framework to instill external knowledge in a ML system to support its learning, we propose a less extreme definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E" [62].

Most of the time, what we call experience or knowledge is obtained by the system through data. ML systems usually receive some quantity of data, the dataset, and iterate over it a certain number of times until some arbitrary condition is met, so that the learning is stopped.
Data can essentially be any digitized representation of something, often vectors or matrices of some shape. We could, for example, decide to represent a human by its weight, height and age; each one of these values would be a feature of our representation.
Deciding on a representation is itself an arbitrary choice: we could in fact decide to represent a person by adding 1 more feature, its BMI, even if it can easily be obtained from what we already feed to the system. Despite this, with most machine learning algorithms our choice of representation and feature engineering end up having substantial implication on the outcome.
Even though the combination of feature engineering and simple ML algorithms is surprisingly effective for a lot of problems, it has the limitation that we need to know or to guess what features needs to be elaborated in order to improve performance. This can be easily explained by posing the following question: given a photograph and having to decide if it contains a cat or not, what kind of features should we provide to the algorithm in order to have it learn properly? We know that a cat's head has a certain shape, we know that they have fur, a tail, and their general shape; but how do we write that as feature that we can compute starting from the pixel values that form the image? We should also take into account shadows, foreground objects partially covering the cat, and the fact that we don't want picture of tigers being miss-classified.
What we would like to do, and what actually happens in most state of the art systems, is to have the algorithm learn a proper representation of the data before deciding on how to weight those features with respect to the final decision. This is what is called representation learning, and it's mostly what is present in systems that are made to solve problems that we previously called "intuitive". These algorithms allow for minimal human intervention while, usually, providing better results, but at the cost of having a computational overhead, that is justified by the ability of learning new features.
More training time is not the only downside, non-trivial applications have data samples that are noisy, and have a high degree of variation and complex interactions between their variables, all these factors lead to the need of a quantity of data that is higher compared to algorithms that limit themselves to simply weighting the features of a representation.
Given this, it should be of no surprise that nowadays a lot of energies are spent into collecting and cleaning data, which quantity can be more important than the specific algorithm of choice among a plethora of "good" algorithms.
Once we have data, the training of a machine learning algorithm is usually some kind of loop that iterates over each sample of the dataset, where the system has to decide on its action, or judgement, and based on the correctness of it, it receives a feedback that will make the algorithm adjust its internal functioning in order to be more correct in its answers.
Once training is completed, we now have a trained model (this particular trained instance of algorithm) that we can use to perform the task we intended it for; this is what is usually called inference time or inference.

After this first rather wordy introduction to machine learning, we can go back to the **Task**, **Experience**, **Performance** loop defined by [62] to better ground each single point.
With task we mean what we actually want our algorithm, once trained, to perform. We can use different algorithms for the same task, some more suited than others. Technically speaking, the term task ends up matching with how we want our ML system to process a data sample. A data sample is usually a vector, $x \in R$, each dimension of x being a feature of our data point, we expect the system to process this vector and return us some value, represented by a single number, or a vector itself.
Most **tasks** can roughly fit among the following definitions:

- **Classification**: given a sample, the program should predict which category it is part of, meaning that the trained algorithm should result in a function $f : R^n \mapsto \{1, ..., k\}$, with n being the number of features of a sample and k the possible categories. Typical real world applications are object detection, face recognition, etc.

- **Regression**, this task is somewhat similar to classification, given an input, the algorithm has to output a (correct) numerical value, by learning a function $f : R^n \mapsto R$. Possible uses are predicting the value of financial stocks given other metrics, the peak temperature during a certain day given the temperatures, humidity and wind speed of a certain amount of previous days, etc.

- Learning a function $f : R^n \mapsto R^k$ is what it is usually called **Structured output**, if the output is not a vector it could be any another structure which contains more than 1 value; the focal point is the presence of strong relationships among the variables of said structure. The output could be of dimensions equal to the input if the task is some sort of transformation or variable-wise annotation, but it is not necessary. Possible use cases are: image segmentation (assigning different parts or pixels of an image to a set of labels, like foreground, background, etc.), generating a sorted group of categories to present alternatives to a user ordered by what we predict its preferences would be, or molecules in the form of matrices to represent their atoms and bonds, and so on.
- **Synthesis and sampling**, which will be the main interest of this thesis, is a task where we expect an algorithm to learn the ability to generate novel samples starting from a given dataset. These samples should of course be similar enough to the original ones to be considered as part of the dataset, but new. This can be useful for different reasons: new samples might take a lot of time to create, it could be hard, expensive or simply boring. This can find use in texture synthesis [34], the generation of faces [47], or even videogames [85].
- **Density estimation or probability mass function estimation** concerns the task of learning $p : R^n \mapsto R$, with p being the probability density (or mass) function over the samples space. This implies explicitly learning the data distribution, which is often a computationally intractable problem. Learning the probability function over the data is something that already happens implicitly in most of the aforementioned tasks, the difference here is that we want an explicit and exactly mapping.

The training phase is where the **experience** concept comes into play, it constitutes the heart of machine learning algorithms and it is ground for thriving research in many different directions. Learning is divided among four -rather blurry - main categories.

- In **supervised learning** the experience is constituted by the algorithm observing a sample which is made by a set of features and a **label**/target, which is the correct answer we want the algorithm to give. The provision of an expected target is the reason this is considered supervised learning. If we denote the set of features as x and the label y, we want our algorithm to learn the probability distribution $p(y|x)$, y could be a category, like type of animals, or a raw value, like the peak temperature of tomorrow.

- The correct expected answer is instead not provided in **unsupervised learning**, the algorithm is simply fed data in the hope that it will learn useful properties, representations, relationships among variables or at the dataset level. One possible application is, for example, the partitioning of the dataset into different clusters of homogeneous samples. Another well known application is finding a better representation for data, like for example reducing the number of features needed to represent the same data point [33], or finding a real valued vector representation of otherwise categorical values, like word embeddings [60].
- **Semi-supervised learning** acts as a bridge between the two approaches, if the two aforementioned forms of learning are no formal definitions, this is even more blurry. Semi-supervised learning is whatever training involves both the presence of a target that the algorithm should match and the lack of it, like a classifier trained on both labeled and unlabeled data, being rewarded for guessing the right target for the former, and for giving a strong score only to one of the possible label categories on the latter, as happens in [88]. **Multi-instance learning** is instead characterized by the fact that labeling happens at a collection level, individual samples are not labeled, whereas the batch is either marked as containing or not containing some class; it can be considered a form of semi-supervision.
- The feedback loop does not have to be limited to guessing a target and receiving a right/wrong response, an example of that is **reinforcement learning**. In this kind of learning, the algorithm is now considered an agent which interacts with an environment in what we call episodes, essentially describing an iterative loop of the agent receiving the current state information as input, deciding on an action to take, changing the environment because of that action and thus receiving some kind of feedback, which in this context is defined as reward.

Reinforcement learning is perhaps the most peculiar among the previously cited learning methods, and it is a world of its own, the consensus on the best resource covering it is [81].

Lastly, we cover what we mean with **performance**. We need a way to measure how good the algorithm is doing during its learning, and through that a way to provide feedback to it on how to change internally.
The performance measure is related to the task at hand, for classification we could, for example, measure the proportion of correct predictions. For regression, we could define performance as the difference between the output value and the target (the lower the better). The choice of a performance measure implies the goodness of the end result of the training phase, and it is not always trivial, especially for tasks such as data sampling; such choice is essentially part of the architectural design of an algorithm and should take into consideration what we expect the system to do.
Should we have a medical application that has to predict cancer have a high false positive rate or trade it off for more false negatives?
As a last note, we generally do not want the model to have a perfect performance on training data, in contrast, such a thing should be seen with suspicion. What we are interested in is the algorithm capacity of performing correctly on samples it has never seen, essentially how it is able to generalize what it has learned over the sample space. On the contrary, over-fitting is the case where the algorithm has been trained too strongly on the training set, resulting in a model that fails to generalize well with new data, which is something that we need to avoid.

2.2 Deep Learning

Deep learning (DL) is a subset of machine learning and representation learning, it makes use of so called neural networks (NNs) to build machine learning systems, so called because of their initial inspiration from an approximated structure of the brain.
DL defines the state of the art for many tasks and its use is expanding for good reasons, it is fairly easy to apply and it is end to end, in the sense that we input something to the network and we get a result, with no human intervention or programming in the middle. This is also a drawback, NNs have in fact been criticized for being black boxes that are hard to debug, and understanding the motivation behind a decision is more often than not downright impossible.
The term "deep" refers to the fact NNs are made of multiple layers, having a flow of information that goes from the first, once input is fed, to the last, traversing all the network in all its depth. Each layer of a NN learns a new representation from the presentation it receives from the previous layer, until we get to a final, usually simpler, representation that is of interest for solving the task.
Each layer is comprised by some parameters θ_i, and the whole parameters θ is what we need to set during the network training, typically, each layer is made of a matrix of parameters that is multiplied with the previous representation (or the input), a vector, to obtain the representation of the current layer, the number of features of each layer determines the size of said layer, the width of the model and the total number of parameters.
Each layer can thus be seen as a mapping from a vector to a vector, or as being composed of n **units** (where n is the size of the layer), which are the neurons in the brain analogy, each of them taking in input the previous units value and deciding on its own.
As we have previously mentioned, representation learning is data hungry and NNs are no exception, on the contrary, they are probably the most data hungry algorithm that we could use. Despite that, they have a widespread use thanks to their raw performance once we have enough data.
The goal of training is to find the correct parameters so that we end up with a mapping $y = f(x;\theta)$ where x is the input and y is output of the network, which should be correct with respect to its task. The mapping is basically the chaining of different functions, each represented by a layer $f_i(x)$, resulting in the composition $f_n(...(f_2(f_1(x))))$.
Up to now we are essentially considering NNs as a chain of matrix products mapping from an input to an output space, this would limit us to a model that has only linear capabilities, and thus would only be good enough for tasks where the output space of interest is linearly separable, e.g. distinguishing between two classes having a representation such that their samples can be partitioned by a hyper-plane. For regression, we would instead be limited to approximating linear functions.
To overcome this, we can apply a non-linear function to the output of each layer, so that layer i, $f_i(x)$, becomes $g_i(x) = \phi(f_i(x))$, ϕ is usually a non-parametric function which is the same over all layers but last, but this is no strict rule.

An exception is made for the output layer, given that the output is related to the task at hand, if such a function is applied, it is usually different from what is applied to the other layers; for classification we could for example apply a softmax function to smooth the prediction values for all categories so that the final output is a vector of probabilities over the possible labels.
Interestingly, it has been observed that neural networks, at least in common tasks such as classification, tend to transform the input space by simplifying it so that the last representation before the output layer has samples belonging to different categories that are linearly separable, so that the final layer can simply learn the correct linear function to do so.
Technically speaking, functions applied to the output of a layer are named activation functions, during the years many have been proposed, which one works best is usually related to the specific task, dataset, and architecture. Some example of activation functions are ReLU, tanh, and sigmoid, among many others [66].
One last caveat: to have each layer properly model the needed linear transformation applied to its input we should also need a bias to allow each neuron to shift through the activation function, or equally speaking, to have a prior value on its activation. This can be explained by the fact that without a bias we would have every neuron being a hyper-plane always passing through the origin, which won't allow to properly model all functions.
Each layer then goes from being $\phi(W^T x)$ to being $\phi(W^T x + b)$, with b being a vector of biases, one for each neuron, the parameters in b need to be learned just as the ones belonging to W. Structuring each layer in this way might seem overall simple, stacking such layers is however quite powerful. It can be proven that neural networks are universal function approximators as long as they have more than 1 layer [22], empirically, they are the go-to choice for plenty of tasks.
During training, we provide feedback to the network based on the match between its final output and the label, while we do not interact with internal "hidden" layers, all parameters are then adjusted according to some training criterion, this repeats until an arbitrary stop condition in the training.
In the following sections we better formalize how to measure the performance of a NN, the optimization algorithm behind the success of NNs, and explain the basic end to end algorithm to train a NN.

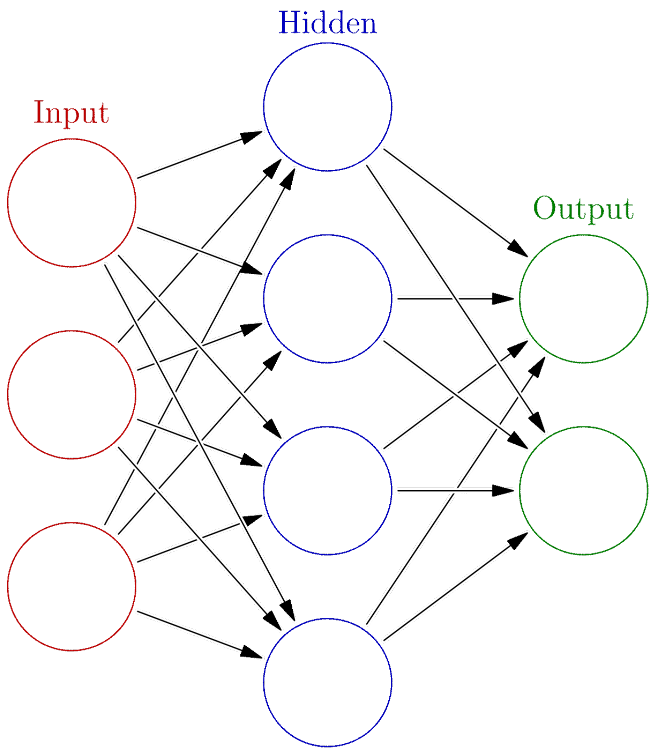

Figure 2.1: Representation of a neural network, image from [4].

2.2.1 Measuring Performance

One of the key elements in machine learning is how we measure the correctness of our algorithm so that we can provide a feedback to correct its internal functioning, NNs are no exceptions.
Luckily, the cost functions applied to neural networks are somewhat identical to the ones applied to other (parametric) models. Most NNs are trained using maximum likelihood, by using the negative log likelihood between data samples and the model prediction:

$$\boldsymbol{J}(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{y}|\boldsymbol{x}), \tag{2.1}$$

from this cost function we can derive many others, which makes using NNs quite versatile given that we can use the same loss(es) for many different architectures.
Training for a classification task will usually lead to using one of the following losses:

- cross-entropy, an equivalent interpretation of negative log likelihood
- hinge loss, $\max(0, 1 - \hat{y} * y)$, with $\hat{y}$ being the predicted probability of the correct class y

While regression will typically use one of those:

- mean absolute error, MAE, $|\hat{y} - y|$
- mean squared error, MSE, $(\hat{y} - y)^2$
- Huber, $L_\delta = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & if\ |(y-\hat{y})| < \delta \\ \delta(y-\hat{y}) - \frac{1}{2}\delta & otherwise \end{cases}$,
  Which is a solution to the fact that MSE tends to be dominated by outliers, while mostly retaining the quality of the MSE to be mean-unbiased and of the MAE, to be median-unbiased.

For regression there is obviously no restriction on the range of values $\hat{y}$ can assume, i.e. it does not represent a probability. Many losses and variants exist, and while for regression and classification we have somewhat of a consensus, for other, more complex tasks research is still moving forward, we leave the analysis of two of those losses for a dedicated section, 2.2.5.

2.2.2 Gradient descent and the optimization of NNs

Gradient descent is an optimization algorithm which we can use to adjust the parameters of a function by updating them in the direction given by negative gradient of some cost function with respect to said parameters.
Gradient descent can be used to optimize pretty much everything that involves parameters and a numerical result as long as the gradient can be computed, i.e. if we do not introduce operations for which the gradient is not defined, so it is not limited to neural networks.
The parameters are updated by a small portion of the gradient, called the learning rate; setting it high means moving faster in the error surface of our function, at the risk of moving past good local optima or not being able to converge, so a lower learning rate is preferred, even if it slows down learning, it has more guarantees on reaching a local minimum. There exist different algorithms and approaches to schedule and manage the learning rate that also take in consideration past gradient values, which we will not cover.
The algorithm is reasonably simple as long as we have an efficient way to compute the gradient with respect to our parameters θ. For simple linear models this is trivial, while for NNs, for a multitude of reasons among which we can include non-linear activation functions and the layered nature of NNs, it is slightly more complex, it can however be computed efficiently nevertheless.
Another problem introduced by the non-linearity of NNs is that the cost functions become non convex, thus their parameters cannot be computed exactly by analytical means and their training loses any global convergence guarantees that simpler models have.
Luckily enough, the gradient of the loss function with respect to θ can be efficiently computed for neural networks. Note that with output we can mean the judgement/classification of the network, but it can also mean the scalar value that we obtain by applying our loss function to the output, as it happens here. After all, by applying the cost function we are simply doing function composition, in the form of $J(f(x, \theta))$.
As we have said, the flow of information usually goes from input through (hidden) layers, then output, this is

what we call the forward pass of a network, also called forward propagation. To compute the gradient of J (our loss) with respect to all the parameters θ, we need to invert the flow of information, starting from the actual scalar loss, the output, we must go through each single layer and determine its gradient, until we get to the very first layer. This is known as the back-propagation algorithm [73], and while it is mostly known for being applied to NNs, it can be used to compute the gradient of any parametric function as long as all operations applied have a defined gradient.
Backprop works by applying the chain rule in order to traverse the composition of functions $J(L^n(L^n - 1...(L^2(L^1(x)))))$ (L stands for layer) and obtaining the derivative of J with respect to each layer of parameters, starting from the last. The chain rule does in fact state that:

$$\frac{dj}{dx} = \frac{dj}{dy}\frac{dy}{dx} \tag{2.2}$$

Without delving into the complete back-propagation for a deep network, we simply demonstrate its application on a network with a hidden layer and no biases. With L_i being the pre-activation output of a layer in the form of $L_i = W_i^t I_i$, where I_i is the input of said layer.

$$Loss(\theta) = J(\phi_o(L_o(\phi_h(L_h)))) = J(\phi_o(W_o^T \phi_h(W_h^T X))) \tag{2.3}$$

We would have that the derivative of J w.r.t W_o would be:

$$\frac{dJ}{dW_o} = \frac{dJ}{d\phi_o} \cdot \frac{d\phi_o}{dL_o} \cdot \frac{dL_o}{dW_o} = \frac{dJ}{d\phi_o} \cdot \frac{d\phi_o}{dL_o} \cdot \phi_h(W_h^T X) \tag{2.4}$$

While the derivative of J w.r.t would W_h would instead become:

$$\frac{dJ}{dW_h} = \frac{dJ}{d\phi_o} \cdot \frac{d\phi_o}{dL_o} \cdot \frac{dL_o}{d\phi_h} \cdot \frac{d\phi_h}{dL_h} \cdot \frac{dL_h}{dW_h} = \frac{dJ}{d\phi_o} \cdot \frac{d\phi_o}{dL_o} \cdot W_o \cdot \frac{d\phi_h}{dL_h} \cdot X \tag{2.5}$$

The recursive behaviour makes it so that most elements are, or at least should, be recomputed, we can instead simply store intermediary values and be efficient. The above equations can be rewritten so that during our backward pass we only need to pass the previous layer gradient to the next layer we need to compute the gradient of, ending up with the following rules:

$$\begin{aligned} \frac{dJ}{dL_o} &= \frac{dJ}{d\phi_o}\frac{d\phi_o}{dL_o} \\ \frac{dJ}{dL_h} &= \frac{dJ}{dL_o}\frac{dL_o}{d\phi_h}\frac{d\phi_h}{dL_h} = \frac{dJ}{dL_o}W_o\frac{d\phi_h}{dL_h} \\ \frac{dJ}{dW_i} &= \frac{dJ}{dL_i}\frac{dL_i}{dW_i} = \frac{dJ}{dL_i} \cdot \phi_{i-1} \end{aligned} \tag{2.6}$$

Now that we can measure the performance of our model and have the gradients of such performance with respect to θ at our disposal, we can apply gradient descent to optimize the NN parameters.

