Alma Mater Studiorum - Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento DISI

Corso di Laurea in INGEGNERIA INFORMATICA MAGISTRALE

TESI DI LAUREA

in

INTELLIGENT SYSTEMS M

Model Agnostic solution of CSPs with Deep Learning

CANDIDATO:
Giovanelli Francesco

RELATORE:
Chiar.ma Prof.ssa Milano Michela

CORRELATORI:
Dott. Lombardi Michele
Dott. Galassi Andrea

Anno Accademico 2018/2019
Sessione I

Abstract

In recent years, deep learning techniques have been widely used for many different real world problems, such as image recognition, natural language processing, game playing, and classification tasks, yielding remarkable results. Deep learning is a sub-symbolic technique, meaning that it is not necessary to explicitly represent the problem's characteristics to be able to learn something from the problem itself, so it could be interesting to use Deep Neural Networks to tackle problems with constraints, also known as Constraint Satisfaction Problems (CSPs), without relying on explicit knowledge about the problem's constraints. This approach is called Model Agnostic, to indicate that the models employed during the experimentation should have the less possible *a priori* knowledge about the problem analyzed.

This approach is particularly useful, since a CSP can be difficult to formulate and describe in all its details: there could exist constraints or preferences which are not explicitly mentioned, that are only implicitly expressed in previous solutions. In such cases, a Deep Learning model that is able to learn the CSP's structure could have meaningful applications on practical problems. Furthermore, from the theoretical standpoint, it is interesting to evaluate if Deep Learning techniques will be able to learn the structure of a CSP.

In particular, in this thesis work we investigate whether a Deep Neural Network can be capable of learning how to solve a classical combinatorial problem, the 8-Queen completion.

We create two different Deep Networks, a Generator and a Discriminator, that are able to learn the characteristics of the problem examined, such as implicit constraints (preferences) and mandatory constraints (hard constraints). The Generator network is trained to produce a single, globally consistent assignment, for any given partial solution in input, to hopefully be able to employ it as an heuristic algorithm to guide a search strategy. The Discriminator network is trained to distinguish between valid or invalid solutions, so that it could become an heuristic algorithm controller, that is capable to determine if a partial solution being constructed from the heuristic can be completed to a full feasible assignment.

Afterwards, we combine the two networks together in a Generative Adversarial Network (GAN), so that they are able to exchange knowledge and information about the problem, hoping to improve both network's performances.

To evaluate these models, different experiments are performed, each one using a distinct family of datasets, and each dataset family having different data characteristics.

Negli ultimi anni, le tecniche di deep learning sono state notevolmente migliorate, permettendo di affrontare con successo diversi problemi, come il riconoscimento delle immagini, l'elaborazione del linguaggio naturale, l'esecuzione di giochi e la classificazione di esempi. Il deep learning ha un approccio sub-simbolico ai problemi, perciò non si rende necessario descrivere esplicitamente informazioni sulla struttura del problema per fare sì che questo possa essere esaminato con successo; dato ciò, l'idea è di utilizzare reti neurali di Deep Learning per affrontare problemi con vincoli, conosciuti anche come Constraint Satisfaction Problems (CSPs), senza dover fare affidamento su conoscenza esplicita riguardo ai vincoli dei problemi. Chiamiamo questo approccio Model Agnostic, per indicare che le reti utilizzate negli esperimenti hanno la minor quantità possible di conoscenza a priori sul problema da analizzare.

Questo approccio può rivelarsi molto utile se usato sui CSP, dal momento che è difficile esprimere tutti i dettagli a riguardo: potrebbero esserci vincoli, o preferenze, che non sono menzionati esplicitamente, e che sono intuibili solamente dall'analisi di soluzioni precedenti del problema. In questi casi, un modello di Deep Learning in grado di apprendere la struttura del CSP potrebbe avere applicazioni pratiche rilevanti. In aggiunta a ciò, è interessante verificare dal punto di vista teorico se, tramite Deep Learning, sia possible far apprendere a un modello la struttura di un problema con vincoli.

In particolar modo, in questa tesi si è indagato sul fatto che una Deep Neural Network possa essere capace di risolvere uno tra i più classici dei problemi combinatori, ossia il rompicapo delle 8 regine. Sono state create due diverse reti neurali, una rete Generatore e una rete Discriminatore, che hanno dovuto apprendere differenti caratteristiche del problema, come vincoli impliciti (detti anche preferenze) e vincoli espliciti (i vincoli veri e propri del problema). La rete Generatore è stata addestrata per produrre un singolo assegnamento, in modo che questo sia globalmente consistente, per ognuna delle soluzioni fornitegli come input; l'ipotesi è che questa rete possa essere poi usata come un'euristica, capace di guidare un algoritmo di ricerca nello spazio degli stati. La rete Discriminatore è invece stata addestrata a distinguere tra soluzioni ammissibili e non ammissibili, con l'idea che possa essere utilizzata come controllore dell'euristica; in questo modo, essa ci può indicare se una soluzione parziale, costruita tramite funzione euristica, possa essere estesa ad un assegnamento completo o meno.

Infine, sono state combinate le due reti in un unico modello, chiamato Generative Adversarial Network (GAN), in modo che esse possano scambiarsi conoscenza riguardo al problema, con l'obiettivo di migliorare le prestazioni di entrambe.

Sono stati eseguiti diversi esperimenti che hanno permesso di valutare queste reti; ogni esperimento ha sfruttato una diversa famiglia di dati, e ogni famiglia di dati aveva caratteristiche diverse dalle altre.

Contents

Chapter 1

Introduction

Machine Learning techniques have been studied and applied on real world problems for many years, allowing machines to learn from a certain set of examples. Their goal is to automatically find patterns and models from the analyzed data, in order to gain an insight on how data is structured; Machine Learning methods can successfully address the need to extract knowledge from an increasingly large amount of information.

In recent years, a particular machine learning technique, called artificial neural network, became more popular. Its design is loosely inspired by the human brain, and it is modeled as a combination of multiple, simple, entities (called neurons).

While neural networks are useful, their effectiveness increased significantly when they evolved into larger and more complex networks, called Deep Neural Networks, thus creating a new approach called Deep Learning.

Deep learning techniques have been widely used for many different real world problems, such as image recognition, natural language processing, game playing, and classification tasks, yielding remarkable results. Deep learning allows us to extract complex and abstract features from data, by introducing representations that are expressed in terms of other, simpler representations [1]. Deep learning is a sub-symbolic technique, meaning that it is not necessary to explicitly represent the problem's structure, or to manually insert complex features, to be able to learn something from the problem itself.

Knowing this, we want to assess if a Deep Network can be used effectively to tackle decision problems with constraints, also known as Constraint Satisfaction Problems (CSPs). In particular, the purpose of this thesis is to investigate whether a Deep Neural Network (DNN) can be capable of learning how to solve a combinatorial problem, with no explicit information about the problem itself, such as the problem's constraints.

This approach is called Model Agnostic, to indicate that the model should have the least possible *a priori* knowledge about the problem analyzed.

A Model Agnostic approach is particularly useful for our analysis, since a CSP can be difficult to formulate and describe in all its details: there could exist constraints or preferences which are not explicitly mentioned, that are only implicitly expressed in previous solutions. In such cases, a Deep Learning model that is able to learn the CSP's structure without relying on any additional knowledge other than (partial) solutions, could have meaningful applications on practical problems, such as scheduling or timetabling ones.

Knowing this, the goal is to to design and test two different DNNs, a Generator network (G), and a Discriminator network (D), that should be able to gain knowledge about one of the classic CSPs, the 8-Queen completion problem. The reason of this choice is that this line of research is at an early stage, so instead of solving more complex, real-world problems, it is better to work with one that has well known properties, thus creating a controlled and more manageable setting to perform the experiments.

For this thesis work, the initial goal is to create a G able to learn preferences (soft constraints) with respect to different problem states. The G is trained to produce a single, globally consistent assignment, with the idea of employing G as a sort of heuristic algorithm, thus helping to guide a search strategy to traverse the search space.
On the other hand, the second objective is to create a D able to understand and learn the problem's hard constraints. The D knows nothing about the preferences, but after the training process it should be able to distinguish between valid or invalid (possibly partial) solutions. In this way, the D could become an heuristic algorithm controller, that is capable to determine whether a partial solution, being constructed by the heuristic, can be extended to a full feasible assignment of all decision variables.

After performing the two separate training processes of G and D, the idea is to combine them together in a Generative Adversarial Network (GAN). The goal of this experiment is to see if G and D can exchange knowledge and information between each other, so that G may become better in generating feasible solutions, while D can improve in detecting when a preference is respected. To do this, we employ pre-trained and untrained versions of both G and D, and we also test an improved GAN model, called Wasserstein GAN with gradient penalty. The evaluation of G, D, and GAN on the 8-queens completion problem is performed in three distinct experiments, one for each dataset family.

The documents is structured as follows: Chapter 2, defines what is a CSP, its main characteristics an applications in real-world cases, and also describes two classical CSPs such as the Partial Latin Square problem, and the N-Queens completion problem (the one used for this thesis work). Chapter 3 examines different machine learning techniques, such as Support Vector Machines and Deep Neural Networks. Chapter 4 illustrates what is a Generative model, and why they are different from Discriminative ones; furthermore, it describes two different generative models, such as Generative Adversarial Networks and their Wasserstein counterpart. Chapter 5 reports different studies on similar subjects, and describes the main idea behind this thesis work. It also illustrates the characteristics of the different dataset families, employed during the experiments. Chapter 6, 7 and 8 are the ones detailing the experiments. Each one of this chapters corresponds to a certain set of experiments, performed using a different dataset family. Finally, chapter 9 contains the final observation for the different experiments, along with suggestions for further developments and improvements.

Chapter 2

Constraints Satisfaction Problem

Constraint Satisfaction (or Solving) Problems (CSPs) are problems based on constraints. These constraints are basically *rules*, that need to be respected in order to correctly solve the problem at hand; for example, consider the restrictions in timetabling problems or transportation scheduling problems.
A CSP is solved when each variable of the problem has a value assigned to it, that satisfies all the restrictions (constraints) on the variable itself. So, to properly solve the problem it is necessary to fully comprehend its restrictions, and to correctly choose the values for the variables.

Formally a Constraints Satisfaction Problem can be defined on three components [2]:

- A finite set of variables **X** $(X1, X2, ..., Xn)$
- A finite set of domains **D** $(D1, D2, ..., Dn)$
- A finite set of constraints **C** $(C1, C2, ..., Cn)$

These elements create a tuple $< X, D, C >$, so that each value V, that is part of a domain D, can be assigned to a variable X; to ensure feasibility, each assignment of values to variables must be valid with respect to the constraints C.
Each constraint C between k variables is a subset of the Cartesian product of the variable's domains D_k, and specifies which values of the k variables are compatible with one another [3]. This subset is characterized by a set of relations.
When all the variables have assigned values, and every assignment is compatible with the constraints, a final solution is obtained. Otherwise, if at least one variable has not a value assigned, the generated set of assignments is called partial solution [2].

2.1 Constraint Programming - structure and applications

The constraints in CSPs are basically restrictions, thus it is possible to identify many real life problems that require decisions based on certain rules and conditions to be properly solved.
Constraint Programming (CP) is a very useful and popular technique, employed to solve CSPs and COPs (Constraint Optimization Problems) [3]. This technique is very helpful for many real world problems, due to the fact that CP guarantees fast prototyping, it is easy to maintain and to apply modifications and it is extensible and flexible, so adding new constraints or customize the search process is not a time-consuming task [3].

CP is generally composed of two interconnected parts, a modelling one and a solving one: in the first one it is necessary to choose which variable to use and which constraints to enforce, while in the second one it is required to choose a search algorithm, to integrate local consistency and propagation, and to choose an heuristic for the branching process (such as which variable to branch on) [4].
CP can be applied to multiple use cases, such as to reduce delay and allow more trains to operate in a railway system, to schedule events in an operating system [4], to improve elderly care by optimizing the shift and visit schedules, to implement adaptive planning for agricultural and urban areas, thus improving tourism and revenues and reducing the environmental impact at the same time [3].

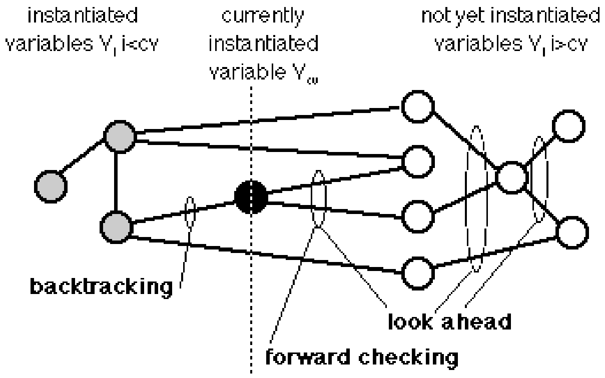
2.2 State space search algorithm for CSP

In CSPs a state is defined by variables X with assigned values taken from domains D, while the operators are labeling operations (assignments of values to variables). CSP problems can be solved using different state space search algorithms, such as Depth-first Search or Breadth-first Search [2].
A basic algorithm has an initial state as the empty assignment. At each step a *successor* function assigns a value to an uninitialised variable, so that the new assignment is compatible on the basis of previous assignments. At each step the goal test is performed: if all the variables are bounded, meaning that a value taken from the domain has been assigned to each variable, then the assignment process will be complete.
This basic algorithm is the same for all CSPs, and has depth limited to the number of variables n. A depth-first search can be used to explore the state space.

Since CSPs are NP-hard problems [4], each solution technique for CSP uses a decision tree, directly or indirectly [5]. A CSP decision tree has multiple levels, where each level is the assignment of a different value to a variable, and every node represents a choice on which one of the possible values will be assigned to the variable corresponding to the node's level. Each leaf can be seen as an assignment on all the possible variables, and it represents a final solution if the assignment is compatible with the constraints, otherwise it is marked as failure.
Given this tree representation, searching for a solution requires finding a leaf corresponding to a goal state. If n variables are involved, and the domains have the same cardinality d, the number of leafs in a decision tree, corresponding to the number of possible assignments, will be d^n [2].

2.3 Propagation algorithms

Propagation algorithms use the constraints to reduce the number of legal values for a variable, then for another variable, and so on, with the goal of eliminating inconsistent values throughout the graph [2]. The main idea is to avoid (or at least reduce) the need for recovery after reaching a failure state, such as backtracking in depth-first search, an inefficient process that it only performs checks with the current and past variables.
Using propagation, filtering trough a constraint leads to filtering through other constraints, thus pruning the decision tree *a priori* during training, and reducing the state space. In this way, tree branches that will lead to failure are pruned, and the state space search is more efficient.
Algorithms that use propagation techniques are Forward Checking and Look-Ahead [5].


instantiated variables V_i i<cv
currently instantiated variable V_{cv}
not yet instantiated variables V_i i>cv
backtracking
look ahead
forward checking

(a) Forward Checking and Look Ahead propagation techniques [6]

2.4 Consistency

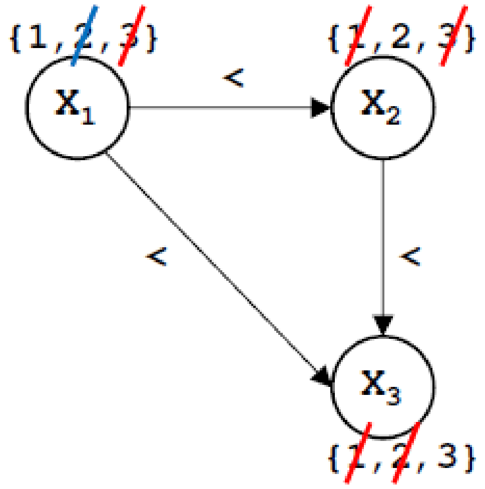
In order to enforce consistency, local consistency techniques can be applied [2]. These techniques do not always apply constraint propagation after the labeling process, instead they are often used to reduce the domains by eliminating values not compatible to a final solution. Consistency algorithms describe a CSP as a constraint graph composed by nodes, representing variables, and arcs, representing constraints. Each constraint arc can have or not have a direction. Binary constraints link two different nodes together, while unary constraints are represented by arcs re-entering the same node.
There are different types of algorithms that enforce consistency, such as node, arc, and path consistency, each one implementing a different level of consistency checking:

- Node consistency: it is the simplest form of consistency. A node is consistent if the unary constraint on the variable X_i is compatible with each value of the variable's domain

- Arc consistency: it is defined for binary constraints. An arc is consistent if, given two nodes-variables connected X_i and X_j, for each value of the domain of X_i, there is at least a value of the domain of X_j that is compatible with the constrain C_t between the two variables. If this is true, each value of the domains will have a support with respect to the constraint C. For each arc of the graph that is not consistent, it is possible to prune all and only the values that lack a support until the graph reaches a stable configuration. The algorithm for enforcing arc consistency is called AC-3

- Path consistency: a pair of variables X_i and X_j is path consistent with a third variable X_k if there is at least one value in the third variable's domain that is compatible with the constraints between X_k and X_i and between X_k and X_j

- Bound consistency: it is more efficient than Arc consistency, but it may prune less. The constraint check is performed on the upper and lower bound of the domains of the variables.

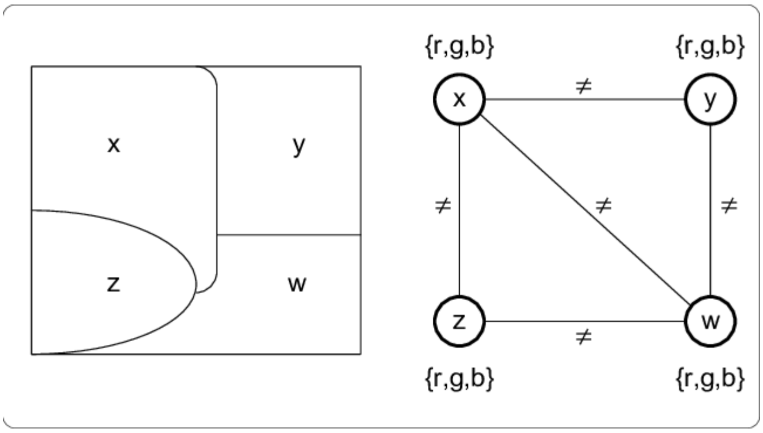


(b) Example of Arc Consistency [5]

There is another category of constraints, called *global* constraints. A global constraint is a constraint that captures a relationship between an arbitrary number of variables [2], and encapsulates a specialized propagation algorithm. This allows the expression of constraints that are otherwise not possible to state using primitive constraints[3].
Some examples of Global Constraints include *Alldifferent*, that enforces each variable in the constraint to have a different value from other variables, and *GCC*, that counts the occurrences of specific values and restricts the maximum and minimum cardinality[3].

2.5 Classical CSPs

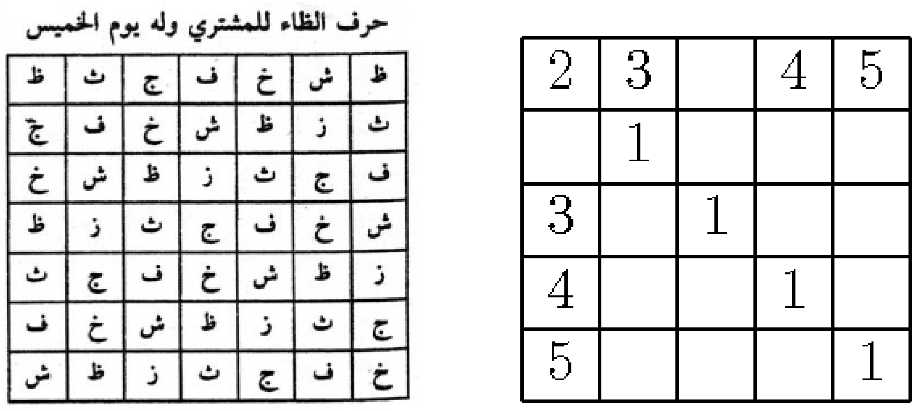
In order to design and evaluate an appropriate deep neural network for this thesis work, it is useful to study and employ one of the classic games with constraint. Classical problems are useful because let us work in a controlled setting with well known properties, and simplifies drawing scientific conclusions.
There are many different classical problems with constraints, usually these are in the form of games or logic puzzles, such as the Map Coloring Problem, Nine Men Morris game[7] and Sudoku.
In the following sections, we will examine two different games with constraints, the Partial Latin Square Problem, and the N-Queens Completion problem.



(c) An example of the Map Coloring problem [8]

2.5.1 Partial Latin Square Problem

Partial Latin Square (PLS) is a common CSP problem, of the NP-Complete family [9].
A Latin Square (LS) of order n is an $n \times n$ array, and each entry is an element from the set $\{1, ..., n\}$. Each row contains each element exactly once, and each column contains each element exactly once. A PLS is a Latin Square where each entry is either empty or contains an element from the set $\{1, ..., n\}$.
The first known occurrences of latin squares seem to be their use on amulets and in rites in certain Arab and Indian communities. Latin squares have applications in statistics, and were introduced to the mathematical community by Leonhard Euler, who has been the first to investigate their properties mathematically since at least 1779. The name *Latin Square* comes as a reference to the earlier usage of Latin and Greek letters instead of numbers [10].
LS work as a base for the sudoku puzzles, because a completed sudoku is a latin square of order 9 with the additional condition that each 9 natural 3 × 3 sub-square contains all 9 symbols. The task for an incomplete sudoku is to complete the square from a given set of entries.
PLS are very similar to an incomplete sudoku, indeed the main goal to solve a PLS problem is to complete it to a LS without adding rows, columns or elements, and the solution must be compatible with the row and column constraints. Deciding whether a PLS can be completed is an NP-Complete problem [9].



حرف الظاء للمشتري وله يوم الخميس

ظ	ث	ج	ف	خ	ش	ظ
ج	ف	خ	ش	ظ	ز	ث
خ	ش	ظ	ز	ث	ج	ف
ظ	ز	ث	ج	ف	خ	ش
ث	ج	ف	خ	ش	ظ	ز
ف	خ	ش	ظ	ز	ث	ج
ش	ظ	ز	ث	ج	ف	خ

(d) Historical Latin Square [10]

2	3		4	5
	1			
3		1		
4			1	
5				1

(e) Example of a Partial Latin Square [11]

Figure 2.1: Different Latin Squares

2.5.2 N-Queens and N-Queens Completion Problem

The n-Queens problem consists in placing n chess queens on an $n \times n$ chessboard so that no two queens are on the same row, column or diagonal. In this way, based on the basic rules of chess, no queen is able to attack each other queen.
This problem has been used as an example and benchmark problem in many classic AI paper: in *Backtrack Programming Techniques* by Bitner et al. [12], the problem is used as a reference for the use of macros as a technique to gain a substantial reduction in computation time when searching for a solution; in *The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems* by Mackworth et. al [13], the N-queens problem is used as an example to analyze the time complexity of several node, arc and path consistency algorithms; in *Symmetry-Breaking Predicates for Search Problems* by Crawford et al. [14], the N-queens problem is employed to study how symmetries can be used to add additional constraints to the search problem.

Furthermore, the N-Queens problem has been employed to develop several different applications, such as parallel storage schemes, traffic control and deadlock prevention [15]. The n-Queens Completion problem is a variant of the standard n-Queens problem, in which some queens are already placed and the solver is asked to place the rest. Each queen can be represented as a single variable X_i, where i represents the row (or column), with a total of n variables, and each variable can have values from 1 to n, where each value represents the column (or row) where the queen is placed [5]. An alternative representation of the problem could be viewing each queen as an ordered pair of integers, (α, β), where α represents the queen's column, and β its row on the chessboard, and the values $\alpha + \beta$ and $\alpha - \beta$ represent the two diagonals the queen is on [16].
For the n-Queens problem, the decision problem is solvable in constant time since there is a solution for all $n > 3$, while the best approach to solve the counting version of the problem (how many solutions to n-Queens there are) is currently an optimised exhaustive search [16].
Because of the ease of finding a solution, the n-Queens problem has been subject to controversy in AI over whether it should be used as a benchmark at all, but recent studies show that, as an NP-Complete and #P-Complete problem, its variant n-Queens Completion does provide a valid benchmark problem [16].

8-Queens Problem

The N-Queens problem is a generalization of the 8-Queens problem: this problem consists in placing exactly 8 queens in a 8×8 chessboard, in a way that no queen threatens another queen. Gauss is often cited as the originator of this problem or the first to solve it, but recent findings shown that he was neither the first to come up the problem, nor the first to solve it, having found only 72 of the 92 total solutions [15]. Bezzel seems to have come up with the problem before Gauss, while Nauck was the first to solve the problem, in 1850, by finding all 92 existing solutions. Later, the 8-queens completion problem was proved complete by Pauls [15].

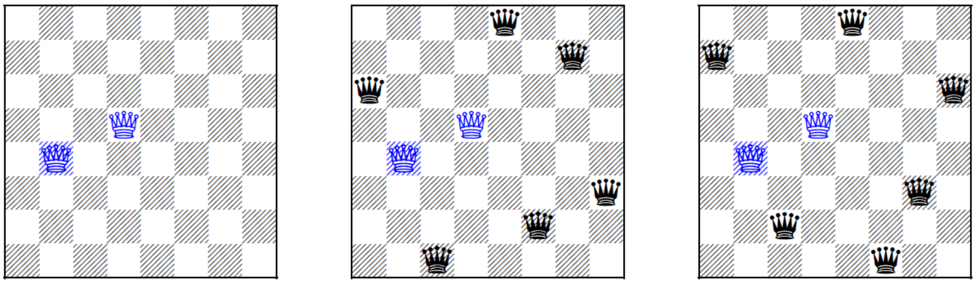


Figure 2.2: Two possible solutions for a given 8-queen completion problem [16]

Given the nature of this problem, and the availability of all the existing solutions, we decided to use the 8-queens problem as the constraint programming problem for this thesis work. This will allow us to use a dataset composed of discrete data, in particular binary data, so that the data model can be agnostic to the problem. Furthermore, we will use this problem as a reference, and apply Deep Learning techniques and Generative Adversarial Networks on it, so that the generated models can gain knowledge about the problem itself, thus being able to automatically generate feasible solutions.

Chapter 3

Machine learning techniques

Machine learning techniques have been developed to allow machines to learn pattern and models relative to a certain set of examples. The idea is that machines can be able to automatically gain knowledge on the data provided during the training phase, with the goal of generalizing and be able to perform well even on unseen examples sampled from the same application domain.

The insight that a machine gains during training is the key this learning process, allowing to tackle complex, real-world problems, that are otherwise too complex to be addressed manually. Simpler machine learning algorithms, such as logistic regression, naive Bayes, and Support Vector Machines (SVM), depend heavily on the representation of the data they are given [1]. This means that these techniques works well when it is possible to manually create appropriate features, but this is not always a viable path, especially for complex scenarios.

On the other hand, more recent and complex techniques, such as Neural Networks, do not rely on carefully crafted features, meaning that it is possible to learn complex characteristics of the dataset by combining the more simpler ones. This is why Deep Neural Networks are used more and more to successfully tackle real-world problems, and also why many different variations of neural networks are being designed and studied in recent years.

3.1 Support Vector Machines

Support Vector Machines (SVMs) is a class of learning methods that have a sound theoretical foundation, requires a small set of examples, and it is quite insensitive to the number of dimensions of the features [17].

A SVM is usually employed in classification tasks, and it tries to find the best classification function that is able to distinguish between the classes in the training set.

One of the main characteristics of SVMs is that they rely on an hyperplane to segregate the examples. This technique tries to find the best possible hyperplane, thus the best classification function, that maximizes the margin between the classes. Geometrically, the margin corresponds to the shortest distance between the closest data points of the other class to a certain point on the hyperplane [17].

