UNIVERSITÀ DEGLI STUDI DI TRENTO

Department of Information Engineering and Computer Science

Master's Degree in Computer Science

Final Dissertation

Constrained Adversarial Networks

Supervisor

Prof. Andrea Passerini

Student

Gianvito Taneburgo

Co-supervisor

Dr. Paolo Morettin

Academic year 2016/2017

Contents

List of Acronyms

Artificial Intelligence	AI
Artificial Neural Network	ANN
Boundary-seeking Generative Adversarial Networks	BGANs
Constrained Adversarial Networks	CANs
Constrained Optimization Problem	COP
Central Processing Unit	CPU
Constraint Satisfaction Problem	CSP
Deep Learning	DL
Feedforward Neural Network	FNN
Generative Adversarial Networks	GANs
Graphics Processing Unit	GPU
Machine Learning	ML
Recurrent Neural Network	RNN
Propositional Satisfiability Problem	SAT

Abstract

Many real-world applications involve domains subject to a set of constraints, such as syntactic rules in natural language processing and physics laws in probabilistic robotics. Reasoning in these constrained domains can be challenging. For instance, solving constrained problems can be computationally expensive, and some decision problems, such as propositional satisfiability, are known to be NP-complete.

The goal of this work is to propose a novel approach to deal with constraints in the context of generative models involving artificial neural networks. In particular, it aims at finding effective ways to instil prior knowledge in the form of constraints in the game theoretic scenario of generative adversarial networks. This task is not much investigated in literature and, due to its potential implications, deserves further research.

In the original adversarial model, two networks, a discriminator and a generator, compete against each other in a minimax game. The goal of the generator is to produce objects similar to the training examples and trick the discriminator into believing they are real, while the goal of the discriminator is to minimize the chances of this to happen. If the game is well-balanced, the generator will learn how to produce new objects resembling those in input, enabling a wide range of applications.

The proposed model of constrained adversarial networks enables the encoding of useful prior knowledge in this framework by introducing a set of penalty functions in the training algorithm. The final result is that the new generator is able to produce samples that simultaneously resemble those in the training set and satisfy input constraints in expectation.

The conducted experiments reveal how the novel architecture is able to approximate the data distribution more closely than its constraints-unaware counterpart, with a performance gap that seems to increase proportionally with the amount of information conveyed by constraints. Furthermore, by exploring different design choices and analysing several other factors, this work offers a number of insights on how these may individually impact on the final performance.

Constrained adversarial networks provide a flexible model to learn how to produce objects satisfying in expectation both constraints that can be formally expressed and constraints that are hard to encode, either from a practical point of view (e.g. in terms of image pixels) or from a conceptual one (e.g. formalize the idea of *beauty*). This result is achieved by evaluating penalty functions and by looking at examples in the same learning algorithm and in a fully-differentiable setting.

Besides the obvious task of object generation, the architecture can be useful for multiple purposes in different constrained contexts, such as for efficient rejection sampling. Our preliminary results on synthetic data sets are encouraging, justifying further research.

Chapter 1

Introduction

Since ancient times, organisms of any species on Earth have faced situations characterized by the presence of constraints. At the beginning they were related to the limitation of resources, such as food, now they involve many different areas, such as mechanics, engineering, economics, finance and biology. Modern problems appear to be very distant from those posed by nature millennia ago, yet many optimization methods take advantage of heuristics based on social behaviour of certain animals, such as bats [8], bees [15], ants [4], lions, wolves and dolphins [29].

Dealing with constraints is a challenging task that appears in a wide variety of contexts, such as sampling or optimization. In an attempt to solve this kind of problems, mathematicians have explored new fields of study and devised new tools, contributing to the overall progress of science. It is known, for instance, that Greek mathematicians Euclid and Hero of Alexandria already solved some optimization problems related to their geometrical studies [5][32].

New methods to deal with constraints have been designed after the invention of computers and our capability to solve large scale problems is increased with the development of faster processors and, in recent times, graphics processing units. In particular, general-purpose computing on graphics cards has enabled the execution of algorithms that were once considered computationally infeasible, widening research horizons.

The availability of larger amounts of data generated via Internet has improved the effectiveness of machine learning algorithms, especially those involving the training of artificial neural networks. Nowadays, these architectures often represent the state of the art for many different tasks, even achieving superhuman results in some of them [31].

The goal of this work is to describe a novel approach to instil prior knowledge in the form of constraints in a deep generative model. Multiple experiments on synthetic data sets have been executed to measure the impact of several factors on the final performance of the proposed model. Different design choices involving both the architectures and the training procedure have been explored, providing useful insights on a task that is not extensively analysed yet in literature.

Our results are encouraging since they show that prior information can effectively guide the generative model at approximating the data distribution more closely than its constraints-unaware counterpart. Further research is thus justified because the proposed model can be useful in many other contexts dealing with constraints. For instance, due to its characteristics, it can be used to perform rejection sampling efficiently.

Contents are organised as follows: chapter 2 provides background information on artificial neural networks and constrained problems; chapter 3 describes the new deep generative model and how it can solve known issues related to constrained problems; chapter 4 presents the conducted experiments and their results; chapter 5 reports the conclusions drawn and future work ideas.

Chapter 2

Background

Some important concepts regarding deep generative models and constrained problems are now presented. They are an essential tool to understand how modern algorithms can be effectively used to solve well-known complex problems. It is out of the scope of this work to discourse on the history of artificial intelligence and deep learning, as well as on any philosophical consideration regarding how technological progress will impact on our lives. The only relevant aspect is that in the last two decades machine learning techniques have experienced a resurgence due to concurrent advances in computational power, availability of large amounts of data and theoretical understanding, becoming an essential part of the technology industry and helping to solve many challenging problems in different fields.

2.1 Artificial intelligence

Today, artificial intelligence (AI) is a thriving multidisciplinary field of research, involving, among the others, computer science, mathematics, psychology, linguistics and philosophy. Despite this, it is difficult to exactly define what AI is, probably due to the lack of a generally accepted *theory of intelligence*, which is an active research topic on its own [20]. Informally, an intelligent machine should be able to mimic human cognitive functions to solve complex problems, such as searching through a large space of solutions, recognizing patterns, learning from experience, using planning methods and reasoning about complex models via induction [26]. These capabilities seem to be the quintessence of intelligence and are required to solve many real-life problems, such as mechanical translation, game playing, theorem proving and natural language understanding. This is the reason why researchers are concentrating their efforts to study this kind of problems: once solved, they would probably offer solutions applicable to wider areas, such as medical diagnoses. On the other side, as machines become increasingly capable, tasks once considered as requiring intelligence are often removed from the definition, becoming "simple computation". This phenomenon is known as the *AI effect* and has lead to the witty Tesler's "theorem" [12]:

Theorem 1. *AI is whatever hasn't been done yet.*

In computer science, there is no established unifying theory or paradigm that guides AI research. Some of the approaches include traditional symbolic AI, logic programming and statistical methods, most of them relying on a large number of tools such as probabilistic methods for uncertain reasoning or mathematical optimization. Several AI projects have tried to hard-code knowledge about the world in formal languages, in order to allow computers to automatically reason about statements using logical inference rules. This is known as the knowledge-base approach to AI. The difficulties faced by these systems suggest that they need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as *machine learning*.

2.1.1 Machine learning

Machine learning (ML) explores the study of algorithms that can automatically learn from examples and then make predictions on new data. It is employed in a wide range of computing tasks where

designing and programming explicit algorithms with good performance is difficult or infeasible. Some applications involve, for instance, spam filtering, detection of network intruders, optical character recognition and computer vision.

A formal definition is provided by Mitchell [27]:

Definition 1. *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.*

This definition of the tasks in which machine learning is concerned is *operational* and does not involve any cognitive term. It also follows Turing's proposal of focusing on whether machines could *do* what humans can do, acting indistinguishably from them, rather than concentrating on *thinking* in a broader sense [35].

ML tasks are typically classified into two broad categories, depending on whether there is a learning feedback available to the systems: *unsupervised* and *supervised learning*. Unsupervised learning is the task of inferring a function to describe hidden structures from unlabelled data, input observations that do not include any explicit information related to the expected output of the algorithm. On the contrary, supervised learning is the task of learning a function that maps an input to an output based on training data consisting of labelled pairs. More formally, the goal of a supervised learning model is to approximate a function $f^* : \mathbb{X} \to \mathbb{Y}$ of interest for the given task, with $\mathbb{X}$ and $\mathbb{Y}$ some generic input and output domains. For instance, in order to build a classifier, a system should try to approximate at its best the function $y = f^*(\boldsymbol{x})$ that maps an input $\boldsymbol{x}$ to a category y. In case of supervised learning tasks, the system will be given training data pairs $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{X} \times \mathbb{Y}$; in case of unsupervised learning tasks, on the contrary, it will only receive some examples $\boldsymbol{x} \in \mathbb{X}$. As special cases, the input signal can be only partially available or restricted to particular feedback. For instance, in *semi-supervised learning* the algorithm is given a training set with some target outputs missing; in *reinforcement learning* training data, in form of rewards and punishments, are given only as feedback to the program's actions in a dynamic environment.

The core objective of a learner is to generalize from its experience [2]. Generalization is the ability of a computer program to perform accurately on new, unseen examples or tasks after having experienced a learning data set. Typically, the training examples come from an unknown probability distribution and the learner has to build a general model about this space that enables it to produce sufficiently accurate predictions in new cases.

The performance of ML algorithms heavily depends on the *representation* of the data they are given. This is a general phenomenon that appears throughout computer science and even daily life. To recognize the importance of data representation consider how fast can be searching a collection of data if it has been structured and indexed appropriately or how easy is for people arithmetic on Roman numerals with respect to the one on Arabic numerals. Some tasks can be solved by carefully designing the right set of *features* and then providing these features to simple models. However, in many tasks it is difficult to know in advance which features are more relevant. One solution to this problem is to use ML to discover not only the mapping from representation to output but also the representation itself. This approach is known as *representation learning*. Learnt representations often result in better performances than those obtained with hand-designed representations. Furthermore, they also allow systems to rapidly adapt to new tasks with minimal human intervention. A family of techniques exploiting the idea of automatically finding an appropriate representation of data in order to solve ML tasks is *deep learning*.

2.2 Deep learning

Deep learning (DL) is a class of ML algorithms and techniques that, in recent years, has seen a tremendous growth in popularity and applicability due to some key factors. It has already proven useful in many software disciplines, including computer vision, natural language and audio processing, robotics, bioinformatics, chemistry, video games, search engines, online advertising and finance. The main ideas behind DL have been around for many decades, if not centuries, but they have been

effectively applied only in the last few years due to more powerful central processing units (CPUs), general-purpose computing on graphics processing units (GPUs), larger available data sets and clever training techniques. In [9], in fact, the authors point out that the *Age of Big Data* has made ML much easier because the key burden of statistical estimation has been considerably lightened. In addition, this trend is generally expected to continue well into the future.

Modern DL draws inspiration from many fields, especially linear algebra, probability, information theory and numerical optimization. However, neuroscience is regarded as one of the most important sources of inspiration for DL researchers, even if today there is no enough available information about the brain to use it as a guide. Nevertheless, it is worth noting that understanding how the brain works on an algorithmic level is currently studied in *computational neuroscience*.

DL achieves great power and flexibility by automatically learning the best representation for a given task as a nested hierarchy of concepts, with each concept defined in relation to simpler ones, and more abstract representations computed in terms of less abstract ones. If one draws a graph showing how these concepts are built on top of each other, the result is often a deep stack made up of many layers. For this reason, this family of algorithms is called *deep* learning.

A similar learning mechanism can be observed in human brains, where neurons are connected in a complex network and cooperate to compute some kind of functions. To some extent, they can be considered computational units, each one retaining a piece of information or part of a representation, organized in a meaningful structure. By drawing inspiration from this model, ML researchers have focused their efforts theorizing a mathematical counterpart. The result has been the development of *artificial neural networks*.

2.2.1 Artificial neural networks

An artificial neural network (ANN) is a function $f : \mathbb{X} \to \mathbb{Y}$, with $\mathbb{X}$ and $\mathbb{Y}$ some generic input and output domains. An ANN f is typically defined as a composition of other functions $f^{(1)}$, $f^{(2)}$, ..., $f^{(n)}$. It is precisely due to this composition that they are called *multilayer* networks. Depending on how these functions are composed together, an ANN can be classified as a *feedforward neural network* (FNN) or as a *recurrent neural network* (RNN). In FNNs the functions composition forms directed acyclic graphs, while in RNNs graphs can contain cycles. This work focuses on the former.

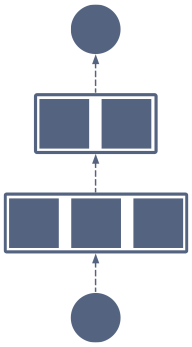

Figure 2.1: Schematic representation of a feedforward neural network.

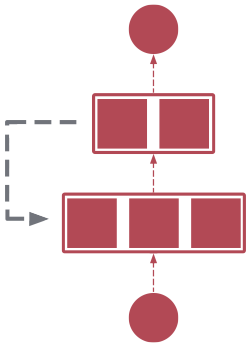

Figure 2.2: Schematic representation of a recurrent neural network.

An example of FNN is the function obtained by chaining three components $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$, yielding $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. In this case, $f^{(1)}$ is called the *first layer* of the network, $f^{(2)}$ is the *second layer* and $f^{(3)}$, the final layer, is the *output layer*. The overall length of the chain gives the *depth* of the model. Training examples specify directly only what the output layer should return, with no clue on the behaviour of other layers. It is up to the ML algorithm to decide how to use those layers to produce the desired output that approximates f^*, the target function to learn, of interest for the given ML task. For this reason, they are called *hidden layers*.

Hidden layers are usually vector-valued and their dimensionality determines the *width* of the model. Every element of the vector may be interpreted as a human neuron and the layer can be considered as consisting of many *units* that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron since it receives inputs from many other connected units (or from the external environment, in case of the first layer) and then computes its own activation value. However, the goal of ANNs is not to perfectly model the brain, even though the choice of the some layer functions $f^{(i)}$ is sometimes guided by neuroscientific observations on biological neurons.

ML tasks often require to learn a nonlinear function f^*. In order to compute a nonlinear function of $\boldsymbol{x}$ at least one layer $f^{(i)}$ of the ANN must involve a nonlinear transformation, otherwise the final output of the network would remain a linear combination of its intermediate results. Most FNNs rely on an affine transformation on every hidden layer, followed by a fixed nonlinear function called *activation function*. The output layer, on the contrary, does not usually apply any further nonlinear transformation. As a result, every hidden layer consists of a vector of hidden units $\boldsymbol{h}^{(i)}$ computed by the corresponding nonlinear function $f^{(i)}(\boldsymbol{x}; \boldsymbol{W}^{(i)}, \boldsymbol{c}^{(i)})$, where $\boldsymbol{W}^{(i)}$ provides the weights of the linear transformation and $\boldsymbol{c}^{(i)}$ the biases. The network thus computes the values of some hidden units and then uses them as input for the next layer. By doing so, information flows in a unidirectional way from one hidden layer to the next, up to the output layer. This is the reason why this kind of neural network is called feedforward.