Algorithm 1 Stochastic Gradient Descent

```
Require: starting parameters $\theta_0$, learning rate $\eta$
1: repeat
2:     permute dataset
3:     for all training examples $(\boldsymbol{x}, \boldsymbol{y})$ do
4:         $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta, x, y)$
5: until convergence
```

SGD is one of the most simple training algorithms, as we have said different algorithms of scheduling of the learning rate and ways of updating the parameters exist; similarly, different techniques of initializing θ have been devised, which we won't cover, it suffices to say that all these strategies tend to initialize the network parameters with low random values.
A rather simple improvement to SGD is the computation of gradient in a batch wise manner, which tends to offer

a much more precise gradient in contrast with using one sample at a time. In practise, the strict definition of SGD is rarely used and using a batch of size 16, 32, 64 is common, this approach is called **mini-batch gradient descent**.

Algorithm 2 Mini Batch Gradient Descent

```
Require: starting parameters $\theta_0$, learning rate $\eta$, batch size $b$
1: repeat
2:     sample a batch of $b$ samples from the training set, $\{(x_1, y_i), ...(x_b, y_b)\}$
3:     $\theta \leftarrow \theta - \eta\frac{1}{b}\nabla_\theta \sum_i J(\theta, x_i, y_i)$
4: until convergence
```

2.2.3 Convolutional Layers

The need to introduce convolutional layers (CLs) stems from their use in this thesis, covering the whole idea would be out of the scope, so we only introduce a few key concepts.
Beyond normal, “fully connected” (FC) layers, where the entire representation in the form of a vector is multiplied to a matrix of parameters, there exist other, more specialized layers for particular types of data or tasks. One of those is the convolutional layer (CL) [54], networks making use of such a layer are called convolutional NNs, or CNNs, we must however note that nowadays most NNs making use of convolution also tend to eventually have some FC layers to elaborate on the presentation resulting from a CL.
The point of CLs is exploiting the fact that for some kind of grid-like data we know a priori of the existence of local relationships between the variables of said grid. Think of photos, or a time series from a stock market, where we know that pixels, features, or values that are near each other in the grid are most likely related to each other in the form of some dependence, co-variance etc.
CLs can provide efficiency by finding and synthesizing local, low level features into higher level concepts that we then pass to higher level layers. The upper layers of a NN to classify pictures do not need to know exact pixel values, but are probably more interested in the presence, or lack of, of certain features in certain positions of the image, such as edges, or shapes made of those edges. We could do the same thing with a FC layer, but a CL can leverage important factors if we know a priori that there exist a locality in the interaction between variables, these factors are: sparse interactions and parameter sharing.
A single unit of a FC layer will interact with all units of the previous layer, for each of these interactions we end up having a separate parameter. With **sparse interaction** we mean the fact that a CL is instead built by having each unit only interact with a subset of the input features. We would go, given an input of m features, from having a FC layer of mn parameters to kn, with n being the number of units and k the number of connections each unit will have. Again, thanks to locality and the small (compared to the whole sample) number of features, we often end up having a k which is way lower than m, leading to savings in the memory and computation.
Note that as we stack CLs on top of each other, units in the deeper layers will be indirectly connected to more and more features from the lower levels, because of the hierarchical contribution of each unit. **Parameter sharing** is given by the fact that, to compute the feature map of a CL, we slide the kernel of the layer as a window over the grid, essentially reusing it (and thus reusing the parameters) multiple times to get local features over multiple localities of the input. This makes sense because the same features (such as the ear of a cat) could appear in different locations of the input.
Parameter sharing combined with sparse interactions means that we are using a minimal amount of weights, k, with the added bonus that these parameters receive a more stable feedback through the gradient given that each of them is applied multiple times by using them on different portions of the input.
Up to know we have reasoned about a grid over which we slide our kernel, or filter, to obtain a new feature map; before concluding this section we can pile up three simple concepts.
First, we can imagine a mn grid, like a gray scale picture, if they picture was in another format, it might have 3 **channels**, red, green and blue, resulting in a $3mn$ grid. We can extend the concept of a kernel of k parameters to a kernel made of $3k$ parameters, resulting in the fact that the output of all units will receive a contribution from all channels. This still makes sense given that the grid like and local nature of our input holds even if we have more than 1 channel.
The second idea is somewhat related, it is reasonable to expect that for non-trivial tasks we would need to detect different kind of features, such as horizontal edges, vertical edges, etc. To do that, we can simply make use of

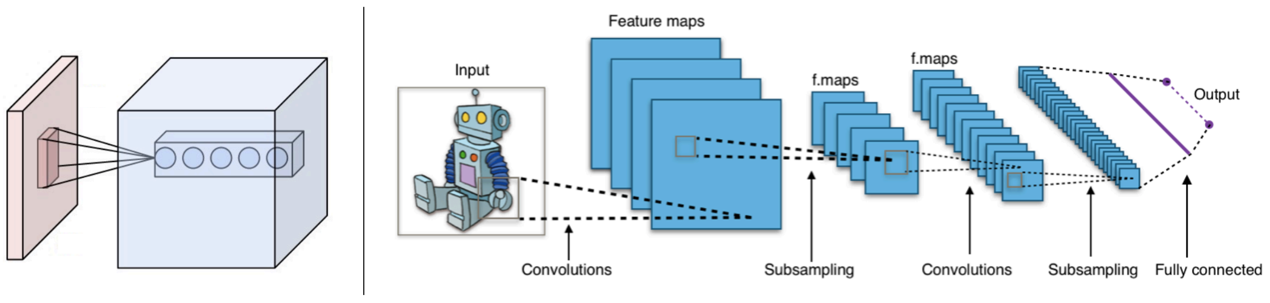

Figure 2.2: Left: units of a convolutional layer with an input feature map in red. Right: an example of a CNN architecture [5].

different kernels, essentially having a convolution that has o output channels instead of a single one, resulting in ko parameters. Combining the two ideas we can have h input channels from the previous layer (or the input), o output channels for the current CL, resulting in a total hko parameters for a single CL.
Last but not least, there are different ways in which we could slide our filter over the input data, in a 2d context, having a kernel made of kk parameters (imagine a small square moving over a greater one), we could move our kernel by 1 step at a time, first moving it over the first dimension, then wrapping around and repeating this but 1 level lower over the other dimension, not much differently from 2 nested for cycles. The step size, or **stride**, is a parameter that is decided a priori before training, and based on that, the number of output features would obviously change, the higher stride the lower the output features.
CLs are often paired with other techniques that can reduce the output map further, which we don't delve into, the purpose of the last paragraph was fixing the concept of the reduction in size of the feature map, so that we can introduce the opposite concept, the up-sampling of the feature map, which is a key step in many generative neural networks.

Convolutional Graph Layers

The theory behind this type of layer can be quite convoluted and is out of the scope of this work, however, such layers are used in chapter 4.3, so it is important to at least denote the existence of said layers.
The key concepts of convolutional layers can virtually be translated for any representation where the same meaningful patterns can be present in different locations, so that can they can be detected by the same filters and later aggregated hierarchically.
While CLs are originally defined for euclidean spaces, they can be adapted to work on graphs [15, 30, 49]. This has the advantage to make a discriminator network making use of such a layer be invariant to node ordering, which might not be the case if, for example, we work on sequential structured objects.
The most important fact is that after an arbitrary number of graph convolution layers, where information is propagated through the graph in order to obtain an embedding for each node, it is possible to obtain a final encoding of the whole graph, said encoding is in the form of a vector that can simply be passed to FC layers for further processing.
The particular implementation used in section 4.3 in order to obtain a node embedding is the one from [17, 76], whereas [55] is used to combine all the **node embeddings** into a final graph level encoding.
For each node v_i in the graph, the layer propagates values in the following manner:

$$h_i^{l+1} = \tanh(f_s^l(h_i^l, x_i) + \frac{1}{|N_i|} \sum_{j=1}^{N} \sum_{y=1}^{|X_{edge}|} A_{ijy} f_y^l(h_j^l, x_j)) \tag{2.7}$$

With h_i^l being the embedding of node v_i obtained by the previous layer, f_s^l is a linear transformation that makes it so that the new representation is informed about the previous value it assumed. For each node v_i, x_i stands for its annotation/label vector. f_y^l are instead affine transformations (one for each edge label, for each layer). $1/N_i$ acts as a normalization to maintain values independent of the number of neighbours of a node, and A stands for the labeled transition matrix. X_{edge} is the set of possible edge labels. The first graph convolution layer can take as input any feature, e.g. a one-hot vector (that should obviously be different for each node), or a transformation

of said vector.
Finally, the **graph embedding** h_g is obtained by combining the different node embeddings:

$$h_g = \tanh(\sum_{i=1}^{N} \sigma(i(h_i, x_i)) \odot \tanh(j(h_i, x_i))) \tag{2.8}$$

Again, for each node v_i, x_i stands for its annotation/label vector. Both node and label are fed to two functions i,j, that can be implemented by virtually any NN, although they are usually very simple FC layers; σ is the sigmoid function, and $\odot$ represents an element wise multiplication.

2.2.4 Transposed convolution

If CLs can help in compressing a grid-like representation into a one which is more compact, we might need to do exactly the opposite. This might be the case in generative adversarial networks, which we will introduce later. The key idea is that starting from a rather limited size representation, e.g. a simple noise vector, we want our NN to output a full data sample, which might be a high resolution photograph. As with CLs, we could - theoretically - do this using FC layers of increasing size, simply reshaping the final output into the expected shape, but again, this would be extremely inefficient, and harder to train.
For this reason, we can use what is called transposed convolution (TCL) [29], it still makes use of kernels, and still exploit locality, sparse connections and parameter sharing. In a CL a kernel with k parameters would, iteratively, multiply its parameters with k values from the input representation to compute a single output unit. A TCL layer can be roughly explained by doing the opposite, having a kernel of k values, we go over each single input feature, the feature is multiplied with all parameters in the kernel, obtaining a matrix of the same shape of the kernel, where each entry m_i is the product of the entry k_i multiplied by the aforementioned feature. We can think of the "upscaled" output map as initially having all units set to 0, each time we compute a matrix from an input feature, given the position of the feature we will know where to position our matrix over the output map (the matrix will, of course, only cover a small window of the output), summing its contribution (the products) to the values of the output units.
The size of the upscaled feature map will change accordingly with the kernel and step size. Again, as for the CL layers, we could reason about input and **output channels**, which we avoid to not be too cumbersome, believing the main idea should suffice in understanding the rest of the work.

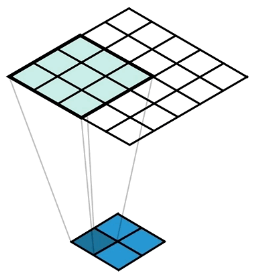

Figure 2.3: Example of transposed convolution, input in blue and output in cyan, generated from [84].

2.2.5 Generative Adversarial Networks

As previously mentioned, ML models are not limited to transforming input data to some kind of answer, new representation, or interesting feature. They can be used to imitate a distribution in order to later use them to generate new samples, once training is done. DL can be considered the state of the art for a generous amount of generative tasks, such as face generation [47] or learning the drawing style that characterizes a cartoon [20]. The main engine behind this success is a category of NNs known as generative adversarial networks (GANs) [38], while the name might sound intimidating, breaking it down helps in clarifying the whole matter: generative stands of course for the fact this model is used to generate new samples given a data distribution, networks (note the s) indicates that we have more than one NN that is being trained (usually 2), adversarial explains the

relationship between those two networks, they are in fact competing with each other over some results, i.e. the loss of one will opposingly depend on the other.
The networks in GANs can make use of all kind of layers and techniques of regular NNs, although some heuristic adjustments are often needed to get better results [69]. Parameter optimization is unchanged, making use of back-propagation to adjust parameters with respect to some loss.
The architecture is developed starting from a game theoretic scenario where the generator network G competes with the discriminator network D. G outputs samples by transforming some input noise z, acting as a function $g(z, \theta_g)$, in contrast, D needs to judge both real (from the dataset) and fake (from G) data, D is then a function $d(x, \theta_d)$ which has to give a higher score to samples it believes are real, and lower to fake ones, the exact form depending on the loss function at work, while G attempts to produce samples that are as believable as possible to fool D.
At the fundamental level, we could devise the learning process of GANs as a zero-sum game regulated by a value function $v(\theta_g, \theta_d)$ where G sees a reward equal to $-v$, while D receives v.
Although convergence in some cases can be proven [38], so that G eventually learns to output samples that cannot be differentiated from the original distribution, the general case is that there is no real guarantee of reaching an equilibrium, simple cost functions can be devised to show that the two players will "orbit" around some values, or increase/decrease forever their parameters, etc [37].
Training GANs is much more difficult than training classical NNs, and the stabilization of their training is still an open problem, the same could be said for their cost functions, which by now there exist many of them, (e.g. [11, 42, 63], but without a real winner [57].
The lack of a good way of scoring performance is especially aggravating, GANs could do very poorly according to some metrics, e.g. by assigning 0 probability to samples in the test set, while producing results that a human observer judges to capture the essence of the generative task [37]. Because of this, there exist different metrics that try to heuristically capture the goodness of the generation process, such as inception score [74] and FID score [43], even this thesis makes use of some specialized metrics (later presented) to measure results in a more manageable way

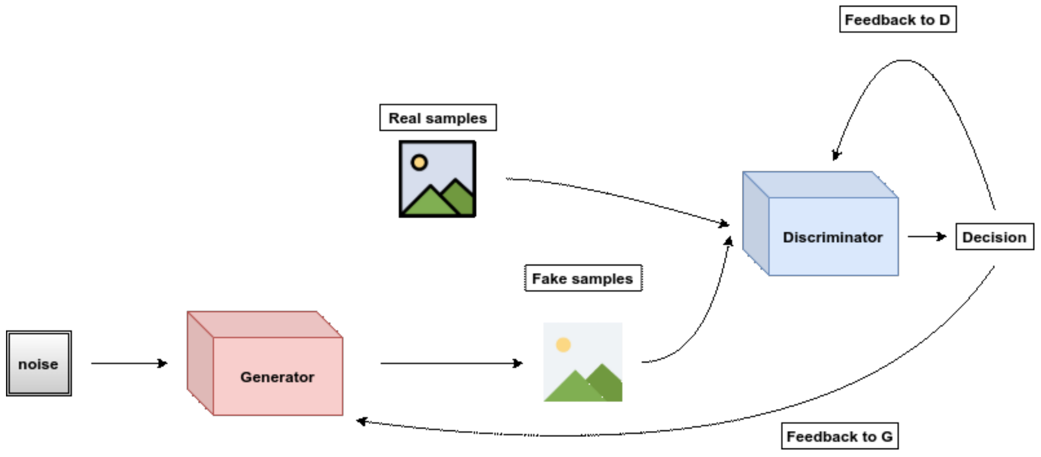

Figure 2.4: Representation of a generative adversarial network.

GAN losses

We present the two losses that have been used in the different experiments of this paper. Their choice was heuristically based on the quality of results over a grid of hyper-parameters. Mapping experiments to their respective loss/hyper-parameters is reserved for a later section.

The first loss we have used in our experiments is a modified version of the original one found in algorithm 3, proposed by the same authors. Only the generator loss is modified, in a way which "causes the derivative of the generator's cost function with respect to the discriminator's logits to remain large even in the situation when the discriminator confidently rejects all generator samples". Note that both in the original and in the modified loss the output of D is a probability, e.g. a sigmoid is applied.

Algorithm 3 The original GAN algorithm, [38]. Mini batch stochastic gradient descent training of generative adversarial nets k is a hyper-parameter representing the number of training step to apply to the discriminator for each iteration.

```
1: for number of iterations do
2:     for k steps do
3:         Sample from the prior noise distribution $p(z)$ a batch of $b$ samples, $\{z_1...z_b\}$
4:         Sample from the real distribution $p_{data}(x)$ a batch of $b$ samples, $\{x_1...x_b\}$
5:         update $D$ by ascending its stochastic gradient
6:             $\nabla_{\theta_d} \frac{1}{b} \sum_{i=1}^{b} [\log D(x_i) + \log(1 - D(G(z_i)))]$
7:     Sample from the prior noise distribution $p(z)$ a batch of $b$ samples, $\{z_1...z_b\}$
8:     update $G$ by descending its stochastic gradient
9:         $\nabla_{\theta_g} \frac{1}{b} \sum_{i=1}^{b} [\log(1 - D(G(z_i)))]$
```

$$L_g = -\log(D(G(z))) \tag{2.9}$$

The discriminator loss basically makes it so that D wants to minimize its mistakes over both real and fake data, whereas G optimizes to increase the probability of D making a mistake.
The second loss we have used is the Wasserstein Distance, GANs using that are therefore called Wasserstein GANs [11], or WGANs. The idea is to use the Wasserstein Distance because it leads to a smoother gradient of G even in the worst conditions (e.g. if D is absolutely winning). Despite the equation for the distance being intractable, it is possible to simplify the distance between the original and fake distribution using the Kantorovich-Rubinstein duality, obtaining:

$$\begin{aligned} &W(P_{data}, P_g) = \sup_{||f||_{L \leq 1}} E_{x \sim P_{data}}[f(x)] - E_{x \sim P_g}[f(x)] \\ &|f(x1) - f(x2)| \leq |x1 - x2| \end{aligned} \tag{2.10}$$

The role of f is played by the discriminator, with its output not being interpreted as a probability but being a raw value defining a reward for the generated samples.
The WGAN loss over a batch for D and G respectively is then:

$$\begin{aligned} &\frac{1}{b}\sum_{i=1}^{b}(D(G(z_i)) - D(x_i)) \\ &-\frac{1}{b}\sum_{i=1}^{b} D(G(z_i))) \end{aligned} \tag{2.11}$$

D is optimized to increase the distance between the score given to real and fake samples, while G needs to get fake samples be judged as highly as possible.
To make D 1-Lipschitz, [11] proposed to apply weight clipping, meaning that the parameters of D are limited between an arbitrary range of values. A following work [42], which improves upon WGAN, proposes to apply a gradient penalty in place of the clipping, so that the discriminator is furtherly penalized by a contribution given by the squared difference of the norm of the gradient of D parameters and 1.

Mode Collapse

As we have said, training GANs is notoriously hard. A major problem is the so called mode collapse, where the generator tends to focus on very limited area(s) of the distribution, this might be for different reasons. It could be that some subsets of the dataset are simply easier to model and imitate, or it may be that the adversarial training makes it so that G is guided towards samples that D easily recognizes as true. Most GAN losses do not

contain an explicit reference to a sense of diversity in the generated samples, focusing on a per-sample real/fake score instead.
The end result is a very limited variety in the generator output to an extent that we need to try again with different hyper-parameters, this is because coming back from mode collapse using gradient descent is really hard. Once we are near a total collapse on one point the gradient of each generated sample will likely point in a similar direction that is an improvement with respect to the judgement of D. The lack of coordination between each sample gradient will eventually push all outputs towards a single point, which D will then recognize as fake after some training. All these almost identical samples will then be pushed together around the sample space by the gradient of D for an indefinite time, without converging.
There are a number of proposed solutions to limit G from mode collapsing, although all of them are empirical. One of them is mini-batch discrimination [74], D will essentially be made to look at samples in combination by a mini-batch layer in D. This special layer projects the features of each sample to a new matrix (one matrix per sample) which rows will be used to make comparisons between samples by applying a negative exponential to the L_1 distance between said rows. Each sample will then obtain a new feature (concatenated to the original features) for each row of its matrix, by summing the aforementioned exponentials obtained by confronting said row with the (relatively) same row of all other data points. The discriminator will still have to score samples individually.
Another option is spectral normalization [63], it aims to impose the Lipschitz constant of D by setting the spectral norm of each one of its layers. The Lipschitz constant of a linear function (such as the matrix product of a layer) is its largest singular value, which coincides with the spectral norm of said matrix. To set the spectral norm of each layer we can simply divide the weights of each layer by its spectral norm, which can be cheaply computed through power iteration. The power iteration algorithm only needs to be iterated k times (e.g. 1) for each forward pass to eventually converge to the right value, this is possible thanks to the fact that D weights change slowly through gradient descent.