Given the fact that it is not always possible to find an hyperplane that perfectly separates the instances of the classes, SVMs adopt a *soft* margin; this means that examples which cross the margin are considered training data errors. The goal is to allow the small amount of noisy data-points as possible, leading to an hyperplane that almost separates perfectly the classes.

SVMs can be extended to work even when data is not linearly separable, by using kernel functions that map data into a new space with higher dimensions, called *feature space*.

SVMs can be also employed in multi-classification tasks: this can be done by repeatedly using one of the classes as the positive class, and the rest as the negative classes (one-vs-all method)

[17].

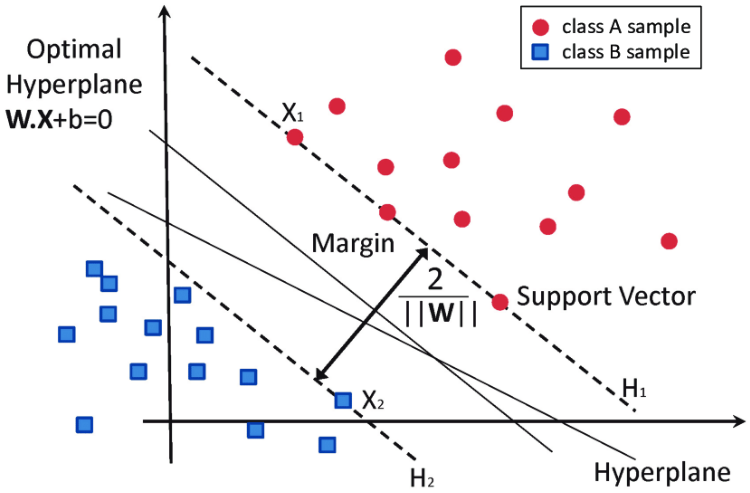


Figure 3.1: Classification of data by support vector machine (SVM) [18]

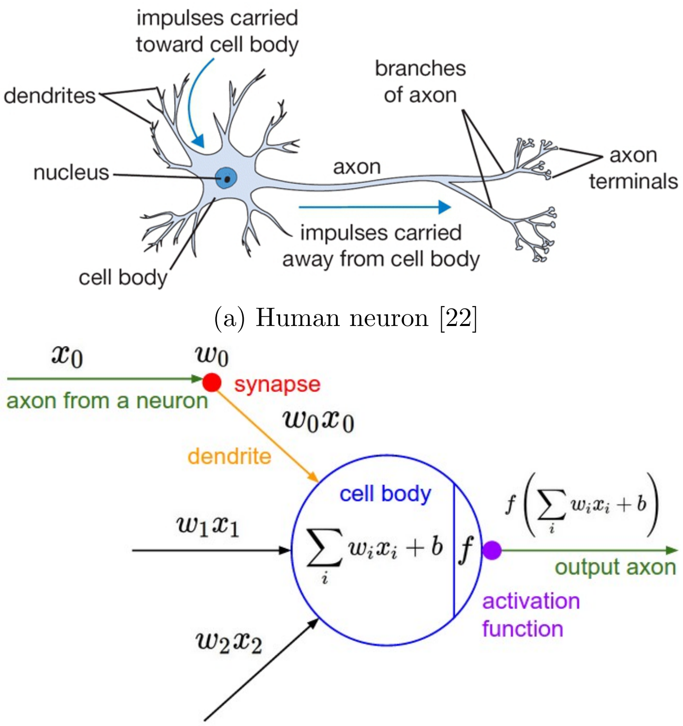
3.2 Neural Networks

Artificial Neural Networks, or more simply Neural Networks (NN), have been receiving more and more attention in recent years, both in the academic and in the business world, and they are used for many different tasks and real-life applications. The result is that the development of different NN architectures has increased year after year.
NNs are a sub-symbolic approach to the problem of learning [7], and they are able to gain knowledge and learn important properties about different problems by storing numbers between the connections of the base units. These base units are also called *neurons*, and the stored numbers represent the weights of the connections between each neuron.
During its training, the network is able to learn from the data by constantly modifying the strength of the connection between the units. Basically, the weights are used to encode the knowledge given by the training examples [19].
By gathering knowledge from experience, this approach does not rely on human operators to formally specify all the knowledge that the computer needs [1]. Once a NN is trained, it can be employed to perform evaluations and predictions. One of the main advantages of NN is that, while the training process can require a lot of time, during the evaluation process the trained network does not need to change its weights, thus resulting in a fast elaboration of data, making it useful also in real-time scenarios.
In the following sections, we will examine the Perceptron, a base unit for the NNs, and we will explore more details about the NN's architecture and different NNs variations.

3.2.1 Perceptron

The origin of NN can be set around the mid 1940s, thanks to the studies on computational models of biological neural networks, by McCulloch et al. [20].
Later on, in 1957, the main element of a NN was designed by Frank Roseblatt, the Perceptron The main idea behind the Perceptron, is to create a simplified mathematical model of how the neurons in our brains operate [21].
The Perceptron, also called *artificial neuron*, is a simple computational unit that receives a set of inputs x_i, coming from the output on nearby neurons. These input values are then employed in a weighted sum, using also the weights w_i to emulate the synapse strength to each nearby

neuron. Finally, a bias b is added to the weighted sum, and the outcome of this operation is passed through an activation function f, to produce an output y. The activation function is also called *threshold* function, since it transforms the incoming value into a 0 or a 1, according to the initial value.



(a) Human neuron [22]

(b) Artificial Neuron (Perceptron) [22]

Figure 3.2: Human and artificial neurons compared

In practice, a Perceptron is also called Linear Perceptron, since it is a linear combination of weighted inputs [21]. The main characteristic of the Perceptron is its ability to learn and gain knowledge about the data that is fed to it in input, called *training set*. This is possible due to the fact that, if an instance is not correctly classified, the value of the weights will be changed accordingly, so that the instance can be properly classified. This weight calibration process is performed for every entry in the training set, and, if the problem is linearly separable, the algorithm is able to converge to a final solution (thus a final configuration of the weights) [19].

3.2.2 From Perceptrons to Neural Networks

Using the learning ability of the Perceptron as reference, NN were designed as a natural evolution from it. The binary output of the Perceptron makes impossible to perform classification tasks on data having more than two classes; to overcome this, many Perceptrons can be arranged together in an hierarchical structure, so that limit of linear decision boundary is no longer existing [19]. This enables NNs to be able to learn and properly work also on non linear problems, the main limitation of Linear Perceptrons.
Indeed, there was another characteristic of the Perceptron that was hindering its performance: the activation function was not-differentiable, since it was designed as a function with binary step; this did not allow the implementation of one of the most important techniques in NNs, called *backpropagation*, that allows information to flow backwards through the neural network.

The NN architecture is made of different *layers*, where each layer is composed by multiple Perceptrons (neurons). Each neuron is basically a signal processor with threshold, thus using

the same idea of Perceptrons, and the signal transmission between neurons is weighted by a set of weights, which change over time during the learning phase.
There are similarities with the brain structure: the brain neurons receive signals and emit signals to other neurons through axons. This is similar to artificial neurons, where we have an activation function that takes the inputs and the weights and computes the output [22].
In NNs, information is represented through real numbers, and the activation function is modeled as a continuous and differentiable mathematical function. We will see multiple activation functions in the following sections.

3.2.3 Feed-forward Neural Networks

Feed Forward NNs are networks where data flow from input to output layers, so each layer computes the output that will be used as an input for the next layer.
In a simple feed-forward NN model, the neurons are grouped in layers, stacked one on top of another: the result is a sequence of an input layer (I), an hidden layer (H) and and output layer(O). As already stated, each layer is connected by a set of weights. Furthermore, in feed-forward models each node of one layer is connected to all the nodes of the following layer [19]. Basically, the hidden layer represents the value of scores for each one of the possible outcomes (with respect to the inputs) that are contained in the set of weights between I and H; the second set of weights, the one between H and O, influences the importance of the H scores to compute the final output score [22].

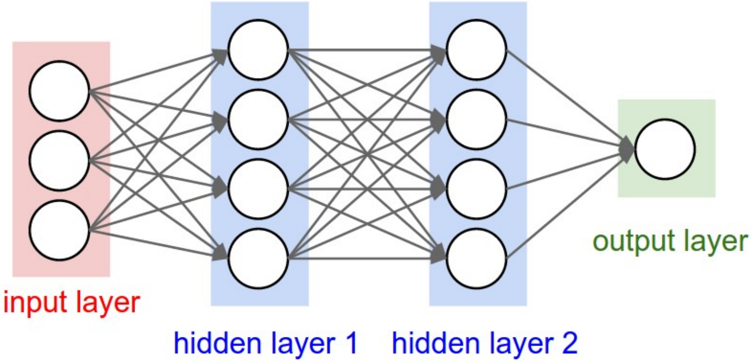


Figure 3.3: An example of feed-forward Neural Network [22]

The NN can be customized through hyper-parameters, which are choices on parameters of the model that are problem-dependent and data-dependent. Usually they are tuned using an independent dataset.
Training a NN means to adapt its hyper-parameters to get the desired outcome, so that during the training process the weights can change accordingly. The goal of training is thus optimizing a loss function, representing the discrepancy, or error, between the outcome of the NN and the desired target; the result is a modification of the weights in the network.

3.2.4 Designing Neural Networks

Hyper-parameters and NN customization

In NN, initializing the weights in an appropriate way can give a significant boost to the overall network's performance [23]. This is because if every neuron has the same weights configuration, the weight updates will be the same and every neuron will learn the same concepts of the others.

So it's important to use weight values that are not too small and not too big, and employ weight initialization techniques.
There are different techniques to initialize the weights, such as random initialization or Xavier initialization [24], and since there isn't a standard procedure that is good for every problem, the choice must be done by looking at each particular problem instance.

Another NN customization point is the number of neurons in hidden layers: while the number of nodes in the output layer is strictly related to the number of desired outcomes, the size of hidden layers can be modified by the network's designer, according to the problem in hand. A larger network could yield better performance, at the cost of a longer training time.

Furthermore, the network can be customize by modifying the regularization term in the loss function: using a value greater than zero, the loss value is increased, since it is composed by data loss and regularization factor.

The learning rate, that influences how strong the weight update is amplified, must be adjusted to the right value, in order to have a sufficiently fast learning process but avoid the *explosion* of the gradients in the network. Usually it has an high value in the first epochs, then decreases gradually in the following epochs, in order to have greater precision in searching the (local) minimum of the loss function. This technique is called *decay* of the learning rate [22]. All the optimization algorithms seen in previous sections, such as SGD and Adam, have the learning rate as an hyper-parameter.

Activation functions

Activation functions are key elements in the design of NNs, mapping inputs incoming in a neuron to a certain output. These functions are usually not linear and differentiable, thus permitting the NN training by using the Backpropagation technique. There are different types of activation functions:

- Sigmoid: non-linear function that maps the input to a range [0,1], the main problem is that it can drastically reduce the gradient if the input is too low or too high

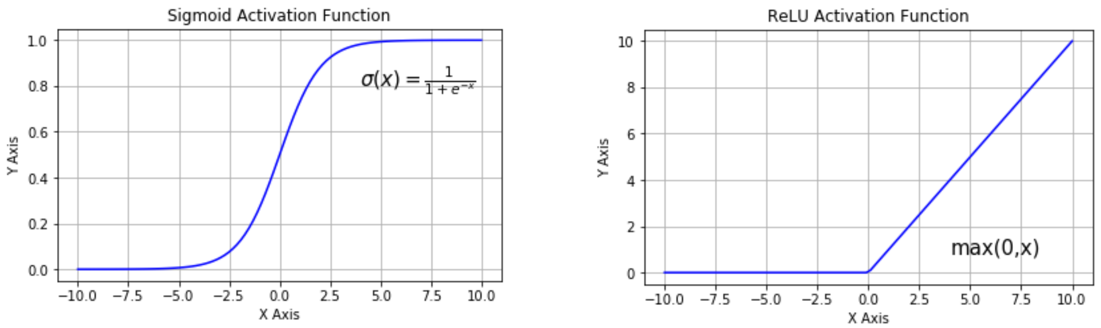
$$f(x) = \frac{1}{1+e^{-x}} \tag{3.1}$$

- Hyperbolic Tangent (Tanh): non-linear function that maps the input to a range [-1,1]

$$f(x) = tanh(x) \tag{3.2}$$

- Rectified Linear Unit (ReLU): if the input is negative it will be mapped to 0, otherwise the output is x. It is very efficient but has the problem of *killing* the gradient in the region where $x < 0$.

$$f(x) = max(0, x) \tag{3.3}$$



(a) Sigmoid activation function [25]

(b) ReLU activation function [25]

Figure 3.4: Sigmoid and ReLU activation functions

- ELU: it has all the benefits of the ReLU but it is more robust to noise and more expensive, because the negative inputs are not mapped to 0 but to an exponential function

$$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases} \tag{3.4}$$

- Softmax: this function works with multiple input data, instead of a single value like the previous functions. The idea is to take a vector of K real numbers in input, and to produce a normalized vector of K probabilities in output. Each value in the output probability distribution is between 0 and 1, and the sum of the K elements is 1. For each element, the Softmax function is defined as:

$$f(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{3.5}$$

The Softmax is usually employed in the final layer of NN classifier that works with multiple class problems. In this instance, the values in output are interpreted as the probabilities that the input belongs to one of each output classes

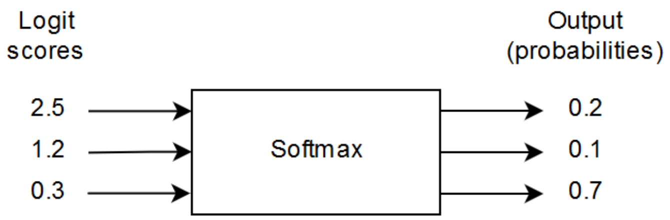


Figure 3.5: An example of Softmax classification with 3 classes

Regularization techniques

The goal of training is to obtain high accuracy not only on the training set but more importantly on the test set, so the accuracy gap between the two, as we already seen, should be small.
In NN we can improve the performance on unseen data using the regularization process, that prevents the network to fit too well the training data, thus preventing overfitting. This can be done in different ways:

- Regularization term: it adds an extra term to the (data) loss in the loss function, in this way the generalization capabilities of the network are increased
- Dropout: in every forward step, and for every layer, a random percentage of neurons is set to zero (in the activation function). This is performed only during the training phase, and helps preventing overfitting, because the NN does not rely too much on the training set data and extracts only the more useful features. When the NN is used after training, all the units are active, thus working as an ensemble of multiple different models working together, and improving performances [26]

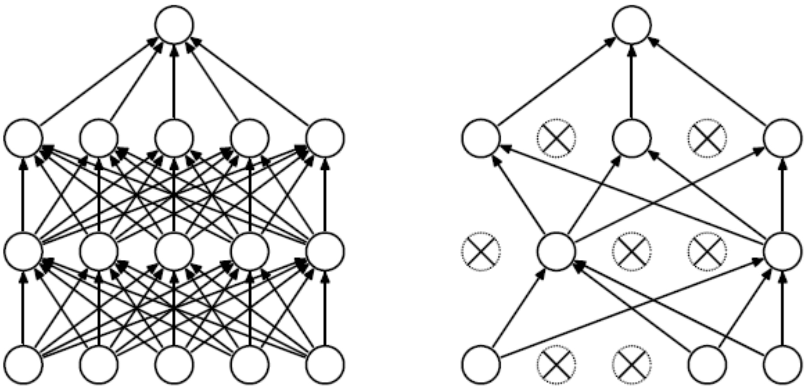


Figure 3.6: NN before (left) and after (right) Dropout [26]

- Batch Normalization: a standardization of the values in output from a layer, computed on mini-batch of data. Batch mean and batch standard deviation are used to obtain normalized values. Batch Normalization forces the outputs in the forward phase to a gaussian unit, normalizing by mean and variance the data in the mini-batch [27]
- Data Augmentation: random transformations on the input data, such as using a mirrored image or small fractions of the original image

With small data we can use transfer learning to improve the performance of the network.

3.2.5 Training Neural Networks

The training algorithm of a NN is made of two distinct phases, the *forward* phase and the *backward* phase. In the first one, the network is fed with the training data, the output of the network is computed, and the error between real outcome and desired outcome is calculated using the loss function. During the second phase, the derivatives and weight corrections are computed for both output and hidden layer, and the weight updates are propagated backwards through the NN using Backpropagation.
We say that an *epoch* of training is passed after all the training data has been fed to the NN, thus indicating the completion of both forward and backward phases. Usually, to properly calibrate the network's weights, several epochs of training are necessary; this is due to the fact that epoch of training corresponds to an alteration of the weights, so the weights of the following epochs will be different.

Backpropagation

To properly train a NN, the updated weights must be propagated through the network. Since the weight update depends on the loss function, thus on the outcome of the NN in the output

layer, it is necessary to transmit the information backwards, from the output layer to the input layer. This technique is called *Backpropagation*.
Every node in a NN has local inputs and an output. In Backpropagation, the incoming gradient (from the output layer) is used in combination with the local neuron gradient, to compute the outgoing gradient. This outgoing gradient is computed for the inputs values, so it needs to flow in backwards direction. To sum up, backpropagation is a process going from the output to the inputs that recalculates the weights between neurons, based on the weight update function that uses learning rate and gradient to recompute the weights [22]. For each neuron, the error on the weights (in output) is used to compute the variation of the different weights in input, with the goal to update every weight in input of the current node.

Loss Function

The loss function evaluates the quality of the weight matrix W, thus helping in finding the best W configuration possible. The loss function compares the output scores of the NN and the expected outcome y. The goal is to find the W configuration that minimizes the loss function. The loss function is the sum of data loss, difference between desired and current NN outcome, and regularization term; this term, usually added to the loss function, tells the model to use a *simpler* W configuration, and penalizes model complexity, so the model tends to go towards a simpler structure and a better generalization ability [22]. One example of regularization term is the L2 norm, used as a common regularization penalty [7], and added to the loss function with a custom weight λ. There are also many regularization techniques, such as Dropout and Batch Normalization, that we will see in the following sections.

An example of loss function, commonly used in many NN models, is the *Cross-Entropy* loss: this function takes the unnormalized probabilities, and applies the exponential function on them, in order to have positive values; the next step is to normalize the values (obtaining probabilities from 0 to 1) and to apply the negative log of the probability of the real class, in order to calculate the final loss value [22]. The Cross-Entropy loss is defined by the following expression:

$$L(y, \hat{y}) = -\sum_{i=1}^{\mathrm{N}} y_i \log(\hat{y}_i) \tag{3.6}$$

Basically, Cross-Entropy loss is used as an indicator of the distance between the distribution output of the model, and the real data distribution.

Optimization

As we stated before, during training it is necessary to optimize the loss function: the goal of optimization is to find the best weights configuration, in order to minimize the loss function value, thus minimizing the error between outcome of the NN and desired target.
There are different algorithms to do this: a first simple idea is to use a random approach, but this would lead to inefficiency and massive time consumption ; a more feasible alternative consists in employing a Slope Following technique [22]. The slope is defined as the derivative of a function given a point. In multiple dimensions it is possible to use the gradient to implement a Slope following technique.
The gradient is as a vector of partial derivatives, in this instance representing partial derivatives of the error as a function of the weights of the NN. The gradient gives an approximation of the function, so it can be recomputed and used to update the W matrix [19].

Given a set of weights w_{ij}, and a learning rate λ, the weights can be updated in the following way:

$$w_{ij} = w_{ij} - \lambda \frac{\partial E(w)}{\partial w_{ij}} \tag{3.7}$$

So, gradient descent is a way to minimize the loss, and the learning rate is the hyper-parameter that changes the convergence *speed*. Basically, the learning rate can be modified to influence training, as a trade-off between speed and precision [19].
There exist different update rules of the weights, based on the gradient descent information:

- Standard Gradient Descent: it needs to calculate the gradients for the whole dataset to perform one update of the weights. Each iteration updates the weights w on the basis of the gradient [28]

  $$W_{t+1} = W_t - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_w Q(z_i, w_t) \tag{3.8}$$

  where Q is a loss function, and γ is an appropriate gain value

- Stochastic Gradient Descent: performs a parameter (weights) update for each training example, and the one example comprising each batch is chosen at random. In SGD, each iteration estimates the gradient on the basis of a single randomly picked example z_t [28]

  $$W_{t+1} = W_t - \gamma_t \nabla_w Q(z_t, w_t) \tag{3.9}$$

  When employing SGD, it is possible that during the optimization process the function reaches a local minima, and if the gradient is zero, the function will be unable to move in the direction of other minimum points (such as the global minima). Another limit of SGD is that the gradient is computed with respect to a batch of data, so noise is introduced, therefore the descent process can be slower [22].

- Mini-batch Stochastic Gradient Descent: instead of working with the entire dataset, it uses a mini-batch (small set) of examples in order to compute the loss and gradient, and re-calibrate the weights. The parameters update process is based on the estimate of loss value and gradient, in order to have better efficiency, especially with large data

- Adam [29]: it uses the *momentum* technique, combined with the raw gradient, in order to step in the direction of the velocity and not the direction of the gradient vector. *Momentum* means that some fractions of the previous update is added to the current update, so that repeated updates build up momentum, thus enabling to move faster in a particular direction. In this way, local minima and saddle points can be overcome. Another improvement of Adam, with respect to other update rules, is that each parameter has its own learning rate

Overfitting

NN are usually trained by employing two different sets of data.
The *training* set is made of data used to actively train the network, so it corresponds to the data that is passed as input to the NN during training. The result is that the NN's weights will be changed accordingly to the data in the training set.
A different, distinct set, called *test* set, is used only to evaluate the network's performance. Every instance in the test set must be new to the NN, in order to assess its ability to generalize.

After the training process is complete, if there is a big gap between accuracy on test set and on training set this will be a symptom of *overfitting*. This means that the NN has learnt very well how to handle data that it has seen during training, but it is much less cable of performing well when unseen data is fed as input to it. In this scenario, the main purpose of NNs, which is to generalize from a certain data distribution, is lacking.
To avoid or lessen this behaviour, it is possible to employ one or more regularization techniques, as explained in the following section. Furthermore, an early stopping mechanism can be used to prevent the model to overfit, halting the training when a certain measure, such as the validation set accuracy, does not improve for a certain number of epochs.

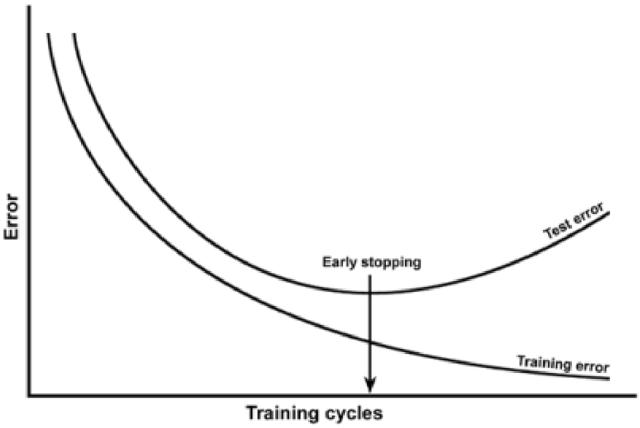


Figure 3.7: Early stopping mechanism to prevent overfitting [30]

3.3 Deep Neural Networks

We saw NN composed by an input, an hidden, and an output layer. This is the simplest form of neural network, but it has also limitations in terms of learning and generalization capabilities. A more recent approach is to organize NN into multi-layered models, where the number of hidden layer is much higher. This hierarchy of layers enables the computer to learn complicated concepts by building them out of simpler ones [1]. The series of multiple hidden layers allows the network to learn gradually more abstract feature of the training data, resulting in an overall improvement in the generalization ability of the model.
Deep learning has been applied to a large set of applications, and many different deep networks have been developed and studied in recent years. One of the most relevant deep networks is the Residual Neural Network.

3.3.1 Residual Networks

Residual Neural Networks (ResNets) [31] are a class of Neural Networks that help to drastically reduce the training difficulty of the standard Deep Neural Networks (DNNs), while enabling

a larger number of layers in the model architecture. The goal is to provide a clear path for gradients to back-propagate to early layers of the network.
The ResNet structure consists in a reformulation of the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. This results in networks that are easier to optimize, and can gain accuracy from considerably increased depth, while having lower complexity (in terms of FLOPs, floating point operations per second, a measure of computer performance) with respect to the typical DNN [31].
ResNets success can be explained investigating a crucial characteristic of a typical DNN, the depth (number of layers) of a network. Indeed, DNN's performance is hindered by the problem of vanishing or exploding gradients [32], partially solved by normalized initialization and intermediate normalization layers, and, when the DNN's depth starts increasing, by a saturation of the accuracy that leads to a fast degradation of the accuracy itself. This degradation is not caused by overfitting, so adding more layers does not lower the training error [31].
A ResNet can solve the accuracy degradation problem. In a ResNet, that is a Deep Network by definition, residual learning is applied to every few stacked layers (basic block). The layers fit a residual mapping, defined by the following function:

$$F(x) = H(x) - x$$

In this instance, x is the input of previous stacked layers, instead of *H(x)*. The original mapping is recast into

$$H(X) = F(x) + x$$

by performing a *shortcut connection* (or *skip connection*), links which skip one or more layers and simply perform identity mapping (when input and output size are equal), and their outputs (x) are added to the outputs of the stacked layers (*F(x)*).

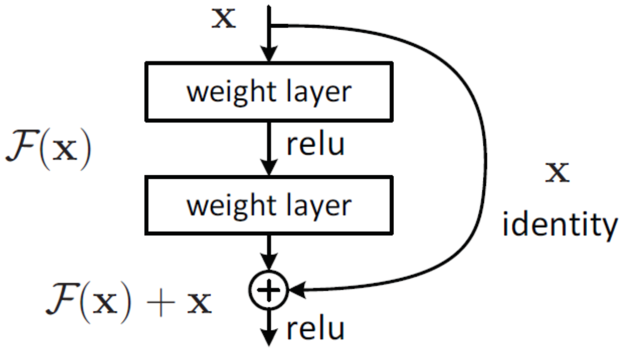


Figure 3.8: Residual Network building block [31]

This results in an easier optimization process, while retaining basically the same computational complexity. The ResNet can be trained with any optimizer, the most popular ones being Stochastic Gradient Descent and Adam, along with Backpropagation, and the training process is faster and more accurate than the typical DNN training process, because the *shortcut connections* are always alive and the gradients can easily back-propagate through them. Giving that, ResNets are not only easy to optimize, but also enjoy accuracy gains from greatly increased depth.
Despite their higher accuracy ResNets are more prone to overfitting [31], therefore different methods to reduce it, such as Dropout, should be applied.

Following the success of ResNets, its development yielded a second version of the model, usually called ResNetv2. This ResNet goal is to create a *direct* path for propagating information, not only within a single Residual Unit, but through the entire network [33]. This is done by using identity mappings: if both *H(x)* and *F(x)* are identity mappings, the signal will be directly propagated from one unit to any other units, in both forward and backward passes [33].

This results in different Residual Units, where there is a sequence of Batch Normalization, Activation function and Dense Layer, and no Activation function is placed after the sum between the input and the output of each Residual Unit. In this way, training becomes easier and the overall performance is improved.