The computational process of a FNN just described can be now formally defined. Given a FNN $f : \mathbb{X} \to \mathbb{Y}$ composed by n functions (layers) $f^{(1)}$, $f^{(2)}$, ..., $f^{(n-1)}$, $f^{(n)}$, we have

$$f(\boldsymbol{x}) = f^{(n)}(f^{(n-1)}(...(f^{(2)}(f^{(1)}(\boldsymbol{x}))))), \tag{2.1}$$

where

$$f^{(i)}(\boldsymbol{x}) = \begin{cases} g^{(i)}(\boldsymbol{W}^{(i)\top}\boldsymbol{x} + \boldsymbol{c}^{(i)}) & \text{if i = 1} \\ g^{(i)}(\boldsymbol{W}^{(i)\top} f^{(i-1)}(\boldsymbol{x}) + \boldsymbol{c}^{(i)}) & \text{otherwise} \end{cases}, \tag{2.2}$$

with $g^{(i)}$ arbitrary activation functions, $\boldsymbol{W}^{(i)}$ layers weights and $\boldsymbol{c}^{(i)}$ layers biases.

For convenience, all these parameters are usually grouped in a vector $\boldsymbol{\theta}$. Training an ANN amounts to finding a $\boldsymbol{\theta}$ such that $f(\boldsymbol{x}; \boldsymbol{\theta})$ well approximate $f^*(\boldsymbol{x})$. This is usually done with information provided by a *loss function*.

2.2.2 Loss functions

In mathematical optimization, a loss function $l : \mathbb{A} \to \mathbb{R}$ assigns a cost to an event or value. This number intuitively measures how bad the effects of an event are or how distant is the input value from the target one. An optimization problem obviously seeks to minimize the loss function. In some cases, the objective function is the opposite of l, it is called *reward function*, and the equivalent goal is to maximize it.

The loss functions of ANNs are similar to those of other parametric models. In most cases, a network defines a distribution $p(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\theta})$, so it is possible to estimate $\boldsymbol{\theta}$ via *maximum likelihood estimation*, a popular method of estimating statistical model parameters given observations.

ANNs are thus often trained using the cross entropy between the empirical distribution defined by the training set $\hat{p}_{data}$ and the model distribution p_{model}:

$$l(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{y}|\boldsymbol{x}), \tag{2.3}$$

where l is the loss function associated to the entire network.

An advantage of deriving the loss function from maximum likelihood is that it removes the burden of designing a new loss function for each model: specifying a model $p_{model}(\boldsymbol{y}|\boldsymbol{x})$ automatically determines a cost function $\log p_{model}(\boldsymbol{y}|\boldsymbol{x})$.

Regardless of which loss function is chosen to train the model, the minimization of the error is achieved through an optimization method, such as *gradient descent*. For such reason, the loss function must be differentiable. In addition, in order to serve as a good guide throughout all the training algorithm, its gradient should always be large and predictable. Functions that saturate undermine

this objective because they produce very small gradients. In many cases this happens because the activation functions used to compute the output of the hidden units or the output units saturate. The logarithm in the negative log-likelihood loss function succeeds in undoing the problematic effect caused by output units involving exponential functions that saturate when their argument is very small.

Error minimization through gradient descent [11] in the parameters space of complex, nonlinear, differentiable systems has been discussed at least since the early 1960s, initially within the framework of Euler-Lagrange equations in the calculus of variations [6]. However, an efficient procedure to minimize loss functions was first described only a decade later [23][24], with the design of the algorithm of *backpropagation*.

2.2.3 Backpropagation

Backpropagation is an efficient method used during ANNs training to compute how weights connecting units should be updated with respect to the value of the loss function. The systems of the 1960s transmitted derivative information through standard Jacobian matrix calculations from one "layer" to the previous one, without explicitly addressing either direct links across several layers or potential additional efficiency gains due to network sparsity, typical of ANNs. Modern backpropagation overcomes these limitations with a computational cost essentially equivalent of the forward pass.

Equations 2.1 and 2.2 concisely describe how ANNs layers as a whole interact with each other. However, to easily describe the backpropagation algorithm, it is handy to define some additional ones that involve single units.

We define $w_{i,j}$ as the weight connecting two units h_i and h_j belonging to different layers. $\mathbb{W}$ is the set of such weights. An iterative optimization algorithm should try to optimize each $w_{i,j}$, step by step, according to the error measured by l. This is done with gradient descent in the following way:

$$w_{i,j} \leftarrow w_{i,j} - \eta \frac{\partial l}{\partial w_{i,j}}, \tag{2.4}$$

where $\eta \in \mathbb{R}$ is the *learning rate*.

We define $pred_j$ as the set of input units connected to h_j:

$$pred_j = \{h_i | w_{i,j} \in \mathbb{W}\}$$

and, similarly, $succ_j$ as the set of output units connected to h_j:

$$succ_j = \{h_i | w_{j,i} \in \mathbb{W}\}$$

Supposing a fixed differentiable activation function φ for every unit, we define net_j as the input h_j receives from the units it is connected to:

$$net_j = \sum_{k}^{pred_j} w_{k,j} \cdot o_k,$$

where $o_k = \varphi(net_k)$ is the output of h_k. If the unit h_k is in the first layer after the input layer, than o_k is just $\boldsymbol{x}_k$, with $\boldsymbol{x}$ being a training example.

Each weight $w_{i,j}$ should be updated according to:

$$\frac{\partial l}{\partial w_{i,j}} = \delta_j \cdot o_i, \tag{2.5}$$

where

$$\delta_j = \begin{cases} \frac{\partial l}{\partial f(\boldsymbol{x}_j)} \cdot \varphi\prime(net_j) & \text{if unit } h_j \text{ in output layer} \\ \sum_{k}^{succ_j} \delta_k \cdot w_{j,k} \cdot \varphi\prime(net_j) & \text{otherwise} \end{cases}.$$

The full derivation of 2.5 can be found in appendix A.

The resulting iterative procedure to train an ANN via gradient descent is described by algorithm 1.

Algorithm 1 Gradient descent algorithm to train an ANN

Require: ANN f
Require: initialized set of weights $\mathbb{W}$
Require: loss function l
Require: learning rate η
1: **while** the stopping criterion is not satisfied **do**
2: **for all** training examples $(\boldsymbol{x}, \hat{\boldsymbol{y}})$ **do**
3: $\boldsymbol{y} \leftarrow f(\boldsymbol{x})$ ▷ compute ANN output
4: **for all** weight $w_{i,j} \in \mathbb{W}$ **do**
5: $\Delta w_{i,j} \leftarrow \delta_j \cdot \boldsymbol{y}_i$ ▷ see equation 2.5
6: $w_{i,j} \leftarrow w_{i,j} - \eta \Delta w_{i,j}$ ▷ update weight according to loss

During the initialization phase, weights are usually given small random values. The stopping criterion (line 1) can be any predicate reasonable for the learning task, such as *all training examples are classified correctly* or *the error computed by the loss function l is below a given threshold*. This procedure is usually repeated several times; the number of times training examples are used once to update the weights is an *epoch*.

Many widely adopted techniques in DL are discovered empirically and systematically applied because they work well in practice, even if their theoretical background is still uncertain and sometimes it happens to see little algorithmic tricks that effectively improve the overall performances of the ANNs. In the same way, it is also common to see approximations that make computation faster or even feasible, without a significant loss in performance. For instance, the gradient descent algorithm is usually modified to process little groups of training examples, called *mini-batches*, instead of one example at a time (line 2) or by randomly shuffling the training set at each new iteration (line 1), giving rise to the *stochastic gradient descent algorithm* (algorithm 2). It is now worth pointing out some other theoretical considerations.

Algorithm 2 Stochastic gradient descent algorithm to train an ANN

Require: ANN f
Require: initialized set of weights $\mathbb{W}$
Require: loss function l
Require: learning rate η
Require: mini-batch size m
1: **while** the stopping criterion is not satisfied **do**
2: $mb \leftarrow$ data mini-batch $\{(\boldsymbol{x}^{(1)}, \hat{\boldsymbol{y}}^{(1)}), (\boldsymbol{x}^{(2)}, \hat{\boldsymbol{y}}^{(2)}), ..., (\boldsymbol{x}^{(m)}, \hat{\boldsymbol{y}}^{(m)})\}$ randomly sampled from training set
3: **for** $k \leftarrow 1, m$ **do**
4: $\boldsymbol{y}^{(k)} \leftarrow f(mb_k[\boldsymbol{x}])$ ▷ compute ANN outputs
5: **for all** weight $w_{i,j} \in \mathbb{W}$ **do**
6: $\Delta w_{i,j} \leftarrow 0$
7: **for** $k \leftarrow 1, m$ **do**
8: $\boldsymbol{x} \leftarrow mb_k[\boldsymbol{x}]$
9: $\boldsymbol{y} \leftarrow \boldsymbol{y}^{(k)}$
10: $\Delta w_{i,j} \leftarrow \Delta w_{i,j} + \delta_j \cdot \boldsymbol{y}_i$ ▷ see equation 2.5
11: $w_{i,j} \leftarrow w_{i,j} - \eta \cdot \frac{1}{m} \cdot \Delta w_{i,j}$ ▷ update weight according to average loss

2.2.4 Theoretical considerations

A number of considerations about DL should be kept in mind when designing ANNs.

First of all, the *"no free lunch" theorem* for ML [36] states that, averaged over all possible data generating distributions, every classification algorithm has the same error rate when classifying previously unobserved examples. In other words, no ML algorithm is universally any better than any other. However, the goal of ML research is not to seek a universal learning algorithm, but to design systems that perform well on specific tasks. For such reason, there is no need to average over all possible data distributions, since only a tiny subset of them are relevant for the real world we experience. The theorem suggests that ML algorithms should be developed by encoding a set of preferences into the learning algorithm itself. If this is done appropriately, than the algorithm will solve the task it is designed for.

Secondly, the central challenge in ML is generalization. The generalization error of a ML model is typically estimated by measuring its performance on a test set of examples unavailable during training. The factors determining how well a ML algorithm will perform are its abilities to make both the training error and its gap with the test error small. These two factors correspond to two known problems in ML: *underfitting* and *overfitting*. The former occurs when the model is not able to obtain a sufficiently low error value on the training set, the latter when the gap between the training error and the test error is too large. One can control whether a model is more likely to overfit or underfit by altering its *capacity*, the ability to fit a wide variety of functions. What should be kept in mind is that, typically, as model capacity increases, training error decreases until it asymptotes to the minimum possible value, while generalization error has a U-shaped curve and, after a certain point, it will start increasing. With *regularization* it is possible to reduce the generalization error without any impact on the training error.

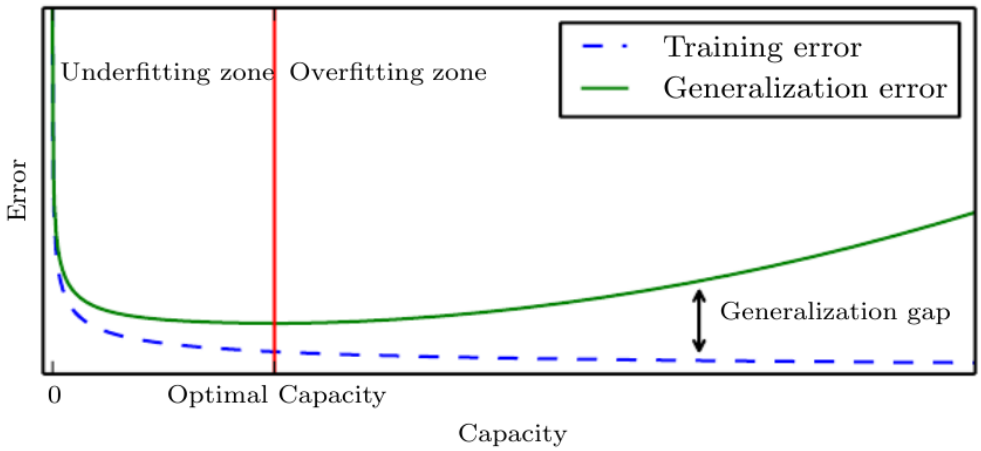

Figure 2.3: Typical relationship between capacity and error as illustrated in [9]. At the left end of the graph, training error and generalization error are both high, in the underfitting regime. As capacity increases, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error and overfitting regime is reached.

A final considerations regards *hyperparameters*, a set of values defining higher level concepts about the model, such as the network learning rate. Since they cannot be learnt directly from the training data, they are usually empirically found by testing different models on a validation set, another set of data unavailable during training, and then selecting the best ones. Hyperparameters tuning can be expensive and is an active research field since these values can largely impact on the final performance of the network.

2.3 Deep generative models

ML models are typically divided into two complementary classes: *discriminative models* and *generative models*. From a probabilistic point of view, discriminative methods try to learn a mapping from input

variables x to output variables y directly modelling the conditional probability distribution $p(y|x)$. On the contrary, generative methods try to learn the joint probability distribution $p(x, y)$ underlying the data. Since $p(x, y)$ can be used to compute $p(y|x)$, generative models can also be used to make predictions. However, probably due to the easier nature of their goal, discriminative algorithms often achieve better results in classification tasks and are generally preferred [28]. Nevertheless, generative models are actively studied because of their peculiar capability of generating new data resembling those in input once $p(x, y)$ is learnt.

Some generative models allow the probability distribution function to be evaluated explicitly, while others only support operations that implicitly require knowledge of it, such as sampling. Techniques to automatically learn an approximation of the real data distribution are particularly useful when learning the exact one is hard or even impossible. Indeed, modern generative networks, when given enough training examples and capacity, often succeed in learning a distribution very similar to the true one. The key insight is that ANNs used for generative models are forced to learn representations in a space significantly smaller than the one of training data and so they are forced to learn meaningful features in order to later generate new data.

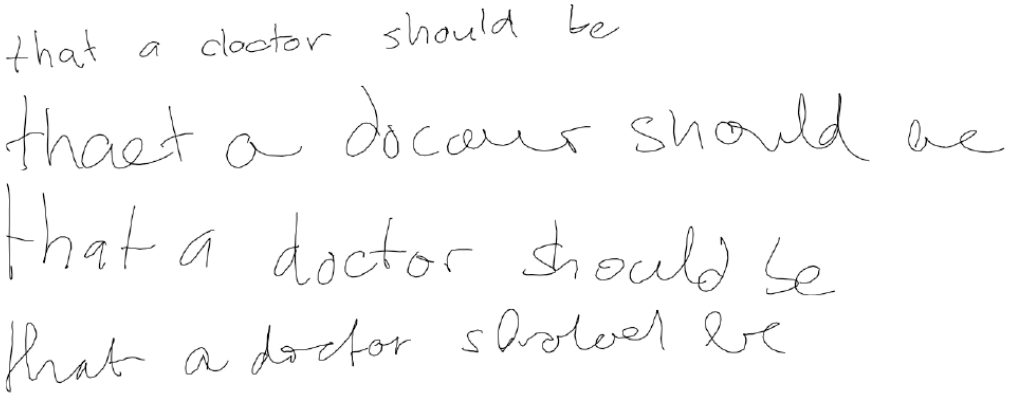

Figure 2.4: Different examples of the words sequence "that a doctor should be" drawn from a RNN generator [10].