2.3 Constraint Programming

The work of this thesis aims to combine GANs with an external supervision to exploit their capability to capture aesthetic and hard to define features with the exactness of constraints programming. This is because while GANs (and NNs in general) can lack exactness, what we can express via constraints is limited, if we do not want to incur in intractable computation times.
Constraints are ubiquitous in computer science, and many well known problems can be mapped to a set of constraints to be solved by the proposed solution. Stating the constraints to be respected is a paradigm that is akin to a declarative language, whereas finding a solution respecting those constraints is often left to a specialized algorithm, engine or library.
Constraint programming (CP) is done via statements that can be - seemingly - extremely simple, such as $A \wedge B \vee D$, or $(y + x)^2 >= 10$, or $\sum_i^m x_i < 4.0$. The simplicity is however, merely apparent, the use of CP is widespread in many domains where finding a useful combination of a certain number of options is needed, e.g. for placing objects or allocating resources [35], and so on. Depending on the problem, solving a set of constraints is clearly NP-complete, one obvious example would be SAT, a Boolean formula is, after all, a constraint itself.
Depending on the problem and the way we choose to model it, solving constraints can be seen as two different categories: constraint satisfaction problems (CSPs) and constraints optimization problems (COPs).
CSPs are characterized by a set of categorical variables $\mathbb{X} = \{x_1, x_2, ..., x_n\}$ that can assume values from certain domains $\mathbb{D} = \{d_1, d_2, ..., d_n\}$ and a set of constraints $\mathbb{X}$ and $\mathbb{C} = \{c_1, c_2, ..., c_n\}$ over these variables. These problems are of combinatorial nature and tend to be solved via search methods, backtracking, constraint propagation, heuristics, etc.
Each constraint in C can be seen as a function over the variables or an explicitly set of valid combinations, solving a CSP is then the task of finding at least 1 tuple that respects all constraints, or, more elegantly put, a mapping between $\mathbb{X}$ and $\mathbb{D}$ that does the same thing.
Constraint optimization is the task of setting the value of the variables we are in control in order to optimize a

certain objective cost function f while respecting a set of constraints, thus a COP takes the form of:

$$\begin{aligned} \text{min} \quad & f(\boldsymbol{x}) \\ \text{subject to} \quad & g_i(\boldsymbol{x}) = c_i \quad \text{for } i = 1, ..., n \\ & h_j(\boldsymbol{x}) > d_j \quad \text{for } j = 1, ..., m \end{aligned}$$

While g_i and h_j are what we define as hard constraints, there is also the possibility of having soft constraints, which concur to a penalization of the cost function if not satisfied, proportional to the extent of not being respected.
COPs are perhaps of least interest to us given that the variables we work with, the generator output samples, are binary, and the constraints we impose form a CSP, although one could argue that we are actually working with a COP, given the later explained constraints and way of quantifying how much they are respected by G. The constraints we apply are all expressed in propositional logic, and thus expressed through a Boolean formula, their exact definition is reserved for the experimental section.

2.4 Knowledge Compilation and Weighted Model Counting

As we have said, (some) constraints can be expressed in the form of propositional logic constraints, such a set can be considered a propositional knowledge base in the sense that we know what must be true given over the considered variables, or, equivalently, a collection of sentences that must be satisfied.
In the same way the same constraints can be expressed in a number of ways, there exist different representations of the same knowledge base. Representations tend to have different pros and cons, the right choice is dependent on the task at hand. Some representations can be more compact and have a lower memory footprint, while some others help in reducing the computational complexity of a category of queries.
Succinctness refers to the final size of the knowledge base once it has been transformed into the new representation, whereas with tractability we mean what set of queries or operations that can be performed in polynomial time. Most often obtaining a representation that is succinct leads to a drawback in tractability and vice-versa, broadly speaking there is an exponential upper bound in terms of computation time or representation size with respect to the number of variables, clauses, etc.
A model of a knowledge base is an assignment of either true or false to the value of each symbol (variable) such that the propositional formula represented by the knowledge base evaluates to true. The problem of determining if a propositional formula is satisfiable, i.e. if there exist a model that satisfies it, is called SAT. #SAT, or propositional model counting, is instead the problem of computing the number of models for a given formula in propositional logic. SAT is NP-complete, whereas #SAT is #P-complete, which is the set of counting problems related to decisional problems that belong to NP.
With weighted model counting (WMC) we mean the sum of the weights of all models of a formula, where the weight of a model is given by the product of the weight of its literals. If set to true, a variable will have weight $p(x) \in [0, 1]$, whereas it will have a weight of $1 - p(x)$ if instead it is considered false.
We can use weighted model counting as a mean of doing probabilistic reasoning by considering $p(x)$ the probability of a variable being set to true, and the resulting weighted model count as the probability of the propositional sentence assuming value true when evaluating a truth assignment which is sampled by the aforementioned variable probabilities.
For example, WMC can be used as a tool for probabilistic inference by encoding a probabilistic model, such as a Bayesian network, into a propositional knowledge base where each model is associated to a weight according to the network parameters [19].
An arithmetic circuit is a directed acyclic graph (DAG) where nodes having in-degree zero are considered inputs, and can take the value of a variable or a constant, all other nodes are either sums or products, evaluating the root node of said DAG is equivalent to evaluating the circuit and the polynomial it represents.
Each node can be seen as computing some polynomial, thus leading to the naturally recursive evaluation where a product node will perform the product of the polynomials computed by its children, whereas a sum node will sum said polynomials. Input gates assume the value of a constant or the variable they are assigned to.
An arithmetic circuit, or a structure that can be transformed into one [32, 12], can serve as a propositional knowledge base. The appeal of using a circuit (or an equivalent structure) is that if some properties are respected, the (weighted) model counting of the represented constraint can be computed in linear time with respect to the size

of the circuit [32, 68, 12], note that this says nothing about the size of the circuit, which upper bound might be exponential with respect to some parameter describing the encoded logic formula (e.g. the tree-width of a CNF).

2.4.1 Sentential Decision Diagrams

A Sentential Decision Diagram (SDD) [68, 23] is a representation for propositional knowledge bases, different SDDs can be combined using Boolean operators in a polynomial time. Given some additional properties the representation is canonical and, starting from a conjunctive normal form (CNF) [16] of n variables and tree width w the upper bound of the size of the resulting SDD is $n2^w$.
SDDs are based on two properties added to a negative normal form (NNF) [16], structured decomposability and strong determinism.
A NNF can be seen as a root DAG where each leaf is related to either the value true, false or a literal, whereas internal nodes are conjunctions or disjunctions. A NNF is said to be decomposable iff all the childs of a conjunction node share no variables among them. A NNF is deterministic iff there are no shared models between the children of a disjunction node. Adding these properties leads to a subset of NNF which is deterministic and decomposable, d-DNNF.
To define *structured* decomposition, we first need to define a structure called V-tree [67], given a set of variables X, a V-tree for X is a full rooted binary tree where there is a one to one mapping between its leaves and the variables in X. A V-tree can also be seen as related to a formula if it maps the variables in said formula in the aforementioned manner.
To get to **structured decomposability**, we need to define a structured subset of DNNF, first, we say that a DNNF respects a V-tree if for each one of its conjunctions $\alpha \quad \& \quad \beta$ there is a node n in the V-tree where $variables(\alpha) \subseteq variables(left\_child(n))$ and $variables(\beta) \subseteq variables(right\_child(n))$.
From this, the set of DNNFs respecting a V-tree T is defined as $DNNF_T$, from that, structured DNNF (SDNNF) is the set of $DNNF_T$ over all possible V-trees.
Given a V-tree T, the set of $d-DNNF_T$ is a subset of $DNNF_T$ formulas that satisfy determinism, finally, deterministic structured decomposable NNF (dSDNNF) is the union of all $d-DNNFT$ over all possible V-trees.
To introduce **strong determinism** we must first introduce the concept of decomposing a Boolean function f which maps variables in Z to either $false$ or $true$. Given Boolean function $f(X,Y)$ with non-overlapping set of variables X and Y, if $f = (p1(X) \wedge s1(Y)) \vee ... \vee (pn(X) \wedge sn(Y))$ then $(p1, s1), ..., (pn, sn)$ is called an (X,Y)-decomposition of f as it allows one to express f in terms of functions on X and on Y only [68].
If $pi \wedge pj$ is inconsistent for all $i \neq j$ then we have a strongly deterministic composition, such a property leads to a lower bound of the number of (pi, pj) pairs in a decomposition (its size).
We can use the two aforementioned properties to guide the decomposition of any Boolean function f. Given such a function we can split its variables into two disjoint sets (X, Y) and - always - [21] decompose it in the following way: $f = (p1(X) \wedge s1(Y)) \vee ... \vee (pn(X) \wedge sn(Y))$ while retaining the following properties: p_i over i are non-false, exclusive and exhaustive, while s_i over i are distinct. Note that the decomposition is also unique given the split of variables. This is what is called the compressed $(X-Y)$-partition of f.
The choice on the variable split and thus decomposition can be fixed by using (thus respecting) a V-tree, it can be proven that given a V-tree there is a unique SDD, hence the SDD can be made canonical.
An SDD can then be built by recursively computing the compressed $(X-Y)$ partition for f at first, and then for the primes and subs of the partition recursively; recursion stops after reaching false, true and literals. Note that the used V-tree can be optimized by local search to reduce the size of the SDD.
An SDD, as seen in figure 2.5, is made by two kinds of nodes: circles stand for the disjunction of its children, whereas a box represents the pair, and thus the conjunction, $(p \wedge s)$. Recursively, a conjunction may contain a decision node (which is a decomposition), constants or literals. Each node is related to some sub-function, be it prime or sub, of the decomposition, with the root node being f itself.
Substituting circle nodes with “OR” and boxes with “AND” the d-SDNNF becomes evident.
Decomposability and determinism allow to answer queries about models efficiently. This however, is only of limited interest to us. To combine this representation with differential optimization we need one more step, which is using an SDD to represent the probability distribution of the model of a propositional theory.
Rather simply, we can transform the SDD by substituting ANDS represented by pairs (p_i, s_i) with products and ORS given by decompositions with normalized sums. This leads to an arithmetic circuit that has variable

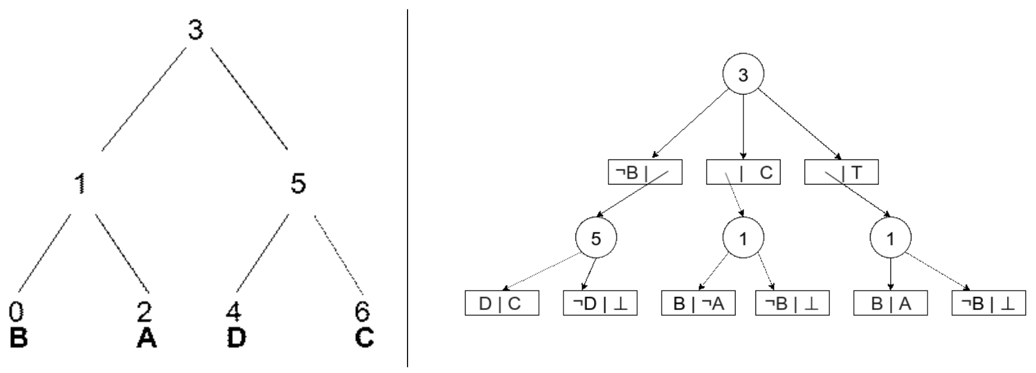

Figure 2.5: Example of sdd and the related vtree, [21].

nodes as its inputs, whereas false and true are mapped to the constants 0.0 and 1.0 respectively. The circuit is perfectly differentiable and its output corresponds to the WMC we would obtain given the plugged in variable probabilities, the output tensor of the generator in our case.
The compactness of the circuit is especially important given the fact that we can trade-off a way more efficient forward pass through the circuit, that happens many times during training, for a single computation at compile time. Most importantly, thanks to the built-in properties of said circuit we can evaluate the circuit and the gradient of its result with respect to the leaf variables in linear time, with respect to its size.

2.5 Semantic Loss

As mentioned in the previous section, we can use propositional knowledge compilation techniques as a bridge between the output tensors of a neural network and logical constraints, coupling symbolic reasoning and representation learning. The semantic loss (SL) [88] is not the first attempt at doing that, other examples are neural Turing machines [39], neural theorem proving [61] and relational embeddings [89].
Technically speaking, going from the WMC in the form of the output of the arithmetic circuit obtained by the SDD to the SL is equivalent to simply taking the negative logarithm of said WMC, so that a sample perfectly respecting the imposed constraints would have a loss of 0. Given this quite simple implementation step from WMC to SL we use this section to better formalize it and enumerate some of its properties.
Definition(from [88]): Let p be a vector of probabilities, one for each variable in X, and let α be a sentence over X. The semantic loss between α and p is

$$L(\alpha, p) \propto -\log \sum_{\boldsymbol{x} \models \alpha} \prod_{i:\boldsymbol{x} \models X_i} p_i \prod_{i:\boldsymbol{x} \models \neg X_i} (1 - p_i) \tag{2.12}$$

Which means that the SL is proportional to the negative logarithm of the probability of sampling a valid combination of variables from p, with p seen as a distribution, i.e. p_i is the probability of X_i being true. In our case, p is a tensor in the computation graph of the neural network in the range [0, 1], made true by using a sigmoid function or other means.
Note that the SL is a generalization of the cross entropy over arbitrary constraints.
We conclude with some properties of the SL taken from [88].

Label-literal correspondence The SL of a single literal is proportionate to the cross-entropy of the equivalent labeling: $L(x, p) \propto -\log(p)$ and $L(\neg x, p) \propto -\log(1 - p)$.

Value symmetry For all p and α, $L(\alpha, p) = L(\bar{\alpha}, 1-p)$, with $\bar{\alpha}$ replacing each variable in α by its negation.

Monotonicity If $\alpha \models \beta$ then $L(\alpha, p) \geq L(\beta, p)$.

Semantic equivalence If $\alpha \equiv \beta$ then $L(\alpha, p) = L(\beta, p)$. This is especially important because if implies that the SL is independent of syntax.

Moreover, it can be proved that the SL as defined in equation 2.12 satisfies all these properties and the remaining ones as defined in [88] and is the unique function (up to a multiplicative constant) that does so.
The work of [88] is limited to discriminative settings, whereas we will try to apply the SL to generative ones.

2.5.1 From Propositional Constraints to Semantic Loss: Pipeline

As we have seen, to go from an arbitrary propositional knowledge base to the final arithmetic circuit some passages are involved.
Given an arbitrary constraint in propositional logic, we need to convert it to either CNF or DNF, a possible tool to do that is sympy [59], the output needs to be in the DIMACS format. Given a CNF or DNF in DIMACS, we can compile it to an SDD by using the SDD package [2], once that is done, we can add the WMC to our computation graph by importing the SDD and visiting its nodes bottom-up to construct the circuit, using the desired tensor variables as the leaf nodes; the PSDD package [12, 50] implements this.
Obviously, the SDD can be deallocated from memory once the circuit is built, and the circuit itself can be discarded once training is done.
A drawback of this approach is that the arithmetic circuit does not exploit matrix or vector operations, creating instead one node in the computational graph for each sum and product. When the constraint is particularly complex or the number of variables is high this can make computation slow or the graph too large to fit in memory, an alternative solution based on an approximation of the SL is offered in the next chapter.

2.6 Related Work

Many recent improvements on deep generative modeling such as VAEs, flow-based approaches, autoregressive models and GANs are not intrinsically designed to support the generation of structured objects. More suited methods like graphical models and probabilistic grammars might be inappropriate or scale badly given the task, e.g. constrained image generation, moreover, sampling under constraints is likely to be inefficient.
Tractable circuits, such as - probabilistic - sentential decision diagrams [12], allow for fast inference while respecting constraints, their parameters can be efficiently learned from data. Given their upper bound in size they might become cumbersome as the number of variables grows (e.g. with images). Moreover, (P)SDDs are not as flexible as NNs, by using GANs we can exploit past works about NNs, such as embeddings [60], transfer learning [82], knowledge distillation [44], adding other losses and objectives and so on. Having a NN architecture that can respect constraints as thus the obvious advantage of achieving what we need while also not setting a ceiling about what we can do.
Mixing machine learning and knowledge instillation is a challenge that has been tackled for a long time. Attempts range from grounding specific weights in Markov Logic Networks [56], to using specific architectures to consider constraints, such as the inclusion of a sub system dedicated to learning constraints [86, 80] and so on.
Other frameworks coupling deep learning with logical constraints exist, such as [71, 45, 26, 28], it is usually done by transforming the constraint into a differentiable form by replacing implications with inequalities or changing logical operators to fuzzy T-norms. However, such alterations affect the information conveyed to the network, and depend on syntax. Factor graphs are a possible, more exact, encoding but require loopy belief propagation to obtain their output and its gradient [64], which scales badly (given the number of passes we require for training a NN), among other issues [78].
In section 4.3 we present our results on the task of molecule generation. Said task, being of especially relevant pharmaceutical interests, has a long trail of research to its name. Most works are defined by how molecules are represented, some of them use the SMILES [87] format, a string based representation used in the aforementioned domain, while others use fixed size representations such as matrices. Examples of the former are CharacterVAE [36], and GrammarVAE [51]; using a sequential representation without a fixed size as the relevant problem that optimizing sequences is usually harder, or at least way slower, than using fixed size representations.
Among the works having a fixed size format we mostly find that the graph representing a molecule is seen as a transition matrix with the possible addition of a node matrix if nodes can assume different labels. A portion of said works uses variational auto-encoders to simply perform link prediction [49, 40, 24], whereas others aim to generate the whole graph in a one-shot manner [77, 75].
Given the scope of this work we are interested in the subset of these architectures using GANs, the two most relevant works in our case are ORGAN [41] and MolGAN [25]. The former uses a sequential architecture [90]

to generate SMILES strings, while the latter makes use of two matrices for a fixed size representation; both these models use some form of reinforcement learning loss to impose arbitrary constraints (related to chemical properties, later explained) on top of the adversarial loss, and both intend to produce the whole molecule at once.
We choose to work with MolGAN given its overall simplicity and training speed gained by using a more simple representation, and the fact it performs better with respect to the possible chemical properties of interest.