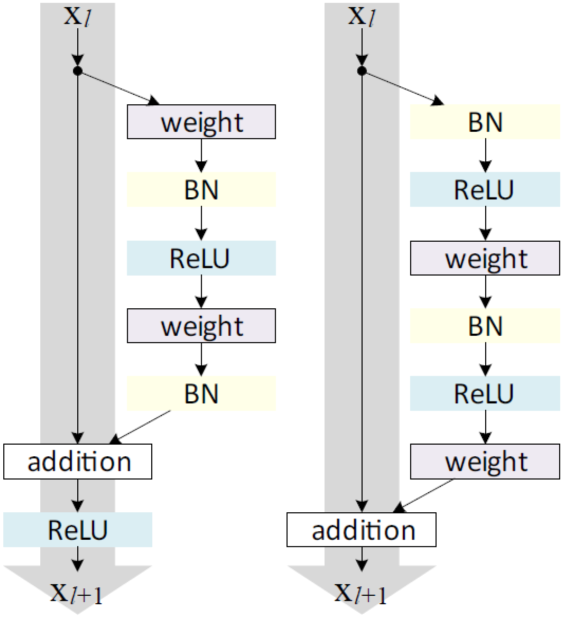


Figure 3.9: Old ResNet unit (left), alongside the improved one (right) [33]

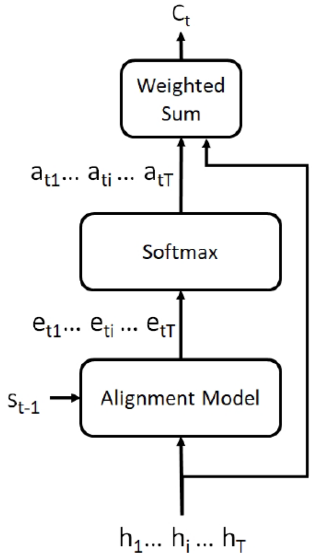
3.3.2 Attention Models

Attention mechanism can be used to increase the performance of a Neural Network (NN), and can also be used to better understand the behaviour of NN that is usually non easy to assess. Due to the sub-symbolic nature of NNs, knowledge is stored in numeric values that do not offer an easy interpretation of the internal evaluations for producing certain output of the NN. By using weights computed by attention and visualizing them, it is possible to gain more knowledge on the internal network behaviour and its relation with output data.

The attention mechanism can be seen as a part of a neural architecture that enables to dynamically highlight important features of the input data according to the given task, and it is able to work with raw input or with a more abstract representation of it [34]. The goal of attention is to compute a weight distribution based on input data, and highlight the most relevant elements by giving their weights an higher value [34].

In most of the practical applications an attention model is trained alongside the NN architecture. This is done using unsupervised training for the model itself, but if knowledge of the desired weight distribution is available, then it will be possible to train the model in a supervised way.

An Attention Model usually produces a context vector, a compact representation of the characteristics of the pivotal input elements, that is useful to lessen the computational load of the NN and yield a performance gain.



c_t
Weighted Sum
$a_{t1} \dots a_{ti} \dots a_{tT}$
Softmax
$e_{t1} \dots e_{ti} \dots e_{tT}$
s_{t-1}
Alignment Model
$h_1 \dots h_i \dots h_T$

Figure 3.10: Attention model of RNNSearch [35] network [34]

The architecture of an attention model depends on the structure of the data and the desired output, but an unified attention model can be designed. This model is composed of a core, that maps a sequence K of keys, representing the encoded data features on which attention is computed, to a distribution a of weights called *attention weights* [34].
Often a second input element, called *query*, is used to give emphasis to the input, based on its relevance with respect to the query. If the query term is missing for some reason, attention can be computed using only the input sequence K, thus creating a *self-attention* model.
The compatibility function f uses keys and query to output a vector of energy scores e. The energy scores vector is used by the distribution function g to compute the attention weights vector a (output of the core attention mechanism). These weights evaluate the relevance of each element to the task.
It could happen that an additional representation of the inputs is needed. This is modeled as V, a sequence of vectors used to apply the attention values previously computed. This combination of V and a gives a set of weighted representations of V, called Z. Finally a compact representation of Z is produced, called *context vector*, that is composed by values with highest attention weights.
The context vector can be used to provide a computational gain [34], because it can encode the input in a compact form, thus working as a feature selection process. In some environments the information given by the context vector could be used to extract the most significant features that a NN uses to make its predictions, thus improving the network's interpretability.
Attention can also be used as a way of injecting knowledge into the NN model, as a mechanism to combine symbolic and sub-symbolic systems, and as a guide to unsupervised learning architectures [34].

Deep attention, as designed and developed by Pavlopoulos et al. [36], is a possible implementation of an attention mechanism. Deep attention is different from a standard attention mechanism, because it can be used in a classification setting, in which there is no previously generated output sequence to drive the attention; it is implemented through a small multi-layered neural network, that takes the inputs, it elaborates them, and finally it outputs a collection of weights in the form of Softmax output. These outputs can be used to highlight the most relevant input values to the task.

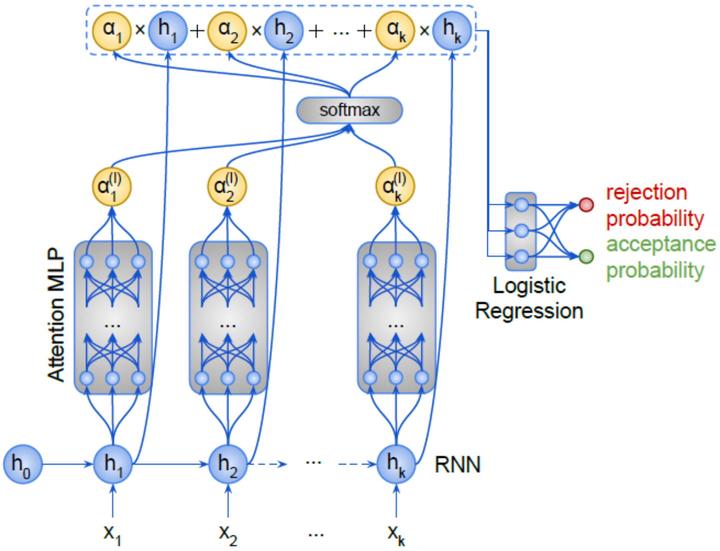


Figure 3.11: Deep attention mechanism for Recurrent Neural Networks, as seen in Pavlopoulos et. al [36]

Chapter 4

Generative Models

In supervised learning, training data x and their labels y are known, and the goal is to learn a function that maps inputs to the right label (eg. Classification tasks).
In unsupervised learning the training data is unlabeled, and the goal is to learn some hidden structure of the data, thus extracting salient features of the input (eg. Clustering, feature learning).
We refer to Generative Models (GM) as a class of models for performing unsupervised learning, where given training data the goal is to generate new samples for the same distribution of the input.
The main difference with respect to Discriminative Models (DM) is that, while DM predicts a label or category to which that data belongs, given certain data features (so $p(y|x)$, the probability of y given x), GM map labels to features, so the goal is to find the best probability distribution of features given a class (so $p(x|y)$, predicting features given a label) [22].
There is a taxonomy of GM, with main categories of explicit or implicit density models. Variational Autoencoders (VAE) are in the explicit-approximate density category, Generative Adversarial Networks (GAN) are in the implicit density category [22].

4.1 Variational Autoencoders

We refer to Autoencoders (AE) as an unsupervised approach to learn some features of the input data. The encoder maps input data x into features z, and it can be implemented as deep, fully connected network, or a CNN. z is usually smaller than x, in order to capture meaningful features [22].
The features z can be used to output data similar to the original data, thanks to decoders; so, basically data flows from input x, to a low dimensions feature space z, to reconstructed input data *x'*.
It is necessary to use a loss function, such as the L2 loss function, to evaluate the error between reconstructed data and original input data. Since training is unsupervised, no labels are used during the process.
After training the model, the decoder can be discarded, and encoder and feature space can be used to create a supervised model with predicted label and classifier [22].
VAE add probability evaluation on AE, thus it is possible to sample from the model to generate data. From z, containing latent attributes extracted from the input, it is possible to sample according to a certain probability distribution (such as Gaussian distribution). In order to generate new data from the model the parameters must be estimated.
The first step is to choose a distribution for z, such as Gaussian, then the decoder model must estimate the true parameters of the input.
The encoder and decoder network are probabilistic, since the model works with probabilistic

generation of data. So both the encoder and the decoder networks are producing distributions, and we can sample on these distributions to get values from them (z and x'). The goal of VAE is to maximize the likelihood of the original input being reconstructed, and by varying samples of z we get different characteristics on generated data.

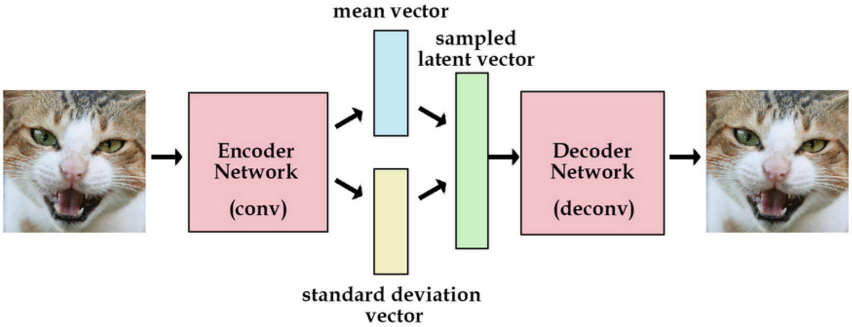


Figure 4.1: Simplified model of a VAE [37]

4.2 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a class of Neural Networks for learning generative models. The GAN architecture consists of two entities: a Generator Network (G) and a Discriminator Network (D). The main goal of a GAN is to train a G that takes in input a vector of noise z and produces samples from the data distribution $p_{data}(x)$, so that the generated data is not distinguishable from real data, thus fooling the D to accept the G's outputs (p_g) as real. Meanwhile, the D is trained to distinguish samples from the G distribution from real data, and provides a training signal for the G. D's output is a single scalar, which represents the probability that the input data came from real data rather than G's output.

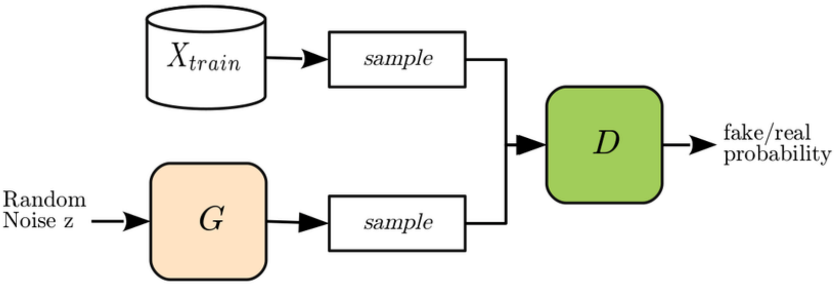


Figure 4.2: Generative Adversarial Network model [38]

Training a GAN basically consists in finding an equilibrium between two players, G and D, in a non-cooperative minimax game, where each player tries to minimize its cost function [39]. Competition drives both G and D to improve their methods (the quality of *counterfeit* data and *discrimination technique* respectively). In order to minimize each player's cost simultaneously, a traditional gradient minimization technique can be used to train both models, alongside backpropagation and dropout algorithms.
G can be trained to minimize $\log(1-D(G(z)))$, where $D(G(z))$ is the output of D representing the probability of input data being real. Unfortunately, this equation may not provide sufficient gradients for G to learn well, so rather than training G to minimize that equation it is usually

better to train G to maximize $logD(G(z))$, thus providing much stronger gradients early in learning [39].
D is trained to maximize the full objective function of the minimax game:

$$\min_G \max_D V(D,G) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{4.1}$$

Training is performed alternating between some steps (k) of optimizing D and one step of optimizing G, resulting in D being maintained near its optimal solution. The global optimum is found when generated data is indistinguishable from real data, so when $p_g = p_{data}$ [39]. Recent papers and researches discovered that instead of trying to optimize the number of steps needed for G and D, it is better to train D and then immediately train G, or try to balance training with a principled approach rather than intuition, such as performing multiple steps of training on one of the two networks if the loss becomes too big or too small [40].
D, for each step, samples a minibatch of m items from $p_g(z)$ (the G output), then samples a minibatch of m examples from the real data distribution $p_data(x)$, and finally updates the D by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^i) + \log(1 - D(G(z^i)))] \tag{4.2}$$

Meanwhile, during each training iteration, the G samples a minibatch of m noise items from noise $p_g(z)$ and updates the G by ascending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^i))) \tag{4.3}$$

4.2.1 Improved GAN training

There are several techniques to improve the stability of training and the resulting quality of GAN's samples, thus encouraging the convergence of the GAN min-max game [41], such as:

- Feature matching: a different objective function is employed for G, one that prevents it from overtraining on the current D [41]. Indeed, the G is now trained generate data that matches the real data statistics; D is employed to specify which statistics of the real data are more relevant to the task. In this way, D tries to find those features that helps us distinguish the most between real and generated data, while G tries to produce data that match the expected value of the features extracted by D. The objective function has a fixed point where G exactly matches the distribution of training data;

- Minibatch discrimination: this technique was introduced in order to reduce the main failure mode of GANs, that is the collapse of the G's output to an emission of the same point. This is caused by the absence of a mechanism to tell the outputs of the G to become more different to each other, and the result is that all outputs converge towards a single point. The D is unable to separate identical outputs, and the algorithm cannot converge.
  To overcome this problem, D must be able to look at multiple data examples at once, performing a *Minibatch discrimination*. The D's job is still to output a likelihood value for each example, discriminating between real or generated data, but it is able to use extra information coming from the other examples in the minibatch. In practice if all the different samples in the minibatch are very similar, the D can recognize the *collapse* and reject the samples as fake;

- Historical averaging: each player's cost includes a term representing the value of the parameters in the past, thus allowing to compute the historical average of the parameters and use it during the training process

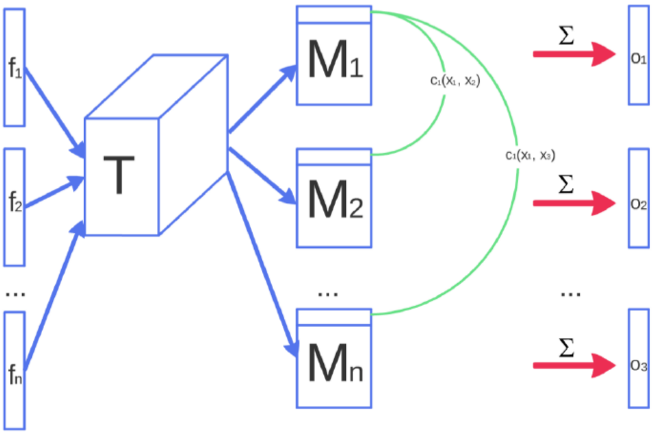


Figure 4.3: Minibatch discrimination in GANs. Features $f(x_i)$ from sample x_i are multiplied through a tensor T, and cross-sample distance is computed [41]

GANs have many advantages: only backpropagation is used to compute gradients, so there is no need of using Markov chains [42] or inference during training, thus making the learning process more reliable and stable; moreover, a wide variety of functions can be incorporated into the model; finally, the G is updated only with gradients flowing through the D, not directly with data examples, so input features are not copied directly into the G's parameters.

The main disadvantages are that there is no explicit representation of $p_g(x)$ and that D and G must be synchronized carefully during draining, so that G is not trained too much without updating D. Another downside for GANs is that a large number of object classes is particularly challenging for them, due to their tendency to underestimate the entropy in the distribution [41].

Ultimately, GANs are able to generate high quality examples thanks to a structure based on a *student-like* G and an *expert*, the D, coaching the G on specific tasks and guiding its development.

4.3 Wasserstein GAN

Training GANs is well known for being delicate and unstable [43], so it is useful to implement techniques to make this process more reliable.

Wasserstein-GAN (WGAN) is a form of GAN that mitigates the main training problem of GANs, thus not requiring neither maintaining a careful balance in training G and D, nor a careful design of the network architecture [43].

A WGAN tries to minimize an approximation of the Earth Mover distance, also called Wasserstein distance

$$W(P_r, P_g) = \underset{\gamma \in \prod(P_r, P_g)}{inf} E_{(x,y)\sim\gamma}[\|(x-y)\|] \tag{4.4}$$

where P_r and P_g are two distributions, representing real and generated distributions respectively, and $\prod(P_r, P_g)$ is the set of all possible joint probability distributions between P_r and P_g.

$\gamma(x, y)$ is one joint distribution, and it indicates how much *mass* must be transported from x

to y in order to transform the distributions P_r into the distribution P_g [43]. The cost of the optimal transport plan (the minimum one) can be seen as the value represented by the Earth Mover distance.

The Earth Mover distance is better than other divergence measures, such as the Kullback-Leibler one, since even for distributions located in low dimensional manifolds, it can provide a meaningful and smooth representation of the distance. This means that the gradient is always continuous, thus improving learning in a lot of situations.

The idea is to use the Earth Mover distance as the GAN loss function, but since is intractable to try all the joint distributions to find the minimum, instead of measuring the minimum value it is necessary to measure the maximum. If we have a prarametrized family of functions f_{ω_ω}, the problem to solve becomes

$$W(P_r, P_g) = \underset{\omega \in W}{max} E_{x \sim P_r}[f_\omega(x)] - E_{z \sim p(z)}[f\omega(g_\theta(z))] \tag{4.5}$$

The goal is to find a function f that solves the maximization problem in the equation. To approximate this, it is possible to train a neural network parameterized with weights w lying in a compact space W, and then backpropagate as it is done in standard GANs [43]. In order to have parameters w lie in a compact space, thus enforcing the Lipschtiz constraint, the weights of the network are clipped to a small range of values after each gradient update.

In WGANs we do not have a standard Discriminator, because it is not distinguishing between fake and real samples anymore; instead, there is a Critic that helps in estimating the Earth Mover distance between real and generated data distribution. Since the Earth Mover Distance is continuous and differentiable, the Critic can and should be trained until optimality [43].

If we compare WGANs with standard GANs, we will see that in the latter the Discriminator learns quickly to distinguish between fake and real data, thus providing a weak signal to the Generator, that is unable to improve significantly; on the other hand, in WGANs we have a Critic that can not saturate [43], so gradients are always sufficiently strong to train the Generator.

Another important characteristic of WGANs is the absence of mode collapse: in original GANs, the Generator converges to an optimal configuration by adapting to the signal coming from the Discriminator, and tends to produce a gradually lower number of unique examples; in WGANs, since the Critic can be trained until optimality, the Generator is unable to collapse to a small subset of examples, thus leading to a much larger variance in the generated data.

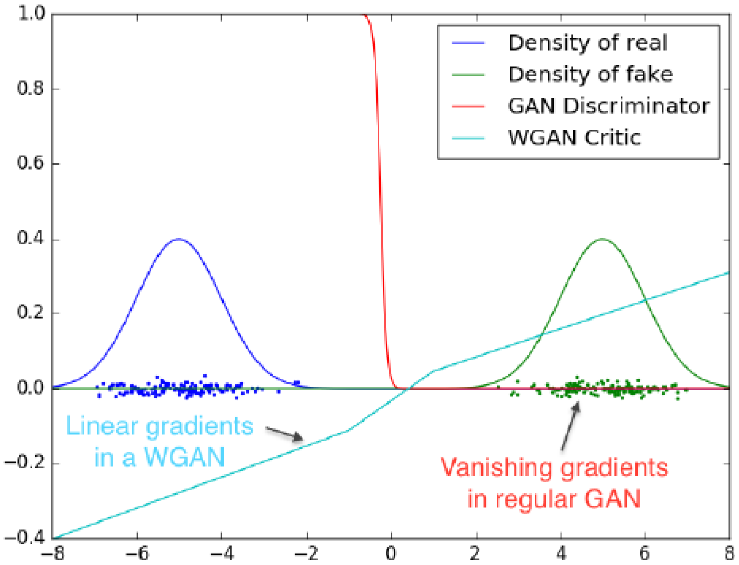


Figure 4.4: Optimal discriminator and critic when learning to differentiate two Gaussian distributions. The discriminator exhibits vanishing gradients, while the WGAN critic provides very clean gradients on all parts of the space [43]

4.3.1 Improved WGAN training

As we saw in the previous section, in the original WGAN weight clipping is enforced during training.

The problem with using a clipping parameter is that, if it is set to a large value, the weights will take a long time to reach the limit, thus making harder to train the critic until optimality [43]. On the other hand, a small clipping parameter leads to vanishing gradients. Clipping weights could also lead to undesired behaviour, such as a bias of the critic towards much simpler functions, thus reducing its capacity, and exploding gradients, due to the interactions between the weight constraint and the cost function [44].

An alternative and better approach, is to penalize the norm of the gradient of the critic, with respect to its input. This approach is called *gradient penalty*, and the resulting model is called Wasserstein GAN with Gradient Penalty (or WGAN-GP).

Since a differentiable function can satisfy the Lipschtiz constraint only if it has gradients with norm at most 1 everywhere [45], WGAN-GP has a term (gradient penalty) in the loss function, which penalizes the network if its gradient norm moves away from 1.

The objective function is

$$L = \underset{\tilde{x}\sim P_g}{E}[D(\tilde{x})] - \underset{x\sim P_r}{E}[D(x)] + \lambda \underset{\hat{x}\sim P_{\hat{x}}}{E}[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2] \tag{4.6}$$

WGAN-GP has demonstrated a significant improvement in training speed and sample quality [45], and it also helps in stabilizing training.

Chapter 5

Model Agnostic Architecture

5.1 Neural Network solutions for CSPs

In the recent years, thanks to the fast development of newer an more effective techniques, Neural Networks (NN) and Deep Learning models have been employed more and more to address many different kind of problems.
While the main fields of application for NNs remain the image recognition task, and the natural language processing task, there exist several instances where NNs have been employed to solve, or help solving, CSPs.

Galassi et al. [46] employed a Deep Neural Network (DNN) that learnt how to construct solutions for two different CSPs, the Partial Latin Square problem, and the 8-Queens completion problem. The DNN in question did not have any explicit information about the constraints of the two problems, and it was trained to extend a feasible partial solution by making a single, globally consistent, variable assignment. Despite the low accuracy, the DNNs displayed a great ability in producing globally consistent assignments, leading to the conclusion that the networks have learnt something about the structure of the problems.

Xu et al. [47] applied deep learning techniques in order to predict the satisfiabilities of CSPs, without relying on humans to select optimal features for the training set. The CSPs analyzed here are random Boolean binary CSPs. The idea is that, by accurately predicting satisfiabilities, the likelihood of choosing a variable that results in a satisfiable sub-problem is increased; the result is a reduced number of backtracking steps needed in a typical CSP solving algorithm. A Convolutional Neural Network was employed, in order to take a data point (CSP) as input, and predicting its label; the network reached a very high prediction accuracy ($> 99.9\%$).

Xue et al. [48] propose an hybrid solution, called DDGan, that is able to both capture implicit user preferences and to generate solutions that satisfy physical or operational constraints, thus combining the benefits of machine learning and constraint reasoning. A Generative Adversarial Network (GAN) is employed to learn user's preferences, and the solutions generated by it are filtered through a decision diagram module (DD) that ensures the feasibility of the solution itself.
This architecture is motivated by the fact that many decisions depend on implicit factor, or preferences, that can not be easily captured by a mathematical objective function.
DDGan is applied to solve routing and scheduling problems, and it shows the ability of producing a significant number of feasible solutions, especially if compared to a standard GAN model.

Shao et. al [49] employ a Deep Neural Network (DNN) to solve the travelling officer problem (TOP).
The TOP is an NP-hard problem, and it describes the problem of an officer that tries to max-

imize a cumulative reward, in this case parking violations. The officer is able to move through a fully connected graph.
DNNs are useful in this context, because they can be trained offline, and perform inference runs in real-time, an ideal behaviour due to the fast evaluation needed for the TOP.
Optimisation methods are used to produce the labels for DNN training, and a data segmentation technique changes the problem from an optimization one into a classification one, so that DNNs can be effective.
The result is a framework that incorporates optimization approaches and DNNs, using a DNN to approximate the greedy algorithm. The output of the network represents the next node that the officer should travel to.
Overall, the performance of the DNN was consistent, but worse than the one obtained using only the greedy algorithm.

Finally, Hottung et al. [50] built a deep learning network that can be integrated into an heuristic tree search procedure, to decide which branch to choose next, and to estimate a bound for pruning the search tree of an optimization problem.
Tree search algorithms are often employed to solve NP-Complete problems, and heuristic tree search is often used to provide good solutions without a large time consumption. Usually, the heuristic is designed by domain experts, thus depending on their knowledge, while here the deep learning network is able to work in place of an heuristic function.
The problem analyzed is the container pre-marshalling one (CPMP), which is NP-Complete; it basically consists in re-sorting containers stacked in a terminal, so that they can be quickly extracted from the stacks. Each container has a group that indicates its scheduled exit time. To solve the problem the containers must be sorted to optimize the retrieval operations.
Two NNs are employed: a policy network, that makes predictions about which branch is considered the best, and a value network, that predicts the cost of completing a solution for a node in the tree search. The training sets are derived from optimal, or near optimal, solutions for the problem in hand, so that each network's training example consists of a partial solution that the network should extend.
This approach has shown a significant improvement over the state of the art methods, even on real-world sized instances, an impressive result, given the fact that the user does not need to inject much knowledge about the problem into the model.

5.2 GANs and CSPs

In this thesis work, we want to design and test two different Neural Networks: a Discriminator network (D) and a Generator network (G), and employ them in a Generative Adversarial Network (GAN).
The idea is to build models that are able to learn the structure of a combinatorial problem, the 8-queens completion problem in this instance, and also how to solve them. The key aspect is that all the knowledge is provided *a priori* (offline); furthermore, to be as agnostic as possible to the problem's structure, all input and output data of the networks are represented through bit vectors, making the encoding general.
The G is trained to produce a single, globally consistent assignment. The training set consists only of positive examples, with the goal of teaching G to follow the desired assignments, thus mimicking specific construction sequences. In this way, G should be able to learn constraints that are implicit, often called *preferences* or soft constraints, and also hard constraints, the ones relative to the problem at hand; what G should not be able to do, is to distinguish between the two types of constraints. If G becomes good in the task it has been trained for, it will be used to guide a depth first search process.

On the other hand, the D is trained on both positive and negative examples, with the goal of making it a differentiable *checker*, so it could be used in different situations.
D does not learn preferences, instead it should able to memorize the *hard* constraints, that depend on the problem in hand, so that it should able to tell whether a provided solution is feasible or not (with respect to the problem constraints). D could be employed as a checker during for an heuristic algorithm, helping in determining whether the solution found by the heuristic method is right or wrong.

After performing the two separate pre-training processes of G and D, the idea is to combine G and D together in a Generative Adversarial Network (GAN). Given the duality of the knowledge that G and D have about the problem, the hope is that by making them work together in a unified model, G can learn to imitate better the *real* solutions, while D can improve in detecting when a preference is followed.

Finally, we want to assess the ability of G to produce full solutions that are feasible, starting from empty ones. The goal is to compare the results obtained by training G trough different training models, such as GANs and WGANs, to the stand-alone case, and to, hopefully, improve G's ability with respect to the one observed using the stand-alone training.

5.2.1 GANs limitations with discrete data

Before analysing the generation of datasets, and the experimental results obtained using the previously described NNs, it is crucial to highlight the pros and cons of using discrete data in adversarial networks training.
GANs have enjoyed great success in modeling continuous distributions; however, applying GANs to discrete data it is not an easy task, in that it is difficult to optimize the model distribution toward the target data distribution in a high-dimensional discrete space [51].

As we will see in the following chapters, even if GAN experiments yielded interesting results there is still margin of improvement; this could be caused by the fact that, by keeping the model as agnostic as possible to the problem, it is necessary to represent the problem data a discrete distribution, in particular by employing binary data. This works well as an abstraction technique, but it hinders the overall GAN performance, since GANs are optimized to work with continuous data.
Indeed, in a GAN the data provided to the D should be composed only by real or fake data [39]. If we look at the experiments executed for this thesis, this guideline is respected for the positive examples, since they are sampled directly from the D's dataset that contains only feasible solutions; the same is not true for the negative ones (fake data), because they are made of data generated from the G's predictions, and these predictions contain a significant portion of perfectly feasible data, thus creating a mix of feasible and unfeasible solutions.