It is likely that in the near future ANNs will be able to generate samples depicting entirely plausible images or videos. This may by itself find use in multiple applications, such as on-demand generated art [39] [25] (see figure 2.5). However, one important application of these networks is the generation of synthetic data to augment real data sets, especially for pre-training, a procedure during which the model is trained in advance on data similar to those of the task to avoid random network parameters initialization. Besides these speculations, present known applications include image denoising, inpainting [37], super-resolution [16] [19], exploration in reinforcement learning and neural network pre-training in cases where labelled data are expensive.

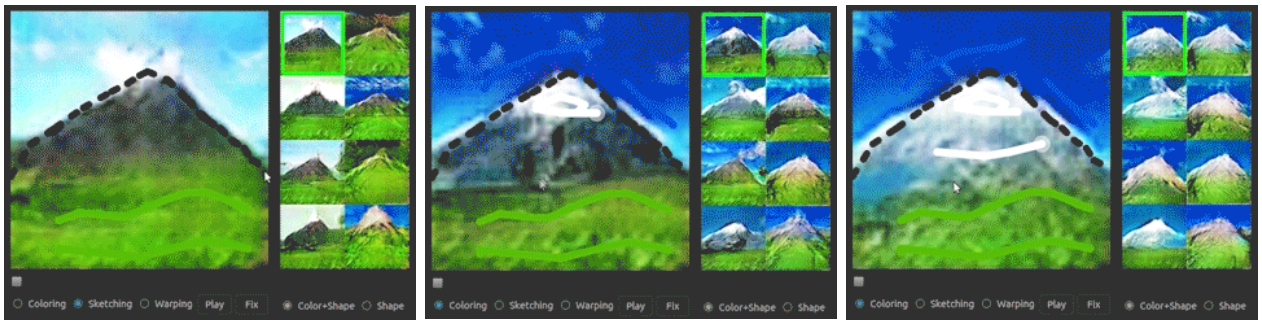

Figure 2.5: Example of interactive image generation from [39]. A system can produce photo-realistic samples that best satisfy the user edits in real-time.

This work focuses on a family of generative models called *generative adversarial networks*.

2.3.1 Generative adversarial networks

Generative adversarial networks (GANs) are based on a game theoretic scenario in which two networks, a *discriminator* and a *generator*, compete against each other. The goal of the generator is to produce objects similar to the training examples and trick the discriminator into believing they are real. This model can be thus thought as a game between a counterfeiter, trying to produce fake currency, and a policeman, trying to detect it. The idea to infer models in a competitive setting was first used for behavioural inference in the field of ethology [21] and to find binary factorial codes [30].

Formally, we define the discriminator as a network $D(\boldsymbol{x}; \boldsymbol{\theta}^{(d)})$ and the generator as a network $G(\boldsymbol{z}; \boldsymbol{\theta}^{(g)})$, where $\boldsymbol{z}$ is a random input noise drawn from a distribution p_z. We define the generator's probability distributions over data $\boldsymbol{x}$ as as p_g.

The output of the generator $\boldsymbol{x} = G(\boldsymbol{z})$ represents a generated object that should resemble those of the training set, while the output of the discriminator $y = D(\boldsymbol{x})$ represents the probability that $\boldsymbol{x}$ was sampled from p_{data} rather than p_g.

The goal of the model is to train simultaneously D and G: the discriminator should try to maximize its output probabilities for training examples; the generator should try minimize the probability of generating objects that will be rejected by D (i.e. those receiving a low score). So, D and G play the following minimax game with *value function* $V(D, G)$:

$$\min_G \max_D V(D, G). \tag{2.6}$$

A typical choice for $V(D, G)$ is

$$V(D, G) = \underbrace{\mathbb{E}_{\boldsymbol{x} \sim p_{data}}[\log D(\boldsymbol{x})]}_{D \text{ goal}} + \underbrace{\mathbb{E}_{\boldsymbol{z} \sim p_z}[\log(1 - D(G(\boldsymbol{z})))]}_{G \text{ goal}}. \tag{2.7}$$

However, when G has not learnt yet how to produce good objects, D can reject samples from p_g with high confidence. In this case $\log(1 - D(G(\boldsymbol{z})))$ saturates, preventing learning. To avoid this issue, G can be instead trained to maximize the opposite function, $\log D(G(\boldsymbol{z}))$. Although mathematically equivalent, the second formulation provides better gradients in early training.

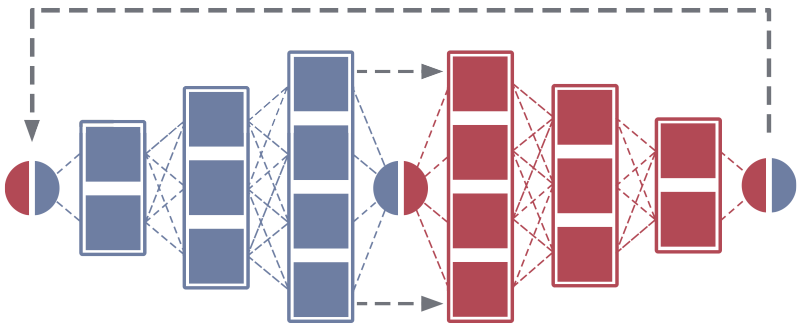

Figure 2.6: Schematic representation of GANs with a generator (blue) producing objects for the discriminator (red).

In practice, D and G are trained via traditional backpropagation using an iterative approach consisting of two alternating loops of k steps of D optimization and one step of G optimization. This results in D being maintained near its optimal solution as long as G changes slowly enough. Algorithm 3 shows the mini-batch stochastic gradient descent training procedure for GANs.

In a scenario in which the two networks have access to infinite examples, after several training steps, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{data}$. In this situation the discriminator will be unable to differentiate between the two distributions and every example will have the same probability of being real or generated, i.e. $D(\boldsymbol{x}) = \frac{1}{2}$. In general, given a fixed generator G, the optimal discriminator is

Algorithm 3 Stochastic gradient descent algorithm to train GANs

Require: ANN discriminator D
Require: ANN generator G
Require: discriminator parameters $\boldsymbol{\theta}^{(d)}$
Require: generator parameters $\boldsymbol{\theta}^{(g)}$
Require: learning rate η
Require: mini-batch size m
Require: discriminator training steps number k

while the stopping criterion is not satisfied **do**
  for k steps **do**
    $z\_mb \leftarrow$ noise mini-batch $\{\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}, ..., \boldsymbol{z}^{(m)}\}$ randomly sampled from p_z
    $mb \leftarrow$ data mini-batch $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(m)}\}$ randomly sampled from p_{data}
    $\Delta\boldsymbol{\theta}^{(d)} \leftarrow \nabla_{\boldsymbol{\theta}^{(d)}} \frac{1}{m} \sum_{i=1}^{m} [\log D(mb_i) + \log(1 - D(G(z\_mb_i)))]$ ▷ compute gradient
    $\boldsymbol{\theta}^{(d)} \leftarrow \boldsymbol{\theta}^{(d)} + \eta\Delta\boldsymbol{\theta}^{(d)}$ ▷ update D by ascending gradient
  $z\_mb \leftarrow$ noise mini-batch $\{\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}, ..., \boldsymbol{z}^{(m)}\}$ randomly sampled from p_z
  $\Delta\boldsymbol{\theta}^{(g)} \leftarrow \nabla_{\boldsymbol{\theta}^{(g)}} \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z\_mb_i)))]$ ▷ compute gradient
  $\boldsymbol{\theta}^{(g)} \leftarrow \boldsymbol{\theta}^{(g)} - \eta\Delta\boldsymbol{\theta}^{(g)}$ ▷ update G by descending gradient

$$D_G^*(\boldsymbol{x}) = \frac{p_{data}(\boldsymbol{x})}{p_{data}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \tag{2.8}$$

The main motivation for the design of GANs is that the learning process requires neither approximate inference nor approximation of a partition function gradient, but only traditional backpropagation. Furthermore, the model is very flexible since D and G can be any ANN and eliminates the need of Markov chains.

On the other side, GANs do not offer an explicit representation of p_g. Furthermore, some critical issues may be hidden in the training procedure. First, D and G must be synchronized well in order to avoid the scenario in which the generator collapses too many values of $\boldsymbol{z}$ to the same value of $\boldsymbol{x}$. This usually happens when G is trained too much, without updating D. Second, simultaneous gradient descent in the adversarial game setting is not guaranteed to reach an equilibrium. In fact, it is possible for the two players to take infinite turns increasing and then decreasing the value function forever, rather than landing exactly on the saddle point of V where they cannot further reduce their costs. Note that the equilibria for a minimax game are not local minima of V, but saddle points which are local minima with respect to the first player's parameters and local maxima with respect to the second player's parameters. Finally, GANs framework is designed for fully differentiable networks D and G. This is problematic since many important real-world data sets, such as word-base representations of language, are discrete. However, there exist variants that can deal with discrete variables, such as *boundary-seeking generative adversarial networks*.

2.3.2 Boundary-seeking generative adversarial networks

Boundary-seeking generative adversarial networks (BGANs) provide a unified framework to deal with discrete and continuous variables. For the present work, only the former are of interest and will be further analysed.

Given a fixed generator G, from equation 2.8 it is possible to derive the density of data:

$$p_{data}(\boldsymbol{x}) = p_g(\boldsymbol{x}) \frac{D_G^*(\boldsymbol{x})}{1 - D_G^*(\boldsymbol{x})}. \tag{2.9}$$

This indicates that, in the limit of a perfect discriminator D^*, the true data distribution can be perfectly estimated from such discriminator and any fixed generator G, even if that generator is

not perfectly trained. A sample from the correct data distribution can thus be obtained by simply reweighing samples drawn from the imperfect distribution p_g according the ratio $\frac{D^*_G(\boldsymbol{x})}{1-D^*_G(\boldsymbol{x})}$.

Unfortunately, it is unlikely that the perfect discriminator D^* can be learnt or even exists. However, given an imperfect discriminator D, the true data distribution p_{data} can be approximated in the following way:

$$\widetilde{p}_{data}(\boldsymbol{x}) = \frac{1}{Z} p_g(\boldsymbol{x}) \frac{D(\boldsymbol{x})}{1 - D(\boldsymbol{x})}, \tag{2.10}$$

where

$$Z = \sum_{\boldsymbol{x}} p_g(\boldsymbol{x}) \frac{D(\boldsymbol{x})}{1 - D(\boldsymbol{x})}$$

is the normalization constant that guarantees $\widetilde{p}_{data}$ to be a proper probability distribution. The bias $\widetilde{p}_{data}$ is affected to only depends on the quality of D: the closer D is to D^*, the lower the bias. This is a nice property, since training the discriminator (a standard probabilistic binary classifier) is generally easier than training the generator (whose objective function is a moving target as the discriminator adapts to the generator and vice versa). Note that the optimum for the generator occurs at $D(\boldsymbol{x}) = D^*(\boldsymbol{x}) = \frac{1}{2}$, so that $Z = 1$ and $p_{data} = \widetilde{p}_{data} = p_g$. This occurs at the decision boundary of the discriminator, when generated samples have the same probability of being classified as real data or samples. This is the reason for the name BGANs.

GANs have serious limitations on the type of variables they can model, because they require the composition of the generator and discriminator to be fully differentiable. With discrete variables this is not true. For instance, consider using a step function at the end of a generator in order to generate a discrete value. In this case, backpropagation alone cannot provide the training signal for the generator, since the derivative of a step function is 0 almost everywhere and so would be the backpropagated signal. The goal of BGANs is to train a generator outputting discrete values by estimating gradients for $\boldsymbol{\theta}^{(g)}$ via the exclusive Kullback-Leibler divergence between the two joint distributions $p_{data}(\boldsymbol{x}, \boldsymbol{z})$ and $p_g(\boldsymbol{x}, \boldsymbol{z})$:

$$\nabla_{\boldsymbol{\theta}^{(g)}} D_{KL}(p_{data}(\boldsymbol{x}, \boldsymbol{z}) || p_g(\boldsymbol{x}, \boldsymbol{z})), \tag{2.11}$$

or by using the biased estimator:

$$\nabla_{\boldsymbol{\theta}^{(g)}} D_{KL}(\widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z}) || p_g(\boldsymbol{x}, \boldsymbol{z})). \tag{2.12}$$

The generator distribution $p_g(\boldsymbol{x})$ can be parametrized as the marginalization of a joint density

$$p_g(\boldsymbol{x}) = \sum_{\boldsymbol{z}} g(\boldsymbol{x}|\boldsymbol{z}) p_z(\boldsymbol{z}), \tag{2.13}$$

where $g(\boldsymbol{x}|\boldsymbol{z})$ is the conditional distribution of the generator function.
The result of the computation of the gradient in equation 2.12 is:

$$\nabla_{\boldsymbol{\theta}^{(g)}} D_{KL}(\widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z}) || p_g(\boldsymbol{x}, \boldsymbol{z})) \approx -\mathbb{E}_{\boldsymbol{z} \sim p_z(\boldsymbol{z})}[\sum_{m=1}^{M} \widetilde{w}^{(m)} \nabla_{\boldsymbol{\theta}^{(g)}} \log g(\boldsymbol{x}^{(m)}|\boldsymbol{z})], \tag{2.14}$$

where

$$\widetilde{w}^{(m)} = \frac{w^{(m)}}{\sum_{m'} w^{(m')}}$$

and

$$w^{(m)} = \frac{D(\boldsymbol{x}^{(m)})}{1 - D(\boldsymbol{x}^{(m)})}$$

are the normalized and unnormalized importance weights and $\boldsymbol{x}^{(m)}$ are samples from the generator for the given $\boldsymbol{z}$. The normalization can help to generate a relative learning signal that is particularly useful when samples are all fairly bad from the point of view of $D(\boldsymbol{x})$.

This allows to compute gradients from a mini-batch of samples from $p_z(\boldsymbol{z})$ and to train discrete GANs without relying on backpropagation. Compared to normal GANs, this method requires M times more space, but computing the M (instead of 1) scalar values of $D(\boldsymbol{x}^{(m)})$ can be parallelized, as samples from $g(\boldsymbol{x}|\boldsymbol{z})$ can be drawn independently.

The full derivation of 2.14 can be found in appendix B.

2.4 Constrained problems

Many real-world applications involve domains subject to a set of constraints, such as grammar rules for sentences, design constraints for modelling, safety rules for airline protocols or game rules. These applications typically solve one of the two following kinds of problem: *constraint satisfaction* or *constrained optimization*.

A constraint satisfaction problems (CSP) is defined as a set of objects whose state must satisfy a number of constraints or limitations. Formally, a CSP is defined as a triple $\langle \mathbb{X}, \mathbb{D}, \mathbb{C} \rangle$, where $\mathbb{X} = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$ is a set of variables, $\mathbb{D} = \{D^{(1)}, D^{(2)}, ..., D^{(n)}\}$ is the set of the respective domains and $\mathbb{C} = \{c^{(1)}, c^{(2)}, ..., c^{(m)}\}$ is a set of constraints. Every constraint $c^{(i)} \in \mathbb{C}$ is a pair $\langle t^{(i)}, R^{(i)} \rangle$, where $t^{(i)} \subseteq \mathbb{X}$ is a subset of k variables and $R^{(i)}$ is a k-ary relation on the corresponding subset of domains. A solution to the CSP is an assignment of values from $\mathbb{D}$ to all the variables in $\mathbb{X}$ that satisfies all the constraints in $\mathbb{C}$.