3 CANS: Constrained Adversarial Networks

3.1 Motivation

Before explaining what CANS are, we provide the motivation behind them, which is threefold. The first reason is related to the difference between optimization and learning.
With learning, we are interested in generalizing over a certain task, for GANS that would be being able to produce new samples with respect to the training set, over-fitting on the input data or having the GAN mode collapse is not auspicable. In contrast, with optimization we try to find the best possible solution, there is no incentive in diversity and no concept of generalization. Adding a concept of diversity to our constraints can be hard to encode and unnatural to the ask, moreover, using constraint programming has also the limiting defect that we can impose only what we can properly formalize.
GANs have the upside that, being a machine learning model, are intended to generalize, sampling different outputs is a built-in feature and capturing otherwise unformalizable features is an ingrained property of representation learning. Searching through many combinations of variables to find the right ones can take an extremely long time for an optimizer, and we will bear this cost every time we need to search for a solution.
If a GAN can be trained to produce valid solution the cost of training is fixed while sampling is relatively cheap. Of course, there might be no guarantee that each time we sample from the generator we will obtain a valid object, but we do not need to. If such a thing is probable enough we could simply use rejection sampling on the trained CAN knowing that it will take drastically less time than a GAN that allocates 0 or low probability mass to valid samples. We could also take the "almost perfect" object and use a real optimizer from there or rely on human intervention if the use of our output is a starting point to be refined by artists, designers, etc.
The second motivation behind CANs is the lack of control that can transpire from using GANs and NNs in general. NNs are end-to-end black boxes, they are hard to debug and understanding why or what they are learning can be a challenge. Training GANs is worse, hyper-parameters become more important, training slower, losses and performance measures lack informativeness and so on. If we can't open NNs up and change the inner working at a whim, we can use the SL to have a generalized way of explaining to the network the conditional dependencies among the output variables we expect from them.
The third and final point is related to data. Complex constraints might not be correctly modeled by the generator network if training data is limited (or if training time is limited, meaning that a reason to use the SL is to reduce training time to attain valid objects). Data might be inconsistent with the constraints we intend to impose, be it because of a noisy/corrupted dataset or due to the original distribution simply being different from what we want to obtain, e.g. if we intend to have the generator distribution, $p_g(x)$ resembling the original dataset but also having previously unseen relationships between its variables.
Having the target distribution differ from the one we provide to the GAN has the obvious problem that we cannot expect the generator to magically match our desires, this can be formalized given the (idealistic) theorem in [38] which basically says that any global equilibrium that is found between G and D leads to p_g matching p_{data}.
The conjunction of the aforementioned rationales leads to our proposed architecture.

3.2 The CAN architecture

A CAN extends a GAN by augmenting its learning in structured domains by allowing fine grained control over all tensors of the architecture. All this is captured by adding a penalty term, the semantic loss, which captures our intentions without the need for more parameters or networks in the adversarial play. It also does not require any more sampling than a GAN would in order to measure the mass that G allocates to feasible and unfeasible objects. Such penalty encourages the generator to output valid structures at training time, and is based on a procedure that is probabilistically sound and exact.
The SL acts through an arithmetic circuit that takes as input arbitrary variables and outputs a single value that explicitly states the probability of the generator objects to be correct, on which a simple negative log can be

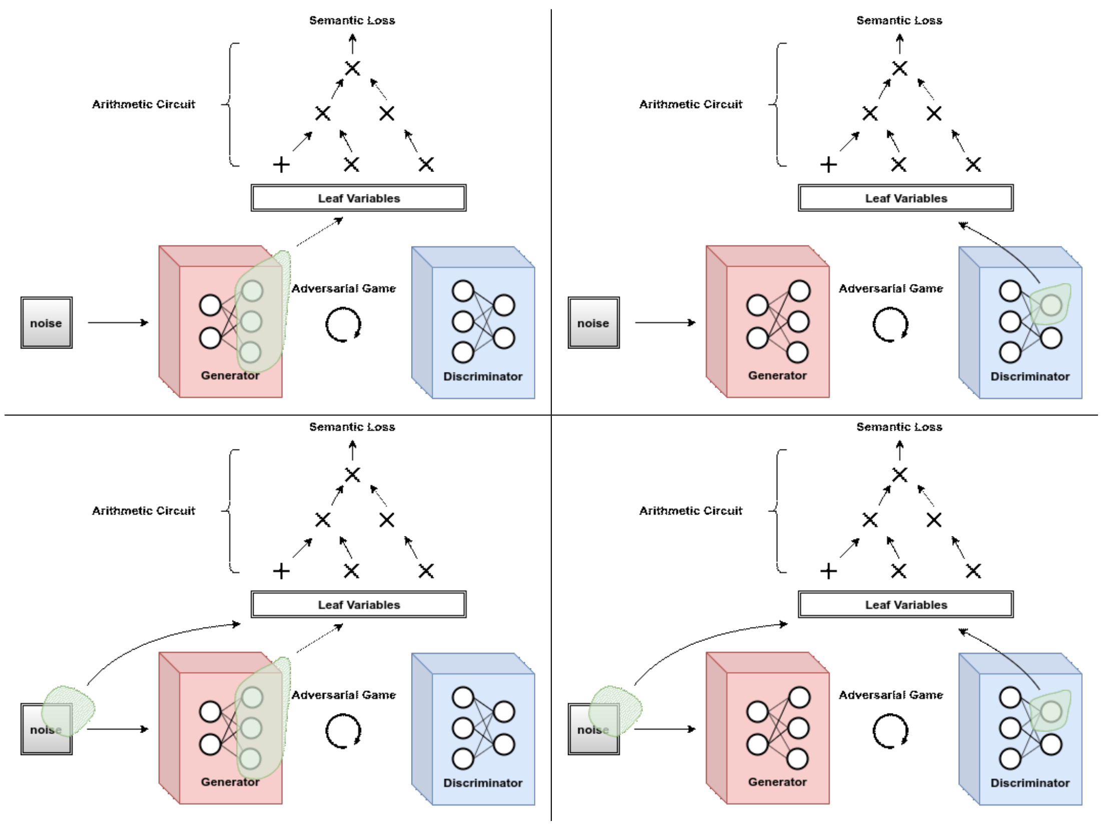

Figure 3.1: Possible variations of CAN, imposing constraints on the output of the generator (top left), on some discriminator output variables (top right), mapping some noise variables to the generator output (bottom left) and mapping some noise variables to some discriminator output variables (bottom right).

used to get to the formal definition. The arithmetic circuit is designed and built at compile time, i.e. before the training even begins, by leveraging knowledge compilation techniques that allow to go from propositional logic to circuit by factorizing the polynomial in equation 2.12 in a more compressed representation, an SDD.
The circuit can, of course, range from very small to quite complex, in the former case this leads to an evaluation of the SL and its gradient that is efficient [88], for the latter situation we propose a solution. The size, or complexity, of the circuit is not bound to the significance of the constraints we impose, given that the variables over which the SL is computed might be tensors that have extremely impacting implications if trained to be conditioned over other tensors; a simple AND might lead to drastically different results compared to a plain GAN.
Given that the circuit measuring the SL is only needed for training, sampling from a CAN afterwards has same computational cost of a GAN, no other optimization steps are involved.

$$L_{CAN}(g,d) = L_{GAN}(g,d) + SL = L_{GAN}(g,d) - \lambda \log(P_g(\alpha)) \tag{3.1}$$

where

$$P_g(\alpha) = \sum_{\boldsymbol{x} \models \alpha} \prod_{i:\boldsymbol{x} \models X_i} p_i \prod_{i:\boldsymbol{x} \models \neg X_i} (1 - p_i) \tag{3.2}$$

With $P_g(\alpha)$ standing for the probability of the generator to output feasible objects with respect to some constraint α, X being the input variables over which the probability is computed (the input tensors, leaves of the circuit), with p_i being the probability of variable X_i being true. A non-negative term λ is used to control the importance of the SL.

3.3 Limits and workarounds

There is no free meal, and the SL is no exception, there are ways to circumvent some of its limits, although by approximation.
The first obvious limit is that the arithmetic circuit is obtained by compiling propositional logic, thus restricting the domain over which we can formally define without getting to constraints (and circuits) that are intractable in size with respect to the number of variables.
That can be fixed, in a limited manner, by adding more players to the adversarial game. Additional NNs can be trained to generalize constraints that we wouldn't otherwise be able to, essentially allowing us to dictate formal constraints over informal ones. As an example, we could have a classifier trained over some categories k_i, and impose on the generator that some other variables x_i are on an equality relationship with the classifier output, like $x_i \implies k_i$.
This has the powerful implication that we can exploit pre-trained NNs and impose constraints over a taxonomy of labels, variables, etc. in a natural and effort-less manner.
The second problem is that depending on the number of variables and complexity of the constraints the arithmetic circuit can get too big, this, coupled with the fact that the ongoing operations are sums and products that do not exploit matrix or algebraic operations, might lead to a computational graph that is either too big to fit on the GPU memory, or that takes a long time to be evaluated. As a counter-measure we propose an approximation of the SL which groups constraints that are equivalent but over different variables under the same arithmetic circuit, in order to exploit vectorized operations to speed up the evaluation of the SL and its gradient.
The method works by reducing the constraints we apply over a single sample to the set of classes of constraints that characterize what we want to impose. An equivalent explanation is given by considering the presence of sub-circuits that are identical but applied to different variables, i.e. they differ only by their literals in their leaf nodes.
Instead of compiling and evaluating all the possible sub-circuits, where doing the former might lead to memory problems and the latter to a slower training, we can limit ourselves to using a single circuit of such type, and reshape our variables accordingly. For each data sample, each leaf of the new circuit can be seen as a vector of variables, so to that sums and products can be done in an element-wise (and vectorized) manner, thus leading to more efficiency.
As a practical example, consider the case where a data point is two dimensional, of size $n \times n$, with n even, and we are interested on imposing an implication between variables having an even column index and the variable right next to them, essentially saying that

$$\forall i, \forall j \mid j \mod 2 = 0, (x_{i,j} \implies x_{i,j+1}) \tag{3.3}$$

The number of pairs bound by the implication grows quadratically with n, and so does the circuit size, compilation and evaluation time. We are, however, applying the same sub-circuit representing an implication over an over on the whole sample, with the final circuit being the AND of the smaller ones.
A more efficient alternative would be to use a single circuit and reshape the sample accordingly before feeding it to said circuit, so that the left leave of the circuit is the vector containing all the variables that would have been antecedents in the implication, while the right leave would be the vector containing the consequent.
Starting from leaves as vectors, the output will itself be a vector, having each dimension as the probability of each implication being respected; to obtain the final scalar output, equivalent to the original circuit, we can simply perform the product over all dimension of the vector, "ANDING" the different weighted model counts.
The concept can work similarly when circuits of the same type are in an OR relation, simply requiring the use of a normalized sum instead of a product.
In the presented case there is no approximation, because we haven't disrupted any of the SDD properties. This can change even by apparently small details, consider the previously imposed constraints, only that now we drop the requirement over j being even:

$$\forall i, \forall j, (x_{i,j} \implies x_{i,j+1}) \tag{3.4}$$

By using the aforementioned approach, we would get to the point of reducing the output vector to a single scalar via product, to "AND" the weighted model counts of the (abstractly) different sub-circuits, by doing this we are however losing the structural decomposability property of SDDs. This is because we now have variables

that are shared among different components of what would be a composition node, given that variables are now taking both the role of antecedent and consequent in different implications, and thus appearing in different siblings of the same decomposition.
One last, subtle problem is related to collateral effects that our constraints might introduce. An implication, $x \implies y$, can be equally rewritten to $\neg x \lor y$, even if our dataset presents cases of solving said imposition in both ways ($\neg x$ being true or y being true, or both), the network may simply mode collapse or find a local optima in which the implication is always solved in the same way, e.g. by always having $\neg x$ being true.
The problem can become more evident if we apply a constraint either on a subset of the distribution or on a sub-part of a single sample, with the former potentially leading to the generator network avoiding said part of the distribution; the latter might become an incentive to G to generate samples that contain evident unintended effects. An example of the second case is provided in section 4.1.3.

4 Experiments

This chapter serves the purpose of enumerating and explaining the datasets the empirical tests were based on, the applied constraints and the results. The experiments are implemented in python and by using the neural network library tensorflow, the software is part of a larger code base meant for studying CANs, the repository is owned by Prof. Andrea Passerini and his research group, and will be disclosed following a (still potential) publication submitted to the eighth International Conference on Learning Representations, ICLR 2020.
For all the following experiments we use a simple GAN architecture augmented with SL unless otherwise specified, the provided results are obtained after searching over a reasonable grid of hyper-parameters, comparison is done by benchmarking the same architecture with the same hyper-parameters except for the use (or lack thereof) of the SL.
Given that we work on discrete data, we have the chance to use metrics that might otherwise be impossible to compute exactly. Validity measures the proportion of perfect objects in a batch, uniqueness measures the fraction of unique types of valid objects with respect to the total valid ones, whereas novelty is the proportion of valid sampled objects that are not in the training data.
Note that validity is essentially a proxy for WMC and it either coincides or is so similar to it that providing both did not make much sense. The third and last experiment uses a mix of the aforementioned metrics and some ad hoc properties coming from the particular domain of application.

4.1 Polygons and Parity Check

4.1.1 Dataset And Architecture

The first experiment is performed on a synthetic dataset coming from previous works carried on under the supervision of Prof. Andrea Passerini, namely by past students Gianvito Taneburgo and Pierfrancesco Ardino.
The dataset is composed of 20×20 discrete and binarized images, each sample has a black background and contains exactly two white regular polygons that are sampled among triangles, squares and rhombi, with a probability of 0.30, 0.30 and 0.40 respectively. The polygons are randomly placed respecting the fact that they should always be different (no image contains two polygons of the same type), they should not overlap with each other and the polygons should be fully contained within the image. Moreover, their total area is the same over all types, 25 pixels. The resulting dataset totals 23040 items.
The GAN (CAN) is implemented by having the generator and discriminator being deconvolutional and convolutional networks, respectively. The generator starts from a noise vector z of dimension 64 and maps it to a new representation with a fully connected layer of 1024 units, followed by a ReLU and batch norm, this happens again with a fully connected layer of 1280 units, then two transposed convolution layers with a kernel size of 5×5 and stride 2 generate the output; the first one of these two layers has 64 filters and is followed by a ReLU and a batch norm layer, while the second one, which is the output layer, has only one output channel given that our dataset is binary.
The generator output layer is followed by sampling from a concrete distribution [58] using the generator logits as starting probabilities, a technique to provide a better gradient to the training of networks that try to mimic a discrete binary distribution.
The discriminator is instead made of two convolutional layers of 5×5 kernels, stride 2, leaky ReLU as the activations and 64, 128 filters respectively. After those, three fully connected layers of 1024, 32, 1 units are applied, with the first two using, again, leaky ReLU as their activation functions.
As for the loss, we use the one referenced in equation 2.9.

4.1.2 Applied Constraints

The outer borders of the image serve a special function, each border pixel is a parity check, corners excluded. Pixels in the top and bottom edge are set to 1 (white) if the sum of the 9 pixels that are on the half column they belong to is odd, 0 (black) otherwise. Pixels in the left and right edge are set to 1 if the sum of the 9 pixels that

are on the half row they belong to is odd, 0 otherwise.
The goal of these pixels is to add a non-trivial relationship between the borders of the image and whatever is inside the remaining 18×18 canvas, to drastically reduce the probability of G generating valid objects without having learned what dependencies model the dataset.
Such constraints can be modeled in the following way using propositional logic, assuming our samples are

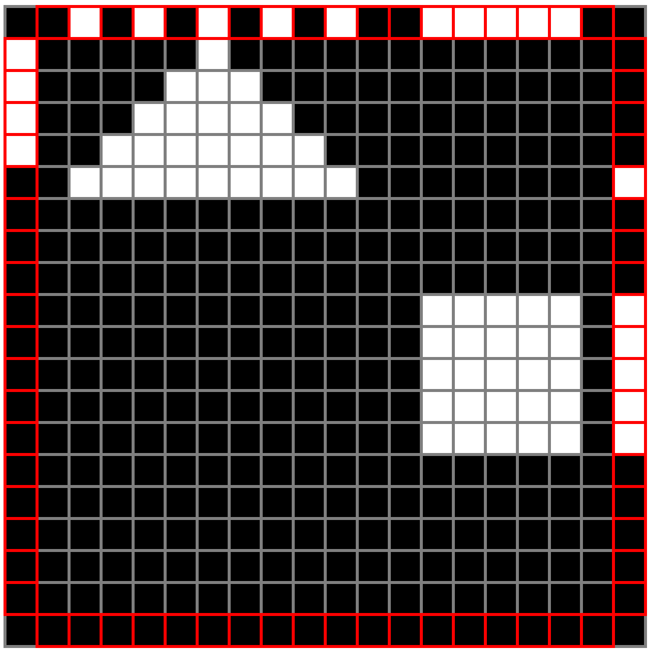

Figure 4.1: Grid cells acting as a (dis)parity check for their respective half row/column.

20×20 binary grids:

$$\begin{aligned}
&\forall i \in \{2...19\} \\
&X[1][i] \iff \bigoplus_{j \in \{2...10\}} (x[j][i]) \\
&X[20][i] \iff \bigoplus_{j \in \{11...19\}} (x[j][i]) \\
&X[i][1] \iff \bigoplus_{j \in \{2...10\}} (x[i][j]) \\
&X[i][20] \iff \bigoplus_{j \in \{11...19\}} (x[i][j])
\end{aligned} \tag{4.1}$$

To keep the size of the resulting arithmetic circuit in check we employ the strategy described in section 3.3, this experiments thus serves as an empirical demonstration of the effectiveness of the technique.

4.1.3 Results and Discussion

The first point to make is that, thanks to the limited size of the circuit, compilation time, memory overhead and computation costs of the SL were negligible.
It can be observed that after a certain threshold of λ, the difference between the validity and WMC of the GAN and the CAN were noticeable. An inverse correlation between λ (and thus validity) and uniqueness/novelty exist, this is reasonable, the more weight we allocate to the SL, the more the generator will tend to mode collapse.
An expected - but pleasant - result is that not only does the SL lead to more valid objects, it does so in a faster way. After 250 epochs of training GANs stand at about 0.57 validity, something that CANs with $\lambda = 1$ reach in about 100 epochs less, taking 0.60 less time to get to the same validity.
These outcomes show that, when dealing with tractable constraints, CANs can achieve better results than GANs,

especially when considering validity as a function of training time. We empirically proved that the proposed method in section 3.3 can work and helps in making the computational cost of SL drastically lower. Note that given the fact that novelty and uniqueness are a function of validity we end up with CANs with λ equal to 0.5 and 1.0 actually having slightly more total novel and unique samples than GAN ($\lambda = 0$).