5.3 Datasets

In this section, we will define and describe all the datasets used in the experimental sections.
A dataset for the N-Queens Completion problem, is built by taking the standard 12 base solutions and by converting them into a binary encoding; this means that, if a queen is present in a certain square of the chessboard it has a corresponding value of "1" in the dataset, otherwise if no queen is present a "0" is used. A one-hot encoding is used in order to represent an assignment, so that only a "1" is present and corresponds to the position of the following queen placed on the chessboard.
This representation can be extended to work with bit vectors, both for the input and the output, and to work with assignments composed by one-hot encoding of multiple values (eg.

the PLS problem), thus making the representation both general and truly agnostic to the problem constraints [46].
For this thesis work, different families of datasets were produced, to meet the needs of the experiments.

In the first family of datasets, designed to be used in the pre-training of both G and D, solutions are simply split into two groups of different dimensions. The result is the generation of two datasets for G, one for training and one for testing purposes, and other two datasets for D.

In the second family of datasets, we introduce a bias by segregating solutions with at least a queen in a central position on the board in one group, and the other solutions in a second group. This helps worsen the performance of the Discriminator Network, so that D has a greater margin of improvement, thus motivating the choice of combining G and D in a GAN model. From the two generated datasets, we experiment with one or the other as the training set, in order to find the best dataset configuration for our needs.

In the third family of datasets, the previous bias during the dataset generation is still present, but this time the solutions with queens in a central position are discarded, thus producing a dataset containing only solutions with no queen in the center of the board. We will experiment with this dataset using a GAN and a WGAN.

5.3.1 First dataset family - Generator

As already stated, the first family of datasets does not have a particular selection criterion; the result is that, starting from the 12 base solutions of the 8-Queens problem, two distinct groups are obtained by randomly selecting 8 base solutions for one group, and the other 4 for another group.
Given the set of binary-encoded base solutions, the training examples are obtained by performing the following steps:

1. Derive two different sets from the 12 base solution, one containing $\frac{1}{4}$ of the solutions that will be used to generate the test set, and one containing $\frac{3}{4}$ of the solution used for the training and validation sets

2. Generate all the possible final solutions by performing the symmetric equivalents and the rotation equivalents of the base solutions, for each set

3. Apply a deconstruction of the generated solutions (for each set), thus obtaining sets composed by pairs of partial solutions and corresponding desired future assignment. The systematic deconstruction process is performed on all the full and partial solutions, generating all possible construction sequences, and it works by undoing an assignment at a time, and by saving the partial solution and the removed assignment into the dataset.

4. After the deconstruction of all the original solutions, the dataset is pruned by considering all groups of examples sharing the same partial solution, and selecting a single representative at random

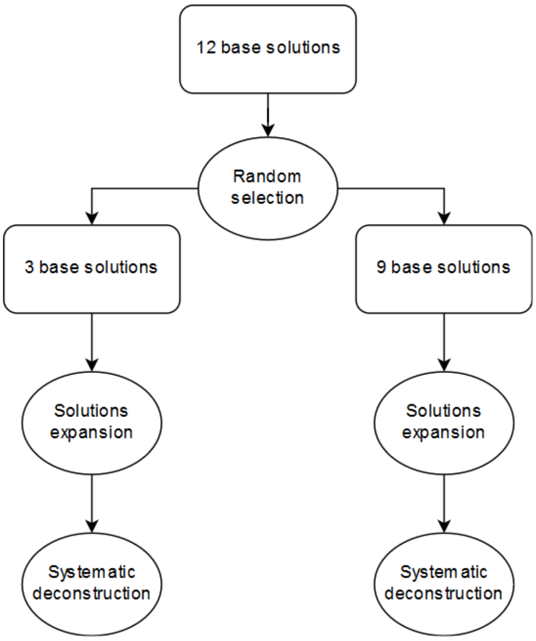


Figure 5.1: Steps to generate the two initial datasets for G

Ultimately, the training and test dataset for the G are obtained. First we have the test set, called *TE-G*, composed by all the partial solutions derived from 3 base solutions, thus representing 25% of all possible solutions; the other set contains all the partial solutions derived from the other 9 base solutions, representing 75% of all possible solutions. The latter set is split into $\frac{2}{3}$ of the data for the training set, called *TR-G*, and $\frac{1}{3}$ of the data for the validation set, called *VAL-G*.

5.3.2 First dataset family - Discriminator

The next necessary step is to produce the datasets for training and testing the Discriminator. The two already existing datasets, TR-G and TE-G, were the ones containing feasible (partial) solutions and their respective assignments; a third dataset, containing all the full feasible solutions (92) for the problem, called *DS-FULL*, was generated as well.
To generate proper training and test sets for D, we perform a sequence of operations:

1. we load two distinct datasets, the first dataset containing a pair of partial solutions and respective assignments, such as TR-G, while the second one contains all the final solutions for the n-Queens completion problem, DS-FULL

2. we merge feasible partial solutions and their respective assignments, both contained in the same dataset. This is performed simply as an element-wise sum of each partial solution and its assignment, thus obtaining a partial solutions with their assignments applied to it. Each new partial solution is used to create a dataset of feasible (partial) solutions, called *FEASIBLE-D*

3. we generate unfeasible assignments by violating row, column or diagonal constraints. This is achieved by iterating on FEASIBLE-D dataset, and for each solution selecting a random violation of the constraints. The next step consists in extracting the position of existing queens in the linearized chessboard, then picking the position of a single queen as a reference for the violation to insert; if multiple queens are present in the selected partial solution, then a random queen is selected. Afterwards, based on the violation

type, a queen is inserted in a illegal position, thus making the solution unfeasible.
The last step is to use the unfeasible solutions generated to create a dataset, called *UNFEASIBLE-D*

4. we create unfeasible partial solutions that can not generate any full feasible solution. This is accomplished by iterating on the feasible partial solutions, then by inserting a queen in a a random position on the board; finally, we check whether this new board can be extended to a final solution or not. If it can not be extended to a final solution, the generated partial solution will be added to the UNFEASIBLE-D dataset

5. we eliminate duplicate partial solutions from the UNFEASIBLE-D dataset, thus removing redundant data

The amount of unfeasible data generated is obviously dependant on the size of the dataset to be transformed: if we use TR-G as input, the script will produce a little more than 39000 unfeasible (partial) solutions, while the dataset with feasible solutions contains almost 21000 elements.
The two datasets generated using the previous steps, FEASIBLE-D and UNFEASIBLE-D, are then merged together, keeping an equal proportion of feasible and unfeasible solutions; the result is a training set for D, called *TR-D*. The same process is repeated using TE-G as input, yielding the final test set for D; we will call it *TE-D*.

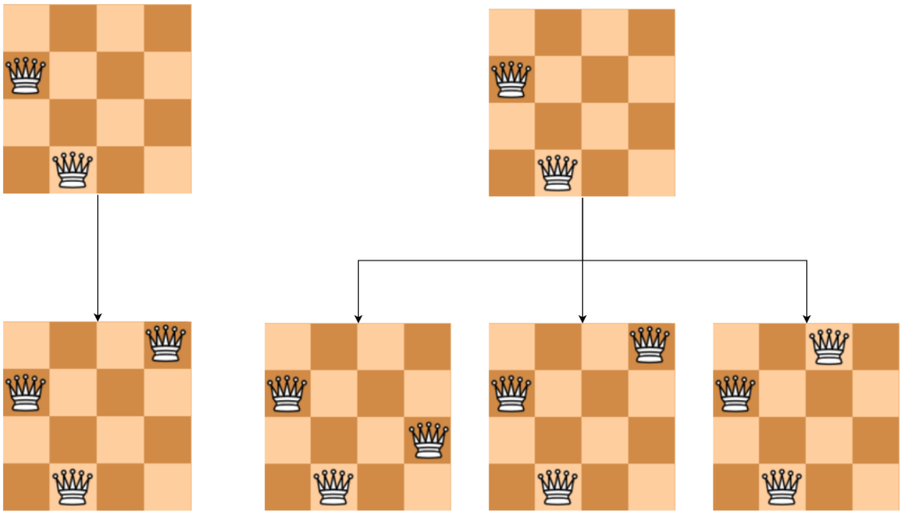
5.3.3 Second dataset family

In the previous section, we saw the generation of datasets for both G and D, by performing systematic deconstruction on two groups of base solution, which were selected and segregated randomly.
Using this family of dataset, we want to obtain a D that has low accuracy (ideally between 0.30 and 0.40) on the test set; this is because if D is already very good in the task it has been trained for, it can not improve by much by training it through a GAN; furthermore, a very strong D can hinder the G learning capability in a GAN setting.
With this goal in mind, the dataset generation process was slightly but significantly changed: instead of dividing the 12 base solutions in two groups without any principle, they were divided according to their characteristics and properties: the first class of base solution is now composed by the 4 solutions with a queen in a central position, where *central* is defined by the 2×2 square in the middle of the chessboard; the other class of solutions is composed by the remaining 8 base solutions, the ones without any queen in the central 2×2 square.

The result is the creation of two datasets for G: the first is derived from the 4 base solutions with queens in a central position, called *DS-G-C*; the other one is created from the other 8 base solutions, and we will called it *DS-G-O*.
For each dataset, two distinct variations are produced:

- Uniques: for each partial solution in a dataset, only one random feasible assignment is chosen. The result is that all the solutions represented by the dataset are unique, so no partial solution is replicated

- Multiple: for each partial solution, all the feasible assignments are present, so the same solution can appear multiple times in the dataset.



(a) *Uniques* dataset variety

(b) *Multiple* dataset variety

Figure 5.2: Here we see a comparison of the resulting solutions, obtained by applying the assignments, for the 4-queens completion problem. In the *uniques* datasets variety, we have only one assignment for each partial solution; in the *multiple* dataset variety, we have all the possible assignments (right).

The *uniques* variety is obviously smaller than the *multiple* one, and it will be used to train both the G and the GAN; the latter will be used only for the training of the GAN network.

Finally, to generate the proper datasets for the D stand-alone and the D in the GAN network, it is necessary to perform the same steps seen in the previous section, using *DS-G-C* and *DS-G-O* as the reference datasets. The outcome is the generation of two new datasets for D, called *DS-D-C* and *DS-D-O* respectively, both in the *uniques* and *multiple* variety.

5.3.4 Third dataset family

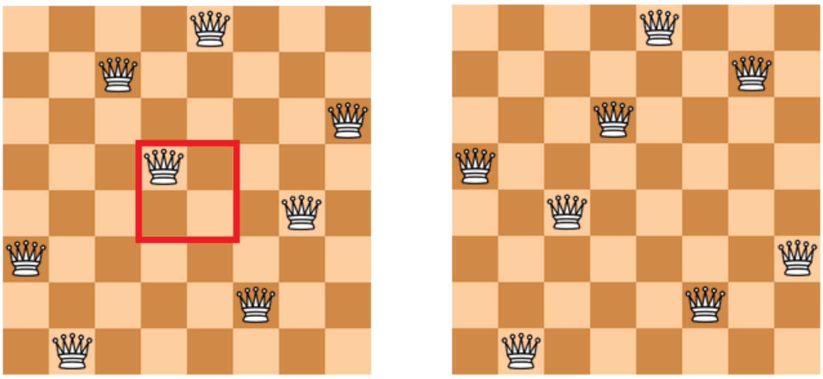
In the previous dataset family, the datasets were generated by selecting solutions with certain characteristics (*bias*), specifically by grouping together solutions that contained queens in a central position (2 × 2 central square in the chessboard). Following this selection process, we produced the final datasets, used alternatively for training or testing purposes.

For the third dataset family we use a slightly different dataset generation process: the selection is still executed using the same criterion, or *bias*, but this time the 4 base solutions with queens in a central position are discarded, leaving only the remaining 8 base solutions as *valid* ones; then, we divide these 8 base solutions into two groups having the same size, so 4 base solutions in one group and 4 base solutions in the other one. This splitting operation is performed randomly.

The goal of using this dataset family for training, is to see if G is able to learn not only to respect the criteria of the problem at hand, but also to produce valid solution with respect to a certain *bias* inserted through data selection in the dataset generation process.

From the two new groups of base solutions, we are able to generate the full datasets for G by expanding the solutions, thus obtaining a new pair of datasets that we will call *DS-G-A* and *DS-G-B*. This results in two datasets that are obviously very similar, in terms of size and type of solutions represented.

As seen in the previous sections, for each dataset two distinct variations are produced: a *uniques* one, and a *multiple* one.



(a) Solution with a central queen

(b) Solution without a central queen

Figure 5.3: Comparison between a base solution with queen in a central position (left) and one without it (right)

Finally, to generate the proper datasets for the D, we employ the same method described in the previous chapters, using DS-G-A and DS-G-B as input. The final outcome is the generation of two new datasets for D, called *DS-D-A* and *DS-D-B*, along with their respective *unique* and *multiple* variety.

It is important to remember that no dataset in this family contains queens in a central position, therefore a strong bias is enforced.

Chapter 6

Experiments with 1st dataset family

6.1 Goal

In a CSP problem, like the 8-Queens Completion problem, there are two kinds of constraints that can be enforced to perform Constraint Optimization:

- Hard constraints: constraints that set mandatory conditions on variable's values, so they must be taken into consideration in order to correctly solve the problem
- Soft constraints: similar to a bias/preference, these are conditions that it is desirable to satisfy

The Generator Network (GN) should be able to learn the preferences (soft constraints) with respect to different problem states, using a custom dataset containing only positive examples that always reflect the desired preferences. G should also learn hard constraints, to be able to create feasible solutions; on the other hand, it should not be capable in distinguishing between the two types of constraints. In this way, after the training process, the GN becomes a sort of heuristic algorithm that can guide a search strategy in order to traverse the search space and reach a goal status.

While the GN learns the soft constraints by following the teachings of a *master*, the Discriminator Network (DN) should be able to understand and learn the hard constraints. These constraints are dependent on the problem, and must be taken into consideration to produce solutions that are feasible and suitable to the task. For instance, in the N-Queens Completion problem there are three different hard constraints: row, column and diagonal constraints.
The DN knows nothing about the preferences (soft constraints), but should be able to choose assignments that follow the hard constraints. This can be done by training the DN using a custom dataset containing both feasible and unfeasible partial solutions. In this way, after the training process, the DN becomes an heuristic algorithm controller, thus helping in determining whether the solution found by the heuristic algorithm is correct and should be pursued.
By modeling and implementing the Discriminator as a Neural Network, we have a differentiable checker that can be used in other applications, such as Generative Adversarial Networks (GANs).

6.2 Generator

6.2.1 Architecture

The GN is designed as a Deep Neural Network (DNN), based on the architecture of the preactivated Residual Network, and the structure of the Neural Network created in the "Model

Agnostic of CSPs with Deep Learning" paper [46].
The GN has been built and trained using TensorFlow, an end-to-end open source platform for machine learning, and Keras, an API framework that offers consistent and simple APIs, and minimizes the number of user actions required for common use cases, making it easy to learn and easy to use, but flexible enough to integrate with lower-level deep learning platforms (such as TensorFlow).
To create the GN, we did not use Convolutional Neural Networks, to be truly agnostic to the problem structure: CNNs work with grid-like data, so, by using CNNs, the 8-queens data should have been organized as a chessboard, thus leading to a more problem-dependent solution. Instead, with Residual Networks, we can organize data simply as a collection of binary elements, resulting in an higher level of abstraction.
Using these guidelines, multiple feed-forward, fully-connected (Dense) layers are employed, composing a series of block structures. In this architecture each layer block (LB) is different from the standard Dense layer, and it is composed of three elements in sequence:

1. Batch Normalization: provides normalization of input data and has also a regularization effect on the network

2. Activation: ReLU activation function has been employed

3. Dense Layer: a typical Dense layer, composed of a certain number of neurons, without a specified activation function

Another key element of the model is the residual block (RB): in this particular case it is composed of two LBs in sequence, the first one composed of 500 neurons, and the second one composed of 200 neurons.
The final GN model architecture is composed of an input layer that takes a two dimensional tensor, where the first dimension is dependent on the batch size and the second dimension represents the 64 variables (positions) in a 8×8 chessboard.
Following the input layer we have the first LB, composed of 200 neurons, then multiple RBs, usually between 20 and 100. The output of the first LB is added to the output of the following RB, thus obtaining the input for the next block. This operation is performed for each RB, using the previous RB output and the current block output to compute the sum, thus implementing a ResNet-like structure.
Finally a Softmax activation function is employed on top of the network, in order to provide a classification score for each output variable. The classification scores sums up to 1, and represent the likelihood of each variable to be the next desired assignment. The idea is to be able to take the variable with the highest likelihood as the network's preferred following assignment.
Dropout has been used, and does provide some increase in performance, despite the fact that the Batch Normalization process seems to already provide a strong regularization effect. Dropout is used immediately after every Batch Normalization function. Multiple trials were performed with Dropout placed before or after every Dense layer, but these configurations did not yield any significant boost in accuracy or feasibility ratio.
The loss function used is the Categorical Crossentropy, an extension of the Binary Crossentropy to multiple (> 2) classes.

6.2.2 Masking

Masking is a filter that can be computed from the input data, with the goal of eliminating unfeasible assignments from the GN's output, in particular to avoid assignments on variables that have a value already assigned. In order to implement it, it is necessary to make an

assumption of knowing that the 8-queens Completion game state is composed of an 8 × 8 chessboard, so it is possible to represent each row as a variable, and each column as the variable's value. The idea is to have a mask that has only zeros on rows corresponding to input variables already assigned (so with a *one* in the row), and only *ones* on rows corresponding to input variables that are unassigned (so with only *zeros* in the row). We can see an example of a masking layer derived from a partial solution in figure 6.1.
The final masking output is computed by multiplying input and mask element by element, thus filtering assignments that are not compatible with the input state, such as assignments on rows already containing a *one* (queen). Masking is performed just before the Softmax activation function of the GN, meaning that the network's output will be greater than zero only for variables corresponding to *free* rows.

4	0	0	0	0
3	0	0	0	0
2	0	0	0	1
1	0	1	0	0
	1	2	3	4

4	1	1	1	1
3	1	1	1	1
2	0	0	0	0
1	0	0	0	0
	1	2	3	4

Figure 6.1: Partial solution in input (left) and the corresponding mask tensor (right), using a mock 4 × 4 chessboard

The implementation of this masking process is executed by inserting a Lambda layer before the Softmax function. This custom layer takes the input and the previous layer's output tensors, then it splits the input tensor into N sub-tensors on the y axis, with $N = 8$ for a 8×8 chessboard, thus representing each chessboard's row as a single sub-tensor.
After the splitting, two different tensors are created with the same dimensions of the single sub-tensor, the first one containing all *zeros*, and the second one containing all *ones*. Then, for each sub-tensor, the sum of each element on the y axis is computed, thus reducing again the size of the tensor, collapsing the y dimension . This represents the sum of the elements in each row, so that a sum of 1 means that that row cannot have another queen assigned, while a sum of 0 means that the row is free.
Afterwards, a partial mask tensor is created by checking the sum tensor values: for each element in the tensor, if it's equal to zero (free row) the corresponding mask's row is populated with the previously created tensor of all ones, otherwise it is populated with the already created tensor of all zeros.
The mask tensors are then concatenated together on the y axis, to form the full mask tensor. Finally, the masked output tensor is computed by multiplying the full mask tensor with the output tensor of the previous network layer, thus reducing to 0 values corresponding to rows/variables that have already a queen assigned.

6.2.3 Attention

An attention mechanism has been employed in the GN to improve the network's performance and, possibly, speed up the training process. As we already stated, the goal of attention is to highlight the most relevant input data to the task, in this particular instance a classification decision, by computing attention weights and using them to obtain the input weights. Different types of attention mechanisms have been evaluated, and the selected one was the Deep Attention one [36].

Deep attention has been used in the GN by passing the batch-normalized network's input to 3 Dense layers in sequence; the first and the second layer is composed of 300 neurons, while the last layer has the same number of neurons as the number of input's attributes (so 64 units). ReLU activation function has been employed for all three layers, then the last layer's output is added element-wise with the input tensor, thus computing directly the input weights that can be used in the following layers of the network.

In the original formulation of deep attention the last layer uses the Softmax activation function, providing a collection of likelihood values for each input data that highlight the input's relevance to the task, called *attention weights*. Only then the input weights are computed, by multiplying the attention weights by the input tensor. This exact process could not be implemented for the N-queens Completion problem since the input is represented through binary encoding, meaning that the product between inputs and the Softmax output would have been greater than zero only for non-zero inputs. An attention model of this kind would have highlighted only the *ones* in the input tensor, while completely ignoring all the *zeros* (including the most relevant to the task). This is the reason why Softmax has not been used in the last layer of the attention network, and also why an addition operation has been employed instead of a multiplication one. The Softmax function could have been used in combination with the addition instead of the multiplication, but since the Softmax's output is composed of small values, especially in a problem with many output classes, the *ones* in the input tensor would have been highlighted far more than the *zeros* and essentially not solving the issue.

With these adjustments, the *zeros* input data can receive an higher amount of attention, thus improving the overall GN's performance both on the training set and on the test set accuracy.

6.2.4 Metrics

To properly train the GN, and achieve the best results, it is important to choose the metric to monitor during the early stopping procedure. The choice was between the validation loss and the validation accuracy, particularly tricky in a multi-class problem with many classes and with the Categorical Crossentropy loss function: during training, the validation loss value can fluctuate, sometimes decreasing while the validation accuracy decreases, or vice-versa. This behaviour is caused by the fact that loss increases while accuracy remains the same when some examples with very bad predictions keep getting worse, or when some examples with very good predictions get a little worse. In order to mitigate these phenomenons the validation accuracy was used as the metric for the early stop process, with a patience of 15 in a small GN, and a bigger patience value (between 20 and 25) for a larger GN.

Another important metric to monitor during training and testing phases, is the accuracy of the GN, respectively on the training and the test set. Accuracy is a metric that reflects how well the network is able to learn the correct variable-value assignment for the problem, so basically how well the network's follows the teaching of its *master*.

While these metrics are useful, there is a more significant metrics for the GN, that we will analyze in detail in the following section: the feasibility ratio.

6.2.5 Feasibility Ratio

In order to properly evaluate the performance of the GN, it is necessary to assess not only the network's accuracy but also the ability of the network to yield globally consistent assignments,

even if such assignments are not the ones chosen as correct in the training or test set. Furthermore, since the main goal is to create a neural network that can guide a search process, generating feasible assignments is more important than reach a high accuracy on the training or test sets.
The feasibility ratio is a metric to evaluate the ratio of partial solutions that could be expanded to full solutions. To compute it, it is necessary to train the network first. After the training, the feasibility evaluation function is called, passing a different data set (training, validation and test set) each time; the function iterates on all the partial solutions and predicts the assignment for each particular partial solution. Afterwards, the number of queens is obtained by counting the number of *ones* existing in the original partial solution. The next step is to apply the predicted assignment, thus placing another *one* in the partial solution, and to check if the new solution is feasible or not feasible. This is performed by counting the total number of queens in the new partial solution, then multiplying each of the 92 full solutions (for the 8×8 8-Queens problem) with the partial solution, element by element; for each result solution, the algorithm counts the number of queens, and checks if this number is equal to the number of queens of the partial solution. If the number of queens is the same, it means that the partial solution is feasible, because all of its queens are placed in a way that allow an expansion to a full feasible solution, otherwise if one or more queen *disappear* it means that some of them were in unfeasible positions. This is obviously possible since all the solutions for the n-Queen completion problem are known *a priori*.
Finally a global ratio is computed, by dividing the total number of feasible solutions by the total number of solutions; single feasibility ratios for each different game state are computed as well, by keeping track of the amount of filled cells in the current partial solution (so the number of existing queens).

6.2.6 Hyper-parameters

An important choice is selecting the number of Residual Blocks, so deciding the size of the GN. Different tests were performed using a bigger network with 100 residual blocks, a smaller network composed of 20 residual blocks, and a medium-sized network composed of 50 residual blocks. With the exception of the training time needed, obviously smaller for the network with less layers, the three networks provided almost the same performance in terms of classification accuracy on the test set, with a small edge in favor of the medium-sized network.
As previously stated, Dropout has been implemented with a rate of 0.1 after each Batch Normalization layer, providing a small boost in performance.
Another hyper-parameter to tune was the number of neurons/units for the network's layers, both inside and outside the residual block. After several trials, a size of 200 units was decided for the first Dense layer and the last Dense layer in the residual block , while the first layer in the residual block was composed of 500 units.
The Adam optimizer was used, with learning rate values between 0.001 and 0.005, a β_1 of 0.9, and a β_2 of 0.999. A custom learning rate decay fucntion has been employed, defined as follows:

$$lr_{new} = \frac{lr}{(1 + (ep \times k))}$$

The k value in the learning rate decay custom function was set to 0.001 after multiple trials with different values.
The Elu activation function was also used for every layer in the network, but since no significant performance increase was observed the ReLU function was used in the final network's model.
For the attention network, different tests were performed to choose the right number of units

in the Dense layers, settling for 300 units for the first two layers.
Finally the impact of masking and attention has been evaluated by performing rounds of training and testing with and without these mechanisms, gaining an insight on the influence of both the techniques on the final classification performance.
All the GN's hyper-parameters were defined after a manual tuning process.

6.2.7 Experimental results

The GN was trained for 200 epochs in a supervised way, using the validation dataset as a benchmark. Early stopping was implemented by monitoring the validation accuracy with a patience of 20 epochs, meaning that training was stopped when validation loss did not improve after 20 consecutive epochs.
In table 6.1, we can see a recap of several evaluations, performed using different models and techniques.
With masking active and attention mechanisms turned off, the best accuracy reached on the test set was 0.14 for the *medium-sized* GN, while the larger GN's best result was an accuracy of 0.13 on the test set, and the smaller GN's best result was an accuracy of 0.14. Overall the accuracy is rather low, but still much higher than the accuracy that we could reach by performing a random guessing ($1/64 \approx 0.015$ for the N-Queens Completion problem).

num. res. block	masking	attention	accuracy test set
100	yes	yes	0.11
100	yes	no	0.13
100	no	no	0.10
50	yes	yes	0.12
50	yes	no	0.14
50	no	no	0.13
20	yes	yes	0.13
20	yes	no	0.14
20	no	no	0.12

Table 6.1: G's accuracy on the test set, using different configurations

During the evaluation of the feasibility ratio it was possible to observe an interesting effect given by the usage of Dropout: as previously stated, Dropout yields a small improvement in the accuracy if the network, but its usage seems to highly increase the feasibility ratio of the generated solutions.
The results, concerning the global feasibility ratio, can be seen in table 6.2; the medium-sized network, containing 50 residual blocks, was able to slightly outperform the deeper one (100 residual blocks) and the smaller one (20 residual blocks). The presence of a masking layer helped to raise the feasibility ratio, while attention did not provide any performance boost. The best global feasibility ratio on the test set, obtained with the medium-sized GN, was 0.60.

num. res. block	masking	attention	feas. ratio test set
100	yes	yes	0.51
100	yes	no	0.54
100	no	no	0.49
50	yes	yes	0.55
50	yes	no	0.60
50	no	no	0.57
20	yes	yes	0.56
20	yes	no	0.58
20	no	no	0.54

Table 6.2: global feasibility ratio of G on the test set, using different configurations

We can take the GN that yields the highest global feasibility ratio, and analyze how the feasibility ratio changes for each of the seven different solution types. Each solution type is identified by the same number of queens.
To have a better idea on the performance of the GN, we can compare the GN yielding the best overall result, in terms of feasibility ratio, with the one exhibited by the Model Agnostic paper by Galassi et al. [46]. In figure 6.2 it is possible to compare the best GN created in previous sections and the best GN of the paper, in terms of achieved feasibility ratio measured on different solution types.