CSPs are the subject of intense research in both AI and operations research since the regularity in their formulation often provides a common basis to solve seemingly unrelated problems. Propositional satisfiability problem (SAT), satisfiability modulo theories and answer set programming can be thought as instances of CSPs. Classic examples of CSPs are the eight queens puzzle, the map colouring problem and Sudoku.

Constrained optimization problems (COPs) involve the process of finding the best values for an objective function with respect to some variables in the presence of constraints on those variables. The objective function is either a cost function to be minimized or a reward function to be maximized. Constraints can be either hard (variables must satisfy them) or soft (variables may not satisfy them with a penalization in the objective function). More formally, a general constrained minimization problem is expressed as

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

Optimization can be an extremely difficult task. ML algorithms try to avoid these difficulties by adopting particular objective functions and constraints to ensure that the optimization problem is convex. When training ANNs, optimization is generally performed in the non-convex case and this can cause many issues, such as finding only local minima.

Many techniques and software exist to solve CSPs and COPs, but their analysis is out of the scope of this work.

Chapter 3

Constrained adversarial networks

"What I cannot create, I do not understand."
Richard Feynman

Though there exist general frameworks that allow the encoding of various structured constraints on latent variable models [7] [38] [22], they either are not directly applicable to ANNs or yield inferior performance [13]. Constrained adversarial networks (CANs) are a generalization of the deep generative model of BGANs, designed to introduce constraints during the adversarial training. The goal of the model is to train a generator that will be able to produce samples that simultaneously resemble those in the training set and satisfy a set of input constraints in expectation.

More formally, CANs consist of a discriminator network $D(\boldsymbol{x}; \boldsymbol{\theta}^{(d)})$, a generator network $G(\boldsymbol{z}; \boldsymbol{\theta}^{(g)})$ and a set of constraints $\mathbb{C} = \{c^{(1)}, c^{(2)}, ..., c^{(m)}\}$, where each constraint $c^{(i)} : \mathbb{X} \to [0, 1]$ is a penalty function. Consistently with GANs, we define $\boldsymbol{z}$ as a random input noise drawn from a distribution p_z, and p_g as the generator's probability distributions over data $\boldsymbol{x}$. The output of the generator $\boldsymbol{x} = G(\boldsymbol{z})$ represents a generated object resembling those of the training set and, possibly, satisfying the constraints, while the output of the discriminator $y = D(\boldsymbol{x})$ represents the probability that $\boldsymbol{x}$ was sampled from p_{data} rather than p_g. In addition, we define a *perfect* object as follows:

Definition 2. *Given a set of constraints $\mathbb{C}$, an object x is perfect if $c(x) = 0 \ \forall c \in \mathbb{C}$.*

3.1 Theoretical motivations

The design of CANs is guided by some theoretical motivations regarding common issues arising when dealing with domains subject to constraints. One of them is that formally encoding constraints may sometimes be hard. This problem is especially true when constraints involve global properties of objects that we, as human, recognize intuitively or can not formally describe. In particular, a constraint can be hard to encode either from a *practical* point of view or from a *conceptual* one. For instance, suppose our data set is made up of images of human faces. A constraint that is practically hard to encode may involve the expected number of eyes on each image. Expressing such property in term of image pixels is clearly unfeasible. However, it is very simple for us to describe what high-level image characteristics we expect to consider the constraint satisfied and, even simpler, to determine if the constraint is not satisfied from a quick glance. On the contrary, it is conceptually harder for us to express the property of beauty. This is not something related to personal taste, rather to the global and somehow implicit nature of some image properties. Nevertheless, we expect a generative model to produce beautiful faces if its training set only consist of pictures of beautiful faces.

GANs are not affected by this problem because, when provided with enough capacity, they can automatically learn how to generate images resembling input data. So, if training set objects satisfy these high-level constraints, the generator will be forced to learn them in order to deceive the discriminator. As a final result, images sampled from p_g will tend to have these desired properties even if they were not explicitly encoded anywhere. CANs inherit this capability from GANs. In addition, they provide the possibility to directly encode all the other constraints that can be easily expressed

to instil external knowledge in the network and to guide the generator training process via penalty functions.

Finding solutions to constrained problem such as CSPs and COPs can be computationally hard. For instance, it is known that SAT is a decision problem NP-complete. The novel approach of CANs consists in replacing the explicit constraints optimization with an implicit polynomial-time learning procedure. During training, in particular, penalty function are evaluated by one of the two networks and their output values are used to guide the generator in better approximating with its probability distribution p_g the goal distribution p_{data}. Exploring different solutions to make this idea effective and efficient has been the research goal. Furthermore, once the networks are trained, generating a possibly perfect object only requires one forward passing of noise through the generator, so it can be done in polynomial-time as well. Also this property of the model is inherited by GANs.

Finally, in some situations finding a single solution is not sufficient for the task being performed and one may be interested in different candidates among those available, perhaps with some kind of statistical guarantee. Many methods are not designed to satisfy this requirement and obtaining different solutions may be particularly inefficient or even impossible. A generative model parametric on some input noise overcomes this limitation by its very nature. In fact, once trained, CANs can be efficiently used for sampling many instances from p_g by simply using different input noise vectors $\boldsymbol{z}$. The more effective the ANNs training procedure is, the higher will be the number of perfect examples produced by the generator. However, the most important consideration involves execution times, since generating a possibly perfect object always requires polynomial time. This allows, for instance, efficient *rejection sampling*, since CANs can be used to produce in polynomial time examples that, in expectation, will not be rejected, making this model useful for many other applications, such as approximate inference or recommendation.

3.2 Design choices

Introducing constraints in the game theoretic scenario of GANs can be done in many different ways. This section describes some of the approaches that have been explored, postponing the presentation of the results and their analysis to chapter 4.

3.2.1 Adversarial models

Since the adversarial training procedure involves two different networks, the first necessary design choice behind CANs regards the *location* in which information coming from penalty functions should be used.

It would seem obvious to add constraints directly in the generator, since they can provide useful clues on how new objects shall be. In this case the generator loss function will include a signal on constraints satisfaction. The loss function to minimize of this new constraint-regularized generator may thus be

$$l_G = \mathbb{E}_{\boldsymbol{z}\sim p_z}[\log(1 - D(G(\boldsymbol{z})))] + \lambda \mathbb{E}_{\boldsymbol{z}\sim p_z}[\sum_{i=1}^{|\mathbb{C}|} c^{(i)}(G(\boldsymbol{z}))],$$

where λ is a regularization term.

This approach can be effective, but requires each constraint in $\mathbb{C}$ to be differentiable in order to backpropagate the learning signal throughout all the generator. Exploring this direction is left for future works.

The opposite approach is to introduce constraints in the discriminator network. Providing D with some oracle penalty functions should enhance its capability to distinguish between training and sampled data. This is especially true when the training set only contains perfect objects. As long as the adversarial training is well-balanced, the generator will be indirectly forced to produce objects satisfying the constraints, especially during the first epochs. From a high-level point of view, this corresponds to strengthen the predictive capability of the discriminator that will, in turn, push the generator to produce better objects in order to effectively compete in the game. The objective function

to maximize of this new constraint-regularized discriminator may thus be

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

This solution is conceptually reasonable and practically flexible. Furthermore, it is constraint-agnostic and can be used even with constraints that are not differentiable. For these reasons, it has been extensively tested.

Another approach is to extend the GANs model to introduce a third network, a *teacher*, forcing the generator to emulate its rule-regularized predictions [13]. This iterative rule distillation process can effectively transfer rich structured knowledge, expressed in the declarative first-order logic language, into parameters of general neural network. Furthermore, this process is agnostic about the ANNs architectures and is left for future works as well.

3.2.2 Discriminator architectures

The second main design choice behind CANs involves the *positioning* of constraints-related hidden units.

Regardless of the internal details about the discriminator architecture, the output layer will necessarily contains one single unit determining the final prediction on the input example. This characteristic enables a straightforward extension that is agnostic about the inner layers and thus applicable to any discriminator network. In this first approach, the original prediction $y = D(\boldsymbol{x})$ is linearly combined with a *penalty vector* $\boldsymbol{c} = [c^{(1)}(\boldsymbol{x}), c^{(2)}(\boldsymbol{x}), ..., c^{(m)}(\boldsymbol{x})]$ and the final result y' is given by a new output layer with a single unit receiving in input

$$\alpha_0 y + \sum_{i=1}^{m} \alpha_i \boldsymbol{c}_i,$$

where α_i are trainable weights balancing the importance of resembling input data and satisfying constraints. Since the output of the discriminator $y' = D(\boldsymbol{x})$ represents the probability that $\boldsymbol{x}$ came from p_{data} rather than p_g and constraints are evaluated via penalty functions assigning greater values to imperfect object, such as those produced by p_g, CANs should automatically learn to assign negative values to all the weights α_i. By doing so, the more an input object violates the constraints, the easier it will be predicted as sampled. Perfect objects from training set will incur in no penalization during all the training procedure, regardless of the current values of α_i, since their penalty vectors will always contain only 0 values. In addition, CANs are given the possibility to learn which constraints are more useful during training since they can assign different scores to the multipliers α_i.

A similar approach consists in using the same penalty vector in an inner layer, before the final output of the discriminator is computed. Of course, such approach is no more agnostic about the inner architecture of the ANN, thus possibly making CANs no more generally applicable around any black-box network D. However, in practice, the requirements to apply this method are easily met since it is reasonable to believe that any network D will have a final hidden layer with some dozens of hidden units. More formally, given a discriminator $D(\boldsymbol{x}) = f^{(n)}(f^{(n-1)}(...(f^{(2)}(f^{(1)}(\boldsymbol{x})))))$ composed of n layers, the final result y' is given by the same output layer $f^{(n)}$ receiving in input

$$f^{(n-1)}(\sum_{i=1}^{u} \beta_u \boldsymbol{h}_u + \sum_{i=1}^{m} \alpha_i \boldsymbol{c}_i),$$

where u is the number of hidden units of the last inner layer and $\boldsymbol{h} = f^{(n-2)}(...(f^{(2)}(f^{(1)}(\boldsymbol{x}))))$ is the the output of the previous hidden layer. The rationale behind this design trick is to slightly increase the impact of penalty vectors, exploiting them before the final decision is computed by the discriminator. Overall, the new discriminator has higher capacity than the one of the first approach and it is the one tested.

3.2.3 Constraints timing

The last important design choice for CANs regards *time intervals* in which hints from penalty functions should be actively used to train the model. The following considerations hold regardless of other design choices, such as those involving alterations of the training procedure or the ANNs architectures. Furthermore, they can be crucial for the final performance and efficiency of the model. This is especially true when evaluating penalty functions has a remarkable impact on execution times.

Many modern deep learning models are parametric on some conditions in order to dynamically change the function computed by the ANN in response to some events. For instance, some layers could be added, others could be dropped or some weights could be frozen for a certain period of time. This flexibility opens up a wide spectrum of possibilities.

The two time-related critical issues for CANs lie in the moment in which constraints should be introduced and in the duration for which they should be used. The number of combinations of the possibilities can be large: only some of them are now discussed.

First of all, constraints can be introduced since the beginning of the adversarial training to immediately provide the discriminator with a powerful tool to distinguish real data from sampled ones. In this case the generator will be soon forced to satisfy constraints, perhaps at the cost of ignoring other implicit global properties.

One alternative is to introduce penalty functions only after a certain number of epochs has passed. By doing so, the generator will first learn how to produce plausible objects and then, hopefully, it will refine its knowledge to also satisfy constraints. The number of epochs after which constraints should be introduced can be decided in advanced or it can be dynamically determined during training, for instance as a function of the current performance of the model on the validation set.

The orthogonal decision is for how long penalty functions should be computed once constraints have been introduced in the learning procedure. For example, once enabled, they could be kept active until the training stopping criterion is reached and learning is stopped, or they could be repeatedly switched on and off. These questions are investigated in the experimental stage.

3.2.4 Secondary issues

Besides the aforementioned design choices, there are some others that are still worth describing, even if their impact on the performance is probably negligible.

A minor design choice regards the initialization of the ANN weights of the penalty vector $\boldsymbol{c}$, that are expected to become negative and progressively smaller as training proceeds. This prior knowledge may be used to set a negative initial values to all the weights, rather than random ones, relieving the discriminator of the burden of understanding that penalty functions can be used to effectively distinguish real and sampled data. This kind of initialization is methodologically fair since it can be considered another form of rule-based knowledge instillation technique of CANs. However, finding the optimal initial values may be non-trivial and it is practically more convenient to let D learn them automatically.

Finally, rather than using penalty functions, CANs could involve some kind of *reward functions*. For instance, the set of constraints $\mathbb{C} = \{c^{(1)}, c^{(2)}, ..., c^{(m)}\}$ could be modified in $\mathbb{C}' = \{1 - c^{(1)}, 1 - c^{(2)}, ..., 1 - c^{(m)}\}$. In this case, the penalty vector $\boldsymbol{c}$ becomes a reward vector and D is expected to learn positive weights for it. From a high-level point of view, this corresponds to increase the probability of considering an object as real if it satisfies the constraints. The two versions seems mathematically equivalent and some preliminary experiments have shown that this modification does not have any notable effect on the final result. For such reason, all the presented experiments only use set of penalty functions.

Chapter 4

Experiments and results

"To apply oneself to great inventions, starting from the smallest beginnings, is no task for ordinary minds; to divine that wonderful arts lie hid behind trivial and childish things is a conception for superhuman talents."

Galileo Galilei

There exists many works in literature regarding controllable content generation, although most of them involve procedural generation with models different from ANNs, such as multi-dimensional Markov chains [33]. Unfortunately, large constrained objects data sets to be used as a common benchmark for this task are still missing since many authors have not shared their data. For this reasons, all the experiments have been performed on synthetic data sets. For sure these results are not sufficient alone to prove the effectiveness of the proposed model on a wide range of applications, nevertheless programmatically generated data are useful to easily test specific design choices, especially in an early research stage.

4.1 Data sets

All the experiments have been performed on two similar data sets. To each of them is assigned a label by which they can be easily later referred to.

The first data set is composed of $23,040$ black-and-white square images of 20x20 pixel. The training set is made up of $18,944$ items, while the test set of the remaining $4,096$ items. Each image contains exactly two different white regular polygons randomly chosen among triangles (30%), squares (30%) and rhombi (40%). Each polygon has an area of exactly 25 pixels. Polygons are randomly placed on a black background without overlapping and avoiding the outer border. This data set is labelled as *poly20*.

The second data set is identical to the first but for the usage of the pixels on the outer border. Each of them now represents a *parity check* of other close inner pixels. In particular, a pixel lying

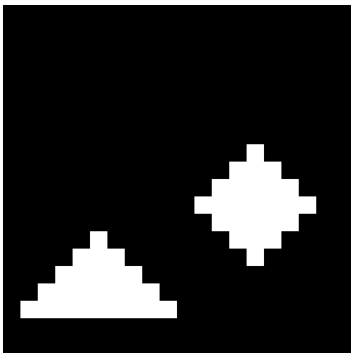

Figure 4.1: Sample from *poly_20* (zoom).