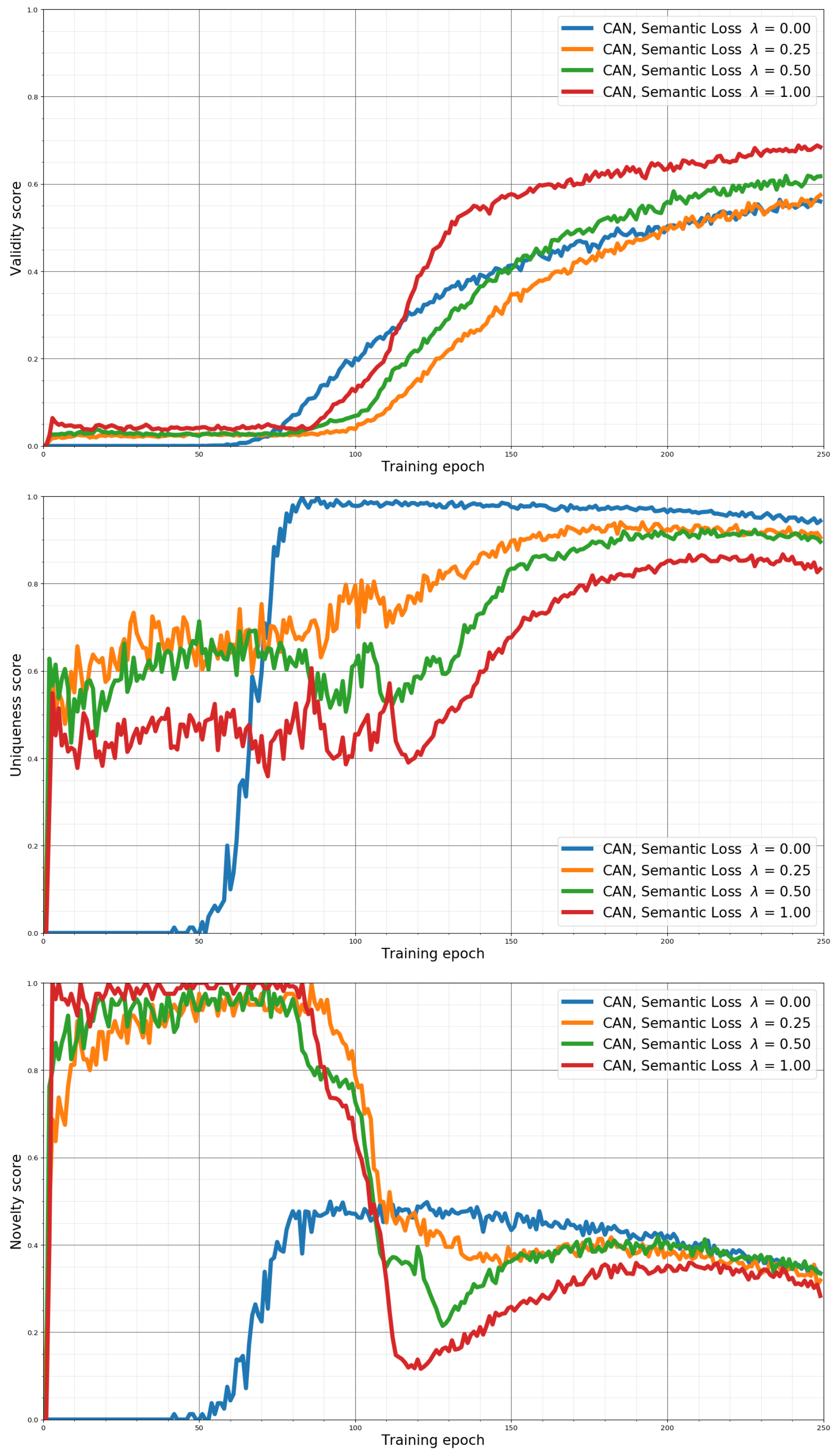

Figure 4.2: Comparison between CANs with different λ and GANs ($\lambda = 0$).

Collateral Effects of The Semantic Loss

As mentioned in 3.3, applying the SL can lead to unforeseen effects due to local optima and the NN taking the easy way out of respecting constraints.
During our experiments we observed such a case, in particular, we applied the parity check constraints only to the top left quadrant of the image, equal to:

$$\begin{aligned} &\forall i \in \{2...10\} \\ &X[1][i] \iff \bigoplus_{j\in\{2...10\}} (x[j][i]) \\ &X[i][1] \iff \bigoplus_{j\in\{2...10\}} (x[i][j]) \end{aligned} \tag{4.2}$$

This resulted in the generator network avoiding the generation of polygons in the top left area of the image, so that constraints would automatically be satisfied by a completely black quadrant.

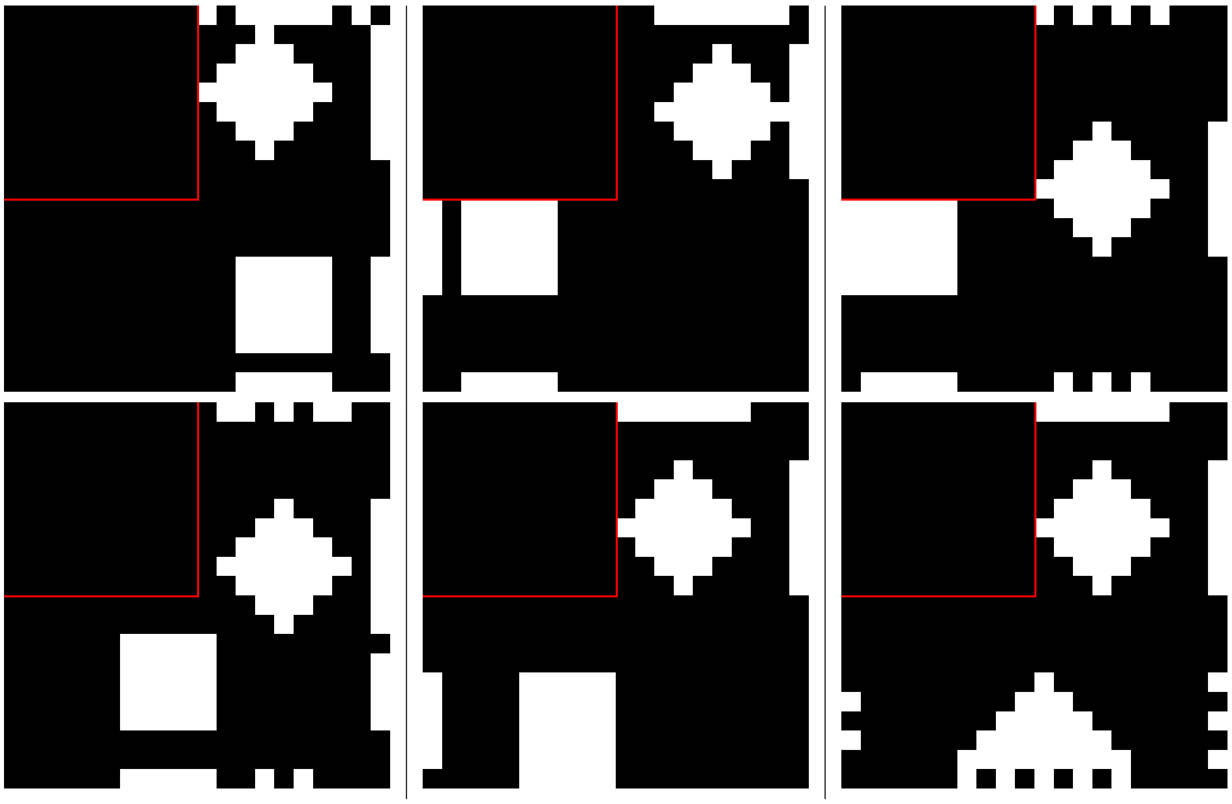
Figure 4.3: Top left quadrant being avoided by the generator when the constraints are only applied to said quadrant.

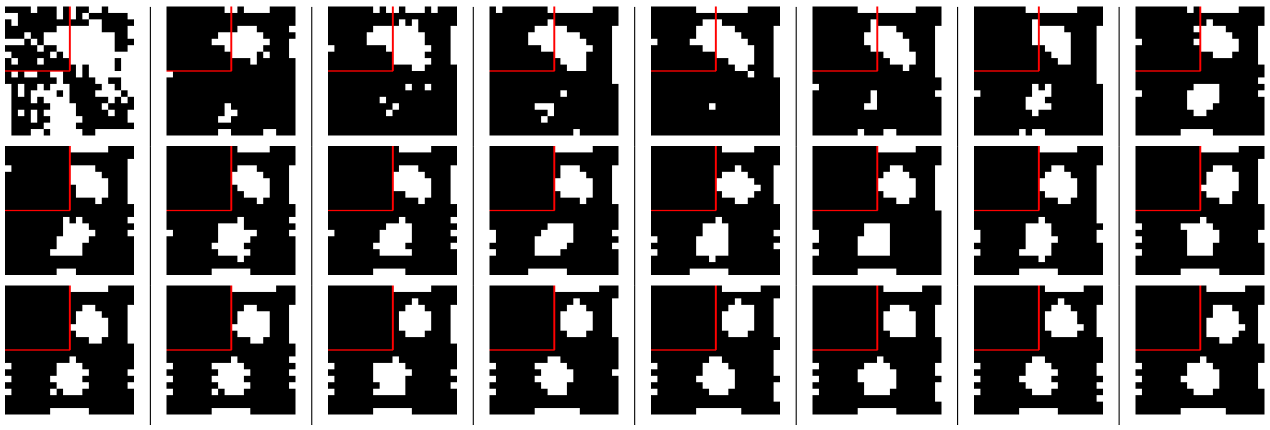

Figure 4.4: Evolution of a sample over time while applying the parity check constraint only on the top left quadrant of the image.

4.2 MNIST Still life

4.2.1 Applied Constraints

For this second experiment we first define what constraints are imposed, then explain the resulting dataset, this is because we take a widely used dataset and radically transform it by imposing some game-changing conditions. We need to start with a preamble about the origin of our rules.

Cellular Automata

A cellular automaton [6] is a discrete model made of a grid of cells, essentially an array, where cells can take a value among a predefined set of states. Theoretically, we can think of this canvas as infinite, of course the dimension are finite in practise.
Beyond having discrete cells with discrete values, cellular automata (CA) also have a discrete concept of time; initialization time is considered at $t = 0$, the values we initialize the cells with can be arbitrarily chosen and determine the end result. Each cell is in a relationship with its neighbours, given its current state and the state of its neighbours a cell will go from one state at time t to another state at time $t + 1$. The rule(s) that define the transformation from one time step to the next one are applied to the whole grid at the same time, and they tend to not change over the iterations.
To synthesize, CA can be seen as arrays over which we iterate to synchronously update all values according to some function, given the starting state and the update rules, this can result in patterns that exhibit different properties.
Most CA are broadly characterized into four categories: patterns evolving into some stable homogeneity, ones evolving in mostly stable or being characterized by a periodical transformation, CA that result in a chaotic evolution, and CA resulting in structures with high complexity and that evolve over a long time, with stability over some sub-parts of said structures.
Some CA have been used to simulate chemical interactions, biological systems, and some are proved to be Turing Complete.
As we have said, CA are computed on a finite grid, this brings the problem on managing border cells, to which there are possible approaches: defining different neighbours for edge cells (thus leading to also defining new rules for them), setting these cells to a constant value, thinking of them as if in a toroidal arrangement, essentially making the considered world a closed loop.

Convay's Game of Life

Convay's Game of Life (GoL) [7, 3] is a cellular automaton designed in order to find a set of rules that met four key requirements, which we quote:

- There should be no explosive growth.

- There should exist small initial patterns with chaotic, unpredictable outcomes.
- There should be potential for von Neumann universal constructors [8].
- The rules should be as simple as possible, whilst adhering to the above constraints.

GoL takes place on an infinite 2d orthogonal grid of square cell, each cell can assume only two states, either alive (1) or dead (0). The neighbourhood of a cell is given by cells that are horizontally, vertically or diagonally next to the considered cell, for a total of height neighbours.
Starting from an initial configuration of said cells (essentially the seed), evolution takes place by following four relatively simple deterministic rules:

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

GoL has captured interest because it is an example of emergent behaviour that can result in automatic organization, moreover, this player-less game is Turing Complete, leading to the philosophically sounding fact that Life is undecidable. Given a pair of patterns, no algorithm exists that can predict if the second pattern is going to be a future state, this is related to the halting problem, determining if a program will eventually reach an end state or if it will continue to run indefinitely.

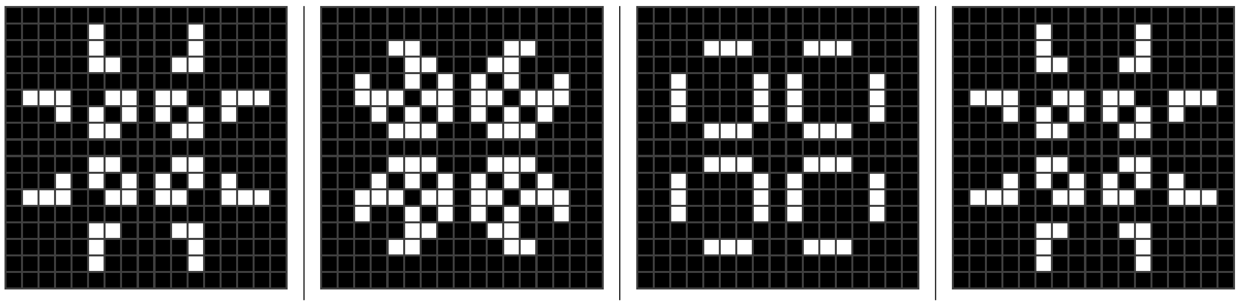

Figure 4.5: Evolution of a CA in Convay's GoL having period 3.

Still Life

As previously mentioned, GoL and CA in general exhibit the presence of patterns that are periodical, we are particularly interested in patterns that do not change over time (having a period of one). In GoL these patterns are called still lifes. We can formalize as a still life a GoL state C, with $c \in [0, 1]$ being a single cell and $N(c)$ the set of neighbours of c, as the following:

$$\forall c \in C \,.\, \left[\left((c = 0) \leftrightarrow (\sum_{n \in N(c)} n \neq 3) \right) \wedge \left((c = 1) \leftrightarrow (\sum_{n \in N(c)} n \in [2, 3]) \right) \right]$$

We are essentially saying that for each dead cell we do not want to trigger GoL rule 4, where having three alive neighbours would make it alive in the next time step. Alive cells need instead to have exactly 2 or 3 alive neighbours so that they maintain their state (rule 2).
The aforementioned constraints are essentially counting, which propositional logic does not allow, so we are left at enumerating all possible valid states; by doing that, we apply the still life constraint on our dataset of choice. As for the previous experiment, the usage of the technique described in section 3.3 was necessary to be able to run the experiment.

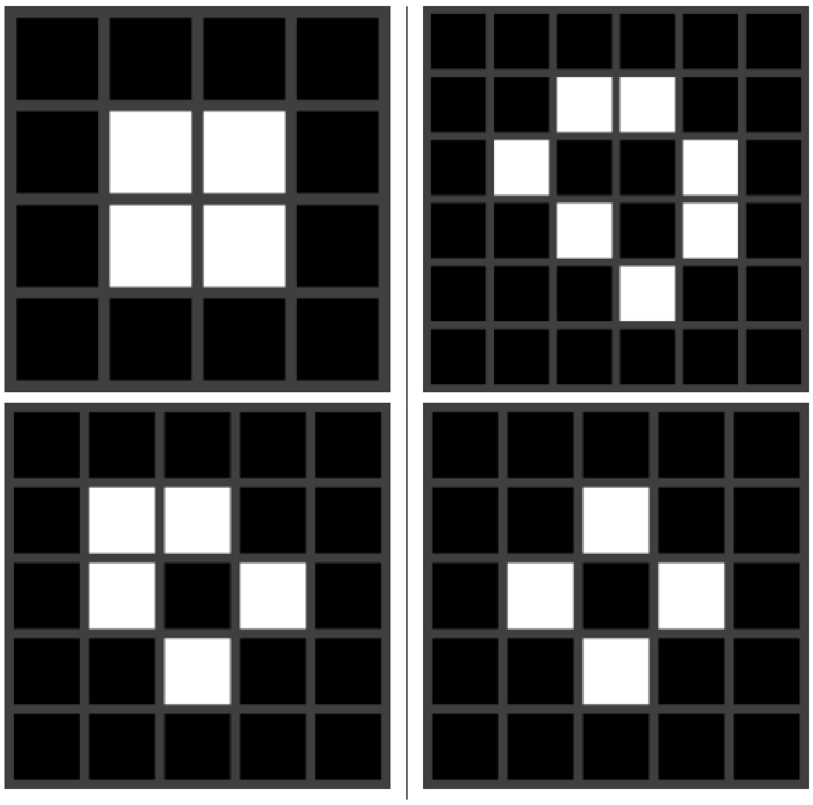

Figure 4.6: Examples of still life in GoL.

4.2.2 Dataset and Architecture

We make use of the widely known MNIST dataset [53], which contains handwritten digits in the form of grey-scale pictures of size 28×28, each sample contains exactly one digit which is roughly centered. To obtain our dataset we started by binarizing the MNIST training set (60000) samples, done that, we used minizinc [65], a constraint modeling language, to pose a COP where the objective was to transform a binarized MNIST sample to a picture respecting the still life rules while minimizing the number of cells (pixels) that had to be flipped from 1 to 0.
An optimizer was then used to search for good solutions while employing a budgeting strategy when optimizing data samples. We performed binary search on the minimum number of alive pixels that we would accept in a solution (this number is, for each sample, starting at half the sum of its alive pixels). For each image, each optimization run had a timeout of 1 second, followed by the update of the minimum threshold according to binary search (rising the minimum requirement if the previous run was successful, lowering it otherwise) and the start of another run, this goes on until the binary search is concluded; the best result is kept.
After processing the whole training set we filtered samples that were too unrecognizable by using a pre-trained MNIST classifier. For each digit category we kept 2000 samples by giving priority to ones having a (correct) higher confidence score by the classifier, resulting in taking samples where said confidence was roughly beyond 0.75. The obtained dataset, labeled MNIST Still Life, is composed of 2000 pictures for each digit, for a total of 20000 training samples.
The generator and discriminator are a deconvolutional and convolutional network respectively. G has three fully connected layers with a ReLU and batch norm layer each, their dimensions are 512, 1024, 3136, then two transposed convolutions follow, both having 4×4 kernels with stride 2; the first one has 256 output channels, is followed by a ReLU and a batch norm, while the output layer is followed by sampling from a concrete distribution using its logits, as in the previous experiment.
The discriminator is comprised of two convolutional layers with 4×4 kernels, stride 1 and 256, 64 output channels; following those, three fully connected layers follow, of size 3136, 1024 and 1 respectively. All layers but the last one make use of leaky ReLUs. We experiment with both a spectrally and a non spectrally normalized discriminator, and we employ the same loss used in the first experiment.

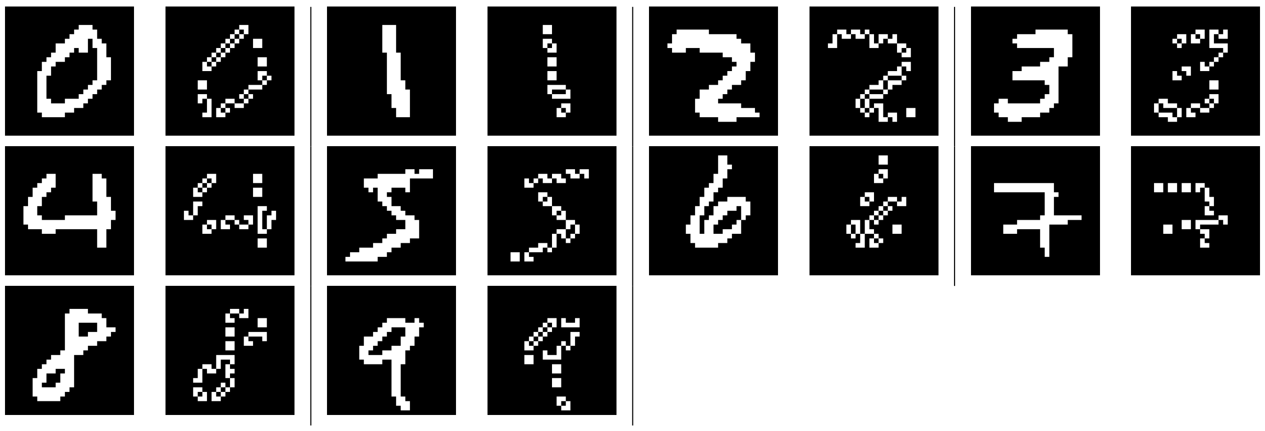

Figure 4.7: Samples from the MNIST dataset and their counterpart in MNIST Still Life.