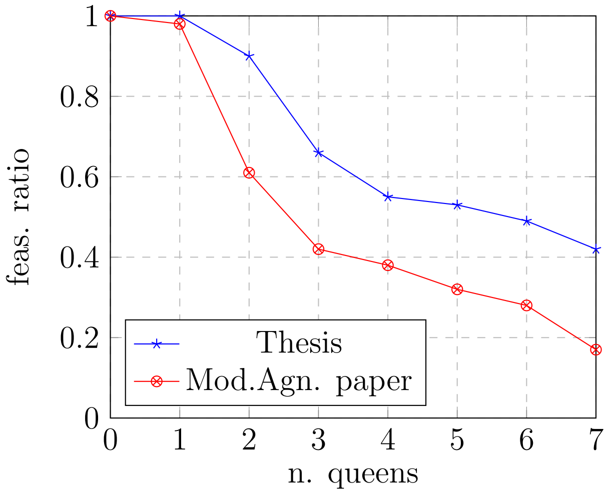


Figure 6.2: feasibility ratios of two different GN, measured on the test set for each solution type.

6.3 Discriminator

6.3.1 Architecture

The DN has been built and trained using the same technologies as the GN, so TensorFlow framework and Keras API.
A key element is that the DN does not have any specific knowledge about the problem, because no knowledge is injected in the network beside the input data (offline injection). Furthermore, the input data is composed only by a sequence of bits, thus making the model truly agnostic to the particular problem to analyse.
The DN has the same base architecture as the GN: it is a Deep Neural Network (DNN) with

a similar design to Residual Networks with pre-activation. The Residual Block's structure is the same as the one in the GN, with Batch Normalization, ReLU activation and Dense layer in sequence, and even the number of neurons in the Dense layers is the same (500 for the first residual block, 200 for the second one). The GN and the DN are mostly similar, but there are several key differences: first, the masking process can not be used in the DN, so it is deleted from the discriminator's model; the loss function used is the Binary Crossentropy function, not the Categorical Crossentropy that it is used in the GN. This is obviously because the DN has a binary output, not a multi-class one; in the last part of the DN it is necessary to use a Sigmoid activation function, in order to output a choice between *feasible* and *not feasible*, and also because Softmax does not perform as intended alongside the Binary Crossentropy loss function.

The same attention mechanism of the GN has been used, but since it did not provide any benefit in terms of performance or training time, it was eliminated from the final discriminator model.

Dropout has not been used, since it doesn't provide any performance increase, and given the fact that the Batch Normalization process seems to provide a strong regularization effect.

Adam optimizer function has been employed, with a learning rate between 0.00006 and 0.00009, a β_1 of 0.9, and a β_2 of 0.999. The same custom learning rate decay algorithm of the GN has been used, where the initial learning rate lr was progressively annealed through epochs, with decay which is proportional to the training epoch ep, and k coefficient set to 0.001, resulting in the following decay fucntion:

$$lr_{new} = \frac{lr}{(1 + (ep \times k))}$$

The DN was trained for 200 epochs in a supervised way, using the validation dataset as a benchmark. Early stopping was implemented by monitoring the validation accuracy with a patience of 20 epochs, meaning that training was stopped when validation loss did not improve after 20 consecutive epochs.

All the DN's hyper-parameters were defined after a manual tuning process.

6.3.2 Hyper-parameters

To train the D, both the training and test set of the first family were used. The validation set VAL-D was obtained from the training set TR-D, by randomly sampling 1/3rd of the training set data.

Validation accuracy was used as the metric for the early stop process, for the same reasons as described for the GN. Various patience values were used, such as a patience of 10 for the smaller network, and a bigger patience value (between 20 and 25) for the larger DNs. Different tests were performed using three different networks, similar to what has been done with the GN: a small network consisting of 20 residual blocks, a medium-sized network composed of 50 residual blocks, and a larger network of 100 residual blocks.

As previously stated, Dropout has not been used for any of the three DNs.

The Adam optimizer was used, with learning rate of 0.0001. The k value in the learning rate decay custom function was set to 0.001 after multiple trials with different values.

The same attention mechanism implemented in the GN was initially used in the DN as well, but since there was no performance improvement it has been eliminated in the following trials.

During training and testing we were interested in measuring the accuracy of the DN, respectively on the training and the test set. The Accuracy of a DN, is a metric that shows how much the network is able to distinguish between feasible and unfeasible solutions, so basically how well the network's follows the teaching of its *master* in discriminating between feasible and unfeasible (partial) solutions.

6.3.3 Experimental results

In table 6.3 we can see a recap of the results obtained using different configurations.
For the DN, we are interested in measuring the accuracy, meaning its ability to distinguish between feasible and unfeasible solutions.
The best accuracy reached on the test set was 0.88, using the *medium-sized* DN, while the larger and the smaller DN's best result was an accuracy on the test set of 0.85. The accuracy of the DN is high, and suggests that the DN is doing very well at the task it has been trained for.
On the training set the accuracy was even higher than the test set one, always around the 0.99 mark, indicating that the DN does not make many mistakes when working with data seen during training.

num. res. block	attention	acc. train set	acc. test set
100	yes	0.98	0.80
100	no	0.99	0.85
50	yes	0.99	0.82
50	no	0.99	0.88
20	yes	0.98	0.81
20	no	0.99	0.86

Table 6.3: D's accuracy on training and test set, using different configurations

6.4 Analysis & Observations

6.4.1 Generator

The low accuracy exhibited by the GN during the experiments, suggests that the GN is not doing particularly well at the task it has been trained for.
On the training set the accuracy was considerably higher than the test set one, a typical case of overfitting. In this instance overfitting is not caused by a defective model or by missing regularization elements, but there is a structural reason for it to happen: the pruning in the last phase of dataset generation introduces a degree of ambiguity, because it is possible that, for the same partial assignment, the training and the test set may report different *correct* assignments that cannot be both predicted correctly.

In terms of feasibility ratio, as expected G performs better when using solutions that have less queens already in place, while its ability of performing feasible assignments decreases as the number of queens in the original solution increase.
By comparing the GN with a similar GN, created for the Model Agnostic paper [46], it is possible to observe a significant improvement for all the different solution types. This means that the GN produced in this section is able to generalize better, thus being capable of producing more feasible assignments on data that it has not seen during training.

While masking helped boosting both test set accuracy and feasibility ratio, the attention mechanism did not provide any benefit, in fact it worsened a little both accuracy and feasibility. Multiple trials were performed, using an alternative attention mechanism having only two layers of 200 and 64 neurons, but even with this configuration the performance was worse than the configuration without attention. This is probably because the *zeros* do not gain a significantly higher value than the *ones*, in fact *ones* are highlighted more than *zeros*, thus failing to placing the focus on the most important *free* cells in the chessboard, resulting in a small decrease in performance. This is why the best overall results were obtained with the attention mechanism deactivated.

6.4.2 Discriminator

As seen in the experimental part, the DN shows a great ability of distinguishing feasible and unfeasible solution, both on training and on test set, meaning that it has gained knowledge about the hard constraints of the problem in hand.

Since the training and test set accuracies are not very different, it is possible to say that overfitting is almost not existent and that the DN is able to generalise very well.

The fact that the DN is able to perform so well can be a problem, because the original idea was to insert GN and DN in a GAN model, with the goal of improving their performance. This can not be done if the DN is too powerful, since a strong DN would definitely hinder the learning of the GN.

In the following chapter, we will address this problem by working with the second family of datasets.

Chapter 7

Experiments with 2nd dataset family

7.1 Goal

In this section, we will train G, D and GAN models using the second family of datasets. It is important to remind that each dataset contains solutions that have different characteristics from the solutions in the other dataset, in particular one is composed of solutions with queens in a central position, while the other has only solutions with no queen in a central position.

The goal of the experiments in this section is to re-train both the G and the D, using the second family of datasets, and determine which dataset should be the training set and which one should be the test set, by looking for the one that worsen the most the performance of D. This is due to the fact that the D seen in the previous experiments was too strong, thus it did not make sense to employ it in a GAN, since it would have hindered G's learning.

The other goal of these experiments is to analyze the performance of a GAN, composed of a G and a D, and to improve GAN's performance through gradients and model architecture analysis.

Initially we try to improve a pre-trained version of G and D by employing them in a GAN, to see if both can improve simultaneously, or if only one it is able to become better instead.

When G and D are pre-trained, their performance, measured by the feasibility ratio for G, and by the accuracy for D, is already excellent on the training set, thus it can not improve; in order to employ them in a GAN, we need to use the test set of the pre-training process as the GAN training set. This helps us in the beginning to properly design the GAN, because pre-trained G and D have a fairly good performance on the original test set and they can still improve. We expect that the performance on the original training set, that is now the test set, will decline a little during the GAN training, since the network will adjust its weights based on the new training data distribution.

Once the GAN is correctly designed, and it is able to learn using pre-trained models, we will employ untrained versions of both G and D, in order to compare the GAN training with the stand alone training process.

Subsequently, we will see how the GAN behaves if we use a mix of noise and training data as G's input, and we will see also how the GAN performs if we employ a G without its masking layer.

7.2 Generator

7.2.1 Design & Hyper-parameters

The G has basically the same architecture of the G seen in the previous experiment.
G is modeled as a Residual Network, so it is composed of multiple Residual Blocks, connected through the addition function. In the final layer we have a Dense layer and a Softmax activation function, that outputs the *preference* of G in deciding the next assignment to make. The loss function is the Categorical Crossentropy.
Dropout is employed, with a rate of 0.1, immediately after every Batch Normalization function, and helps to increase the test accuracy and, more importantly, the feasibility ratio of the generated assignments.

Adam optimizer function has been employed, with a learning rate between 0.005 and 0.0001, a β_1 of 0.9, and a β_2 of 0.999. A custom learning rate decay algorithm has been used, where the initial learning rate lr was progressively annealed through epochs, with decay proportional to the training epoch ep, resulting in the following decay fucntion:

$$lr_{new} = \frac{lr}{(1 + (ep \times k))}$$

with a k of 0.01.
The GN was trained for 200 epochs in a supervised way, using the validation dataset as a benchmark. Early stopping was implemented by monitoring the validation accuracy with a patience of 10 epochs.
The following experiments were performed using a medium-sized network composed of 50 residual blocks.
The number of neurons/units for the network's layers is the same as the previous G, so 200 units for the first Dense layer and the last Dense layer in the residual block, while the first layer in the residual block is composed of 500 units.
The ReLU function was used as the activation functions in all the network's layers.
Attention has not been employed, since results in the previous experiment were worse when using it.
On the other hand, the Masking layer is active since it has shown a positive impact on the performance.

7.2.2 Experimental results

The training of G is performed in the same way as the previous experiment, this time using DS-G-O as training set, and DS-G-C as the test set, then switching them for the following training process.

In terms of accuracy, we can see the results in table 7.1. The training set accuracy is rather low, due to the fact that the early stopping mechanism halted the training when it was clear that the G was overfitting, so it is safe to say that without this mechanism the value could be much higher.
We do not see significant differences between the two training experiments, since the test set accuracy is the same, and the training set accuracy very similar.

train set	test set	train set accuracy	test set accuracy
DS-G-O	DS-G-C	0.41	0.08
DS-G-C	DS-G-O	0.43	0.08

Table 7.1: G's accuracy on the test set, using different training sets

In table 7.2 it is possible to observe the global feasibility ratio exhibited by G, for both training and test sets. No significant differences can be seen between the first and the second evaluation; as expected, G's feasibility ratio is slightly higher using DS-G-O as the training set, since in this way G is trained on a larger quantity of data, but overall the two assessments yield similar results.

train set	test set	train set feas. ratio	test set feas. ratio
DS-G-O	DS-G-C	1.00	0.38
DS-G-C	DS-G-O	1.00	0.31

Table 7.2: G's global feasibility ratio, using different training sets

Finally, we can see the feasibility ratios of G for each solution type in figure 7.1, when using DS-G-O as the training set. The results are worse than the ones seen in the previous experiment, but that's to be expected, since in this instance training and test set data have meaningfully different characteristics.

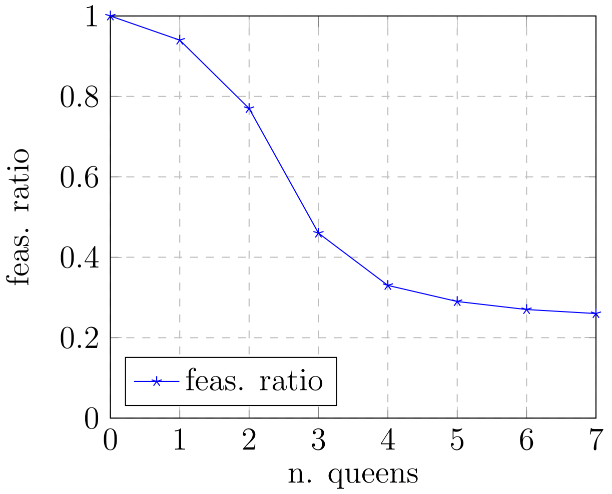


Figure 7.1: feasibility ratios of G-O, measured on the test set for each solution type.

7.3 Discriminator

7.3.1 Design & Hyper-parameters

The network's architecture is very similar to the one described in the previous experiment: D is a deep network modeled as a Residual Network, with 50 residual blocks. Each residual block is composed of two sub-blocks, containing a Batch Normalization layer, an Activation layer (ReLU activation function) and a Dense layer. Dropout has been applied only on the first Dense layer. The optimizer is Adam optimizer, with a learning rate between 0.0005 and 0.005. An early stopping mechanism is employed, with a patience of 20 on the validation accuracy. Using this setup, the D is trained multiple times using with the DS-D-O dataset as the training set, and the DS-D-C dataset as the test set. The validation data is obtained from the training set, sampling randomly $\frac{1}{3}$ of the training data.

7.3.2 Experimental results

The best training accuracy reached using DS-D-O as the training set is 0.99 after 72 epochs of training, with a test set accuracy of 0.42.
By training the same D network, but with the DS-D-C dataset as the training set, and the DS-D-O dataset as the test set, we get similar results, with a maximum training set accuracy of 0.99 after 88 epochs of training, and a test set accuracy of about 0.40. These results can be seen in table 7.3.

train set	test set	train set acc.	test set acc.
DS-D-O	DS-D-C	0.99	0.42
DS-D-C	DS-D-O	0.99	0.40

Table 7.3: D's performance using different training sets

If we compare these results with the ones obtained using the first family of datasets, it is evident that D suffered a significant loss in accuracy. This was expected and it was the desired outcome, because in this way D is not too strong and can be employed in a GAN without blocking G's improvements.

7.4 GAN

7.4.1 Original GAN design

A GAN is composed of a G and a D network working together, so in order to create a GAN model it is necessary to load or build both G and D. After the respective training process, the structure of the models, their weights configurations, and the state of their optimizers, were saved on file, in order to be able to restore them when building the GAN.
The first step of the GAN network builder consists in loading the discriminator model. Then, the following step is to re-compile discriminator: by compiling a model again, the optimizer states are lost, but the weights of the network will not change. This is not mandatory, since the D is pre-trained and already compiled, but it helps in resetting the status of the optimizer, thus providing a faster training process.
Using the same approach, we load the G.
To provide the G's output as input to the D, an intermediate layer is required. This is due to the fact that G produces a single assignment in output, highlighting the position of the queen to be inserted into a given partial solution, but the D is able to properly work only with partial solutions as its input, not with single-assignment solutions. We will describe and implement the Lambda intermediate layer in the following section.

By creating an unified model, composed of the original G and the Lambda layer, we get a generator that applies assignments to the original partial solutions, thus producing new partial solutions. This model is called *full* Generator.
The next step is to set the *trainable* property of the D model to *False*: this is motivated by the fact that for the combined model, G plus D, we want to train only the G.
The final operation to create the GAN is to create a *combined* model: the outcome of the *full* Generator is provided to the D as its input, so the *combined* model has the partial solutions

tensor as its input, and the outcome of the D as its output. Finally, it is necessary to compile the *combined* model, thus freezing the weights of the D (since its *trainable* property is set to *False*).
Basically, as we can see in figure 7.2, the *combined* model consists in a sequence of G, Lambda merging layer, and D, and the goal is to train the generator to fool the discriminator. It is crucial to remember that the original D model is still trainable, and it will be trained in parallel of the *combined* model, so its weight can change and the weight alteration will be reflected on the D in the *combined* model.

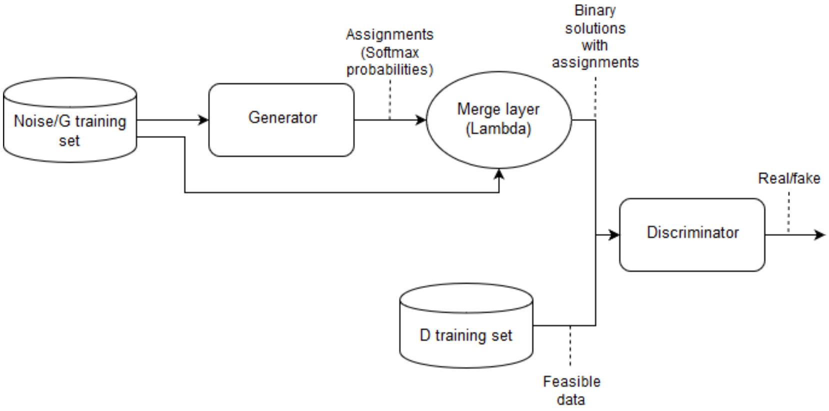


Figure 7.2: Original GAN structure

7.4.2 Merging layer

This intermediate layer is implemented as a Lambda layer, positioned immediately after the G: the Lambda layer takes two tensors as input, one is the tensor containing the partial solutions (input of the G), and the other is the tensor containing the assignments (produced by G). This Lambda layer merges these tensors together, and it must employ only operations that have a defined gradient, so it it not possible to use binary conditions, such as Keras backend *when* function, and operations with no gradient, such as the *argmax* function. Since the assignments tensor is produced by the Softmax output function in the G, the *argmax* would have been extremely useful to select the position of the assignment with the higher value, instead it is necessary to implement the same functionality using operations that allow the training of the network.
The final version of the merging Lambda layer is composed of the following operations:

1. Max: computes a tensor with only the maximum values for each assignment

2. Subtract: subtracts from each assignment tensor its max value, so that the desired assignment has a value of 0, and the other values are less than 0

3. Sign: transforms values that are not the desired assignments to -1, and the desired assignment to 0

4. Ones like: creates a tensor of ones, with the same shape as the assignment tensor

5. Add: performs plus 1 operation, on the *signed* assignments tensor, so that values that were -1 become 0, and values that were 0 become 1 (the desired assignments). This is essentially a binary mask, with a 1 in the position of the desired assignment

6. Add: merges together the binary mask, obtained from the assignments tensor, with the partial solutions tensor used as input to the G

The outcome of this Lambda layer is a tensor having the selected (by G) assignment set to 1 in the original partial solution, so it produces new partial solutions that can be used as the D input. In figure 7.3, it is possible to see an example of an assignments tensor produced by G (Softmax output), and how it is transformed by the Lambda merging layer. The tensors are represented through two smaller mock 4 × 4 chessboards, to simplify the visualization.

4	0.02	0.11	0.015	0.065
3	0.003	0.01	0.03	0.05
2	0.004	0.010	0.5	0.05
1	0.025	0.025	0.005	0.05
	1	2	3	4

4	0	0	0	0
3	0	0	0	0
2	0	0	1	0
1	0	0	0	0
	1	2	3	4

Figure 7.3: G's assignments before (left) and after (right) the Lambda merging layer, using a mock 4 × 4 chessboard

7.4.3 Experimental results for original GAN

Training a GAN requires finding the right balance between the training of G and D.

We first start by using pre-trained versions of both G and D. In the training function, the first thing to do is to load the datasets: since both G and D have already been trained, the training set is defined by the test set used during the original training of G and D, so if DS-G-O (or DS-D-O) was the original training set, now DS-G-C (or DS-D-C) is the GAN training set, and vice-versa.

The D training set contains only the positive examples, not the negative ones. Both G and D dataset come from the *uniques* family. For the purpose of evaluating the performances, it is required to load also the original training and test set for the D, the test set for G, and the dataset containing the 92 full solutions for the n-Queen completion problem (DS-FULL).

To properly train D and the *combined* network, we define the adversarial ground truths as two 2-d arrays: the *valid* ground truth array is composed of ones, while the *fake* one is composed of zeros [52]. Both arrays have N elements, where N is the batch size (usually N is equal to 64). This is because both D and G will be trained batch by batch, and a single batch contains only real or only fake data.

For each epoch, the training process executes a number of steps of D training, and a number of steps of G training:

- Discriminator: for each step of D training, first we perform a random sample from the D training set, by shuffling the dataset and selecting the first N elements, where N is the selected batch size (usually 64). Then we sample N elements from the G training set, using the same methodology. By providing the data sampled from the G training set as input to the *full* G network, using the *predict* method, we get the partial solutions with the applied assignments.

  Next, it is necessary to train the D, both on valid and fake data: this is done by using the *train on batch* function two times, the first time with the sampled feasible solutions from the D training set as the input (x), and the *valid* adversarial ground truth as desired output (y), and the second time using the predictions obtained from the *full* G as input (x), and the *fake* adversarial ground truth as desired output (y). In doing so, the D is trained to recognize the data coming from its training set as valid, and the data produced by the G as fake. Finally, the losses and accuracies of the two training methods are added together, then divided by two, to obtain the D total loss and total accuracy.

- Generator: for each step of G training, a random sample from the generator training set is performed, as shown for the D. Afterwards, we train the *combined* model using the *train on batch* method, providing the sampled data as the input (x), and the *valid* adversarial ground truth as the desired outcome (y), so we train the G to have the D label samples as valid. This is possible since both the Lambda layer and the D model have freezed weights, so the only element of the *combined* model that can be trained is G.

After training D and G, a check on the epoch value is performed. There are two different evaluation intervals: one is faster and less in-depth, in which the program checks the feasibility ratio of the assignments produced by G, by passing a small sample of data from the current training set to the solutions evaluation function shown in the previous chapters; secondly it evaluates D's performance on the current training set. The full evaluation phase is usually executed after more epochs, such as every 20 or 50 epochs, since it is slower than the other one: it performs two assessments on the feasibility of the solutions produced by G, one on the full training set and another on the full test set; afterwards, D's performance is evaluated both on the full training set and on the full test set.

Several training processes were performed, using the Adam optimizer, for both G and D, with a learning rate ranging from 0.0005 to 0.001. Unfortunately, it became clear since the early attempts that this network configuration was incapable of boosting neither G or D's performances. Indeed, the main problem was that D became too strong, and was able to lower its loss both on real and fake data very quickly, while G's loss grew epoch after epoch; the results was that both G and D could not improve, in fact both network's performances were down a little with respect to the original performances. In figure 7.4 it is possible to see an example of the progression of G and D losses during training.
To improve training, several configuration were examined, such as GANs having G pre-trained and D not pre-trained, or vice-versa, with little success. Even with both G and D not pre-trained, the GAN training stability was not found, and the result was at best a small improvement on D.

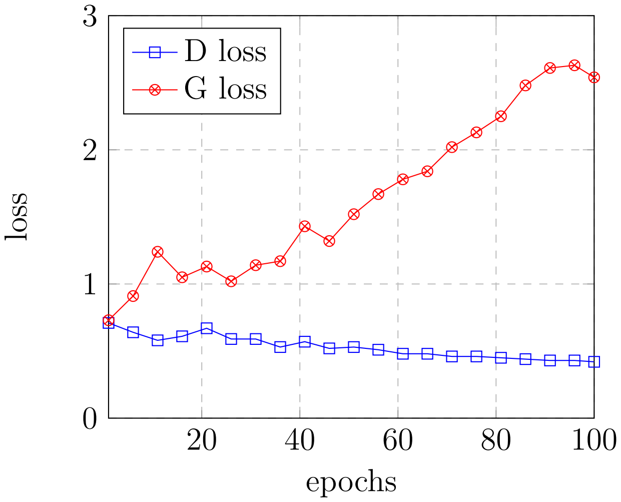


Figure 7.4: G loss and D loss progression over training epochs in the original GAN

7.4.4 Improving gradients flow in GAN

The first step to understand the causes of the poor GAN performance, is analyze the network to see if G and D can be individually trained after the GAN setup, and if the gradients that flow through the *combined* network, from D to G, are bigger enough to guarantee a proper training process.

After loading a pre-trained G, a D, and creating the *full* G and the *combined* models, the training analysis starts from the stand-alone G: by using the *fit* method, with the G original test data as input, it is possible to verify that G is still able to correctly improve by directly training it.

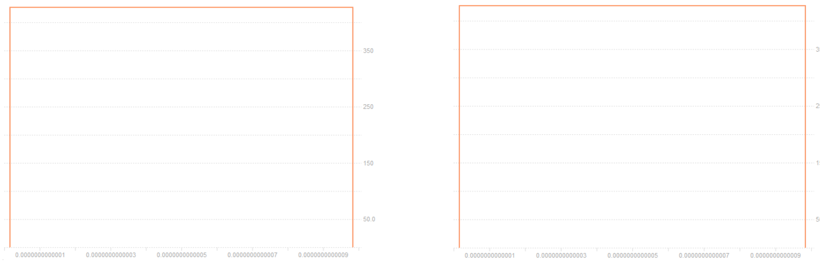
Subsequently, we look at the training performance of the *combined* network, the model composed of the original G, the Lambda merging layer, and D. To perform a training of the *combined* model, it is necessary to use the original G test data as input (x), and an array containing only ones as labels (y). In order to properly evaluate the network, we train the D for one epoch on the training GAN data (the original test data), reaching an accuracy of about 0.75. Given this setup, it is possible to train the *combined* model using the same *fit* function, but this time it is clear that the network does not improve at all. Indeed, by measuring the initial feasibility ratio of G, the accuracy of the *combined* network during training, and the feasibility ratio of G after the training process, we can verify that no increase in performance is obtained, as seen in table 7.4.

Before training		After training	
acc. combined	feas. ratio G	acc. combined	feas. ratio G
0.70	0.41	0.70	0.41

Table 7.4: G and *full* G performance before and after training

This anomalous behaviour is a symptom of poor gradients flow through the network, and by comparing the training performance of *combined* model with the stand-alone G it is easy to see that the Lambda layer is the main culprit for this phenomenon.

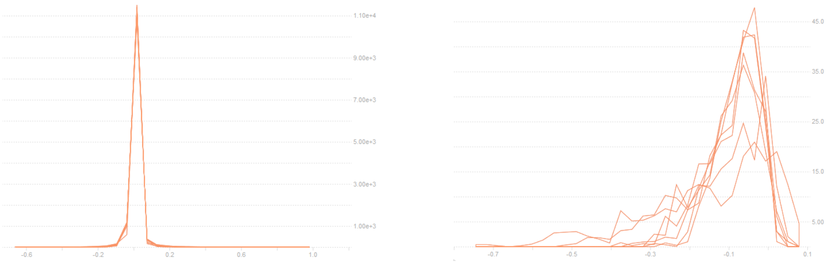
To properly evaluate the gradients flow, it is useful to create a monitor function using the Tensorboard callback, and passing it to the *fit* function when training the *combined* network. Tensorboard is a suite of visualization and evaluation tools, included in the Tensorflow package. As we can see in figures 7.5 and 7.6, that show the values of G and D kernel's gradients on both the first and the last Dense layer, the gradients for D are fine, while the gradients for G are extremely flat and have very small values during all the epochs (three) of training of the *combined* network.