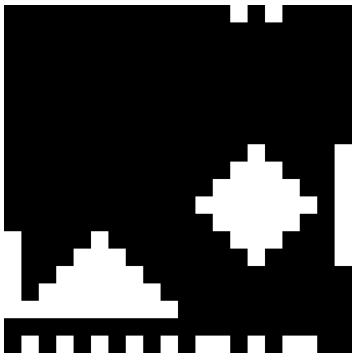

Figure 4.2: Sample from *poly_20_pc* (zoom).

on a vertical border (left or right) will be the parity check of the 9 inner nearest pixels lying on the same row (left-half or right-half). Similarly, a pixel lying on a horizontal border (top or down) will be the parity check of the 9 inner nearest pixels lying on the same column (top-half or bottom-half). By assigning the values of 0 and 1 to, respectively, black and white pixels, we can formally describe the borders of a matrix-like image I:

$$\forall i = 1, ..., 20 : \begin{cases} I[i][0] = \sum_{k=2}^{10} I[i][k] \mod 2 & \text{(bottom-half columns)} \\ I[i][20] = \sum_{k=11}^{19} I[i][k] \mod 2 & \text{(top-half columns)} \\ I[0][i] = \sum_{k=2}^{10} I[k][i] \mod 2 & \text{(left-half rows)} \\ I[20][i] = \sum_{k=11}^{19} I[k][i] \mod 2 & \text{(right-half rows)} \end{cases} . \tag{4.1}$$

The second data set is labelled as *poly20_pc*.

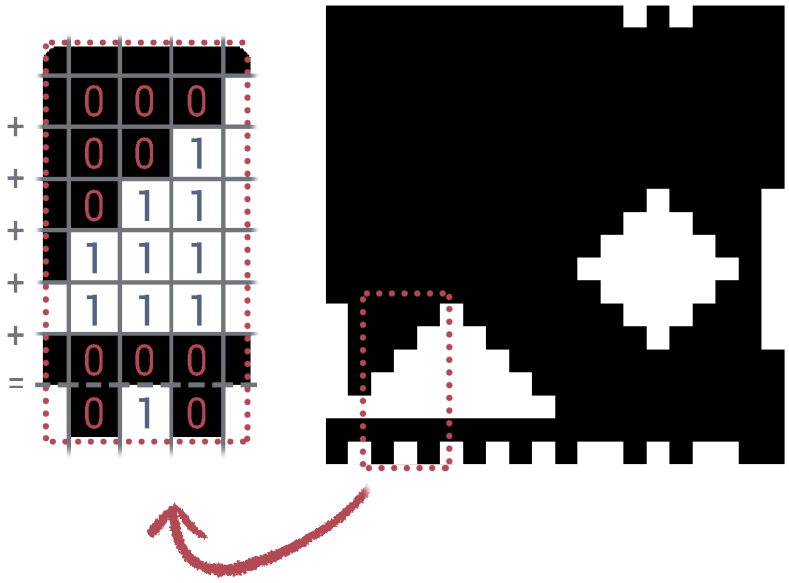

Figure 4.3: Schematic representation of how parity checks are computed.

The two data sets have been generated according to the following criteria:

- the number of items in the data sets is as large as possible, yet bounded by the constraints on the polygons area and their non-overlapping placement;
- small 20x20 pixel images allow a faster training with modest-size ANNs architectures that comes handy in an exploratory research scenario;
- placing two polygons on each image, instead of a single one, avoids the *mode collapsing problem* of GANs. In fact, by doing so, the generator can not try to learn only the easiest shape, that is the one more frequently classified as a training datum by the discriminator. Instead, it is continually forced to learn combinations of them and, consequently, each of them. This evidence is the result of some preliminary experiments on GANs;
- rhombi are supposed to be the most difficult polygons to learn among the three available, so they are slightly more present in the data sets;
- the constant value of 25 for the polygons area maximises the number of convex regular polygons that can be discretely represented;

- each parity check is computed only on one half of the image to model a function of modest complexity. If desired, these constraints can become harder by extending them on entire rows/columns.

A final note regards the choice of parity check pixels in *poly20_pc*. Their goal is to dramatically reduce the probability for the generator to output a perfect object without having first learnt the rules that define the border of the images. Each parity check, in fact, halves the probability to generate by chance a perfect object. In this way, the generator is forced to learn the relations expressed in 4.1. Furthermore, from a computational point of view, checking if a pixel is correct is very fast. Hence, parity checks seem a suitable family of functions to test the proposed deep generative model: functions that are hard to learn and easy to verify.

4.2 Model implementation

The deep generative framework is implemented in Python and will be open-sourced. The input of the software is a JSON file describing the experiment to be run. It contains information about the data set to be generated or reused, the ANNs architectures and some training parameters and hyperparameters. Every descriptor is thus sufficient to train and evaluate the entire model. Experiments are deterministic and reproducible since JSON files also contain the seeds for the random distributions.

ANNs are trained using TensorFlow [1], the open-source ML framework developed by Google. The experiment descriptor includes function names to be used to build the model, together with references to the loss function and the optimization algorithm.

The JSON file can optionally contain a list of constraints. If provided, the resulting generative model will be that of CANs, otherwise simple BGANs. In the first case, constraints will be used during training according to the ANN model specified.

To reduce execution times, penalty functions are dynamically compiled to native machine instructions with Numba [18] and they are evaluated in parallel on sampled data by all available CPU cores, avoiding Python's Global Interpreter Lock. In addition, generated samples are also independent of each other, so they can be further processed in parallel in a thread-safe way. On the contrary, data sets penalty vectors are computed once, at the beginning of the training algorithm, and cached, since their values never change. As a result, this single-GPU/multi-core implementation provides similar running times for BGANs and CANs.

Statistics are collected during the training mainly via TensorBoard, the native visualization tool tightly integrated with TensorFlow. This is the best solution to get insights on the training process since some parts of the computation are compiled for the GPU and not exposed via any API. Statistics can be collected with a customized frequency both on the validation set and on the test set.

4.3 Experimental protocol

Probabilistic generative models can be used for many different tasks, such as compression, denoising, semi-supervised learning or unsupervised feature learning. Given this wide range of applications, a lot of heterogeneity exists in the way these models are trained and evaluated. Some works have proved that good performance with respect to one criterion do not imply good performance with respect to other criteria [34]. Since being able to generate realistic samples from the data distribution is one of the goals of a generative model, sometimes they are simply evaluated by visually inspecting the samples. Typically, the evaluation is carried out by experimental subjects who do not know the source of the samples [3]. Unfortunately, it is possible for a very poor probabilistic model to produce very good samples. However, the framework of this work enables an objective evaluation via penalty functions and no subjective assessment is necessary.

ANNs generally require a significative amount of random numbers to be trained. They are used, for instance, to initialize the weights of the network or to randomly shuffle mini-batches before every new training epoch. GANs-based models, in particular, involve a far greater amount of noise, since new vectors are required to sample data from p_g and this operation is usually performed a million times.

However, due to computational implications, extensively testing ANNs can be really time-consuming and it can be hard to get results with a high significance level.

Experiments have been repeated multiple times with different random seeds. Each plot shows the number of repetitions and highlights the first, second and third quartiles of the aggregated results regarding the performance measure of interest.

Networks have been evaluated for their capability to satisfy constraints individually and simultaneously. In the first case, plots show the average over all constraints of the percentages of objects satisfying each of them. In the latter, they show the percentage of perfect objects produced. In both cases, the images involved in the measurement are those generated during a training epoch on the test set after which no optimization is performed.

The tested BGANs and CANs architectures are equivalent, with the obvious exception of the hidden units reserved for the constraints, that are CANs-specific. Networks are also initialized in the same way and visually evaluated on the same input noise. To measure their capability of generating objects satisfying constraints, a number of statistics are periodically collected during training, such as the percentage of perfect objects being generated or the mean error on each penalty function.

Stochastic optimization is performed via Adam [17] and batch normalization [14] is used on some layers to accelerate training.

The reported execution times refer to the experiments performed on *Maximillian Pegasus*[1]: CPU Intel Core i7-5930K 3.5GHz, 6 cores; 3x GPU MSI GeForce GTX 980, 4GB; RAM 16GB; OS Windows 10; Python version 3.5.2; TensorFlow-GPU version 1.6.0; Nvidia driver 391.01; CUDA version 9.0; cuDNN version 7.0.5.

4.4 Model architectures

Not being interested in finding the best architecture for the task, but only in comparing BGANs and CANs with the same architecture, the experimentation does not involve model selection. On the contrary, the architecture initially adopted, together with all its hyperparameters, is the one described in the original work on BGANs. Preliminary works on BGANs have proved this architecture to have enough capacity to learn in a reasonable amount of time how to generate images of different kinds. Similarly to data sets, also architectures are labelled for convenience.

Generators and discriminators tend to have symmetric configurations. This choice makes the adversarial training procedure quite stable over the epochs and overall effective. Discriminator architectures usually involve convolutional layers to learn spatially local patterns of input polygons. Vice versa, generator architectures use the opposite operation, the transposed convolution layer, often simply called deconvolution, to expand input noise into meaningful shapes.

The following is a schematic description of the architectures. For each of them is provided the complete list of composing layers, along with their parameters and input/output dimensions. The symbol $\xrightarrow{r}$ is used to indicate reshaping operations. The symbol $\xrightarrow{t}$ represents a transpose operation. The symbol $\xrightarrow{c}$ denotes that the layer has been expanded to introduce hidden units to be fed with values computed by penalty functions.

The placeholder ? is used for input values. For generator architectures usually it is $|\boldsymbol{z}| \times |\textit{mini-batch}|$, while for discriminator architectures it is $|\textit{samples}| \times |\textit{mini-batch}|$, where $|\textit{samples}|$ represents the number of samples drawn for each generator output, as specified by BGANs. Default values are $|\boldsymbol{z}| = 64$, $|\textit{samples}| = 20$, $|\textit{mini-batch}| = 64$.

bgan20_gen: generator network originally used for BGANs, adapted for 20x20 pixel data sets.

1. $(?, 64) \rightarrow (?, 1024)$. Type: dense (1024 output units); activation: ReLU; batch normalization.
2. $(?, 1024) \rightarrow (?, 3200)$. Type: dense (3200 output units); activation: ReLU; batch normalization.

[1] Lorenzo, thanks from the bottom of my heart.

3. $(?, 3200) \xrightarrow{r} (?, 5, 5, 128) \rightarrow (?, 10, 10, 64)$. Type: deconvolution; filters: 64; kernel size: 5; strides: 2; kernel initializer: orthogonal; bias initializer: zeros; activation: ReLU; batch normalization.
4. $(?, 10, 10, 64) \rightarrow (?, 20, 20, 1)$. Type: deconvolution; filters: 1; kernel size: 5; strides: 2; kernel initializer: orthogonal; bias initializer: zeros; activation: linear.

bgan20_discr_h32l: discriminator network similar to the one originally used for BGANs, adapted for 20x20 pixel data sets. Layer 4 is added for consistency with the corresponding constrained discriminator layer reserved for penalty function values (*can20_discr_h32l*, layer 5).

1. $(?, 20, 20, 1) \rightarrow (?, 10, 10, 64)$. Type: convolution; filters: 64; kernel size: 5; strides: 2; kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
2. $(?, 10, 10, 64) \rightarrow (?, 5, 5, 128)$. Type: convolution; filters: 128; kernel size: 5; strides: 2; kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
3. $(?, 5, 5, 128) \xrightarrow{r} (?, 3200) \rightarrow (?, 1024)$. Type: dense (1024 output units); kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
4. $(?, 1024) \rightarrow (?, 32)$. Type: dense (32 output units); kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
5. $(?, 32) \rightarrow (?, 1)$. Type: dense (1 output unit); kernel initializer: Xavier; activation: linear.

can20_discr_h32l: discriminator layer with constraints introduced before computing the original output, in an intermediate layer. The choice of 32 hidden units for layer 5 is arbitrary.

1. $(?, 20, 20, 1) \rightarrow (?, 10, 10, 64)$. Type: convolution; filters: 64; kernel size: 5; strides: 2; kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
2. $(?, 10, 10, 64) \rightarrow (?, 5, 5, 128)$. Type: convolution; filters: 128; kernel size: 5; strides: 2; kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
3. $(?, 5, 5, 128) \xrightarrow{r} (?, 3200) \rightarrow (?, 1024)$. Type: dense (1024 output units); kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
4. $(?, 1024) \rightarrow (?, 32)$. Type: dense (32 output units); kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
5. *if* $|\mathbb{C}| > 0$: $(?, 32) \xrightarrow{c} (?, 32 + |\mathbb{C}|)$. Type: dense ($32 + |\mathbb{C}|$ output units); kernel initializer: Xavier; bias initializer: Xavier; activation: linear.
6. $(?, 32 + |\mathbb{C}|) \rightarrow (?, 1)$. Type: dense (1 output unit); kernel initializer: Xavier; bias initializer: Xavier; activation: linear.

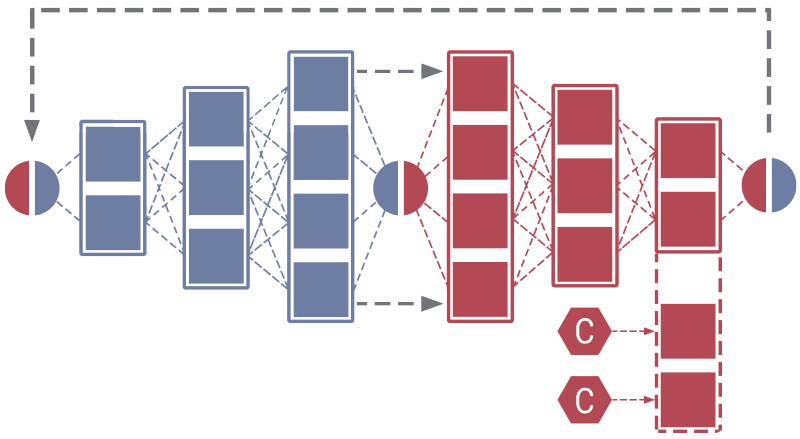

Figure 4.4: Schematic representation of CANs with *can20_discr_h32l*. Constraints are introduced in the discriminator (red) in the final hidden layer.

can20_discr_outl: discriminator layer with constraints introduced after computing the original output, in an additional final layer.

1. $(?, 20, 20, 1) \rightarrow (?, 10, 10, 64)$. Type: convolution; filters: 64; kernel size: 5; strides: 2; kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
2. $(?, 10, 10, 64) \rightarrow (?, 5, 5, 128)$. Type: convolution; filters: 128; kernel size: 5; strides: 2; kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
3. $(?, 5, 5, 128) \xrightarrow{r} (?, 3200) \rightarrow (?, 1024)$. Type: dense (1024 output units); kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
4. $(?, 1024) \rightarrow (?, 1)$. Type: dense (1 output units); kernel initializer: Xavier; activation: leaky ReLU ($a = 0.2$).
5. *if* $|\mathbb{C}| > 0$: $(?, |\mathbb{C}|) \xrightarrow{c} (?, 1)$. Type: dense (1 output unit); kernel initializer: Xavier; activation: linear.
6. *if* $|\mathbb{C}| > 0$: $(?, 1) \xrightarrow{c} (?, 2) \rightarrow (?, 1)$. Type: dense (1 output unit); kernel initializer: Xavier; bias initializer: Xavier; activation: linear.