4.2.3 Results and Discussion

Figure 4.11 shows how the spectral normalization of the discriminator is needed to avoid a strong mode collapse, which is present in all non spectrally normalized experiments. For the spectrally normalized versions, a noticeable correlation between hints of mode collapse (number of ones produced) and, λ, the weight of the SL can be also seen.

Despite the training, the only perfect (valid with respected to the still life constraints) samples are always ones. The other digits sampled from G definitively look similar to the original counterparts, but always have at least one variable/pixel such that the rules of still life are not respected considering its neighbours.

The fact that the only valid outputs come from a single digit category means that the plots in figure 4.8, 4.9 and 4.10 are all related to said category, those statistics were measured by doing five runs for each experiment and taking the average.

The validity trends in 4.8 are essentially a measure of mode collapse or of how many (perfect) ones are being

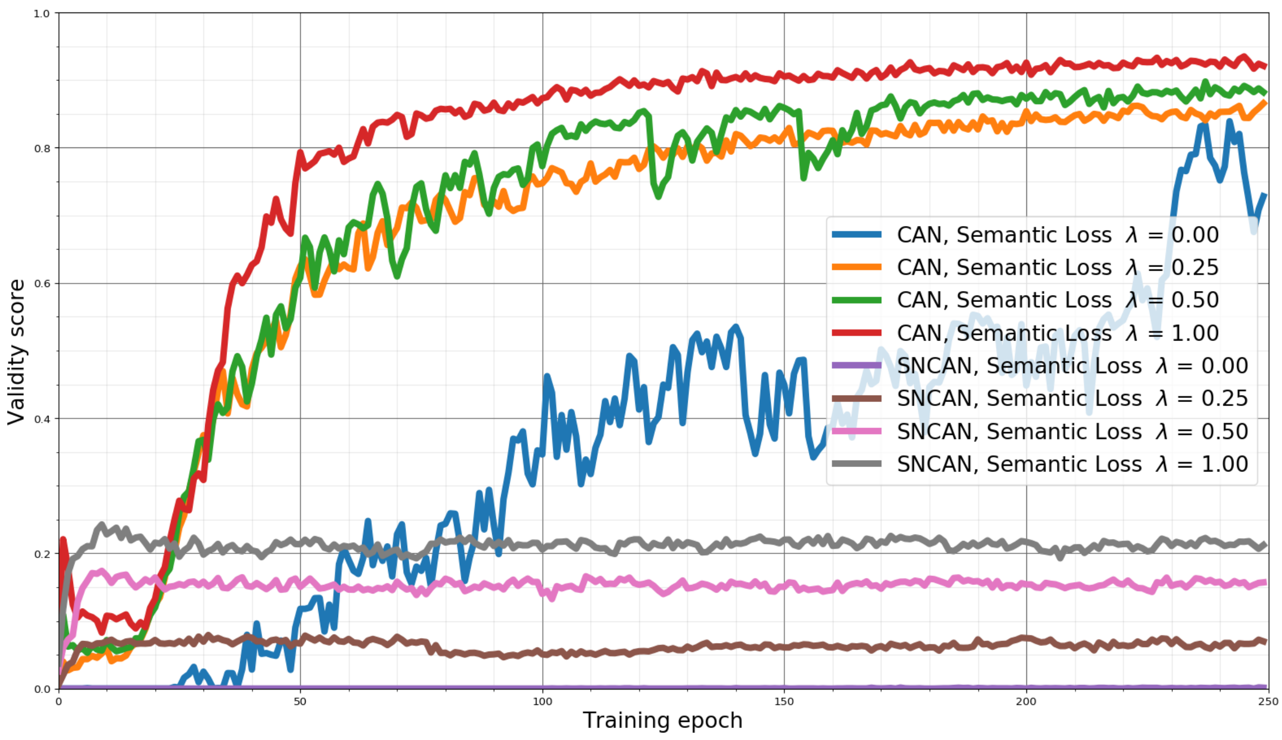

Figure 4.8: Average valid samples over training. In this case validity is a measure of mode collapse given the fact that the only valid objects produced by both architectures were ones.

sampled.

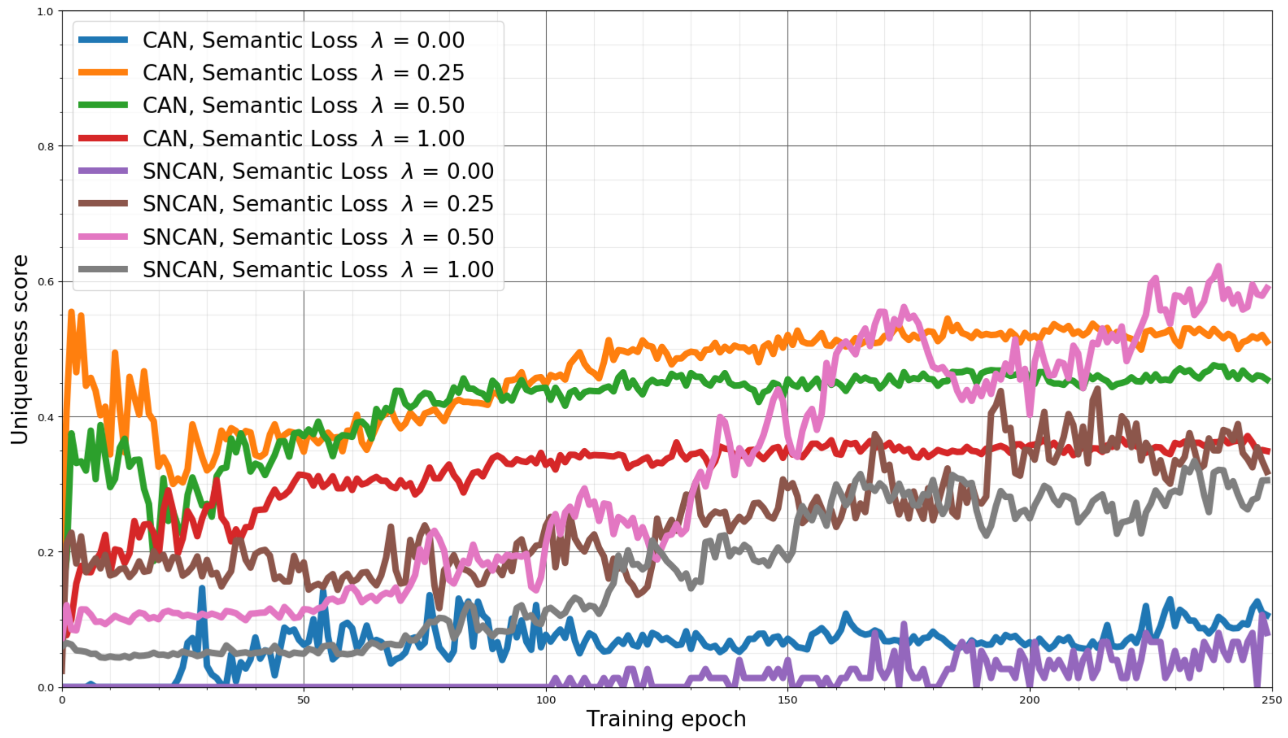

Figure 4.9: Average uniqueness of a batch of samples. Most curiously the spectral normalization is making the network produce different classes of digits, whereas at the single digit level (ones) there seems to be no absolute contrast between SNCAN and CAN in terms of variety.

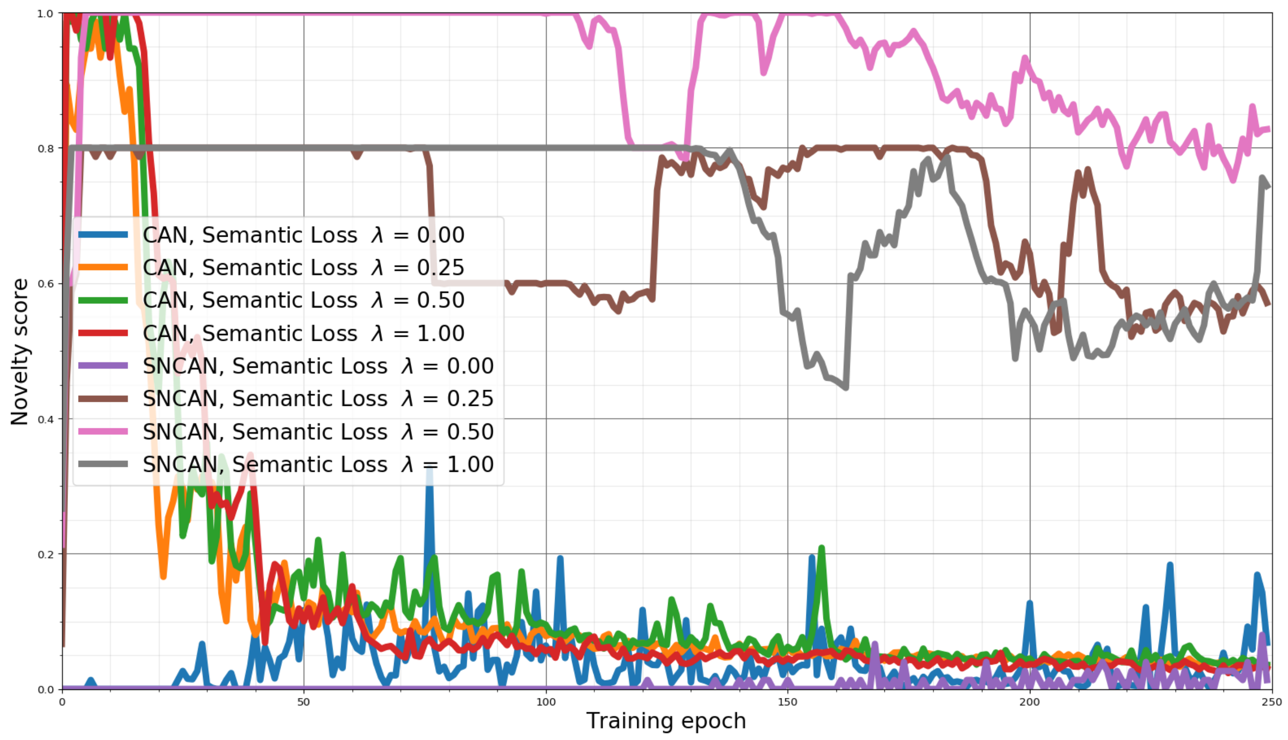

Figure 4.10: Average Novelty of output samples during training.

It is no surprise then that CANs are in the upper end of the plot whereas SNCANs at the bottom, with λ being - naturally - a proxy for a stronger validity.
The novelty trends in figure 4.10 suggest that non spectrally normalized CANs are more likely to output valid objects that are a copy of already existing samples from the dataset. Curiously, this happens with SNCAN with

$\lambda = 0$ as well, this might be related to the fact that said architecture struggles to actually produce any valid object.
We believe the flatness that is present in the novelty trends of SNCAN is due to the fact that SNCAN tends to produce more samples that are valid but that are not really resembling a digit, leading to a high degree of "novelty" that decreases once the adversarial loss helps in correcting said samples.
Interestingly, for all different λ the SNCANs cap out at a certain validity measure (different for each λ) relatively early in the training, showing that while the spectral normalization does indeed stop the mode collapse from happening in our case, the network is not able to produce perfect samples of the remaining digits.
We believe this task is especially hard due the ongoing "global" interactions between variables and constraints. In the first experiment, about polygons, any invalid sample could be simply fixed by flipping parity check bits, given that the constraints were local to half rows/columns. Fixing a still life can instead be much harder because flipping a variable might lead to a cascade of needed changes that easily involve huge portions of the image. This, paired with the fact we are using gradient descent to optimize G on both the semantic and the adversarial loss, can reasonably explain the difficulties that have come to be.

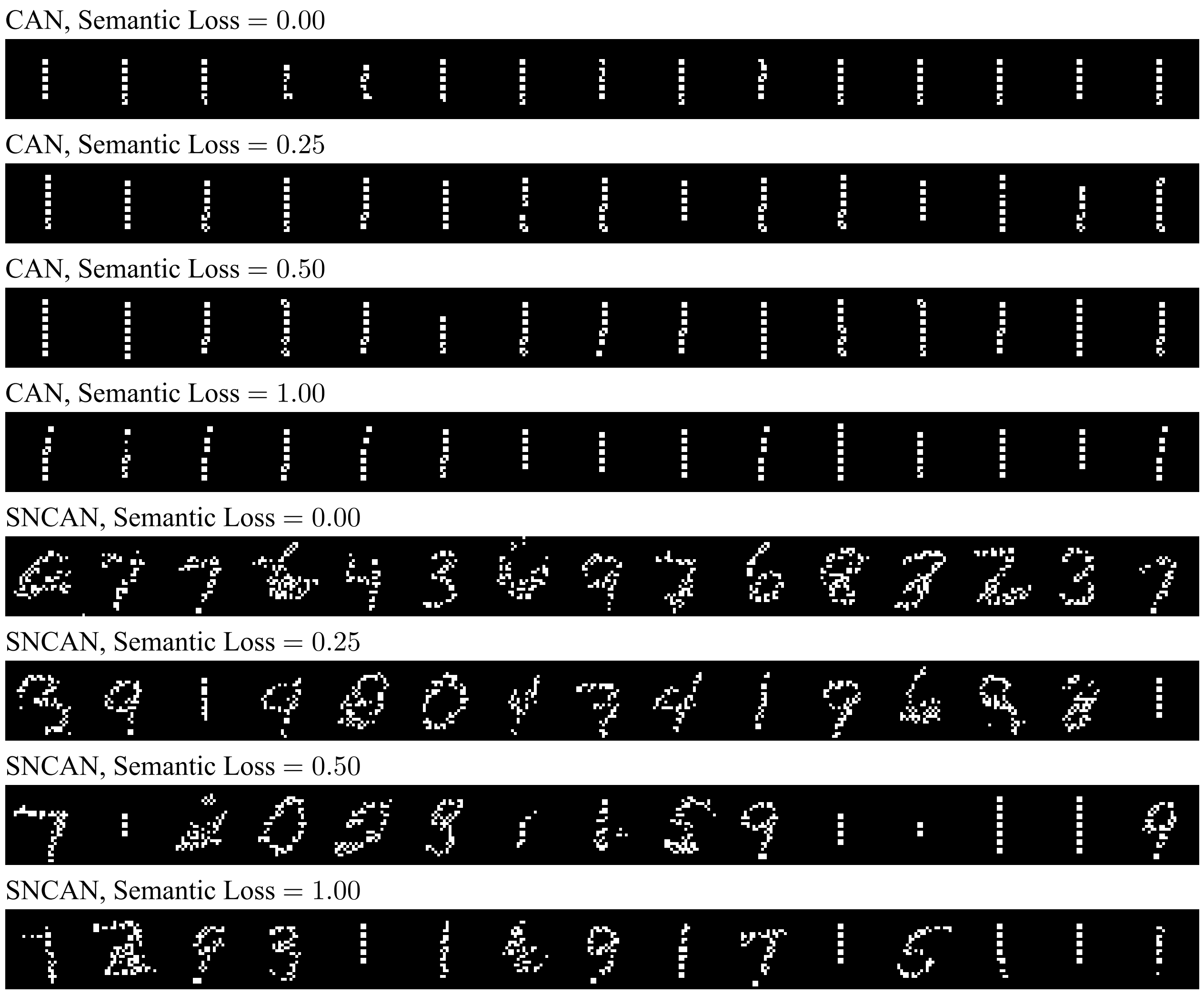
CAN, Semantic Loss = 0.00
CAN, Semantic Loss = 0.25
CAN, Semantic Loss = 0.50
CAN, Semantic Loss = 1.00
SNCAN, Semantic Loss = 0.00
SNCAN, Semantic Loss = 0.25
SNCAN, Semantic Loss = 0.50
SNCAN, Semantic Loss = 1.00

Figure 4.11: Evaluation samples after 250 epochs of training for different values of the SL λ, with and without spectral normalization applied to the discriminator.

Single Digits

Given the inability of the architecture to produce valid items without mode collapsing, we split the dataset into 10 smaller datasets, 1 for each digit, of 2000 samples each. For each of those sets we train a GAN having the architecture and hyper-parameters identical to the experiment on the whole dataset, except for the fact that λ is

always fixed at 1.0 and a run lasts for 500 epochs.
A dropout was applied to the constraints in the image, i.e. for each sample only a random subset (0.05) of the variables was checked for validity and counted towards the SL. Given the technique described in section 3.3, a simple implementation is to set to 1.0 the output of the sub-circuits that were considered dropped out, before applying the "AND" via a product. This dropout was necessary to be able to produce valid/perfect samples.
Curiously, the CAN struggled much more to produce valid objects, despite the number of respected constraints being higher than the spectrally normalized counterpart. We believe this is due to the GAN mode collapsing much faster than the SNCAN, thus leading to D being trained to recognize its valid (but identical) objects as fake, which makes the samples produced by G move around the sample space indefinitely, as described in section 2.2.5. Our intuition is supported by the behaviour shown in figure 4.12.
Figure 4.13 shows that, as explained in 3.3, the network is exploiting sub-patterns that easily lead to a higher validity, at the cost of the results not matching the input distribution properly. Surprisingly, both CAN and SNCAN fail to produce valid digits even on these simplified datasets, showing that the problem is indeed much harder than it seems, at least when producing objects from scratch. GANs (no SL applied) produced no valid samples apart from ones.

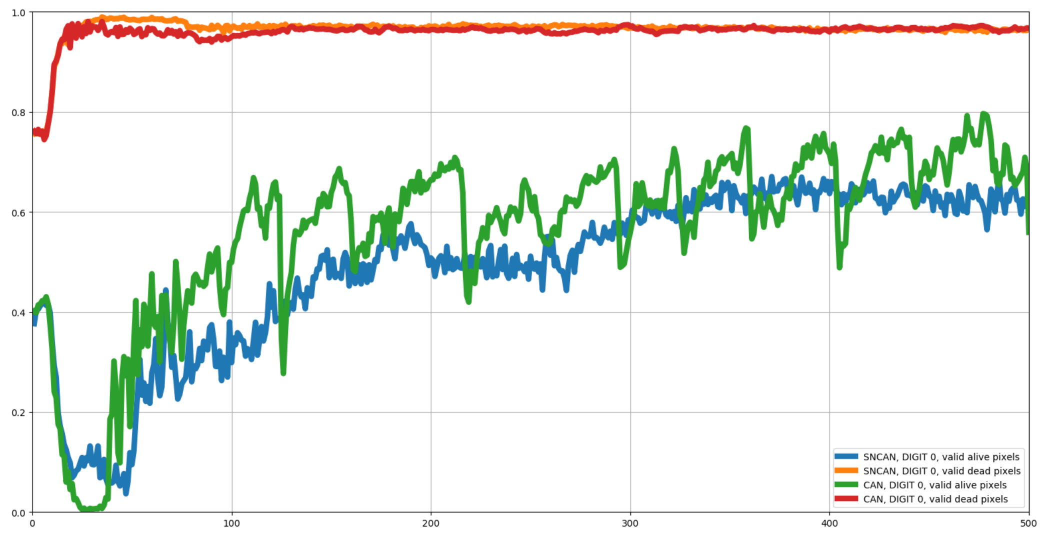

Figure 4.12: Average number of respected constraints for alive and dead pixels separately. A single pixel is considered as respecting the constraints if it would not flip in a successive iteration of the GoL, given its current state and its neighbours. Plot related to the runs on the simplified dataset containing only the single digit 0, for the remaining digits datasets (1-9) it exhibited a similar behaviour. The seemingly good performance on dead pixels is explained by the fact that most of said pixels are the ones surrounding the actually drawn digit, and that totally black areas constitute a valid still life. Moreover, valid dead pixels take less effort to produce, given that they simply need to not have exactly three alive neighbours.