(a) G's first layer

(b) G's last layer

Figure 7.5: Kernel gradients in first and last G layer, respectively



(a) D's first layer

(b) D's last layer

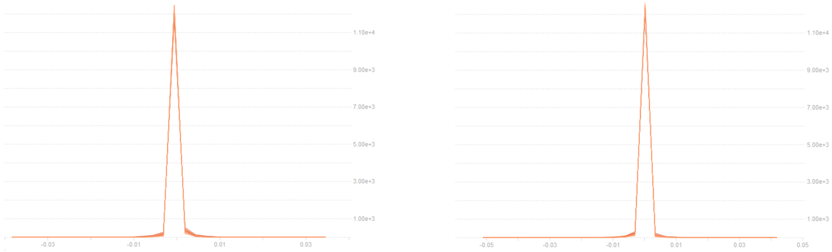
Figure 7.6: Kernel gradients in first and last D layer, respectively

The Lambda layer, as designed, uses a binary mask that drag the gradients value down, furthermore it uses several operations that contribute in reducing the gradients (such as the *sign* function). After multiple trials with different operations in the Lambda layer, no margin of improvement was found, so the Lambda was abandoned in favor of a simpler implementation. Knowing that the Lambda layer is the likely culprit of the poor network's performance, the first thing to do is to remove it, and use an addition layer between the original partial solutions and the assignments computed by the G. This simplifies the network's architecture, and it's a more natural choice, since the G's assignments are directly applied on the inputs. Another advantage of using only the Add layer, is that the solution produced by G become very similar or identical to the original binary solutions, thus helping during the training process of D. By analysing the training performance, it is possible to notice that the *full* G network actually improves if trained directly on a dataset:

Before training		After training	
acc. full G	feas. ratio G	acc. full G	feas. ratio G
0.61	0.41	0.99	0.61

Table 7.5: G and *combined* network performances before and after training (without Lambda layer)

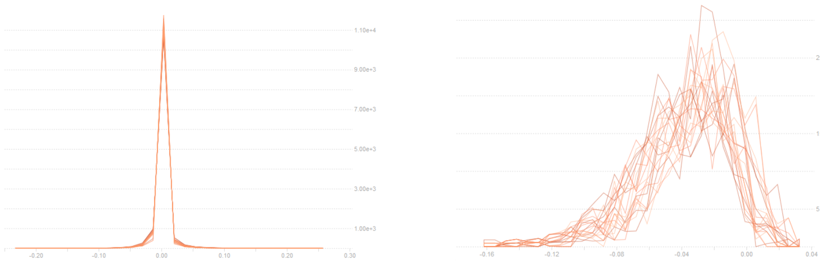
To have a better insight on the network's behaviour, it is necessary to perform a gradients analysis using Tensorboard. This analysis shows that the gradients are much better than before, both on the first and on the last layer of G (figure 7.7), and on fist and last layer of D (figure 7.8):



(a) G's first layer

(b) G's last layer

Figure 7.7: Kernel gradients in G, without using Lambda layer

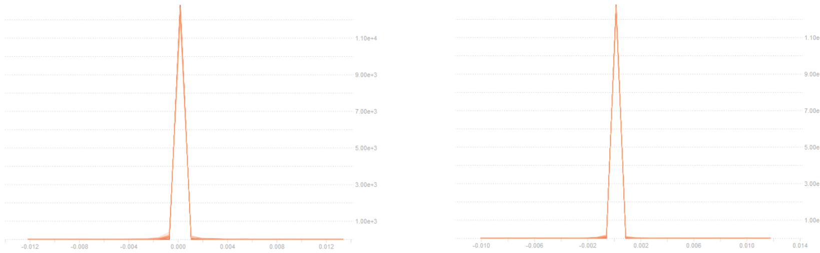


(a) D's first layer

(b) D's last layer

Figure 7.8: Kernel gradients in D, without using Lambda layer

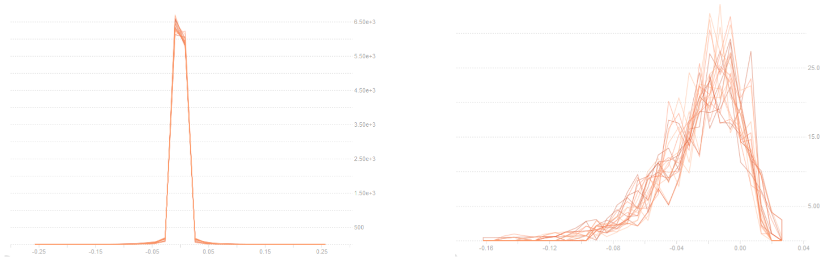
If the Lambda merging layer drags down the overall performance, then it is reasonable to look for another weak point in the architecture: the masking layer. This layer is positioned at the end of the G network, and it's implemented as a Lambda layer that uses a binary mask to filter out assignments that would be placed on a row with an already existing queen. We remove the masking layer from the G, leave only an Add layer between G and D, and try to evaluate the gradients flow in the resulting *combined* network. Even if we expected a drop off, similar to the one experienced using the merging Lambda layer, no such thing happens. In fact, the gradients are very similar to the ones of the previous architecture, leading to believe that this Lambda layer behaves differently than the merging one. This may be due to the fact that the masking layer uses the multiplication, not the addiction, as the operation between binary mask and partial solutions, to compute the desired assignment. Furthermore, the mask tensor is built by working on a array-like version of the assignments tensor, not directly on it. The gradients of the *combined* model having G without the masking layer can be seen in figures 7.9 and 7.10.



(a) G's first layer

(b) G's last layer

Figure 7.9: Kernel gradients in G without masking layer



(a) D's first layer

(b) D's last layer

Figure 7.10: Kernel gradients in D (using G without masking)

In the following sections, different GAN performance evaluations will be performed, both on architectures having a G with masking layer, and ones without it.

7.4.5 The Batch Normalization problem in GAN

We already analysed the behaviour of G and *full* G when training them directly on training data, so what is missing is an evaluation of D. After building the GAN, we take D and train it using the *fit* function, passing the original test set data as input. By tracking the accuracy of D before, during and after training, it is possible to observe an anomalous behaviour: D seems to train correctly, since its loss value reduces epoch by epoch and its accuracy increases, but the measures of loss and accuracy after the training process are often equal, or even less, than the ones before it. This is caused by the Keras implementation of the Batch Normalization layers: this layers perform two operations, they normalize the data by subtracting the mean and dividing by the standard deviation, and they train two additional parameters (per node); somehow, after setting the *trainable* property to False on the D, and after compiling the *combined* model, the additional parameters on the Batch Normalization layers of the original D freeze, but it looks like the layers still perform the normalization operation during training time, using the training batch. This is extremely confusing, and the result is that the stand-alone D is unable to learn correctly, even if it seems that the *fit* function is working correctly. To overcome this, both G and D were re-designed: all the Batch Normalization layer were substituted with Dropout layers, using a dropout rate of 0.1. By performing this modification, it is possible to verify that D is now able to learn correctly even after setting the *trainable* property to False, and after compiling the *combined* model.

7.4.6 Improving GAN design

After verifying that G, *full* G, and D were trainable when using the test data, our goal is to enhance the GAN performance. To accomplish that, we look back at the previous GAN design, and analyse which design choices were wrong and what can be changed to obtain a better and faster training process.

As seen in the previous model design, the Lambda merging layer between G and D was drastically reducing the learning potential of the GAN; moreover, the masking layer in the G contributed a little in worsen the overall performance. While these two layers were part of the problem, not all the issues were solved by simply removing them, thus a detailed analysis of the original GAN architecture is necessary.

The first element to examine is the inequality of the number training steps for G and D: this imbalance doesn't work well, especially in networks that have different performances. Instead of trying to balance the losses via statistics, so trying to find a number of G/number of D schedule to uncollapse training, a better way of doing it using a principled approach: one could train G (or D) multiple times in a row, if its loss becomes too big or too small; a simple but effective alternative is to use the same number of steps for G and D (usually one), so that the training process is more balanced. The latter will be used in the final GAN model, since it is less prone to errors and it is easier to manage during training.

Another element that caused poor performance is the sampling method used to extract data from the datasets and obtain the training batches for G and D, and the prediction batch for G. This sampling process was performed by shuffling the target dataset for each training step, and by selecting the first N elements. Using this method, there is a strong chance that some data is seen by G or D multiple times, while other data is seen far less (or never seen, for that epoch). A better approach is to shuffle the datasets for G and D at each epoch, not each training step, and by selecting, for each training step, a batch of data. This batch starts from the first element in the dataset that the model has not already seen in that particular training step. This guarantees that the training of G and D, and the predictions of G used by D, are executed on all the dataset, for each epoch, thus improving the overall training process.

By looking at the optimizers and the learning rates, it is possible to identify a third improve-

ment: in the original GAN architecture, there were two different learning rate for D and for the *combined* network. This is another source of imbalance in the training process, and can be easily fixed by using the same learning rate for both networks.
Activation functions are a key factor in propagating the gradients through the network. Both G and D employ a pre-activation mechanism, using the ReLU activation function before any Dense layer. Unfortunately, the ReLU activation function has sparse gradients, since it flatten all the negative values to zero, thus the stability of the GAN game suffers a lot. To overcome this, we re-design the G and the D, by replacing the ReLUs with Leaky ReLUs. In the Leaky ReLU, the derivative is 1 in the positive part, and is a small fraction in the negative part, thus improving the gradients flow. We can see the Leaky ReLU activation function in figure 7.11.

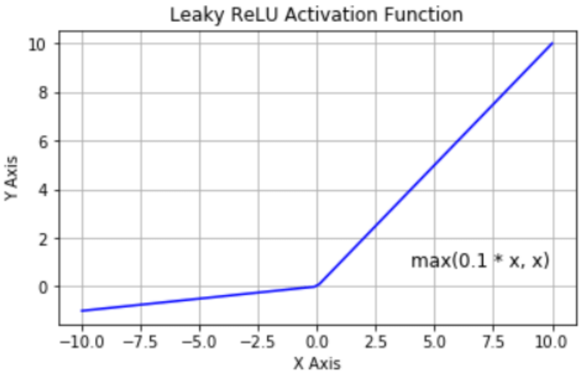


Figure 7.11: Leaky ReLU activation function

The result is that, as seen in figure 7.12, for each residual block in both G and D, we have a sequence of Dense layer, Leaky ReLU activation fucntion, and Dropout layer.

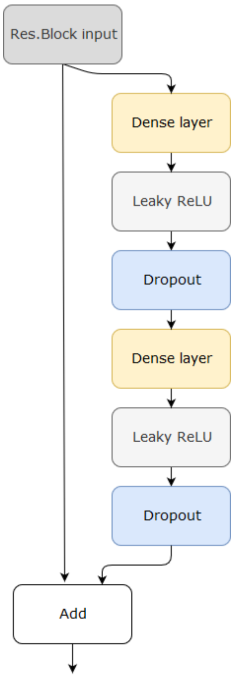


Figure 7.12: New residual block structure

Another trick to improve the training is to use soft labels when training D. Soft labels mean that given two target labels, such as 1 for *real* and 0 for *fake*, then for each incoming sample, if it is real, it the label is replaced with a random number between 0.8 and 1, and if it is a fake sample, it is replaced with a number between 0.0 and 0.2. This weakens a little the D, giving G the time to improve and challenge D. It is also possible to make the labels the noisy for the discriminator, by occasionally flipping the labels when training the discriminator (so labeling as *valid* fake samples, and as *fake* real samples). This technique was not very useful in this instance, so it was not used in the final GAN model.

One final modification of G and D, consists in reducing the number of residual blocks in both networks. By using only 2 residual blocks, instead of the original 50, the models become easier to train, and the overall performance slightly improves, since the vanishing/exploding gradients problem is much less significant in this instance.

After applying all these improvements on the original GAN, we get the final GAN, as seen in figure 7.13. This network is composed of a sequence of the new G, the Add function between G's assignments and the input (partial solutions), and the new D.

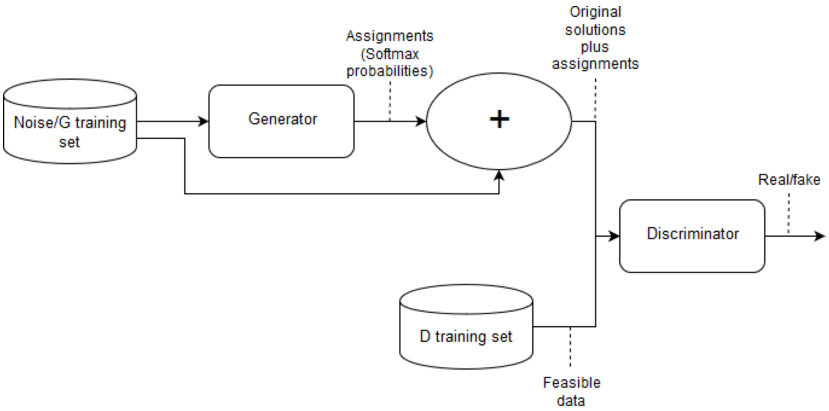


Figure 7.13: Final GAN structure

7.4.7 Experiments using pre-trained G & D

The first GAN evaluation tests, for the second family dataset, are performed using pre-trained G and D.

We start by training G on a certain dataset, such as the DS-G-C, and by testing it on the other dataset (in this case, DS-G-O). After the training process, we save G model, as already described in previous sections. This process is repeated after switching the datasets, so that the previous training set is the new test set, and vice-versa. Later on, the same procedure is applied on D.

In the end, we have four different models, two for G and two for D. Those which were trained on datasets DS-G-C and DS-D-C, and tested on DS-G-O and DS-D-O, will be called G-C and D-C, while the others G-O and D-O.

The GAN training data, for both G and D, is made of data from the test set employed during the pre-training phase; this means that, if G and D were pre-trained using the *C* group of datasets, the GAN training set will be composed of the *O* group of datasets, and vice-versa.

Using the improved GAN, describe in the previous sections, and a learning rate of 0.0002 for both the optimizers of D and the *combined* model, we perform the GAN evaluation.

For the GAN datasets, initially the *uniques* datasets variety are employed, while later the *multiple* one is used.

In the following tables and graphs, the training set is considered as the GAN training set, not the original training set, same thing for the test set.

All the training routines were stopped after 100 epochs, to have a better idea of the pros and cons the different configurations, and to be able to compare them equally. Early stopping was not employed, since we want to improve both G and D, so a single metric that highlights both models performances does not exist.

We can see the results using G-C and D-C and the *uniques* dataset in tables 7.6 and 7.7 respectively.
Both the G and the D improve their performances over the training set (the original test set), but they also become worse on the test set (the original training set). This is an expected behaviour, due to the fact that both networks specialize on the training set family, thus lowering their performances on the other dataset family. It is important to highlight that the feasibility ratio of G on the training set, while significantly improving, does not reach the same value of the stand-alone training process; the feasibility ratio on the test set is constantly decreasing, and at epoch 100 is slightly better than the pre-trained one on the original test set.
After training both G-C and D-C, we perform the same evaluations for G-O and D-O. In this instance, the training set is made of data coming from the *O* datasets, while the test set is composed of data sampled from the *C* datasets.
In tables 7.6 and 7.7, it is possible to verify that the final outcome is very similar to the previous one, so no plot for G feasibility ratio and D accuracy will be displayed for this particular case. Furthermore, since the original training and test set performance is better than the previous case, this one is less interesting and it will not be analysed in detail; for the same reason, the BL configuration will not be evaluated in the following sections either.

The previous performance analysis were achieved using the *uniques* dataset variety. GANs, and classic neural networks, work better when the dataset used are large, so it is interesting to evaluate the same GAN architecture made of pre-trained G-C and D-C, this time using the *multiple* datasets variety to train the network. As previously stated for this configuration, during the GAN training we use the *O* dataset group as the training set, and the *C* dataset group as test set.
The final results, after 100 epochs of training, are displayed in tables 7.6 and 7.7.
In these tables, as well as in the other experimental sections that employ pre-trained models, we show the baselines for the metric to evaluate. These baseline values represent the final values for the metric, obtained during the pre-training process; in other words, they are the starting metric values of the models employ in the GAN.
As expected, using the *multiple* dataset provides a better performance than using the *uniques* one. G feasibility ratio on the training set is higher, meaning that it improves faster than the previous case; the feasibility ratio on the test set is also higher, so G does not worsen its performance as significantly as before on the original training set. The accuracy of D is very similar to the *uniques* case, so it is possible that D reaches a stability around that value, thus maximising the knowledge exchange between D itself and G.

feasibility ratio G					
G model	DS type	train set baseline	test set baseline	train set	test set
G-C	uniques	0.35	1.00	0.80	0.51
G-O	uniques	0.45	1.00	0.86	0.49
G-C	multiple	0.35	1.00	0.90	0.60
G-O	multiple	0.45	1.00	0.91	0.60

Table 7.6: G performance before and after 100 epochs of GAN training

accuracy D					
D model	DS type	train set baseline	test set baseline	training set	test set
D-C	uniques	0.39	0.98	0.70	0.38
D-O	uniques	0.40	0.99	0.66	0.37
D-C	multiple	0.39	0.98	0.70	0.35
D-O	multiple	0.40	0.99	0.68	0.38

Table 7.7: D performance before and after 100 epochs of GAN training

By tracking the results of the evaluation intervals during training, it is possible to observe how the global feasibility ratio of G on the training set changes over time. As we can see in figure 7.14, the feasibility ratio of G, when using a *uniques* dataset variety, is steadily improving through the epochs, with more significant growth at the beginning of the training and some minor setback in later epochs.
In the same figure, we can see that, by using the *multiple* variety of datasets, the feasibility ratio grows faster than the previous case, but the difference between the two seems to become less and less relevant as the training goes on.

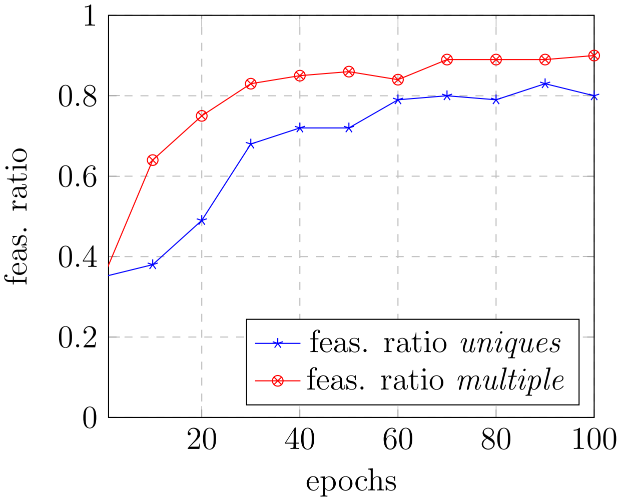


Figure 7.14: how G-C feasibility ratio (on the GAN training set) changes over time, using *uniques* and *multiple* variety of datasets

A more detailed analysis on the feasibility ratio can be done by looking at the proportion of feasible solutions generated by G, at epoch 100, for each type of solution, as seen in figure 7.15. As expected, G is able to produce a large proportion of feasible solutions when the starting one has fewer queens in place, while performance gets worse as the number of queens rises.
By comparing G trained with the *uniques* and *multiple* cases, we can see that the latter performs slightly better than the former when a small number of queens is present; when more queens are present, the feasibility ratio is very similar between the two, and the performance of both instances is still lacking a bit on solutions that have 7 existing queens.

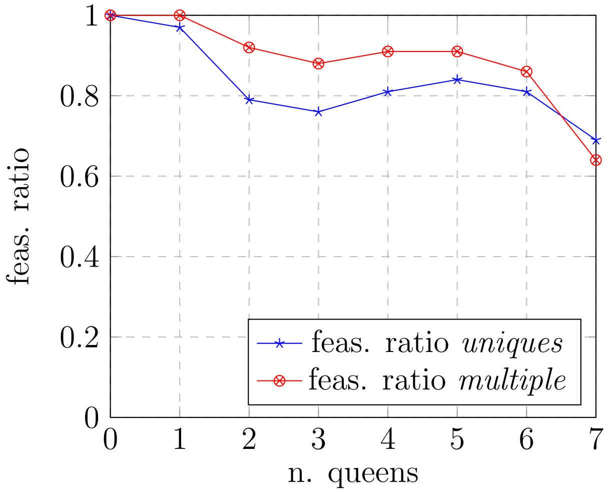


Figure 7.15: feasibility ratio of G-C for different types of solutions, at epoch 100

Another interesting metric to evaluate and track is the D accuracy on the training set, and how it evolves over time.
In figure 7.16 we can see that, as training goes on, D improves its accuracy on the GAN training set during the first epochs, but it soon stabilises itself to a certain value (in this instance, 0.70). This is true for both the *uniques* and the *multiple* variety of training sets, leading to believe that D's performance reaches a limit and stabilizes itself no matter what dataset is used.

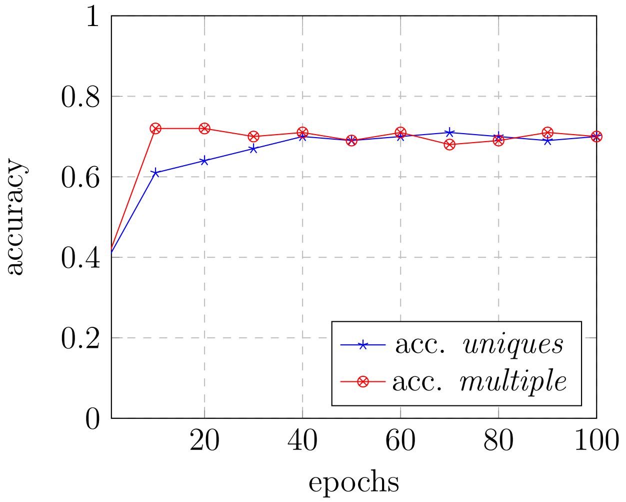


Figure 7.16: D-C accuracy progression during training

Finally, we can also examine the values of G and D losses, as they progress over different training epochs; for the *uniques* dataset variety, the losses are represented in figure 7.17. The losses are sampled every 5 epochs of training.

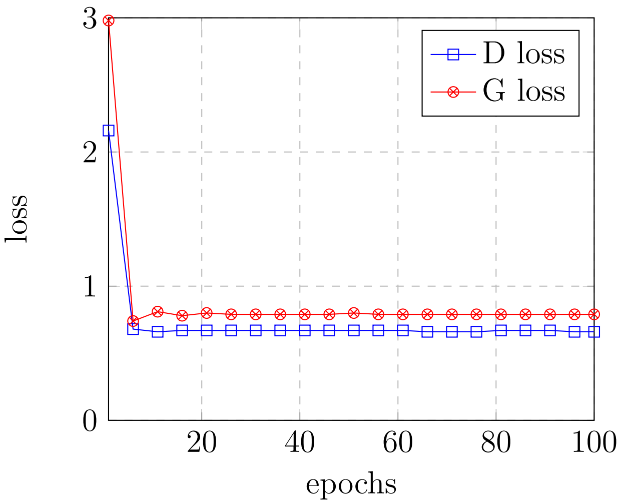


Figure 7.17: G-C loss and D-C loss progression over training epochs

As we can see, both G and D losses start at a high value, but they converge soon to a stable lower value. Both losses exhibit small alterations, showing that the GAN training is balanced, and both G and D are competing fairly.

For the *multiple* case, the losses profile are very similar to the *uniques* one as well, as shown in figure 7.18. After the first few epochs, both losses stabilize and stay almost the same during the remaining epochs of training.

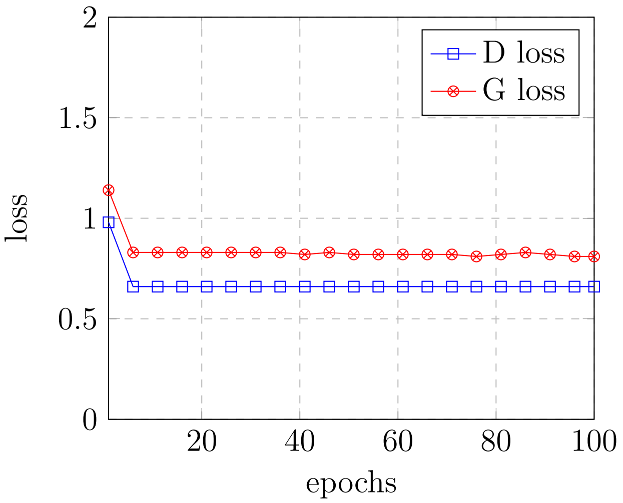


Figure 7.18: G-C loss and D-C loss progression over training epochs (using the *multiple* dataset)

7.4.8 Experiments using untrained G & D

In the previous section, we analysed the performance of the GAN, using pre-trained networks for both G and D. Both the *uniques* and *multiple* datasets family were used, as well as G and D pre-trained on the O dataset group (G-O, D-O), and G and D pre-trained on the C dataset group (G-C, D-C).
In this section, the performance analysis will be performed by using only the *multiple* dataset family, since this setup already proved to yield the best overall results. What changes here is that both G and D are not pre-trained, so they have different starting skills than before. O and C datasets will be used alternatively as the GAN training set. The idea here is to compare this

training process with the one using pre-trained models, and also to the stand alone training performed on G and D, to see if the performances are comparable with respect to quality and time needed to train the networks.

We start by creating the GAN, using both a G and a D made of 2 residual blocks; the G has a masking layer. The networks are configured as previously shown, to obtain a *combined* model made of G, Add layer and D, in sequence.
We employ the O multiple dataset as training set, and the C multiple dataset as test set. This is due to the fact that the O group of datasets contains more data.
The results, obtained after 100 epochs of training, can be seen in tables 7.8 and 7.9.
If we switch the datasets, thus using the C dataset group for training, and the O dataset group for testing, the results are fairly similar; due to this fact, no feasibility ratio, accuracy and losses plots will be shown. The final results using this dataset configuration can be seen in tables 7.8 and 7.9 as well.

feasibility ratio G					
G model	DS type	train set baseline	test set baseline	train set	test set
G-C	multiple	0.17	0.15	0.91	0.27
G-O	multiple	0.16	0.15	0.90	0.37

Table 7.8: G performance before and after 100 epochs of GAN training

accuracy D					
D model	DS type	train set baseline	test set baseline	training set	test set
D-C	multiple	0.49	0.42	0.68	0.41
D-O	multiple	0.47	0.54	0.70	0.41

Table 7.9: D performance before and after 100 epochs of GAN training

By comparing the final values of G's feasibility ratio with the previous experiment that used a pre-trained G, we can see that the feasibility ratio on the training set is basically the same, even if there is a significant difference on the starting values (0.16 instead of 0.35).
The feasibility ratio on the test set is quite different, but while here its value grows over time, in the previous evaluation the G started from a perfect feasibility ratio (1.00) and declined over the training epochs.
In the stand alone G training, using the O dataset group as the training set and the C one as the test set, the feasibility ratios were 1.00 and 0.46 respectively on the training and test set, so it is possible that those are close to the maximum reachable values using this GAN configuration.
Once again, we plot the measured values of the feasibility ratio of G, and of the accuracy of D, during the training process (figure 7.19, left and right graph respectively).