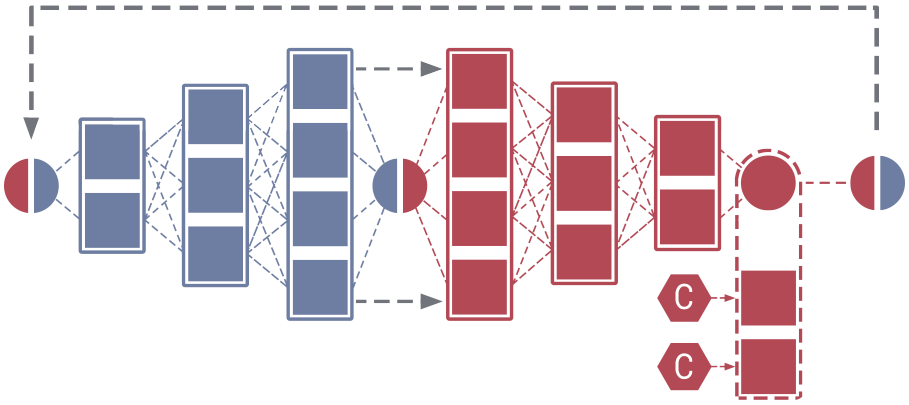

Figure 4.5: Schematic representation of CANs with *can20_discr_outl*. Constraints are introduced in the discriminator (red) after the final hidden layer.

4.5 Penalty functions

According to the criteria by which data sets *poly20* and *poly20_pc* have been generated, it is possible to define some constraints on the images. The following is a schematic list, with their corresponding penalty functions and labels.

smaller_area: require the image to not contain less than the expected number of white pixels (2×25). The penalty is proportional to the underestimation.

$$f(I) : min(1, max(0, 2 \cdot 25 - \sum_{i=1}^{20} \sum_{j=1}^{20} M[i][j])/(20 \cdot 20 - 2 \cdot 25))$$

greater_area: require the image to not contain more than the expected number of white pixels (2×25). The penalty is proportional to the overestimation.

$$f(I) : min(1, max(0, -2 \cdot 25 + \sum_{i=1}^{20} \sum_{j=1}^{20} M[i][j])/(20 \cdot 20 - 2 \cdot 25))$$

convex: require the image to only contain convex shapes. The penalty is proportional to the number of pixels contained in concave zones defined by polygons. Shapes are identified with a breadth-first search in a graph representing the image. Wrong pixels are detected scanning the image by rows, columns and diagonals.

$$f(I) : min(1, (\sum_{i=1}^{20}\sum_{j=1}^{20} M[i][j] \text{ if in concave zone})/(2 \cdot 20 \cdot 20))$$

parity_check_p_i: require the image to correctly set a parity check pixel on the border. The placeholder *p* can be one of {rl, rr, ct, cb} to indicate that the *i-th* pixel involves, respectively, a left-half row, a right-half row, a top-half column or a bottom-half column. The penalty assigned is binary: 0 if the relation 4.1 is true for the *i-th* pixel, 1 otherwise.

All the penalty functions can be straightforwardly extended to data sets of larger images or with a different number of polygons.

Since experiments have been performed reusing the same sets of penalty functions, additional labels are also provided for them (range notation adopted for conciseness):

area_convex: $\mathbb{C} = \{smaller\_area,\ greater\_area,\ convex\}$, $|\mathbb{C}| = 3$.

some_pc: $\mathbb{C} = \{smaller\_area,\ greater\_area,\ parity\_check\_rl\_1\text{-}9,\ parity\_check\_ct\_1\text{-}9\}$, $|\mathbb{C}| = 20$.

all_pc: $\mathbb{C} = \{smaller\_area,\ greater\_area,\ parity\_check\_rl\text{-}ct\text{-}rr\text{-}cb\_1\text{-}18\}$, $|\mathbb{C}| = 74$.

4.6 Experiments

The goal of the first experiment is to compare BGANs and CANs with the simplest data set, *poly20*, and a small set of constraints, *area_convex*. Figures 4.6 and 4.7 show that BGANs are superior to CANs both for the percentage of perfect objects generated and for average constraints satisfaction. The reason is probably related to the constraints involved, that are uninformative and insufficient to better guide the generator in the search space for CANs while introducing additional parameters that unnecessarily burden the training. This evidence justifies the use of the other data set, *poly20_pc*, and more informative penalty functions.

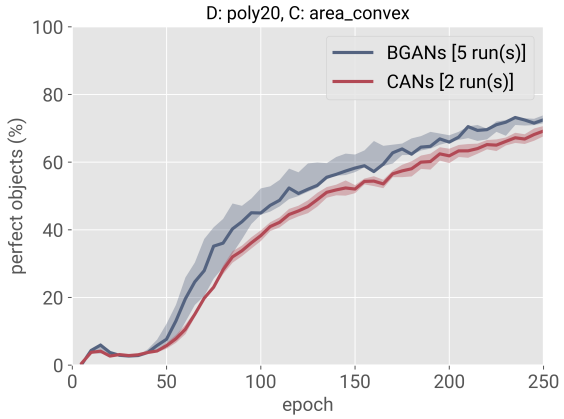

Figure 4.6:
BGANs: *bgan20_gen* + *bgan20_discr_h32l*.
CANs: *bgan20_gen* + *can20_discr_h32l*.

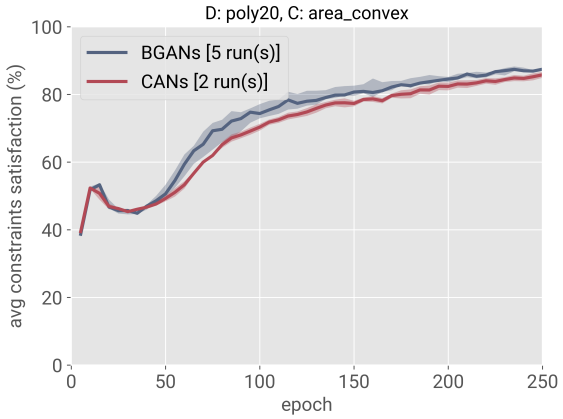

Figure 4.7:
BGANs: *bgan20_gen* + *bgan20_discr_h32l*.
CANs: *bgan20_gen* + *can20_discr_h32l*.

Opposite results are observed when the data distribution to be learnt is harder, such as for *poly20_pc*. In this case, CANs benefit from the constraints expressed in *some_pc* and achieve better performance than BGANs, as depicted in figures 4.8 and 4.9.

The differences between the two models are nevertheless small, with the biggest improvement observed in average constraints satisfaction. Figure 4.10 shows that the gap between BGANs and CANs tends to disappear as training continues. This result is reasonable, since BGANs are provided with enough capacity to learn the correct distribution anyway. Nevertheless, it can be considered significative because CANs only require small additional time to be trained, as demonstrated in figure 4.11.

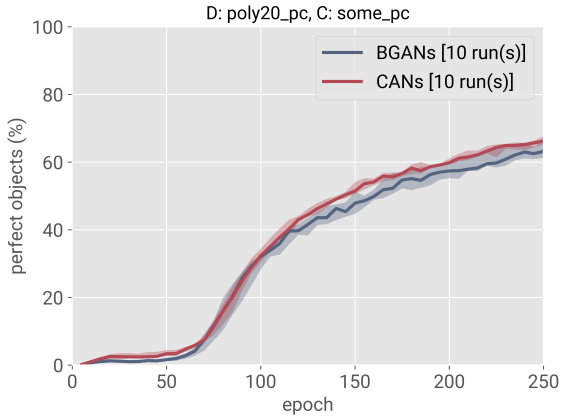

Figure 4.8:
BGANs: *bgan20_gen + bgan20_discr_h32l.*
CANs: *bgan20_gen + can20_discr_h32l.*

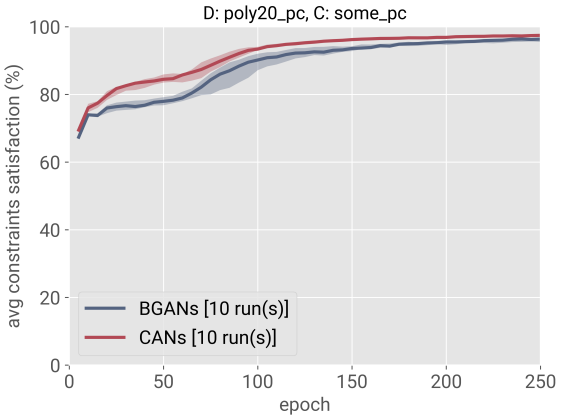

Figure 4.9:
BGANs: *bgan20_gen + bgan20_discr_h32l.*
CANs: *bgan20_gen + can20_discr_h32l.*

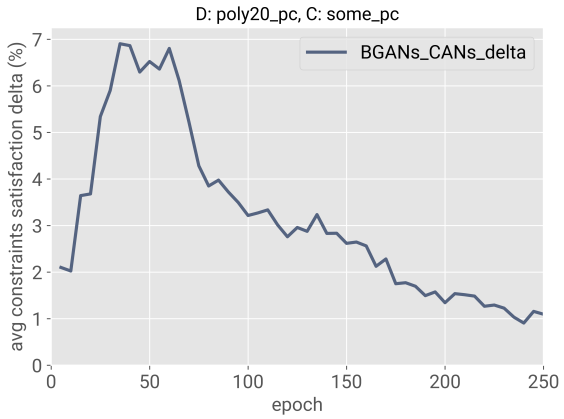

Figure 4.10:
BGANs: *bgan20_gen + bgan20_discr_h32l.*
CANs: *bgan20_gen + can20_discr_h32l.*

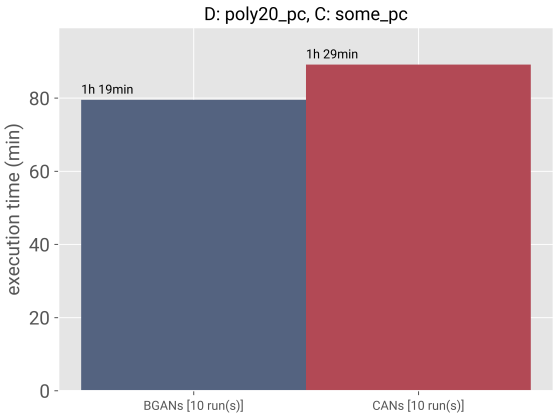

Figure 4.11:
BGANs: *bgan20_gen + bgan20_discr_h32l.*
CANs: *bgan20_gen + can20_discr_h32l.*

Some insights can be drawn when comparing two CANs, one using constraints on the last hidden layer and the other on the output layer. Figure 4.12 shows how the first architecture is more effective in learning each single constraint in the long term, producing many more perfect objects. On the other side, figure 4.13 suggests that modelling the discriminator's output and the constraints as having the same relevance may largely impact the average number of constraints satisfied at the beginning of the training procedure. This trade-off deserves further research in future work.

In all the experiments presented so far using the set of constraints *some_pc* it is evident that networks achieve excellent performances in satisfying each constraint on average (95%+), but not in generating perfect objects (∼65%). The same experiments have also been conducted using a double amount of training epochs. Figure 4.14 shows that, on longer runs, not only the number of perfect objects increases over epochs, but the gap between BGANs and CANs is also preserved.

The ratio behind using only a subset of available parity checks as constraints is that network capable of satisfying some of them should in principle be able to satisfy all of them. This hypothesis is tested by using the complete set of constraints, *all_pc*.

Figure 4.15 reports the results achieved with a set up equal to that of figure 4.8 but for the usage of the extended constraints set. As expected, both the models now require more training epochs before being able to generate perfect objects (at least 50). A more interesting observation regards the same performance measured at the end of the training algorithm. It is possible to notice that CANs succeed in generating objects satisfying the two set of constraints with approximately the same probability, while BGANs suffer a great performance loss. The result is better highlighted in figure 4.16, where a

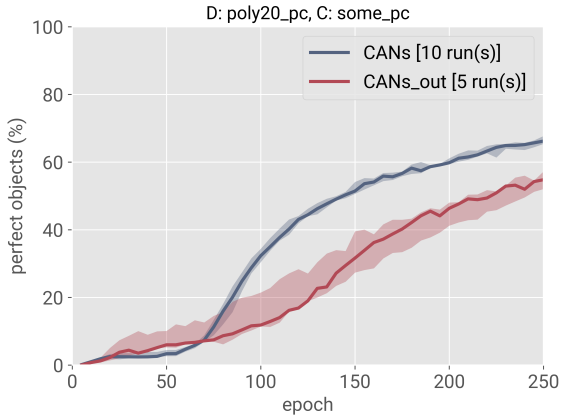

Figure 4.12:
CANs: *bgan20_gen + can20_discr_h32l*.
CANs_out: *bgan20_gen + can20_discr_outl*.

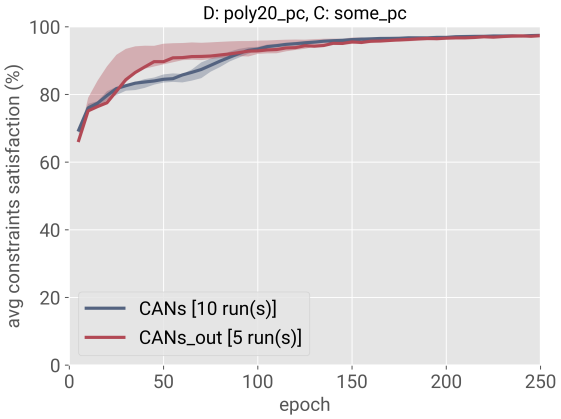

Figure 4.13:
CANs: *bgan20_gen + can20_discr_h32l*.
CANs_out: *bgan20_gen + can20_discr_outl*.

remarkable gap of 16% can be observed between the two architectures at the hundredth epoch.

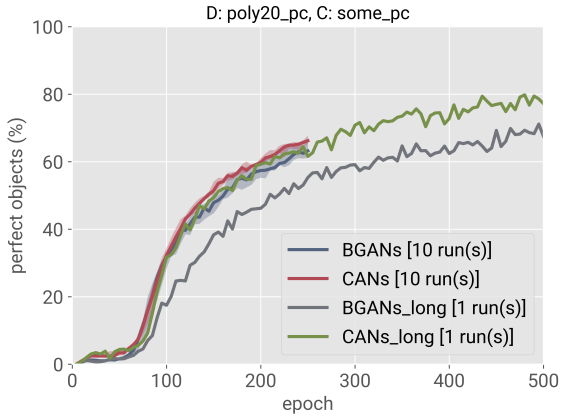

Figure 4.14:
BGANs: *bgan20_gen + bgan20_discr_h32l*.
CANs: *bgan20_gen + can20_discr_h32l*.
BGANs_long = BGANs
CANs_long = CANs.

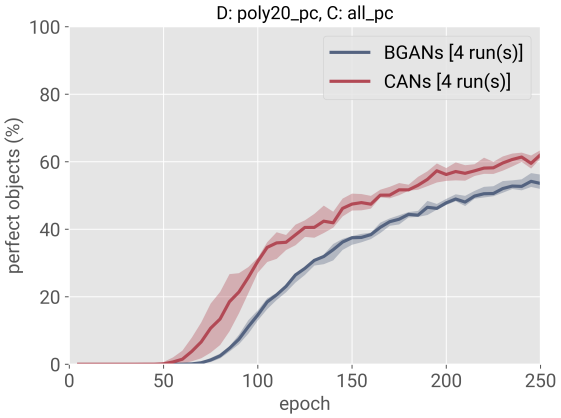

Figure 4.15:
BGANs: *bgan20_gen + bgan20_discr_h32l*.
CANs: *bgan20_gen + can20_discr_h32l*.