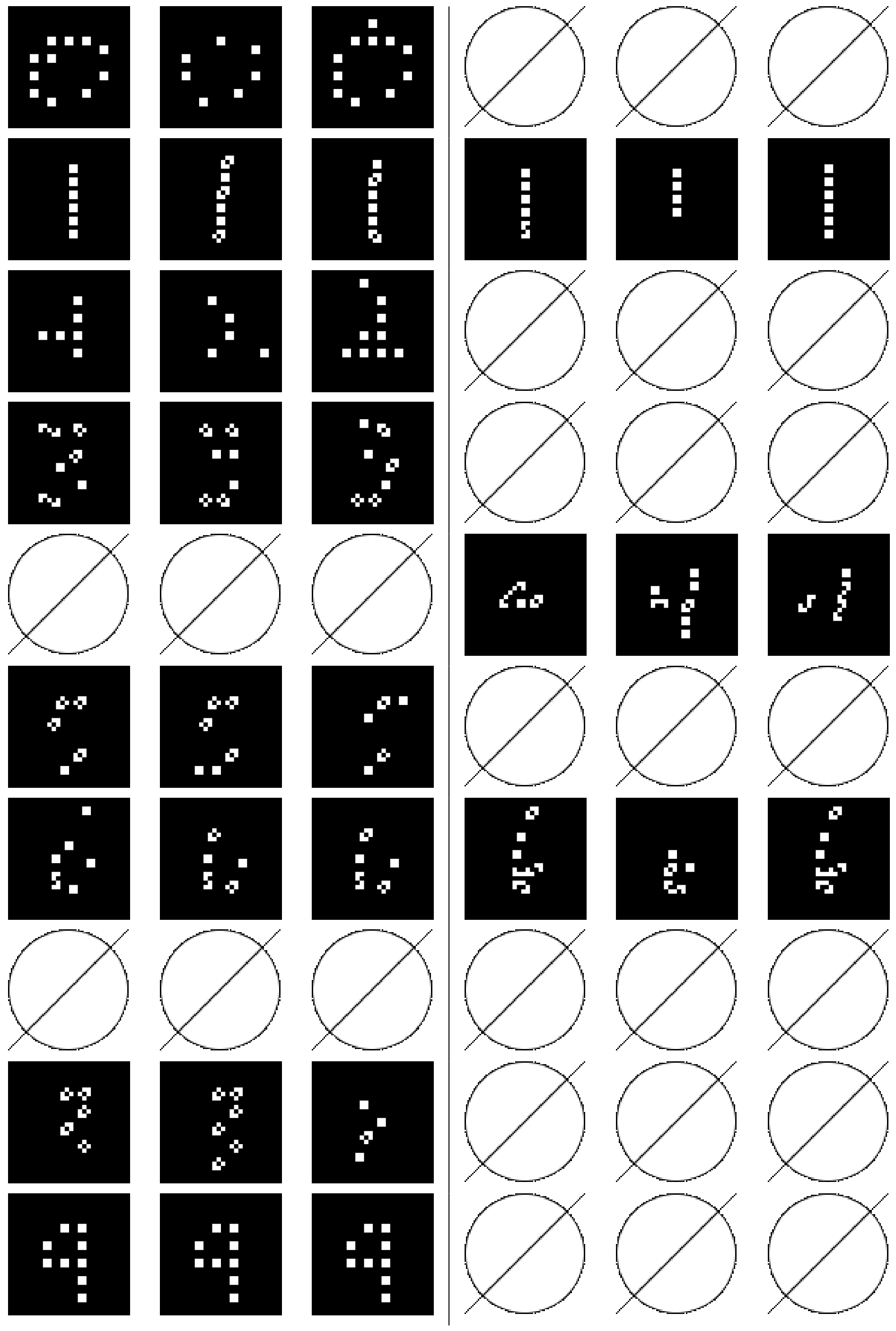

Figure 4.13: Examples of perfect digits produced on the 10 simplified datasets containing 2000 digits of the same type. Left column samples belong to the spectrally normalized CAN, right column samples to the CAN. No picture (empty set symbol) means no valid samples were produced. Repeated images are due to the model not producing different samples among the valid ones.

4.3 Molecule Generation

The third and last benchmark concerns the generation of molecules of pharmaceutical interest. This is an interesting task for different reasons, it is a real-world application of structured objects, molecules in this case, where said objects are actually graphs.

Before potentially hitting the market, a drug must go through a series of steps, each of those may involve years. First we have the drug-discovery phase, then we have pre-clinical and clinical trails, and finally a review for approval, approximately, the expected time for a new drug to complete the whole process is around 12-15 years. The first phase, drug discovery and design, usually requires the proposal of a batch of candidates (thousands) that are then filtered out by the remaining phases. It is a well researched area that can involve long searches over databases with billions of molecules that need to match certain rules, etc, among other scalability issues. This experiment aims to improve on the work of [25] in the task of de novo drug design by applying the SL in a peculiar way.

SMILES (simplified molecular input line entry system) [87] is a popular ASCII character based encoding for compounds, each molecule is represented by a string, this format is well known in the chemistry and pharmaceutical community and research area. SMILES is a rather human friendly solution compared to encoding molecules as vectors, but has the drawback of being a sequence which length is not fixed, even for representing somewhat similar compounds. This poses the obvious problem that, if training a GAN (CAN) can be hard, training a recurrent GAN can be even harder, in terms of hyper-parameters, training time, and overall result.

A matrix (or matrices) representation has the handy feature of being of fixed size and thus representable as the output of a non recurrent GAN. While a fixed length could seem limiting, the given number of atoms and bonds we deal with leads to a combinatorial space of more than 2×10^{31} possible graphs (valid and invalid molecules). Aside from [25], other works have been carried on de novo compound generation, such as ORGAN [41], or [36], we build upon the work of MolGAN because 1) it is a GAN, 2) it easily reaches a high validity, giving us the chance to apply the SL in a more peculiar way compared to the previous two experiments.

4.3.1 Chemistry Refresher

This is an extremely shallow introduction to molecules and atoms that might be needed to better understand what we are attempting to do, and what we cannot do with the SL.

An atom is the smallest constituent unit of ordinary matter that constitutes a chemical element [9], every atom is made of a nucleus, which contains protons and neutrons, and electrons that are bound to said nucleus. Protons have a positive charge, whereas neutrons and electrons have a neutral and negative charge respectively. Having the number of protons and electrons different makes an atom either negative or positively charged.

The chemical element (type) of an atom is given by its atomic number, which is the number of protons in its nucleus.

A molecule consists of an electrically neutral group of atoms that is held together by bonds between said atoms. A bond may happen through atoms sharing a part of their electrons with other atoms, this kind of bonds are called covalent bonds, and are considered “strong bonds”. The sharing of electrons happens in pairs, and the covalency of an atom is defined as the number of electrons it can share when forming said chemical bonds.

Given the scope of this experiment and the dataset we work with, we consider single bonds (one pair of electrons is exchanged), double bonds (two pairs), triple bonds and aromatic bonds. It suffices to say that we cannot consider a generated molecule as valid if it does not respect the covalency of the atoms we work with, carbon, oxygen, nitrogen and fluorine.

4.3.2 Metrics Of Interest

Designing a drug is not limited to finding valid compounds. Along with validity and uniqueness, we consider 4 more domain related metrics, with the last three of them (partially) representing what we are looking for in a drug.

Diversity

The diversity score is a heuristic metric obtained by comparing sub-structures of generated samples against a random subset of molecules from the dataset, to show if patterns happen to be repeated often. The use of this metric stems from the fact that it was used - and defined - by the original MolGAN authors, thus it is an ulterior mean of comparison.

Quantitative Estimate of Druglikeness

Quantitative estimate of druglikeness [13], or **QED**, is an estimator of how likely a molecule is to be a drug, which is one of the key properties for compound selection during drug discovery. QED is introduced as a concept of desirability of a compound, and is used to describe the underlying distribution of molecular properties by combining different metrics and ad hoc functions into a single scalar.
Eight widely used molecular properties concur to the QED by a weighted geometric mean: molecular mass, octanol-water partition coefficient, total hydrogen bond donors, total hydrogen bond acceptors, molecular polar surface area, total rotatable bonds, total aromatic rings and total structural alerts. Each characteristic was selected because it is fast to compute, robust, and it is a partial indicator of drug-likeness.
QED assume values ranging from zero, totally undesirable, to one, extremely desiderable.

Octanol-water partition coefficient

A partition coefficient represents the proportion of a compound in a mixture of two phases that are immiscible and at an equilibrium, it essentially indicates how differently it dissolves in two different phases.
In the pharmacology domain, most often phases are actually solvents, with one being water and the other being hydrophobic, e.g. 1-octanol. In this case, the partition coefficient, now called octanol-water partition coefficient (**logP**), is an assessment on how hydrophilic and hydrophobic a substance is.
An indicator of how water loving or hating a substance is can be used, for example, in determining the distribution of drugs in a person's body, water loving drugs (high octanol/water partition coefficient) will be mostly found in aqueos regions of the body, whereas hydrophobic substances will be in areas lacking water.
Said coefficient "strongly affects how easily the drug can reach its intended target in the body, how strong an effect it will have once it reaches its target, and how long it will remain in the body in an active form" [10].
For this reason, logP is one of the deciding factors during pre-clinical drug discovery to either keep or discard a compound.

Synthetic accessibility score

Even if we find a compound which has nice properties (QED, logP), it being valid, in the sense that it can exist because it respects the rules of physics, chemistry, etc, might not be enough. Said potential drug might not even pass onto the next stage of the process of making a drug. This is because we must also take into consideration how feasible it is to synthesize (make) said drug.
The Synthetic accessibility score (**SAS**) [31], is a heuristic developed to offer an estimate on the ease of synthesis, it ranges between 1 (very easy to make) to 10 (very difficult to make). SAS is computed by considering the fragments of a molecule as having a positive or negative impact on synthesizability. For each fragment, said contribution is determined by comparing it against the fragments of one million known compounds from the PubChem database [48] and then normalized with respect to the total fragments in the molecule.
Compounds are penalized by their complexity by considering the molecule size or the presence of non-standard structural features, like large rings, non-standard ring fusions, and stereocomplexity.

4.3.3 Architecture

The architecture we start from, MolGAN, building upon the code which can be found at [18], is different from the usual GAN, and deserves its own section. First, it is not a two players game anymore, because of the presence of a reward network R, second, it makes use of graph convolution layers to better elaborate on the compounds at hand.
The reward network R, just like the discriminator D, takes as input the output of the generator G or real data, instead of judging samples as fake or real, it has to determine the value of the metric it is optimized for. To compute the different target metrics for both fake and real data we make use of rdkit [52]. Samples are in a fixed size representation composed of 2 matrices, one for mapping (9) nodes to a possible label among 5, the other being the indirect transition matrix between said nodes, each edge can assume one of 5 labels.
R is trained using the mean squared error between the predicted value and the output of the external system; while G is trained against D (and vice-versa) with the Wasserstein loss (plus gradient penalty). G is also

trained to maximize the expected metric(s) score measured by R by using deep deterministic policy gradient, a reinforcement learning objective.
Invalid molecules receive a QED, SA and logP score of 0.0, two different molecules are considered duplicates if invalid; this has the extremely important implication that **optimizing for any of these metrics is also an implicit optimization of validity**.
All three networks are trained simultaneously from the ground-up, although a pre-trained version of R could be used. As the authors of MolGAN report, the loss that G receives from R, the reward loss, should not be used right from the start due to the generator easily diverging if R has not been trained for at least some epochs. As for hyper-parameters, we searched over a grid to try to match the original authors' results as best as possible, in their same manner we first train for 150 epochs without activating the reward loss on G, then let the training continue for **up to** 300 epochs, deactivating the adversarial loss and activating the reward loss in its place.
The SL, applied to G (and later explained), is active for the whole training, and is added on top of the other losses (the reward loss is still used).
The exact number of layers and parameters of G, D and R can be found in [25, 18].

4.3.4 Dataset

Lack of data is not really a problem for molecules. GDB-17 [72] is a chemical database containing "only" molecules having up to 17 atoms, it contains a total of 166.4 billions of entries. QM9 [70], a subset of GDB-17, limits itself to having a maximum of 9 atoms per molecule, with atoms either being carbon (C), oxygen(O), nitrogen (N) and fluorine (F), it contains 133885 samples. We make use of an unnamed subset of QM9 used in [41, 25], which contains 5k compounds.
Said molecules are parsed and brought to an indirect graph representation, a first matrix of shape 9×5 contains atoms and their label (no atom, C, O, N, F), whereas a $9 \times 9 \times 5$ symmetrical matrix contains information about the bonds between said atoms (no bond, single bond, double bond, triple bond, aromatic bond). Each node can obviously assume only 1 label, the same stands for edges.

4.3.5 Applied Constraints

The defining metrics of this task are something that we cannot model with SL. Validity, at a minimum, involves counting, given by checking for the correctness of bonds with respect to the valency of atoms. QED, SA and logP are obtained by aggregating the contribution from different fragments of each molecule, the estimation of chemical properties such as molecular mass, and so on. These metrics, together with validity, are out of the scope of the SL.
The SL can be used to impose arbitrary rules, and not only what can be intuitively seen as a constraint. As we have said, it can be used to impose a relationship among whatever group of variables in our computation graph. Specifically, we use the formula to bind generated samples to latent variables to leverage the SL in order to guide the GAN towards the production of more unique objects. For each sample, a portion of its latent vector whose values can be interpreted as probabilities is used to trigger the activation of constraints that promote the presence of specific atoms in the generated molecule. Given the 4 types of atoms that we deal with (ignoring the padding/no atom label), the last 4 dimensions of the latent vector are considered to be the probability of the related specific type of atom appearing in the molecule, no matter the position.
Let z' be the portion of the input noise z used to impose the presence of a type of atom, A the 9×5 matrix mapping atoms to a label, assuming the first dimension of each label vector (a row in A) is the one related to the padding/no atom label, we could write the aforementioned constraint in the following way:

$$\begin{gathered} \forall i \in \{1,2,3,4\} \\ z'[i] \iff \bigvee_{a=1}^{9} A[a][1+i] \end{gathered} \tag{4.3}$$

The advantage is twofold, we can boost diversity and have more control on the generative process by mapping latent dimensions to specific characteristics of the samples. To empirically prove the soundness of this method we take MolGAN and augment it with this kind of SL, comparing results while using the same hyper-parameters.

The SL is added on top of the other losses, given that we cannot hope to model validity or other chemical properties with it. The point is to still exploit R to optimize said properties while using the SL to improve the uniqueness of results, something which MolGAN is lacking.
Given the simplicity of these constraints, no approximation (3.3) had to be used, the resulting overhead was negligible with respect to the original architecture.

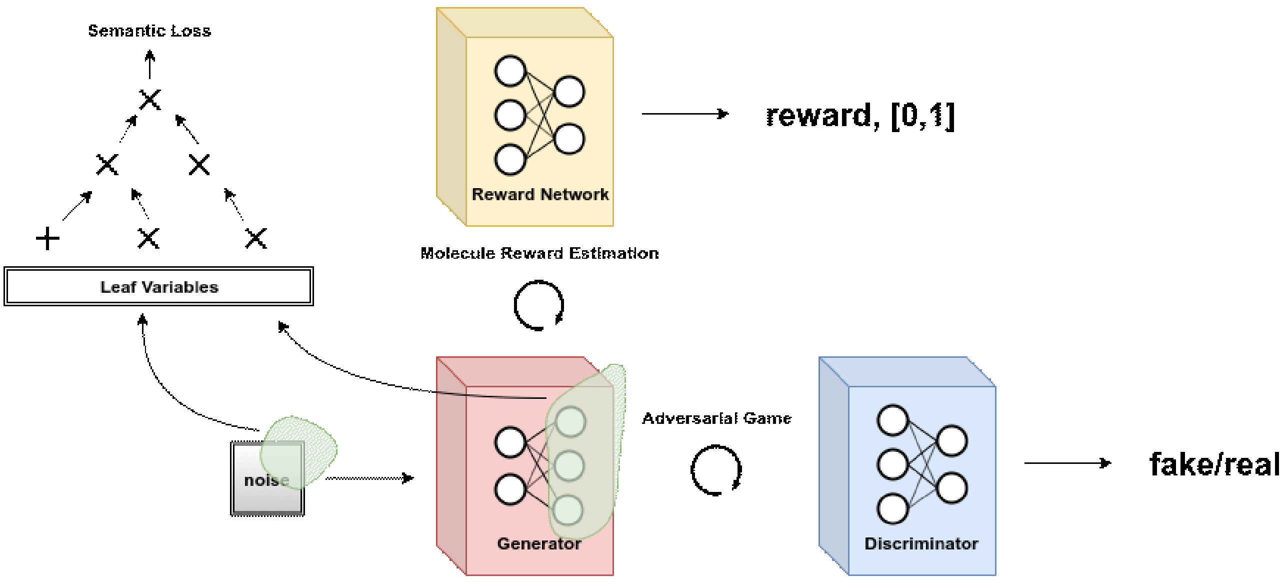

Figure 4.14: The MolGAN architecture augmented with the SL.

4.3.6 Results And Discussion

We augment MolGAN with our constraints conditioned on the latent variables, starting from the first epoch. This is done for two experiments: one where we optimize R (and thus G) on QED, SA and logP, the second where we optimize it on the uniqueness of the batch. All metrics are normalized between 0 and 1, with 1 being the best score reachable by that metric. In the first case R optimizes for the product of the three chemical properties. A uniqueness of 1 means that the 5000 evaluation molecules are all different.
Given its importance, we restate the fact that optimizing any metric among QED, SA, logP and uniqueness is also a proxy for boosting validity, given that invalid molecules are assigned a score of 0 and count as duplicates with other invalid ones.
Experimental settings, architecture and evaluation metrics are identical to [25] except for the introduction of the SL, meaning that when the SL is not used G is trained with the adversarial loss for the first 150 epochs, then the reward loss takes its place.
In contrast, when the SL is active we re-scale the adversarial loss of G by 0.10 and assign the SL a λ equal to 0.9, these values were found by searching over a reasonable amount of values. Due to the Wasserstein loss not being logarithmic, we found it easier to search for the λ parameter by not applying the negative logarithm component and use $1 - WMC$ for the loss instead. Note that, when the SL is used, after 150 epochs the reward loss takes the place of the adversarial loss, while the SL remains active.

Reward for	**Semantic loss**	**validity**	**uniqueness**	**diversity**	**QED**	**SA**	**logP**
QED + SA + logP	False	97.4	2.4	91.0	47.0	84.0	65.0
	True	96.59 (2.52)	2.52 (0.25)	98.81 (2.04)	51.75 (1.55)	90.74 (5.48)	73.62 (1.14)
uniqueness	False	99.19 (0.19)	4.75 (2.35)	66.87 (6.39)	50.10 (1.68)	37.26 (9.55)	32.89 (4.55)
	True	99.27 (0.16)	17.41 (3.07)	91.28 (2.87)	41.54 (1.89)	43.99 (8.38)	33.94 (2.16)

Table 4.1: Results of using the semantic loss on the MolGAN architecture. The first row refers to the results reported in the MolGAN paper. Experiments were run 8 times each (rows 2, 3, 4) to obtain mean and std values, the latter in parentheses. Statistics are collected over 5k samples.