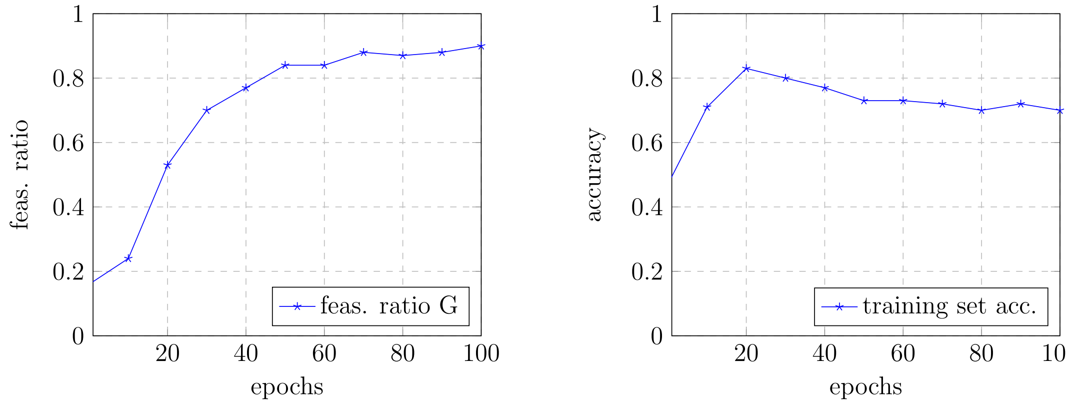


Figure 7.19: how G feasibility ratio changes over time, using untrained G and *multiple* ds (left); D accuracy progression during training, using untrained D and *multiple* ds (right)

These figures show that while G is constantly improving, albeit more slowly after 50 epochs, the D's accuracy reaches its maximum value around epoch 20, then gently declines over time. During this decline, the G's feasibility ratio is still growing, so D is *teaching* correctly to G, and the knowledge is being passed correctly between the two.

If we plot the feasibility ratios (figure 7.20) for each type of solution, we can see a similar pattern as before: G's performance is very good on solutions that have fewer queens, but gets worse as the number of queens already in place increases.

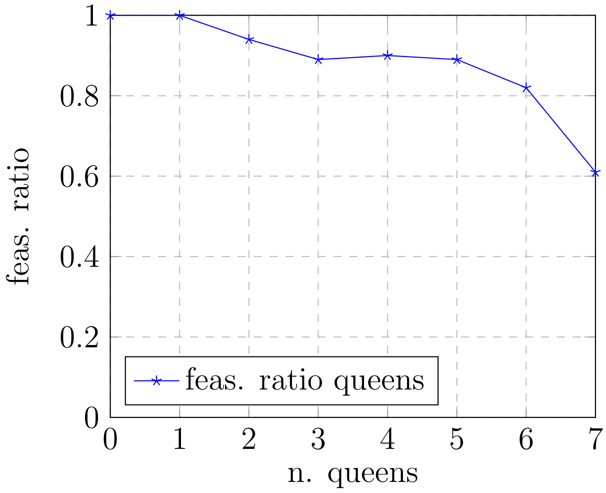


Figure 7.20: feasibility ratio for different types of solutions at epoch 100 (untrained G, *multiple* ds)

Finally, it is useful to plot how G and D losses evolve during the GAN training (figure 7.21).

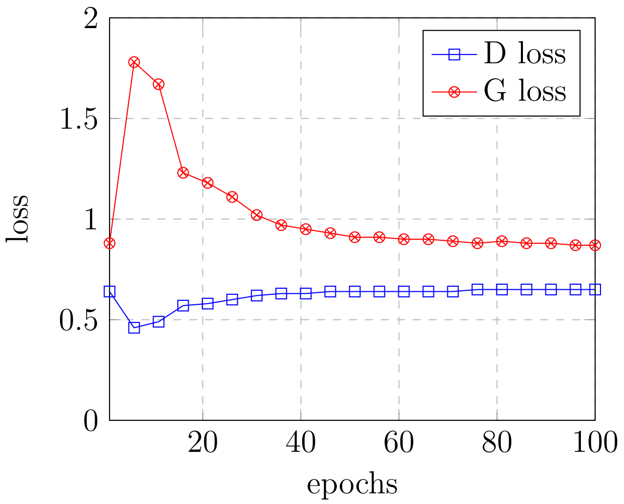


Figure 7.21: G loss and D loss progression over training epochs (using the *multiple* dataset)

The loss evolution for untrained G and D is a little different than the pre-trained case: indeed, in the previous one the losses stabilized almost immediately, while in this instance the G and D losses progressed initially in different direction, G's loss increasing and D's loss decreasing, and later converging to a more stable value.

7.4.9 Experiments using noise as G's input

Traditional GANs, as already stated, work by using a particular dataset as the D's input, and by providing noise as G's input. Since the n-Queens Completion problem is represented using a model agnostic technique, all data is represented as binary data. Due to this fact, it is almost impossible to use data generated from noise for this particular instance. In table 7.10 we can observe how the global feasibility ratio changes as the probability of success of the binomial distribution increases.

prob. success binomial	feas. ratio ds
0.02	0.76
0.04	0.41
0.05	0.29
0.06	0.19

Table 7.10: global feasibility ratios of datasets produced with different binomial noise

The reason why this type of noise can not be used in this instance, is that if we try to generate data by sampling from a random binomial distribution having the number of trials n set to 1, it is possible to observe that sampled solutions that have less than 3 queens have an high value of feasibility, while solutions with more than 3 queens are all, or nearly all, unfeasible. Since no solution is feasible from the start, G would be unable to learn anything, because any solution generated from the desired assignment would be immediately unfeasible itself.

A possible workaround to this problem, is to generate a dataset for G made of data sampled from both noise distribution and a feasible dataset. The goal is to have a final dataset that has good feasibility ratio itself, even at a single-queen level. Initially G will output a small amount of feasible solutions, but it has a margin of improvement on all types of solutions. Moreover, D starts by receiving less feasible data as *fake*, so it should be able to learn better. These are the steps needed to compute the final G dataset, when using a noise source:

1. The first thing to do is to load the feasible dataset, that will complement the noise. We then create a data structure that collects all the solutions from the selected dataset, divided for each possible queens number

2. We iterate between 1 and 8, representing the number of queens to evaluate in each round, to generate the noise-based solutions. We exclude the 0 case, since every assignment produced would be feasible anyway. Subsequently, a certain amount of noise is generated by creating a fixed number of solutions: each solution is produced by inserting the desired number of 1s in a 0s array, then randomly shuffling the solution itself. Since the amount of generated data is customize, we set a lower limit for instances having one or two queens, so that the global feasibility ratio of noise lowers and it is more balanced

3. This step consists in mixing feasible data with previously created noise, so that feasibility is almost equal for the different solution types (number of queens). In order to implement this, it is necessary to iterate between 3 and 8, representing the number of queens to evaluate in each round; there is no need to add solutions with less than 3 queens to the noise, since they already have a very high feasibility ratio. Then, for each solution type, a certain amount of data is sampled from the collection containing feasible solutions (generated during step 1), and added to the final dataset. For solutions having six or seven queens, we sample less element from the feasible collection; this is due to the fact that there is a smaller number of these particular solution types, so we don't want to use them all or run out of them, especially if we use a pre-trained G

After generating the dataset for G, it is necessary to train the GAN, as seen in previous sections. To execute this experiment, we will employ an untrained G network, as well as an untrained D.

We use both DS-G-C and DS-G-O as the source for feasible solution to mix with noise, alternatively. The results regarding G's feasibility ratio, and D's accuracy, obtained after 100 epochs of training, are displayed in tables 7.11 and 7.12 respectively.

feasibility ratio G					
G model	DS type	train set baseline	test set baseline	train set	test set
G-C	multiple	0.08	0.09	0.62	0.24
G-O	multiple	0.09	0.09	0.66	0.25

Table 7.11: G performance before and after 100 epochs of GAN training (using noisy datasets)

accuracy D					
D model	DS type	train set baseline	test set baseline	training set	test set
D-C	multiple	0.43	0.45	0.89	0.41
D-O	multiple	0.41	0.43	0.90	0.40

Table 7.12: D performance before and after 100 epochs of GAN training (using noisy datasets)

By comparing them to the ones in previous sections, it is possible to see that G does not improve as much as before, though a little drop-off is to be expected using noise. It is important to state that the feasibility ratio of G at epoch 100 is still on the rise, and by training the GAN for several more epochs it reaches a maximum value around 0.90, so the performance is comparable to the ones seen in previous sections, albeit the training is overall slower.

On the other hand, D's performance is much better, leading to believe that maybe D is becoming too good in little time, leaving G less room for improvement. Indeed, as seen in plot 7.22, the loss value of D decreases steadily in earlier epochs, and when it stabilizes it has already a lower value than the previous instances; G's loss builds up over time, especially at the start of the training, and it is bigger than the one examined in previous sections.
Since the performance using the C dataset group is very similar the the other case, we report only the feasibility ratio of G, and the accuracy of D after 100 epochs, as seen in tables 7.11 and 7.12 respectively.

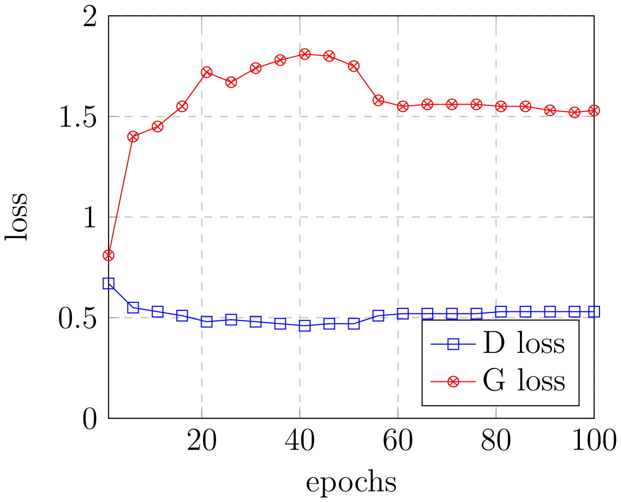


Figure 7.22: G-O loss and D-O loss progression over training epochs (using the *multiple* dataset)

Finally, in figure 7.23 it is possible to see how the global feasibility ratio on the noisy training set changes over time during training; this value is significantly lower than the one displayed in previous experiments, because it is an evaluation on assignments generated by using noisy solutions, so only a portion of the G input data can generate feasible solutions. In figure 7.23 we can also see the feasibility ratios for each possible solution type at epoch 100 of training; the evaluation is performed on the full non-noisy training set (O dataset group, in this instance).

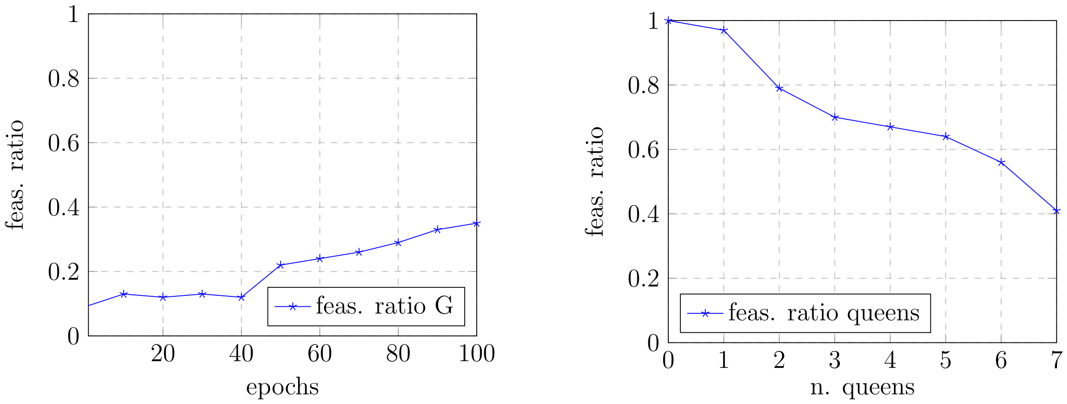


Figure 7.23: how G-O feasibility ratio changes over time on noisy training set (left); feasibility ratio for different types of solutions at epoch 100, using partial solutions from the non-noisy training set (right)

7.4.10 Experiments using G without masking

All the previous evaluations of the GAN were performed using a G having a masking layer. In the gradients analysis section, we stated that no significant advantage had been detected by using a G without the masking layer, since the gradient values were very similar to the masking-G case.
Even knowing this, it can be interesting to evaluate the GAN performance when G does not have the masking layer, so in this section we will perform the same training evaluations seen earlier, using this particular configuration. The training set used are the ones of the O group and of the *multiple* variety, and the test sets come from the C group, *multiple* variety.
It is important to recall that the GAN model is composed of a sequence of G, Add layer between G's assignments and the partial solutions, and D. If the masking layer is absent, then G has a lower baseline for its feasibility ratio. Furthermore, by performing the addition, it is possible that G decides to place a queen in a position where an existing queen is already present. To properly identify such behaviour during training, we define a threshold value of 1.1; this allows us to count the occurrences in which the solutions, resulting from the addition layer, have illegal assignments. This is due to the fact that the G produces an assignment tensor through the Softmax layer, so the sum of its elements will always be 1, thus if after applying the assignments the new solution has values greater than 1, it means that G has computed a value greater than 0 for an assignment in a position with an already existing queen.

We train the GAN using the same mechanism as before, but we also keep track of the predictions produced by G and used as fake data for the D, thus enabling us to monitor how well G is learning to emulate the real data. A new metric is defined, called *illegal assignments ratio*, that measures the proportion of illegal solutions (exceeding the threshold) produced by G, for every epoch of training. The evaluation is performed using both pre-trained and untrained G and D.

If we only look to the evolution of the solutions produced by G, it is possible to identify three distinct phases:

1. Phase 1: it starts at the beginning of the training process, and it lasts for about five epochs. During this period, the queens in the original partial solutions (1s) are slightly changed to bigger values (such as 1.01, 1.02) in G's produced solutions, while the other values are all small ones, fairly equal to one another. The G does not execute any assignment;

2. Phase 2: it lasts for several epochs, usually about twenty epochs. In this phase, G is able to reproduce the original queens (1s) as such, without any alteration. The remaining values in each solution have more variance than the ones in the previous phase, but the final outcome is still not a binary solution. Indeed, for each solution it is possible to identify the desired assignment, that usually have a value between 0.5 and 0.8, while the other values are much smaller;

3. Phase 3: it continues until the end of the training. In this period, G is now able to produce almost binary solutions: the original queens (1s) are replicated perfectly,and the assignments consist of one value set to 1, or very close to it, while the others are set to 0, or values slightly above 0.

Using a mock 4 × 4 chessboard, as seen in figure 7.24, it is possible to represent a binary partial solution that it is passed to G as input during phase 3 (left), and the respective solution in output from the *full* G network (G and Add operation); as we can see, all the original queens are replicated as 1s, and the desired assignment is applied in position (1,3) with a value very close to 1 (0.99); all the other free positions have a corresponding value that is 0 or very close to it.

4	0	0	1	0
3	0	0	0	0
2	0	0	0	1
1	0	1	0	0
	1	2	3	4

4	$2e-23$	$3e-20$	1	0
3	0.99	0	$8e-23$	$4e-22$
2	$1e-20$	0	$7e-23$	1
1	0	1	$3e-23$	$2e-22$
	1	2	3	4

Figure 7.24: original solution (left) and solution produced by G without masking (right), during phase 3; mock 4 × 4 chessboard

Given the behaviour of G during the different phases, it is clear that G is learning to emulate the data distribution of D's input.
To properly analyse the feasibility ratio of G without masking, a small change is needed in the evaluation function: what is missing is a comparison between the number of queens before and after the addition of an assignment to the original solution; using this mechanism, we can easily identify which assignment is placed in an illegal position (basically where a queen was already in place before the assignment). Illegal assignments are counted as unfeasible ones.

In the following paragraph we will examine two different GAN training instances: one employs an untrained version of both G and D, while the other uses G and D both pre-trained; in both instances, G has no masking layer.
For the pre-training instance, the analysis starts by training G on a given dataset. For our experiments, we will use the datasets of the C group, in their *multiple* variety, as we already seen in previous sections. Given the fact that no masking is present, G's performance on the original test set (that will become the GAN training set) is worse than the one observed when the masking layer exists, as expected. D is trained as well, using the same training set as G.
In tables 7.13 and 7.14, we can see how G and D perform after 100 epochs of training, for both pre-trained and untrained instances.
By comparing these results with the ones of the masking case, it is clear that both are very similar, and that G is able to reach a high feasibility ratio even without the masking layer.

feasibility ratio G					
G model	pre-trained	train. set baseline	test set baseline	train. set	test set
G-C	no	0.08	0.08	0.84	0.32
G-O	no	0.09	0.08	0.86	0.31
G-C	yes	0.26	1.00	0.86	0.51
G-O	yes	0.09	0.09	0.85	0.53

Table 7.13: G performance before and after 100 epochs of GAN training (using noisy datasets)

accuracy D					
D model	pre-trained	train. set baseline	test set baseline	train. set	test set
D-C	no	0.51	0.48	0.61	0.37
D-O	no	0.52	0.46	0.65	0.39
D-C	yes	0.39	0.98	0.68	0.36
D-O	yes	0.40	0.99	0.69	0.39

Table 7.14: D performance before and after 100 epochs of GAN training (using noisy datasets)

In figure 7.25 we can see how the illegal solutions percentage, a metric defining the proportion of G's assignments placed on a position where a queen is already present, changes over time, for both untrained and pre-trained G instances.
If we look at the untrained G feasibility ratio, we can see that it is growing in the initial epochs, while constantly decaying as the training process goes on. This is another evidence of G learning how to properly execute assignments, in order to avoid placing a queen in a position where another queen is already in place.
When evaluating a pre-trained G feasibility ratio, we can observe that this ratio is significantly lower than the previous case, and that's to be expected since G is pre-trained and starts from an higher feasibility ratio value than before.

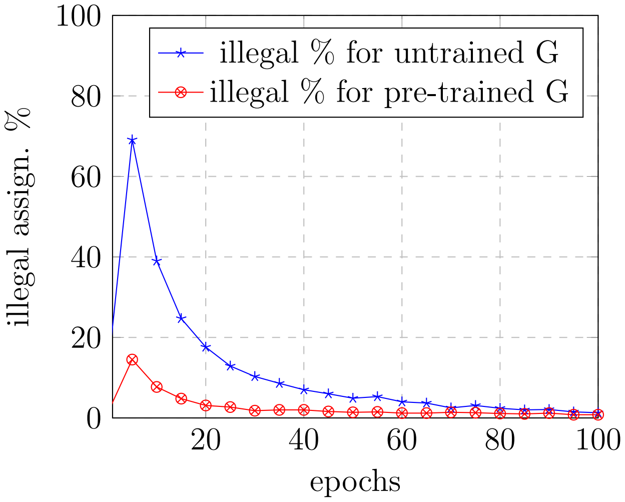


Figure 7.25: how G illegal assignments percentage changes over time (using pre-trained and untrained G without masking, using *C* dataset group of the *multiple* variety)

To properly compare how G is learning with and without masking, we plot how G feasibility ratio fares over time, in figure 7.26.
When looking at the untrained G instance, we can see that G exhibits a similar behaviour than the one observed in the previous experiment, but is seems to reach an upper limit around 0.90; after that, the feasibility ratio slowly decreases instead of growing.
No significant changes can be observed for the pre-trained G instance as well, if we compare these results with the ones obtained with a *masked* G.

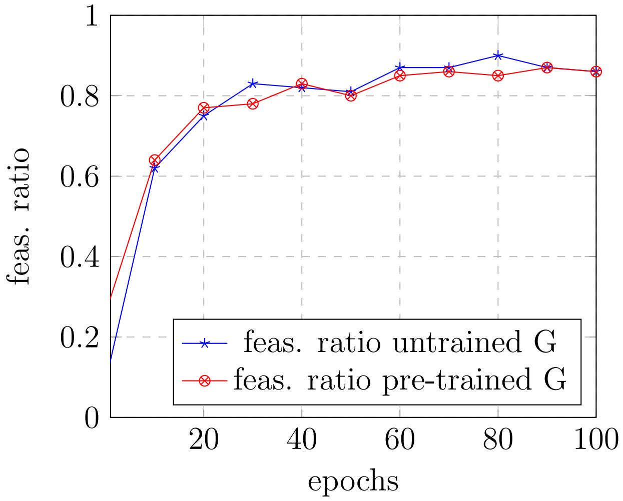


Figure 7.26: how G feasibility ratio changes over time (using pre-trained and untrained G without masking; using C dataset group of the *multiple* variety)

Finally, in figure 7.27, it is possible to see how D's accuracy changes during training. The accuracy's profile is very similar to the previous ones, with the accuracy value growing significantly in the earlier epochs of training, while slowly decreasing its value over time. No significant differences can be found between the untrained and the pre-trained instances.

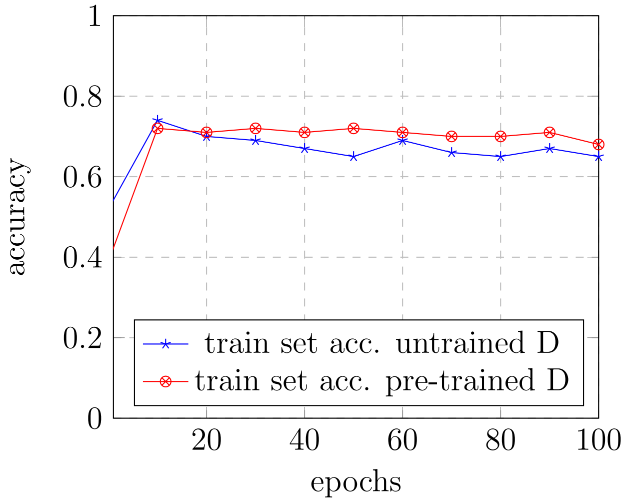


Figure 7.27: D accuracy progression during training (using pre-trained and untrained D; using C dataset group of the *multiple* variety)

Ultimately, both GANs configurations produce almost the same results, even if one features G and D pre-trained and the other not. This leads to believe that there is a sort of upper limit in the learning process of G and D, and that the starting point doesn't matter significantly, impacting only the amount of time needed during training.

7.5 Analysis & Observations

7.5.1 Generator

In the G network's experiments, we saw that G exhibited a slightly different behaviour than the one seen in the previous chapter. This is an expected outcome, and it is due to the fact that the second family of datasets is made of two datasets with different characteristics. This leads to a overall worse G's performance, both in terms of accuracy (follow the teachings of a master) and in terms of feasibility ratio of produced assignments.
The difference between using DS-G-O or DS-G-C as the training set, in terms of accuracy, is negligible, with a training set accuracy of 0.41 and 0.43 respectively, and an identical test set accuracy of 0.08. We can see a slight discrepancy in the feasibility ratio metric, in favor of the first one: DS-G-O has a test set feasibility ratio of 0.38, while DS-G-C of 0.31. Again, this is expected, since the dataset without queen in a central position (DS-G-O) is made of a lot more solutions than the other one, so, by employing it during training, the G is trained with more examples.

7.5.2 Discriminator

As we saw in the experimental section, the results obtained differ from the ones seen in the previous chapter: indeed, using the first dataset family, the D reached high accuracy values both on training and test set; on the other hand, when using the second dataset family, the test set accuracy exhibited is significantly lower. This means that, in this instance, the D has a lot of room to improve its performance, and it makes more sense to employ it in a GAN network.
Given the similarities of the results using the two possible training/test set combination, no default training and test set are selected, leaving both the possibilities as valid, so in the GAN section we experimented using one or the other set alternatively.

7.5.3 GAN

In the GAN experimental sections, we performed several evaluations of the architecture to better explore its ability and its potential to improve.
Initially we tried to train G and D using a standard GAN implementation, but it did not perform as well as expected. Next, we looked at the first GAN implementation, analyzed what worked and what was hindering the model's performance, and successfully tweaked some key design elements, such as the activation functions of G and D, and the intermediate merging layer, to improve the network's effectiveness.
The new GAN architecture was then employed to perform several experiments, with some variations between each other.

During the experiment using a pre-trained version of G and D, we saw that the G improved its ability to perform feasible assignments on the GAN training set by a significant margin, especially in the initial phase of the training process. On the other hand, G's performance on the GAN test set (the original training set) was reduced, since the network adapted its weights to work well with the new training data distribution.
In the same experiment, we saw that D's accuracy improved during the earlier phases of training, and stabilized itself later on.
We experimented using both the *uniques* and the *multiple* variety of datasets, and using the latter exhibited a slightly better training for both G and D.

In the second GAN experiment, we employed untrained version of both G and D, using the *multiple* dataset variety. In this instance, it was possible to observe a significant improvement

for G, in terms of feasibility ratios of produced assignments. The overall result obtained on the GAN training set was comparable to the one in the pre-training case, even if the baseline was much lower.
D improved a little more than the pre-trained case, during the initial training phases, while it stabilized itself on comparable accuracy values later on.

In the third experiment, we tried to use a mix of binomial noise and G datasets as the training set for G, trying to imitate a more classic GAN training process.
In this instance, D improved by a wide margin in terms of accuracy on the training set, while G's improvement, while evident, did not match the ones seen in previous experiments. This can be explained by the fact that G can not be able to produce feasible assignments when receiving as input unfeasible noise solutions, and it has not a feedback mechanism that informs it about this fact. On the other hand, D now receives a much higher ratio of unfeasible solutions as *fake* data, thus becoming much better in distinguishing feasible and unfeasible solutions on the training set.

Finally, we experimented using a GAN composed of an untrained G that was missing the masking layer. Even using this configuration, we saw a significant improvement in the feasibility ratios of G's assignments, yielding comparable results to the first and second experiments.
We measured the percentage of solutions that violated at least a constraint, verifying that indeed the G was learning to have a better understanding of the problem in hand, yielding less and less illegal assignments as the training progressed.
We also performed an evaluation using pre-trained G and D, observing marginal differences with the previous case.

Overall, we saw that G and D are able to improve in a GAN model. The margin of improvement depends on the type of training data, less on the untrained or pre-trained condition, or the presence of a masking layer for G.
Using pre-trained G and D, they became better on the new training data, but they also worsened their performance on the original training set.
G's performance, while improving, seemed to reach a sort of limit in several experiments. This could be due to the fact that the datasets are made of discrete data, so, as G improves, D receives more and more feasible solutions as *fake*, thus being unable to become better and limiting further G's improvements.

Chapter 8

Experiments with 3rd dataset family

8.1 Goal

In this chapter, we will perform several experiments using the third family of datasets.
During the datasets creation, we introduced a bias, by selecting and eliminating the solutions that contained queens in a central position on the board.
For G, D and GAN, we will perform experiments similar to the ones seen in previous chapter. We are interested in measuring how these models perform when the training set contain a strong bias.
We will evaluate also a variety of the GAN model, called WGAN, in the hope of seeing an improvement in terms of a faster and steadier training process.
For comparison purposes, the same experiments will be performed on a generator and a discriminator implemented as Support Vector Machines. The idea is to see if a typical, and less sophisticated machine learning technique, is able to yield good performances as well.

We will also introduce a new process, that allows us to assess G's ability in creating full solutions, when starting from empty ones. We call this process *stochastic generation.*
The main goal of this chapter is to evaluate the ability of G in producing solutions that are valid with respect to the constraints determined by the problem in hand, and by the bias inserted into the datasets; this means not only checking the solution's validity for the standard criteria of the 8-queens completion problem (row, column, diagonal constraints), but also checking whether G's solutions do not have any queens in a central position.

8.2 Generator

8.2.1 Stochastic Generation

In this chapter, we introduce and employ a data generation process, called *stochastic generation*; the results obtained using this process, allow us to evaluate how G fares after being trained using different techniques.
The main goal of the Stochastic generation process is to generate new complete solutions for a problem, the 8-queens problem in this instance. We use the probabilistic output of the G network to simulate the generation of complete solutions, starting from empty ones.
This means that each full solution is constructed by performing one assignment after another, until the solution is complete; for instance, in the 8-queens problem a solution is considered complete when it contains 8 queens. Each desired assignment is chosen using the probabilities that G computes for that specific partial solution.
This probabilistic selection helps in improving solutions variability, otherwise, by choosing always the assignment with the highest probability, we would obtain the same solution over and

over again.