It is worth noting that, due to the kind of constraints involved and to the multi-core implementation, the difference in execution times remains small, as reported in figure 4.17. This proves that the model is able to scale and to learn how to satisfy constraints in polynomial-time if penalty functions can be evaluated in polynomial-time as well. Results regarding average constraints satisfaction are similar to the previous ones and thus omitted.

In all the results presented so far CANs appear to improve the BGANs baseline when they have early access to constraints and when penalty functions can be effectively used to guide the network in the search space. However, none of the experiments has evaluated the impact of the size of the data set. To estimate the impact of the number of training examples on the model performance a new smaller data set, *poly20_pc_small*, is used. It contains one half of the training examples of *poly20_pc*, thus being composed of $13,568$ images ($9,472$ for training, $4,096$ for testing).

Results on the same experimental set up are reported in figures 4.18 and 4.19. With the smaller data set the gap between CANs and BGANs increases considerably according both to average constraints satisfaction and to the number of perfect objects generated. In this configuration constraints are the most effective and CANs score an improvement superior to 20% to their counterpart, as highlighted in figure 4.20.

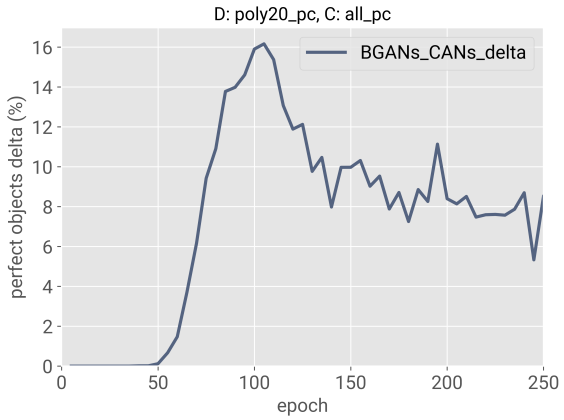

Figure 4.16:
BGANs: *bgan20_gen + bgan20_discr_h32l.*
CANs: *bgan20_gen + can20_discr_h32l.*

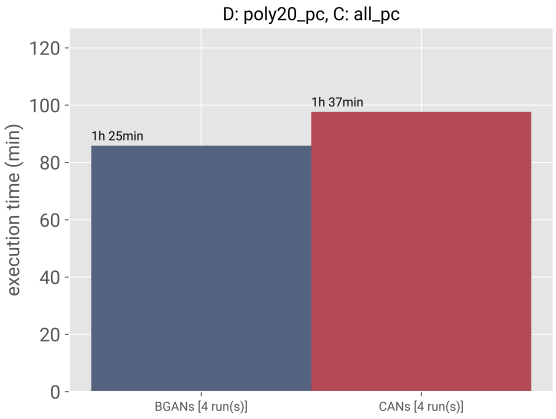

Figure 4.17:
BGANs: *bgan20_gen + bgan20_discr_h32l.*
CANs: *bgan20_gen + can20_discr_h32l.*

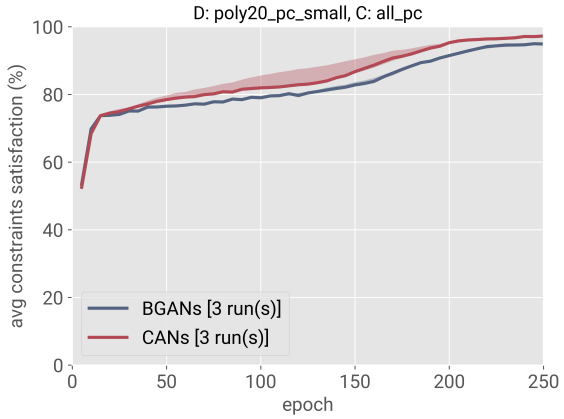

Figure 4.18:
BGANs: *bgan20_gen + bgan20_discr_h32l.*
CANs: *bgan20_gen + can20_discr_h32l.*

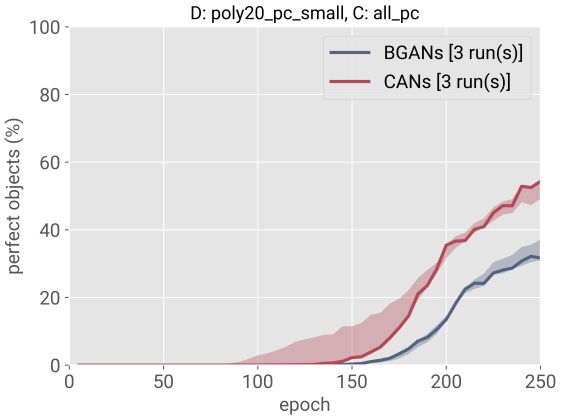

Figure 4.19:
BGANs: *bgan20_gen + bgan20_discr_h32l.*
CANs: *bgan20_gen + can20_discr_h32l.*

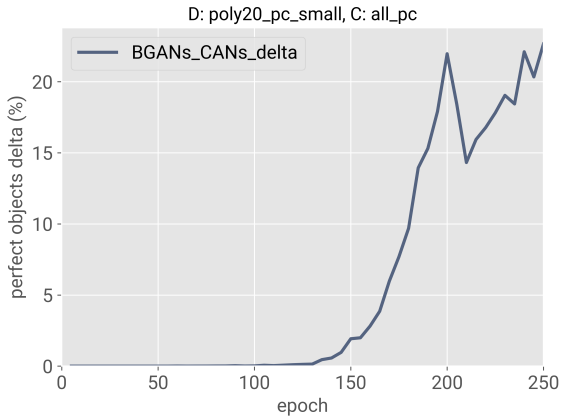

Figure 4.20:
BGANs: *bgan20_gen + bgan20_discr_h32l.*
CANs: *bgan20_gen + can20_discr_h32l.*

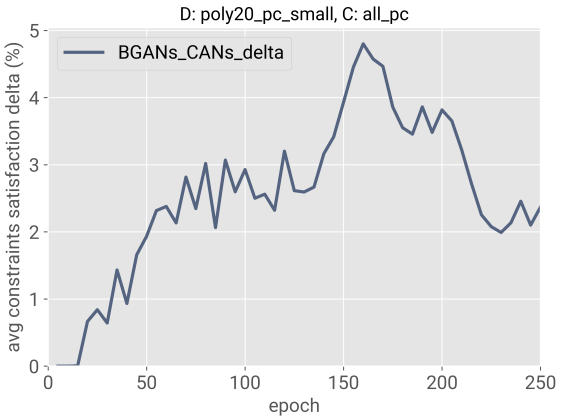

Figure 4.21:
BGANs: *bgan20_gen + bgan20_discr_h32l.*
CANs: *bgan20_gen + can20_discr_h32l.*

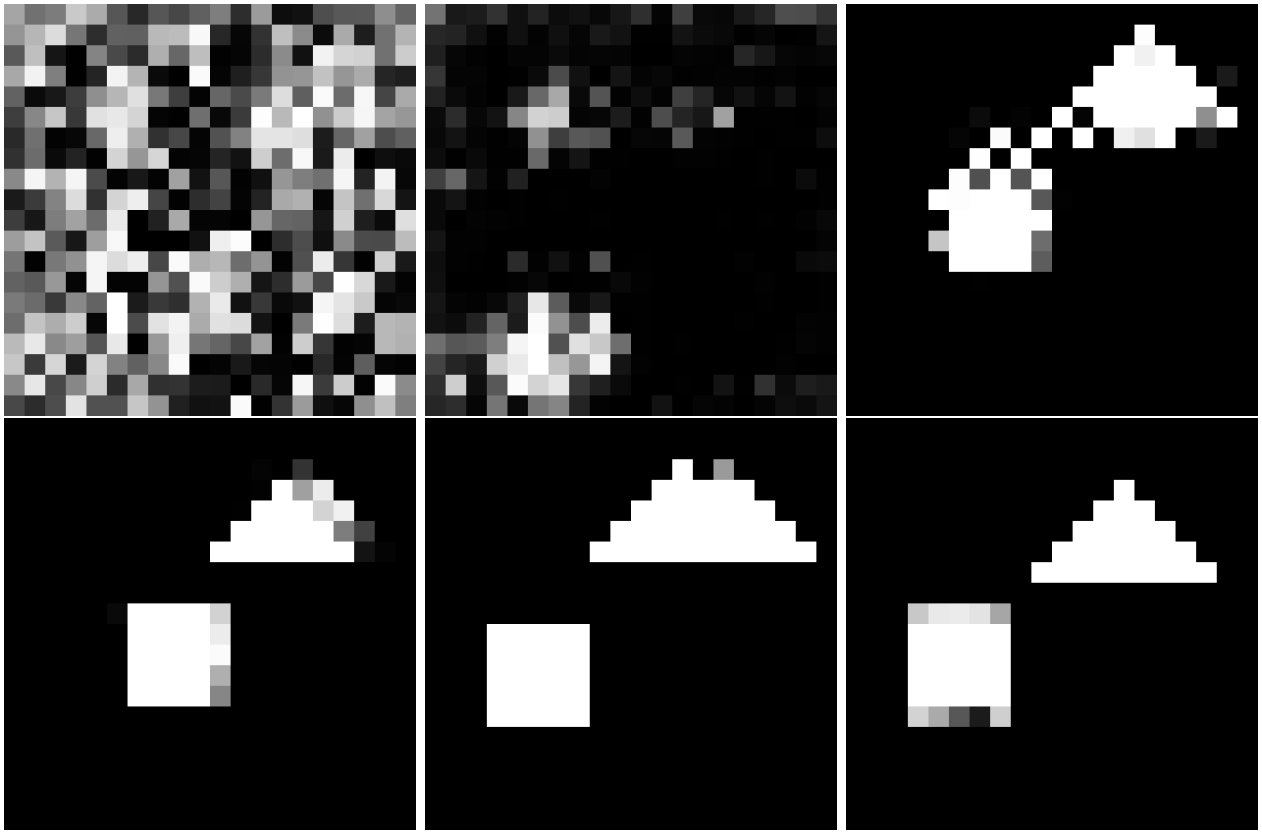

Figure 4.22: Samples drawn at different epochs from the generator trained on *poly20* (zoom).

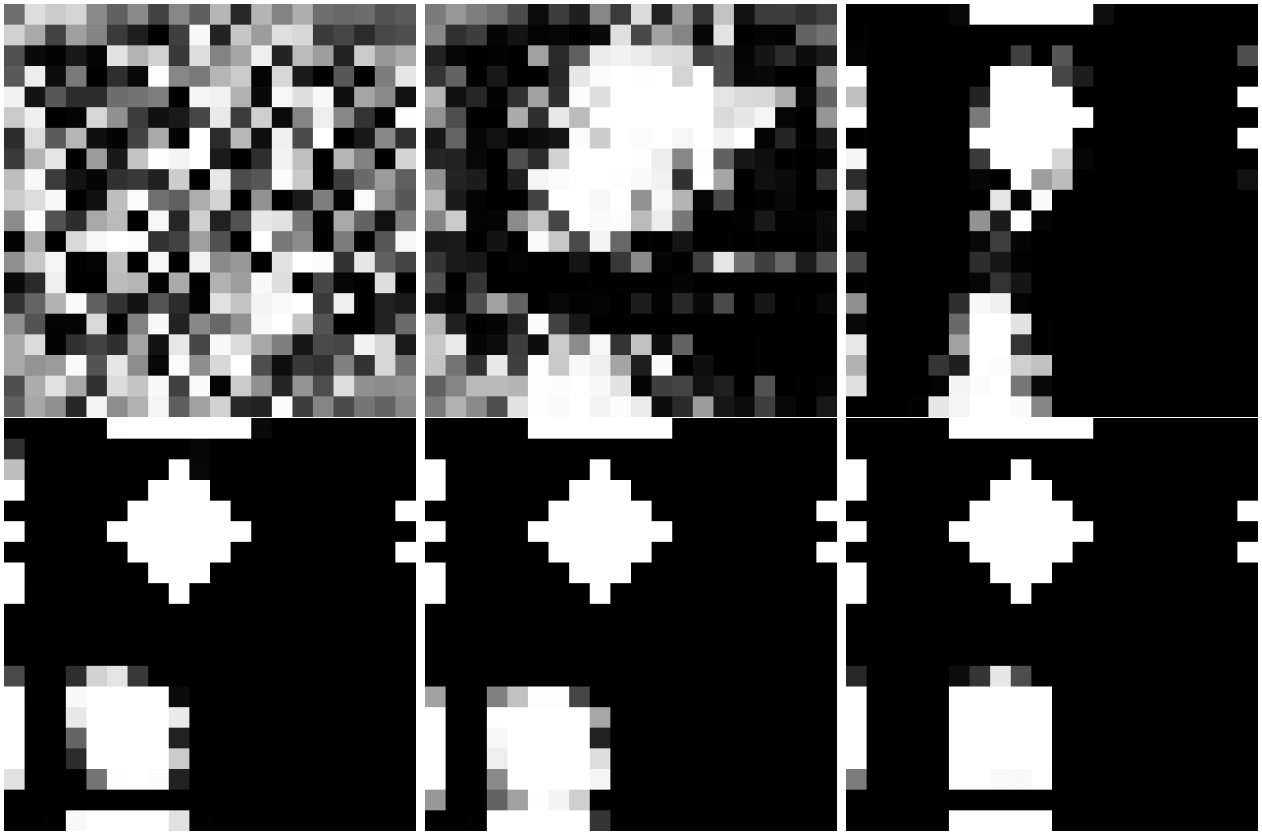

Figure 4.23: Samples drawn at different epochs from the generator trained on *poly20_pc* (zoom).

Chapter 5

Conclusions

The presented results show that constrained adversarial networks enable the encoding of useful prior knowledge in the game theoretic scenario initially proposed by generative adversarial networks. The conducted experiments reveal how constraints on discrete domains can be introduced in the learning procedure to instil knowledge in the deep generative model. The effect is that the proposed architecture is able to approximate the data distribution more closely than its constraints-unaware counterpart of boundary-seeking generative adversarial networks.

Furthermore, by exploring different design choices and analysing several factors, this work offers a number of insights on how these may individually impact on the final performance of the model. For instance, it is observed that the improvement over the baseline seems to increase proportionally with the amount of information conveyed by constraints and becomes particularly relevant when training data is not abundant.

The results also show that the knowledge instillation process is effective even if the information is indirectly provided to the generator via penalty functions computed by the discriminator. Evaluating other design choices for the model is left for future work. Many other architectures, in principle, could provide equivalent or better results. For instance, the original adversarial model may be extended to introduce a third constraints-aware network acting as a regulariser.

Further research is necessary to generalize our results to a wider variety of domains and constraints types. The synthetic data sets used in our experimental setup only allow some preliminary considerations and further testing is required on real-world data. However, the encouraging empirical results indicate a strong potential of constrained adversarial networks to be useful in many other contexts involving constraints beside the obvious one of object generation. For instance, due to its performance in generating new data from input noise, constrained adversarial network could become a building block for larger systems addressing more complex problems, such as optimization, or with specific requirements, such as uniform solution sampling.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2011.

[3] E. Denton, S. Chintala, A. Szlam, and R. Fergus. *Deep generative image models using a laplacian pyramid of adversarial networks*, volume 2015-January, pages 1486–1494. Neural information processing systems foundation, 2015.