Optimizing for QED, SA, logP

This first experiment is a direct comparison with [25], **after 150 epochs the adversarial loss is deactivated and the reward loss comes into play.** As in [25], the training is stopped once the uniqueness falls below 0.02, and the model from the previous epoch is kept. Our diversifying constraint is employed with the intent of slowing down the mode collapse of the network, so that the reward loss can be further optimized before hitting the uniqueness threshold.
Table 4.1 shows the improvement on the desirable properties of the generated samples.

Optimizing for uniqueness

In this setting the reward measured by R is proportional to the uniqueness of the generated batch, in order to boost it. Here training stops once validity reaches 0.99. Again, the reward loss is activated only past 150 epochs, while the SL is active right from the start.
Results, as shown in table 4.1, suggest that the uniqueness metric modeled by R is a good proxy for validity but is not informative enough to lead G to more diversified batches. Coupling the reward loss with the SL offers a significant boost to uniqueness while retaining the validity of the produced objects.

5 Conclusions

We introduced CANs, constrained adversarial networks, a framework to encode prior knowledge into a deep learning generative architecture, in our case, GANs. CANs stem from the successful coupling of discriminative NNs and the SL, which prompted this work.
The role of the SL is to penalize the GAN when it allocates probability mass to objects that we consider invalid with respect to an arbitrary definition. This way we can direct the network towards a behaviour that it would not have normally, due to limits in the architecture, noisy data or lack thereof.
A positive characteristic of the SL is that the encoded knowledge is "baked" into the network, this means that at inference time we incur in the same cost of a normal GAN. The SL can have either a trivial or significant overhead during training, depending on the constraints. We provided a technique to approximate said loss so that it could actually be used in cases where its use would have been impossible otherwise.
We showed potential issues or limits in CANs (and perhaps GANs in general) by highlighting the effects that complex and tightly coupled relationship over variables have on the outcome of training.
We empirically proved that other networks aside from classical GANs can be transformed into CANs, specifically, we improved the performance of a network based on a three players adversarial game, MolGAN, said network was already providing competitive or state of the art results in the task it was designed for.
To synthesize, we showed and empirically proved how it is possible to impose arbitrary propositional logic constraints over GANs to make them better approximate a target data distribution.

5.1 Limits

The limits of CANs are mostly related to the limits of the SL itself, being based on that. First, there is the propositional logic limit, we obviously can't formally and exactly include what is out of scope of propositional logic without grounding each variable and then falling into intractability.
The second limit is related to transforming a propositional knowledge base into an SDD, this first requires a pass to compile it to a statement in CNF or DNF, then another pass to get the SDD. The main issue is that, again, the upper bound is intractability.
Even if we end up with an arithmetic circuit that is tractable with respect to the starting proposition, it might be too large to properly run, be it because of its memory footprint or because of the overhead of evaluating it. While these two issues might be non-existent with most applications of SDDs, training a NN will realistically require the circuit to be on GPU and its evaluation time to be very low given the high number of passes we perform.
At last, we consider a problem that is perhaps related to optimization in general, to be fair, this could simply be considered an issue of semantics (what we are telling the network with the provided losses) rather than a defect. The problem is that satisfying a Boolean formula in one way is generally worth the same as all other possible solutions, at least that is what we are telling with the SL. This means that given the constraint $A \implies B$, which is equal to $\neg A \vee B$, the network could collapse to samples where A is always false, if it happens to be the easiest thing to do between the two. The adversarial loss could help in this regard, but we have shown that this is not always the case, especially if the target distribution contains samples that are much more easy to be made valid.

5.2 Future Work

It goes without saying that applying CANs to different domains and datasets is something to be considered to empirically prove that the framework generalizes.
A first, reasonable future work would be using another NN as an approximate and differentiable function of the imposed constraints, this would be useful to model intractable constraints while allowing the SL to be applied to the trained network output variables in a general purpose. One example would be to use multi label classifiers where the labels are organized in an ontology, such as in [27], and use the SL to help the network understand the relationships among the different labels.

Another research direction would be using the arithmetic circuit (or circuits) as a form of feature engineering. The WMC(s) would be passed to the discriminator together with the samples, this would maintain the circuit differentiable while hopefully making it so that the adversarial loss could also be the carrier of information about constraints, instead of having a separate loss related to validity.
One last proposed idea would be applying the constraint described in section 4.3.5 to frequent item sets that are mined from a discrete dataset, after finding n recurring patterns we could have n latent variables be in an equivalence relationship with the presence or lack thereof of their associated pattern.

Bibliography

[1] URL: https://www.mensa.org/.

[2] URL: http://reasoning.cs.ucla.edu/sdd/.

[3] URL: https://www.scientificamerican.com/magazine/sa/1970/10-01/.

[4] Oct 2019. Page Version ID: 919772681. URL: https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=919772681.

[5] Oct 2019. Page Version ID: 920095111. URL: https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=920095111.

[6] Sep 2019. Page Version ID: 916120382. URL: https://en.wikipedia.org/w/index.php?title=Cellular_automaton&oldid=916120382.

[7] Sep 2019. Page Version ID: 918260007. URL: https://en.wikipedia.org/w/index.php?title=Conway27s_Game_of_Life&oldid=918260007.

[8] Oct 2019. Page Version ID: 919088061. URL: https://en.wikipedia.org/w/index.php?title=Von_Neumann_universal_constructor&oldid=919088061.

[9] Sep 2019. Page Version ID: 917358604. URL: https://en.wikipedia.org/w/index.php?title=Atom&oldid=917358604.

[10] Sep 2019. Page Version ID: 917323124. URL: https://en.wikipedia.org/w/index.php?title=Partition_coefficient&oldid=917323124.

[11] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv:1701.07875 [cs, stat]*, Jan 2017. arXiv: 1701.07875. URL: http://arxiv.org/abs/1701.07875.

[12] arthur. *art-ai/pypsdd*. May 2019. URL: https://github.com/art-ai/pypsdd.

[13] G. Richard Bickerton, Gaia V. Paolini, Jérémy Besnard, Sorel Muresan, and Andrew L. Hopkins. Quantifying the chemical beauty of drugs. *Nature Chemistry*, 4(2), Jan 2012. doi:10.1038/nchem.1243.

[14] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis. *arXiv:1809.11096 [cs, stat]*, Sep 2018. arXiv: 1809.11096. URL: http://arxiv.org/abs/1809.11096.

[15] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv:1312.6203 [cs]*, Dec 2013. arXiv: 1312.6203. URL: http://arxiv.org/abs/1312.6203.

[16] Hans Kleine Buning and Theodor Lettmann. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, Aug 1999.

[17] Nicola De Cao. Deep generative models for graphs - vaes, gans, and reinforcement learning for de novo drug discovery. page 45.

[18] Nicola De Cao. *nicola-decao/MolGAN*. Oct 2019. URL: `https://github.com/nicola-decao/MolGAN`.

[19] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, April 2008. URL: `http://dx.doi.org/10.1016/j.artint.2007.11.002`, `doi:10.1016/j.artint.2007.11.002`.

[20] Yang Chen, Yu-Kun Lai, and Yong-Jin Liu. Cartoongan: Generative adversarial networks for photo cartoonization. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, Jun 2018. URL: `https://ieeexplore.ieee.org/document/8579084/`, `doi:10.1109/CVPR.2018.00986`.

[21] Arthur Choi and Adnan Darwiche. Sdd advanced-user manual version 2.0. page 36.

[22] Balázs Csanád Csáji. Approximation with artificial neural networks. 2001.

[23] Adnan Darwiche. Sdd: a new canonical representation of propositional knowledge bases. AAAI Press, Jul 2011. URL: `http://dl.acm.org/citation.cfm?id=2283516.2283536`, `doi:10.5591/978-1-57735-516-8/IJCAI11-143`.

[24] Tim R. Davidson, Luca Falorsi, Nicola De Cao, Thomas Kipf, and Jakub M. Tomczak. Hyperspherical variational auto-encoders. *arXiv:1804.00891 [cs, stat]*, Apr 2018. arXiv: 1804.00891. URL: `http://arxiv.org/abs/1804.00891`.

[25] Nicola De Cao and Thomas Kipf. Molgan: An implicit generative model for small molecular graphs. *arXiv:1805.11973 [cs, stat]*, May 2018. arXiv: 1805.11973. URL: `http://arxiv.org/abs/1805.11973`.

[26] Thomas Demeester, Tim Rocktäschel, and Sebastian Riedel. Lifted rule injection for relation embeddings. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov 2016. `doi:10.18653/v1/D16-1146`.

[27] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[28] Ivan Donadello, Luciano Serafini, and Artur d'Avila Garcez. Logic tensor networks for semantic image interpretation. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, Aug 2017. URL: `https://www.ijcai.org/proceedings/2017/221`, `doi:10.24963/ijcai.2017/221`.

[29] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv:1603.07285 [cs, stat]*, Mar 2016. arXiv: 1603.07285. URL: `http://arxiv.org/abs/1603.07285`.

[30] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P Adams. *Convolutional Networks on Graphs for Learning Molecular Fingerprints*. Curran Associates, Inc., 2015. URL: `http://papers.nips.cc/paper/5954-convolutional-networks-on-graphs-for-learning-molecular-fingerprints.pdf`.

[31] Peter Ertl and Ansgar Schuffenhauer. Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions. *Journal of Cheminformatics*, 1(1):8, Jun 2009. `doi:10.1186/1758-2946-1-8`.

[32] Abram L. Friesen and Pedro M. Domingos. The sum-product theorem: A foundation for learning tractable models. *CoRR*, abs/1611.03553, 2016. URL: `http://arxiv.org/abs/1611.03553`, `arXiv:1611.03553`.

[33] Karl Pearson F.R.S. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11), Nov 1901. `doi:10.1080/14786440109462720`.

[34] Anna Frühstück, Ibraheem Alhashim, and Peter Wonka. TileGAN: Synthesis of large-scale non-homogeneous textures. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 38(4), 2019.

[35] Malik Ghallab, Dana Nau, and Paolo Traverso. Elsevier, May 2004.

[36] Rafael Gomez-Bombarelli, Jennifer N. Wei, David Duvenaud, José Miguel Hernandez-Lobato, Benjamin Sanchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D. Hirzel, Ryan P. Adams, and Alan Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science*, 4(2), Feb 2018. `doi:10.1021/acscentsci.7b00572`.

[37] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[38] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv:1406.2661 [cs, stat]*, Jun 2014. arXiv: 1406.2661. URL: `http://arxiv.org/abs/1406.2661`.

[39] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv:1410.5401 [cs]*, Oct 2014. arXiv: 1410.5401. URL: `http://arxiv.org/abs/1410.5401`.

[40] Aditya Grover, Aaron Zweig, and Stefano Ermon. Graphite: Iterative generative modeling of graphs. *arXiv:1803.10459 [cs, stat]*, Mar 2018. arXiv: 1803.10459. URL: `http://arxiv.org/abs/1803.10459`.

[41] Gabriel Lima Guimaraes, Benjamin Sanchez-Lengeling, Carlos Outeiral, Pedro Luis Cunha Farias, and Alan Aspuru-Guzik. Objective-reinforced generative adversarial networks (organ) for sequence generation models. *arXiv:1705.10843 [cs, stat]*, May 2017. arXiv: 1705.10843. URL: `http://arxiv.org/abs/1705.10843`.

[42] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. *arXiv:1704.00028 [cs, stat]*, Mar 2017. arXiv: 1704.00028. URL: `http://arxiv.org/abs/1704.00028`.

[43] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *arXiv:1706.08500 [cs, stat]*, Jun 2017. arXiv: 1706.08500. URL: `http://arxiv.org/abs/1706.08500`.

[44] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv:1503.02531 [cs, stat]*, Mar 2015. arXiv: 1503.02531. URL: `http://arxiv.org/abs/1503.02531`.

[45] Zhiting Hu, Xuezhe Ma, Zhengzhong Liu, Eduard Hovy, and Eric Xing. Harnessing deep neural networks with logic rules. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2410–2420, Berlin, Germany, August 2016. Association for Computational Linguistics. `doi:10.18653/v1/P16-1228`.

[46] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *arXiv:1611.07004 [cs]*, Nov 2016. arXiv: 1611.07004. URL: `http://arxiv.org/abs/1611.07004`.

[47] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *arXiv:1812.04948 [cs, stat]*, Dec 2018. arXiv: 1812.04948. URL: `http://arxiv.org/abs/1812.04948`.

[48] Sunghwan Kim, Paul A. Thiessen, Evan E. Bolton, Jie Chen, Gang Fu, Asta Gindulyte, Lianyi Han, Jane He, Siqian He, Benjamin A. Shoemaker, and et al. Pubchem substance and compound databases. *Nucleic Acids Research*, 44(Database issue), Jan 2016. `doi:10.1093/nar/gkv951`.

[49] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv:1609.02907 [cs, stat]*, Sep 2016. arXiv: 1609.02907. URL: `http://arxiv.org/abs/1609.02907`.

[50] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*. AAAI Press, 2014. URL: `http://dl.acm.org/citation.cfm?id=3031929.3031996`.

[51] Matt J. Kusner, Brooks Paige, and JosÃ© Miguel Hernandez-Lobato. Grammar variational autoencoder. *arXiv:1703.01925 [stat]*, Mar 2017. arXiv: 1703.01925. URL: `http://arxiv.org/abs/1703.01925`.

[52] Greg Landrum. Rdkit: Open-source cheminformatics. URL: `http://www.rdkit.org`.

[53] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL: `http://yann.lecun.com/exdb/mnist/` [cited 2016-01-14 14:24:11].

[54] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*. Springer-Verlag, 1999. URL: `http://dl.acm.org/citation.cfm?id=646469.691875`.

[55] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv:1511.05493 [cs, stat]*, Nov 2015. arXiv: 1511.05493. URL: `http://arxiv.org/abs/1511.05493`.

[56] Marco Lippi and Paolo Frasconi. Prediction of protein β-residue contacts by markov logic networks with grounding-specific weights. *Bioinformatics*, 25(18):2326–2333, 2009.

[57] Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. Are gans created equal? a large-scale study. *arXiv:1711.10337 [cs, stat]*, Nov 2017. arXiv: 1711.10337. URL: `http://arxiv.org/abs/1711.10337`.

[58] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

[59] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. URL: `https://doi.org/10.7717/peerj-cs.103`, `doi:10.7717/peerj-cs.103`.

[60] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. *Distributed Representations of Words and Phrases and their Compositionality*. Curran Associates, Inc., 2013. URL: `http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf`.

[61] Pasquale Minervini, Matko Bosnjak, Tim Rocktäschel, and Sebastian Riedel. Towards neural theorem proving at scale. *arXiv:1807.08204 [cs]*, Jul 2018. arXiv: 1807.08204. URL: `http://arxiv.org/abs/1807.08204`.

[62] Tom M. Mitchell. *Machine learning*. McGraw-Hill series in computer science. McGraw-Hill, international ed., [reprint.] edition, 20.

[63] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. *arXiv:1802.05957 [cs, stat]*, Feb 2018. arXiv: 1802.05957. URL: `http://arxiv.org/abs/1802.05957`.

[64] Jason Naradowsky and Sebastian Riedel. Modeling exclusion with a differentiable factor graph constraint. page 5.

[65] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*. Springer-Verlag, 2007. event-place: Providence, RI, USA. URL: `http://dl.acm.org/citation.cfm?id=1771668.1771709`.

[66] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv:1811.03378 [cs]*, Nov 2018. arXiv: 1811.03378. URL: `http://arxiv.org/abs/1811.03378`.

[67] Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In *AAAI*, 2008.

[68] Thammanit Pipatsrisawat and Adnan Darwiche. A lower bound on the size of decomposable negation normal form. In *AAAI*, 2010.

[69] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv:1511.06434 [cs]*, Nov 2015. arXiv: 1511.06434. URL: `http://arxiv.org/abs/1511.06434`.

[70] Raghunathan Ramakrishnan, Pavlo O. Dral, Matthias Rupp, and O. Anatole von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data*, 1(1), Aug 2014. `doi:10.1038/sdata.2014.22`.

[71] Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. Injecting logical background knowledge into embeddings for relation extraction. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

[72] Lars Ruddigkeit, Ruud van Deursen, Lorenz C. Blum, and Jean-Louis Reymond. Enumeration of 166 billion organic small molecules in the chemical universe database gdb-17. *Journal of Chemical Information and Modeling*, 52(11), Nov 2012. `doi:10.1021/ci300415d`.

[73] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988. URL: `http://dl.acm.org/citation.cfm?id=65669.104451`.

[74] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *arXiv:1606.03498 [cs]*, Jun 2016. arXiv: 1606.03498. URL: `http://arxiv.org/abs/1606.03498`.

[75] Bidisha Samanta, Abir De, Gourhari Jana, Pratim Kumar Chattaraj, Niloy Ganguly, and Manuel Gomez-Rodriguez. Nevae: A deep generative model for molecular graphs. *arXiv:1802.05283 [physics, stat]*, Feb 2018. arXiv: 1802.05283. URL: `http://arxiv.org/abs/1802.05283`.

[76] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. *arXiv:1703.06103 [cs, stat]*, Mar 2017. arXiv: 1703.06103. URL: `http://arxiv.org/abs/1703.06103`.

[77] Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. *arXiv:1802.03480 [cs]*, Feb 2018. arXiv: 1802.03480. URL: `http://arxiv.org/abs/1802.03480`.

[78] David Smith and Vibhav Gogate. Loopy belief propagation in the presence of determinism. In *Artificial Intelligence and Statistics*, Apr 2014. URL: `http://proceedings.mlr.press/v33/smith14.html`.

[79] Sam Snodgrass. Controllable procedural content generation via constrained multi-dimensional markov chain sampling.

[80] Gustav Sourek, Vojtech Aschenbrenner, Filip Zelezny, Steven Schockaert, and Ondrej Kuzelka. Lifted relational neural networks: Efficient learning of latent relational structures. *Journal of Artificial Intelligence Research*, 62:69–100, 2018.

[81] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[82] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. *arXiv:1808.01974 [cs, stat]*, Aug 2018. arXiv: 1808.01974. URL: `http://arxiv.org/abs/1808.01974`.

[83] Julian Togelius, Mike Preuss, Nicola Hochstrate, Simon Wessing, Johan Hagelback, Georgios Yannakakis, and Corrado Grappiolo. Controllable procedural map generation via multiobjective evolution. 14, Jun 2013.

[84] vdumoulin. *vdumoulin/conv_arithmetic*. Oct 2019. URL: `https://github.com/vdumoulin/conv_arithmetic`.

[85] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M. Lucas, Adam Smith, and Sebastian Risi. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. *arXiv:1805.00728 [cs]*, May 2018. arXiv: 1805.00728. URL: `http://arxiv.org/abs/1805.00728`.

[86] Po-Wei Wang, Priya Donti, Bryan Wilder, and Zico Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning*, pages 6545–6554, 2019.

[87] David Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *J. Chem. Inf. Comput. Sci.*, 28(1):31–36, 1988.

[88] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. A semantic loss function for deep learning with symbolic knowledge. *arXiv:1711.11157 [cs, stat]*, Nov 2017. arXiv: 1711.11157. URL: `http://arxiv.org/abs/1711.11157`.

[89] Min-Chul Yang, Nan Duan, Ming Zhou, and Hae-Chang Rim. Joint relational embeddings for knowledge-based question answering. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Oct 2014. `doi:10.3115/v1/D14-1071`.

[90] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. *arXiv:1609.05473 [cs]*, Sep 2016. arXiv: 1609.05473. URL: `http://arxiv.org/abs/1609.05473`.