[4] M. Dorigo and M. Birattari. Ant colony optimization. In *Encyclopedia of machine learning*, pages 36–39. Springer, 2011.

[5] Euclid. *The Thirteen Books of the Elements Translated By Sir Thomas Heath 3 Vols.* Dove Publishing, 2000.

[6] L. Euler. *Methodus inveniendi*. 1744.

[7] K. Ganchev, J. Graça, J. Gillenwater, and B. Taskar. Posterior regularization for structured latent variable models. *Journal of Machine Learning Research*, 11:2001–2049, 2010.

[8] A. H. Gandomi, X.-S. Yang, A. H. Alavi, and S. Talatahari. Bat algorithm for constrained optimization tasks. *Neural Computing and Applications*, 22(6):1239–1255, May 2013.

[9] I. Goodfellow, Bengio Y., and Courville A. *Deep Learning*. MIT Press, 2016.

[10] A. Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[11] J. Hadamard. *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*. Mémoires présentés par divers savants à l'Académie des sciences de l'Institut de France: Éxtrait. Imprimerie nationale, 1908.

[12] D. R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1999.

[13] Z. Hu, X. Ma, Z. Liu, E. H. Hovy, and E. P. Xing. Harnessing deep neural networks with logic rules. *CoRR*, abs/1603.06318, 2016.

[14] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[15] D. Karaboga and B. Basturk. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. *Journal of global optimization*, 39(3):459–471, 2007.

[16] T. Karras, T. Aila, S. Laine, and J. Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *CoRR*, abs/1710.10196, 2017.

[17] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[18] S. K. Lam, A. Pitrou, and S. Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[19] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. *arXiv preprint*, 2016.

[20] S. Legg and M. Hutter. Universal intelligence: A definition of machine intelligence. *CoRR*, abs/0712.3329, 2007.

[21] W. Li, M. Gauci, and R. Gross. A coevolutionary approach to learn animal behavior through controlled interaction. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 223–230, New York, NY, USA, 2013. ACM.

[22] P. Liang, M. I. Jordan, and D. Klein. Learning from measurements in exponential families. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, pages 641–648, 2009.

[23] S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's thesis, Univ. Helsinki, 1970.

[24] S. Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[25] P. Merrell, E. Schkufza, Z. Li, M. Agrawala, and V. Koltun. Interactive furniture layout using interior design guidelines. *ACM Trans. Graph.*, 30(4):87:1–87:10, July 2011.

[26] M. Minsky. Steps toward artificial intelligence. In *Computers and Thought*, pages 406–450. McGraw-Hill, 1961.

[27] T. M. Mitchell. *Machine Learning*. McGraw-Hill Education, 1997.

[28] A. Y.-T. Ng and M. I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*, pages 841–848, 2001.

[29] R. Oftadeh, M. J. Mahjoob, and M. Shariatpanahi. A novel meta-heuristic optimization algorithm inspired by group hunting of animals: Hunting search. *Computers & Mathematics with Applications*, 60(7):2087 – 2098, 2010.

[30] J. Schmidhuber. *Neural Computation*, 4(6):863–879, 1992.

[31] J. Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[32] W. Schmidt, L. L. M. Nix, Q. Lūqā, H. Schöne, and J. L. Heiberg. *Heronis Alexandrini Opera qvae svpersvnt omnia ...* Bibliotheca scriptorum Graecorum et Romanorum Teubneriana. in aedibus B. G. Teubneri, 1900.

[33] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius. Procedural content generation via machine learning (PCGML). *CoRR*, abs/1702.00539, 2017.

[34] L. Theis, A. van den Oord, and M. Bethge. A note on the evaluation of generative models. *ArXiv e-prints*, November 2015.

[35] A. M. Turing. Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65. Springer, 2009.

[36] D. H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Comput.*, 8(7):1341–1390, October 1996.

[37] R. A. Yeh, C. Chen, T.-Y. Lim, M. Hasegawa-Johnson, and M. N. Do. Semantic image inpainting with perceptual and contextual losses. *CoRR*, abs/1607.07539, 2016.

[38] J. Zhu, N. Chen, and E. P. Xing. Bayesian inference with posterior regularization and applications to infinite latent svms. *Journal of Machine Learning Research*, 15(1):1799–1847, 2014.

[39] J.-Y. Zhu, P. Krähenbühl, E. Shechtman, and A. A. Efros. Generative visual manipulation on the natural image manifold. *CoRR*, abs/1609.03552, 2016.

Appendix A

Backpropagation derivation

The gradient of the loss l with respect to the weight $w_{i,j}$ is computed by using the chain rule:

$$\frac{\partial l}{\partial w_{i,j}} = \underbrace{\frac{\partial l}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j}}_{\delta_j} \cdot \frac{\partial net_j}{\partial w_{i,j}}. \tag{A.1}$$

Since the activation function φ is differentiable,

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \varphi(net_j)}{\partial net_j} = \varphi\prime(net_j). \tag{A.2}$$

By the definition of net_j,

$$\frac{\partial net_j}{\partial w_{i,j}} = \frac{\partial \sum_{k}^{pred_j} w_{k,j} \cdot o_k}{\partial w_{i,j}} = o_i. \tag{A.3}$$

The gradient of the loss l with respect to the output of h_j depends on whether the unit belongs to the final layer or to a hidden layer of the ANN f:

$$\frac{\partial l}{\partial o_j} = \begin{cases} \frac{\partial l}{\partial f(\boldsymbol{x}_j)} & \text{if unit } h_j \text{ in output layer} \\ \underbrace{\sum_{k}^{succ_j} \frac{\partial l}{\partial o_k} \cdot \frac{\partial o_k}{\partial net_k}}_{\text{total derivative wrt } o_j} \cdot w_{j,k} = \sum_{k}^{succ_j} \delta_k \cdot w_{j,k} & \text{otherwise} \end{cases}. \tag{A.4}$$

Substituting equations A.4, A.2 and A.3 in A.1 we get:

$$\frac{\partial l}{\partial w_{i,j}} = \delta_j \cdot o_i, \tag{A.5}$$

where

$$\delta_j = \begin{cases} \frac{\partial l}{\partial f(\boldsymbol{x}_j)} \cdot \varphi\prime(net_j) & \text{if unit } h_j \text{ in output layer} \\ \sum_{k}^{succ_j} \delta_k \cdot w_{j,k} \cdot \varphi\prime(net_j) & \text{otherwise} \end{cases}.$$

For example, suppose φ is the standard logistic function

$$\varphi(x) = \frac{1}{1 + e^{-x}}$$

with first derivative

$$\frac{d}{dx}\varphi(x) = \varphi(x)(1 - \varphi(x))$$

and the loss function l is the Euclidean distance with first derivative

$$\frac{d}{dx}\varphi(x) = \varphi(x)(1 - \varphi(x)).$$

The ANN f will first compute an output value $\boldsymbol{y} = f(\boldsymbol{x})$ and then update every weight according to A.5

$$\frac{\partial l}{\partial w_{i,j}} = \delta_j \cdot \boldsymbol{y}_i,$$

where

$$\delta_j = \begin{cases} (\boldsymbol{y}_j - \hat{\boldsymbol{y}}_j) \cdot \boldsymbol{y}_j \cdot (1 - \boldsymbol{y}_j) & \text{if unit } h_j \text{ in output layer} \\ (\sum_{k}^{succ_j} \delta_k \cdot w_{j,k}) \cdot \boldsymbol{y}_j \cdot (1 - \boldsymbol{y}_j) & \text{otherwise} \end{cases}$$

and $\hat{\boldsymbol{y}}$ is the truth value associated to $\boldsymbol{x}$ according to the training set.

Appendix B

Gradient derivation for BGANs

The goal of BGANs is to train a generator outputting discrete values by estimating gradients for $\boldsymbol{\theta}^{(g)}$ via the exclusive Kullback-Leibler divergence between the two joint distributions $\widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z})$ and $p_g(\boldsymbol{x}, \boldsymbol{z})$:

$$\nabla_{\boldsymbol{\theta}^{(g)}} D_{KL}(\widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z}) || p_g(\boldsymbol{x}, \boldsymbol{z})). \tag{B.1}$$

$\widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z})$ can be assumed to be constant with respect to $\boldsymbol{\theta}^{(g)}$ if the one from the previous iteration is used. Thus, $\boldsymbol{\theta}^{(g)}$ gradient will not be propagate inside $\widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z})$. Due to 2.10 and 2.13, the following holds:

$$\widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z}) = \widetilde{p}_{data}(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z}) = \frac{1}{Z_{|\boldsymbol{z}}} g(\boldsymbol{x}|\boldsymbol{z}) \frac{D(\boldsymbol{x})}{1 - D(\boldsymbol{x})} p(\boldsymbol{z}), \tag{B.2}$$

where

$$Z_{|\boldsymbol{z}} = \sum_{\boldsymbol{x}} g(\boldsymbol{x}|\boldsymbol{z}) \frac{D(\boldsymbol{x})}{1 - D(\boldsymbol{x})}$$

and $\widetilde{p}_{data}(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})$ is an estimate of the data in the neighborhood of the generated samples defined by $\boldsymbol{z}$.

The gradient can be estimated as follows:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}^{(g)}} D_{KL}(\widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z}) || p_g(\boldsymbol{x}, \boldsymbol{z}))) &= \nabla_{\boldsymbol{\theta}^{(g)}} \sum_{\boldsymbol{z}} \sum_{\boldsymbol{x}} \widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z}) \log \frac{\widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z})}{p_g(\boldsymbol{x}, \boldsymbol{z})} \\
&= \nabla_{\boldsymbol{\theta}^{(g)}} \sum_{\boldsymbol{z}} \sum_{\boldsymbol{x}} \widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z}) (\log \widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z}) - \log p_g(\boldsymbol{x}, \boldsymbol{z})) \\
&= -\sum_{\boldsymbol{z}} \sum_{\boldsymbol{x}} \widetilde{p}_{data}(\boldsymbol{x}, \boldsymbol{z}) \nabla_{\boldsymbol{\theta}^{(g)}} \log p_g(\boldsymbol{x}, \boldsymbol{z}) \\
&\overset{B.2}{=} -\sum_{\boldsymbol{z}} \sum_{\boldsymbol{x}} \frac{1}{Z_{|\boldsymbol{z}}} g(\boldsymbol{x}|\boldsymbol{z}) \frac{D(\boldsymbol{x})}{1 - D(\boldsymbol{x})} p(\boldsymbol{z}) \nabla_{\boldsymbol{\theta}^{(g)}} \log p_g(\boldsymbol{x}, \boldsymbol{z}) \\
&= -\mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z})} \left[\sum_{\boldsymbol{x}} \frac{1}{Z_{|\boldsymbol{z}}} \frac{D(\boldsymbol{x})}{1 - D(\boldsymbol{x})} g(\boldsymbol{x}|\boldsymbol{z}) \nabla_{\boldsymbol{\theta}^{(g)}} \log g(\boldsymbol{x}|\boldsymbol{z}) \right] \\
&\approx -\mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z})} \left[\sum_{m=1}^{M} \frac{1}{M} \frac{1}{Z_{|\boldsymbol{z}}} \frac{D(\boldsymbol{x}^{(m)})}{1 - D(\boldsymbol{x}^{(m)})} \nabla_{\boldsymbol{\theta}^{(g)}} \log g(\boldsymbol{x}^{(m)}|\boldsymbol{z}) \right]
\end{aligned}$$

where we approximated the gradient using M samples $\boldsymbol{x}^{(m)} \sim g(\boldsymbol{x}|\boldsymbol{z})$ for each $\boldsymbol{z}$. Selecting $\boldsymbol{x}^{(m)}$ amounts at sampling the discrete values from the probability $g(\boldsymbol{x}|\boldsymbol{z})$, e.g. a sigmoid function for binary variables.

The samples can also be used to approximate $Z_{|\boldsymbol{z}}$ as:

$$Z_{|\boldsymbol{z}} = \sum_{\boldsymbol{x}} g(\boldsymbol{x}|\boldsymbol{z}) \frac{D(\boldsymbol{x})}{1 - D(\boldsymbol{x})} \approx \sum_{m=1}^{M} \frac{1}{M} \frac{D(\boldsymbol{x}^{(m)})}{1 - D(\boldsymbol{x}^{(m)})}.$$

For binary variables it is possible to approximate $\nabla_{\boldsymbol{\theta}^{(g)}} \log g(\boldsymbol{x}^{(m)}|\boldsymbol{z})$ in the case of a logistic activation function σ as follows:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}^{(g)}} \log g(\boldsymbol{x}^{(m)}|\boldsymbol{z}) &= \nabla_{\boldsymbol{\theta}^{(g)}} \sum_{i=1}^{|\boldsymbol{x}^{(m)}|} \mathbf{1}_{[\boldsymbol{x}_i^{(m)}=1]} \log g_i(\boldsymbol{x}^{(m)}|\boldsymbol{z}) + \mathbf{1}_{[\boldsymbol{x}_i^{(m)}=0]} \log \left(1 - g_i(\boldsymbol{x}^{(m)}|\boldsymbol{z})\right) \\
&= \nabla_{\boldsymbol{\theta}^{(g)}} \sum_{i=1}^{|\boldsymbol{x}^{(m)}|} \mathbf{1}_{[\boldsymbol{x}_i^{(m)}=1]} \log \sigma(h_i(\boldsymbol{x}^{(m)}|\boldsymbol{z})) + \mathbf{1}_{[\boldsymbol{x}_i^{(m)}=0]} \log \left(1 - \sigma(h_i(\boldsymbol{x}^{(m)}|\boldsymbol{z}))\right) \\
&= \nabla_{\boldsymbol{\theta}^{(g)}} \sum_{i=1}^{|\boldsymbol{x}^{(m)}|} \mathbf{1}_{[\boldsymbol{x}_i^{(m)}=1]} \log \frac{1}{1 + e^{-h_i(\boldsymbol{x}^{(m)}|\boldsymbol{z})}} + \mathbf{1}_{[\boldsymbol{x}_i^{(m)}=0]} \log \frac{e^{-h_i(\boldsymbol{x}^{(m)}|\boldsymbol{z})}}{1 + e^{-h_i(\boldsymbol{x}^{(m)}|\boldsymbol{z})}} \\
&= -\nabla_{\boldsymbol{\theta}^{(g)}} \sum_{i=1}^{|\boldsymbol{x}^{(m)}|} \log \left(1 + e^{-h_i(\boldsymbol{x}^{(m)}|\boldsymbol{z})}\right) + \mathbf{1}_{[\boldsymbol{x}_i^{(m)}=0]} h_i(\boldsymbol{x}^{(m)}|\boldsymbol{z}) \\
&\approx -\nabla_{\boldsymbol{\theta}^{(g)}} \sum_{i=1}^{|\boldsymbol{x}^{(m)}|} \max\left(0, -h_i(\boldsymbol{x}^{(m)}|\boldsymbol{z})\right) + \mathbf{1}_{[\boldsymbol{x}_i^{(m)}=0]} h_i(\boldsymbol{x}^{(m)}|\boldsymbol{z}),
\end{aligned}$$

since $\log\left(1 + e^{-\boldsymbol{x}}\right) \approx \max\left(0, -\boldsymbol{x}\right)$.