8.2.2 Criteria evaluation

Once a certain amount of complete solution has been produced, using the Stochastic generation process, it is necessary to perform a check on different kind of constraints. This allows us to analyze the characteristics of the full solutions produced by a G, during the multiple experiments performed in this chapter.
Three different kind of criteria are evaluated:

1. First Criterion: each solution must satisfy the three hard constraints of the n-Queens Completion problem, so no solution can violate row, column or diagonal constraints
2. Second Criterion: each solution must satisfy row, column, and diagonal constraints, and also the *no queens in central position* one. This last criterion is enforced by the so called *bias* of the training set, since no dataset used for training purposes has been generated by base solutions with queens in a central position
3. Third Criterion: each solution must satisfy only the *no queens in central position* criterion, so it should not have a queen in a position represented by the central 2×2 square in the chessboard

The first and second criteria are evaluated in the same way: a comparison between each generated solution, and each solution contained in a certain dataset. For the first criterion, the dataset is the one containing the full 92 solutions, while for the second criterion, the target dataset contains the expanded solutions obtained from the 8 base ones that do not have any queen in a central position.
To evaluate the third criterion, we simply check whether it contains a queen in one of the four central positions.

8.2.3 Design & Hyper-parameters

The G used in this chapter is basically the same G obtained when trying to improve the original GAN performance, as seen in the previous chapter.
G has been implemented as a Residual Network, and by employing the LeakyReLU activation function instead of the ReLU. G made of only 2 residual blocks.
The attention mechanism has not been activated, while the masking layer is present.
The final layer has a Softmax activation function, and the outputs represent the preferences in performing each assignment.
The learning rate used for these experiments is 0.0008.
Dropout has been employed after every Leaky ReLU Activation function, with a probability of 0.1.
An early stopping mechanism is used, with a patience of 5 over the validation set accuracy.
The loss function used is the Categorical Crossentropy.

8.2.4 Experimental section

The training of G is performed by employing DS-G-A as the training, and DS-G-B as the test set, then switching them for the following training experiment.
As we can see in table 8.1, no significant differences can be found between the first and the second evaluation; G's global feasibility ratio is slightly higher using DS-G-A as the training

set, since it is marginally bigger than DS-G-B, but overall the two assessments yield similar results.

Both in terms of accuracy and feasibility ratio, we can see that these results are more similar to the ones obtained using the first dataset family, thus resulting better than the ones obtained using the second dataset family. This is expected, since DS-G-A and DS-G-B are more similar than DS-G-O and DS-G-C.

train set	test set	test set acc.	train set feas. ratio	test set feas. ratio
A	B	0.15	1.00	0.60
B	A	0.14	1.00	0.55

Table 8.1: G's performance using datasets A and B

If we plot the feasibility ratios of different Generators for different solution types, we can see that the G analyzed in this section performs very similarly to the one seen in the first experiment. The comparison can be seen in figure 8.1.

As we already stated, this behaviour is expected, and it is due more to the training data characteristics than the G network architecture.

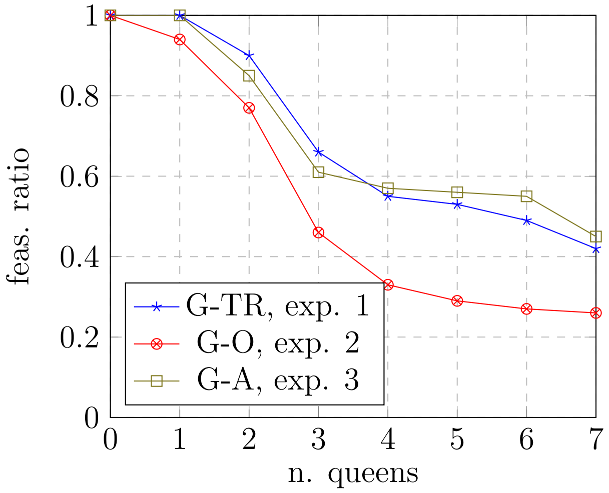


Figure 8.1: G feasibility ratio on the test set for different solution types, comparing the three different experiments

8.3 Discriminator

8.3.1 Design & Hyper-parameters

The D used in this chapter is very similar to the one designed and employed for the previous GAN experiments.

D is a Residual Network, made of 2 residual blocks. The Leaky ReLU activation function has been employed, instead of the ReLU one.

The number of neurons in a single residual block is of 500 units for the first Dense layer, and of 200 units for the second one.

Dropout has been employed after each activation function, with a probability of 0.1.

The optimizer used is Adam, with a learning rate of 0.0008.

An early stopping mechanism has been implemented, with a patience of 5 on the validation accuracy.

The output layer has a Sigmoid function as the activation function, and the loss function used is the Binary Crossentropy one.
No attention mechanism has been employed.

8.3.2 Experimental section

In this experiment, the D is trained multiple times by employing the DS-D-A dataset as the training set, and the DS-D-B dataset as the test set, then switching them for the following evaluation. The validation data is obtained from the training set, sampling randomly $\frac{1}{3}$ of the training data.
The best training accuracy reached is 0.99 after 36 epochs of training, with a test set accuracy of 0.49. By training the same D network, but with the DS-D-B dataset as the training set, and the DS-D-A dataset as the test set, we get similar results, with a maximum training set accuracy of 0.98 after only 15 epochs of training, and a test set accuracy of about 0.48. These results can be seen in table 8.2.

train set	test set	train set acc.	test set acc.
A	B	0.99	0.49
B	A	0.99	0.48

Table 8.2: D's performance using different training sets

If we compare these results with the ones obtained in previous experiments, we can see that, in this instance, the accuracy is higher than the one exhibited by D when using the second dataset family, but it is lower than the one obtained during the first experiment.

8.3.3 GAN

8.3.4 GAN Design & Hyper-parameters

The GAN model used in this section is the same that has been used in the previous experiments.
We will train G and D one time for each batch, so that the training process is more balanced.
The batch data is derived from the training sets, in a way that, for each epoch, all the training set data passes through the networks.
We employ the same learning rate for both networks, 0.0002, using Adam optimizer.
Both G and D employ a pre-activation mechanism, using the Leaky ReLU activation function before any Dense layer.
D has 2 residual block, and each residual block is made of two sequences of Dense layer, Leaky ReLU activation fucntion, and Dropout layer.
We also use soft labels when training D, meaning that, given two target labels, such as 1 for *real* and 0 for *fake*, then for each incoming sample, if it is real, it the label is replaced with a random number between 0.8 and 1, and if it is a fake sample, it is replaced with a number between 0.0 and 0.2.
The G used in these experiments has a masking layer before the Softmax activation function.

8.3.5 WGAN Design & Hyper-parameters

WGANs are employed to yield a better variability in solutions generated by G.
When training using a WGAN, it is important to remember that there is not a *standard* D; instead, a Critic (C) is employed.
The C has the same structure of D, but it doesn't have a Sigmoid activation function as its

last element. The C ends with a Dense layer, having size 1, without any activation function, because it is necessary that its output does not represent a probability. Instead, the output should be as large and negative as possible for generated inputs and as large and positive as possible for real inputs.
The WGAN used for the experiments in this chapter, is the improved one, the WGAN with gradient penalty (WGAN GP). As already stated, this type of WGAN do not use a weight clipping technique, instead it has a term in the loss function which penalizes the network if its gradient norm moves away from 1.
The C used in this instance is made of 4 residual blocks, so it is slightly bigger than the D used in the standard GAN.
The optimizer used is the Adam optimizer, with a learning rate of 0.0001, since WGANs tend to perform better with a lower learning rate [44].
The C is trained 5 times for every G's update.
The gradient penalty weight is set to 10, as seen in the WGAN GP article by Gulrajani et al. [45].

8.3.6 Experiments using pre-trained G & D

In this section, we perform a similar experiment than the one seen in the previous chapter.
For pre-training purposes, we employ datasets from the *A* group as training sets, and ones from the *B* group as test sets, both in their *uniques* variety. Then we use a portion of *B* group datasets as the GAN training sets, and the remaining data as the GAN test sets. We also perform the experiment with using the opposite dataset configuration.
The feasibility ratio of G, and the accuracy of D, obtained after 100 epochs of training, can be seen in tables 8.3 and 8.4. The final outcome is very similar to the one observed in the pre-trained GAN experiments with the second dataset family: we can see that G and D become more capable on the new training set, while their performance worsen on the original one.

feasibility ratio G				
train set	train set baseline	test set baseline	training set	test set
A	0.60	0.99	0.90	0.58
B	0.55	0.98	0.88	0.59

Table 8.3: pre-trained G performance before and after 100 epochs of GAN training

accuracy D				
train set	train set baseline	test set baseline	training set	test set
A	0.49	0.99	0.68	0.40
B	0.48	0.99	0.66	0.40

Table 8.4: pre-trained D performance before and after 100 epochs of GAN training

We further explore the details for the training when using *A* dataset group; in figure 8.2 we plot the measured values of the feasibility ratio of G, and of the accuracy of D, as they evolve during the training process (left and right plot, respectively).

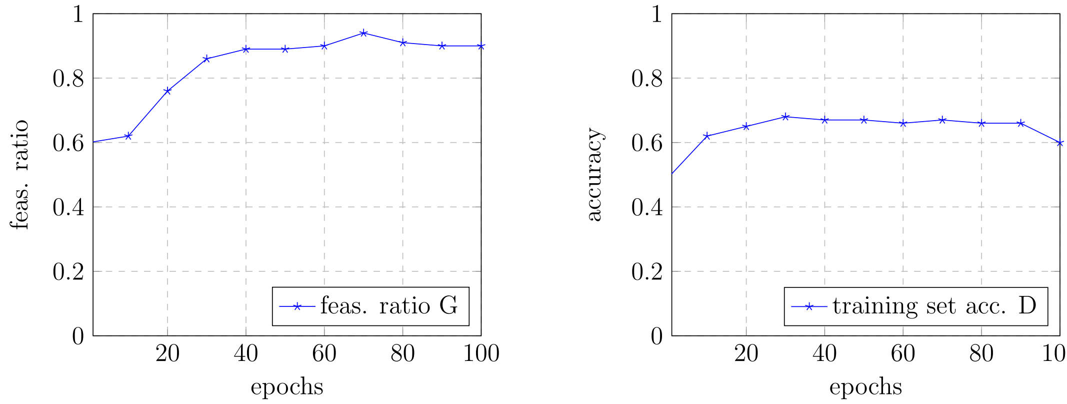


Figure 8.2: feas. ratio of G on the training set (left) and accuracy of D on the training set (right), after 100 epochs of GAN training

In figure 8.3 it is possible to observe how the feasibility ratio changes for each type of solution; we can see a slightly different pattern than before, since the feasibility ratio, instead of constantly decreasing as the number of queens already positioned increase, declines for solutions with two queens already in place, but then starts growing again until the last solution type (seven queens).

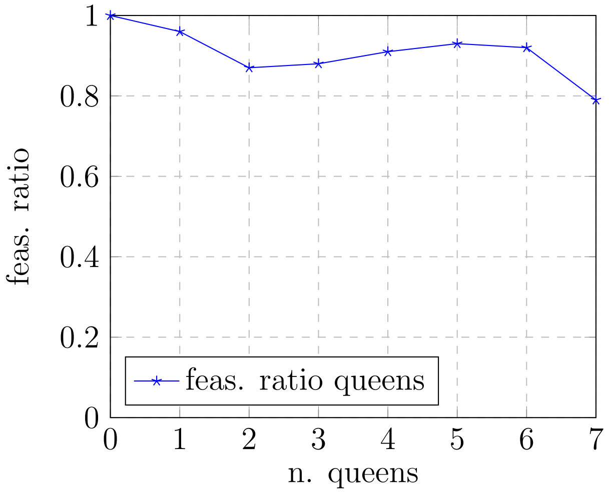


Figure 8.3: feasibility ratio for different types of solutions at epoch 100 (pre-trained G, A unique ds)

Finally, in figure 8.4), it is possible to observe how G and D losses evolve during the GAN training, when using A as the training set. The loss pattern is very similar to the ones seen in previous experiments.

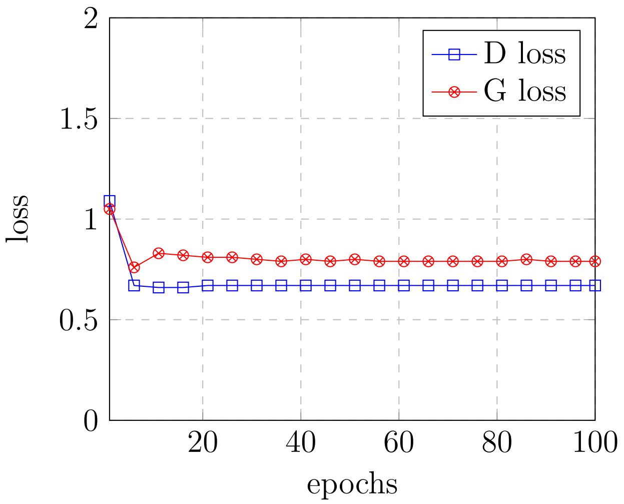


Figure 8.4: G loss and D loss progression over training epochs (using the A unique dataset)

8.3.7 Experiments using untrained G & D

We perform another similar experiment to the one shown in the previous chapter, by employing untrained versions of both G and D, and working with the third dataset family.
The results, obtained after 100 epochs of training, can be seen in tables 8.5 and 8.6. Even in this instance, the final outcome is very similar to the one observed in the corresponding experiment when using the second dataset family; we see a slower learning development during the initial part of the training, and a faster one during the second half of the training process. The test set feasibility ratio is slightly higher than the previous experiment, and this is an expected outcome since the training and test sets are more similar than before.

feasibility ratio G				
train set	train set baseline	test set baseline	training set	test set
A	0.17	0.17	0.85	0.41
B	0.20	0.16	0.86	0.40

Table 8.5: untrained G performance before and after 100 epochs of GAN training (*unique* ds)

accuracy D				
train set	train set baseline	test set baseline	training set	test set
A	0.56	0.57	0.77	0.45
B	0.44	0.48	0.77	0.44

Table 8.6: untrained D performance before and after 100 epochs of GAN training (*unique* ds)

As seen before, in figure 8.5 we plot the measured values of the feasibility ratio of G, and of the accuracy of D, during the training process with A dataset group (left and right side, respectively).

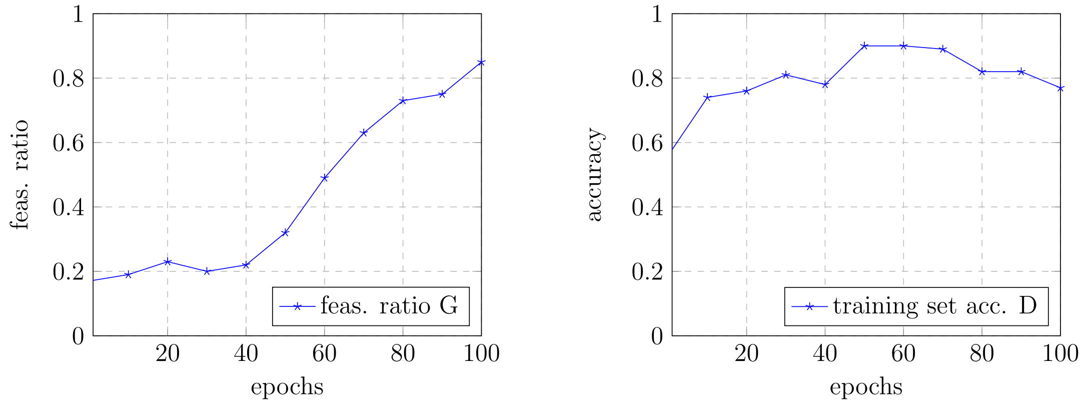


Figure 8.5: feas. ratio of G on the training set (left) and accuracy of D on the training set (right), after 100 epochs of GAN training

These figures show that while G is constantly improving, the D's accuracy reaches its maximum value around epoch 50, then gently declines over time. During this decline, the G's feasibility ratio is still growing. Since we are using a *unique* variety dataset, the learning process is slower than using a *multiple* variety, but the overall maximum value of feasibility ratio reached is still the same. So it is not a question of what value of feasibility ratio it can yield, but how fast the training process goes.

If we plot the feasibility ratios (figure 8.6) for each type of solution, we can see a similar pattern as before: G's performance is very good on solutions that have fewer queens, but gets worse as the number of queens already in place increases.

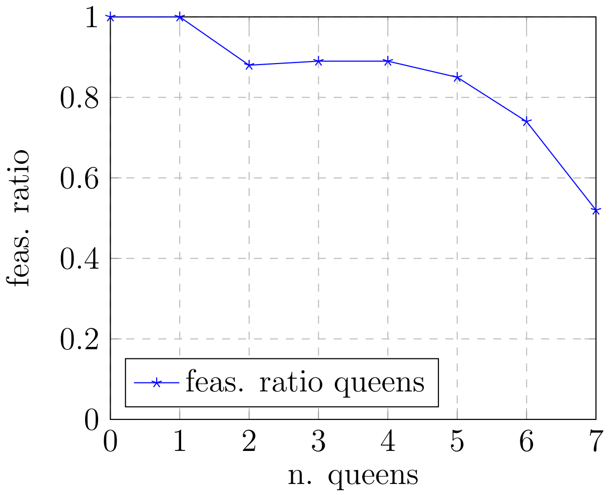


Figure 8.6: feasibility ratio for different types of solutions at epoch 100 (untrained G, A unique ds)

Finally, it is useful to plot how G and D losses evolve during the GAN training, when using A as the training set (figure 8.7).
The loss evolution for untrained G and D is similar to the one seen in the corresponding second family dataset experiments: the G and D losses evolve initially in different direction, G's loss increasing and D's loss decreasing, and later converging to a more stable value. When losses become more stable, the training of G seems to be faster.

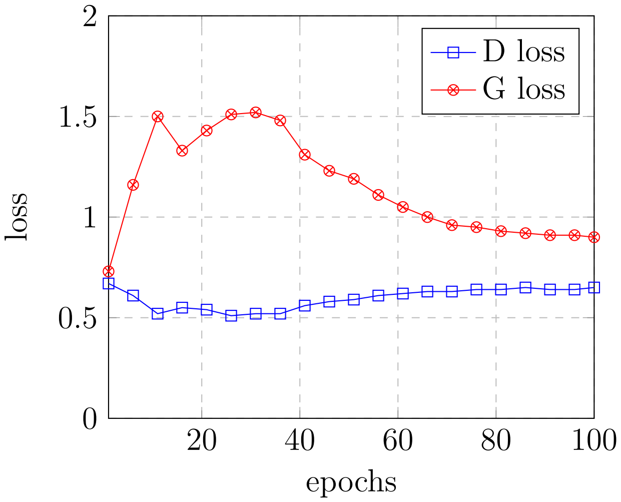


Figure 8.7: G loss and D loss progression over training epochs (using the A unique dataset)

8.3.8 Experiments using WGAN

In this experimental section, we will train an untrained G using a WGAN model instead of a standard GAN; knowing that the WGAN has a Critic model instead of a Discriminator, and that the Critic's job is not to distinguish between real and fake data, we will not report the accuracy of the C.

Initially, the data presented as *fake* to the C will be the data produced by G (G's predictions), instead of using unfeasible partial solutions as in a typical WGAN implementation [44]. WGANs with gradient penalty usually do not employ the predictions produced by G as *fake* data, instead they sample from the same data distribution (noise) used by G to generate its output [44].

In this instance, we don't have a noisy distribution to sample; what we can do is provide as *fake* data the unfeasible solutions that are of the same dataset group as the one of the training set. For example, the DS-G-A training set contains only feasible data; by using the previously described D dataset generator, it is possible to produce unfeasible data for DS-G-A, and use it as *fake* data during C's training.

In this way, the C is trained using both feasible and unfeasible data, while G is still trained using only feasible data as input, and learns through C. This is different from the stand-alone case, in which G learned in a supervised way, thus knowing the labels (y) for each data in input (x). From now on, we will call this WGAN model *WGAN2*.

We start the experiments by performing the same analysis as before, using the previously described WGAN design.

In table 8.7 it is possible to observe the feasibility ratios of G, before and after 100 epochs of training, for both the *standard* WGAN and the WGAN with unfeasible data (WGAN2). The A and B dataset groups are used as the WGAN training set alternatively.

WGAN seems to perform slightly worse than the GAN counterpart, albeit by a small margin; if we look at WGAN2, we can see that the feasibility ratios are even higher, almost comparable to the ones obtained using a stand-alone approach.

feasibility ratio G					
train set type	WGAN type	train set baseline	test set baseline	training set	test set
A	WGAN	0.16	0.17	0.76	0.39
B	WGAN	0.15	0.16	0.77	0.45
A	WGAN2	0.15	0.14	0.94	0.60
B	WGAN2	0.20	0.16	0.86	0.40

Table 8.7: untrained G performance before and after 100 epochs of WGAN training (with and without unfeasible data)

In figure 8.8 it is possible to observe how the feasibility ratio of G evolves during training, for both WGAN instances.
As we can see, training is faster than the one observed using a standard GAN, leading to a much better feasibility ratio during the earlier phases of training, both on training and test sets, and later converging to a more stable value; when observing the feasibility ratio evolution for WGAN2, it it clear that training is much faster than the one using WGAN or alternative GAN architectures, and the feasibility ratio reached is also significantly higher.

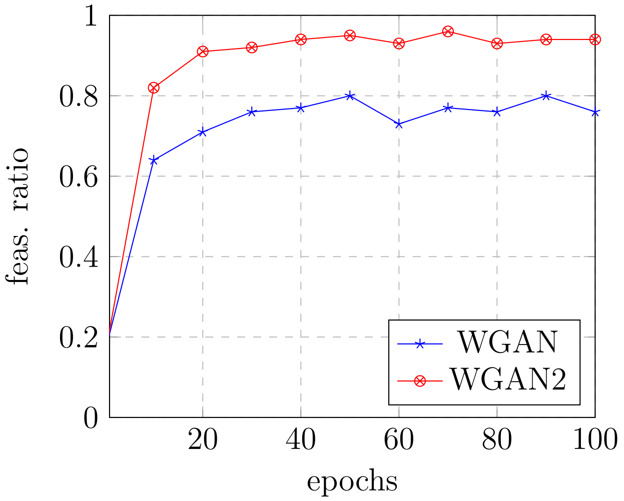


Figure 8.8: how G feasibility ratio (training set) changes over time using WGAN GP (with and without unfeasible data)

Following the evaluations of previous experiments, we plot the feasibility ratios for each type of solution, as seen in figure 8.9.
In the WGAN instance, the plot features a slightly different behaviour as the one seen in previous analysis: the feasibility ratio is higher for solutions with 0 or 1 queen already in place, but it declines significantly for solutions that have 2 queens, then in increases again until the final solution type (7 queens).
The WGAN2 plot shows further evidence that WGAN2 performance is much closer to the one seen in the stand-alone training case, than the one seen with GAN or WGAN.

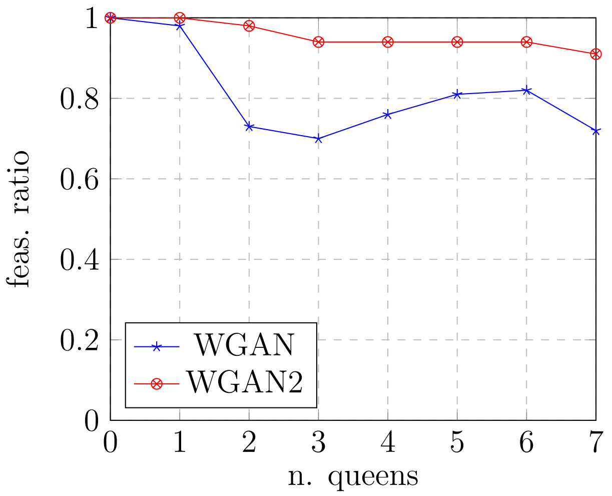


Figure 8.9: feasibility ratio for different types of solutions at epoch 100 (with and without unfeasible data)

Finally, it can be interesting to show also how G and C losses evolve during the WGAN training, as shown in figure 8.10.

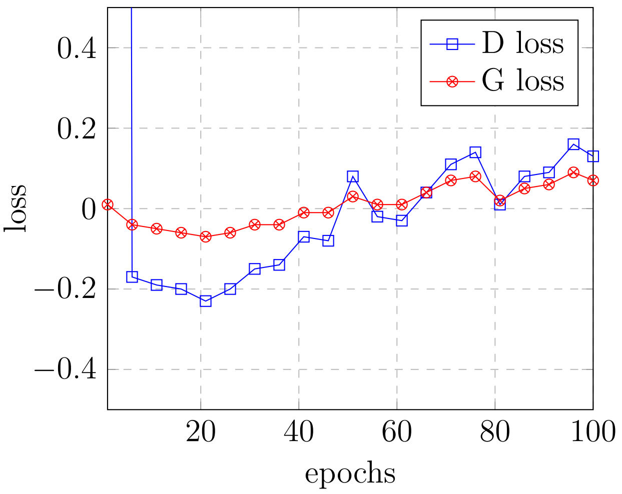


Figure 8.10: G loss and D loss progression over training epochs, using WGAN

The loss evolution during the WGAN training is much different than the one seen using the standard GAN: the C's loss is constantly decreasing and it has always a negative value during the early training phases, then it converges around 0; the G's loss hovers almost always around 0, with minimal changes over time.

If we look at the losses evolution for WGAN2, as we can see in figure 8.11, we see that during the early epochs, when training is faster, D's loss is stationary around zero; after 40 epochs, D's loss decreases constantly, while G's loss slowly converges to zero.

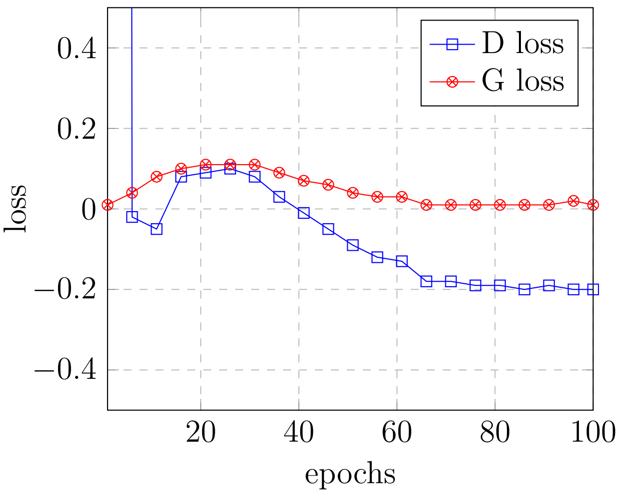


Figure 8.11: G loss and D loss progression over training epochs, using WGAN2

8.4 Support Vector Machines experiments

We perform the same experiments as the ones in previous sections, this time using Support Vector Machines (SVMs).

The first SVM is designed as a generator (SVM-G), mimicking the behaviour of the Generator neural network. This means that SVM-G is trained on a certain dataset of the third family, so A or B datasets, and it must learn to perform the desired assignment for each partial solution in input.

SVM-G is implemented as an Scikit-learn SVM [53], using a *one-vs-one* approach for the training, and enabling class membership probability estimates, instead of the original scores.

In table 8.8 we can see a comparison of the results obtained through SVM training, with the ones obtained using stand-alone and WGAN training. It is evident that SVM-G has a similar performance of the one exhibited by the stand-alone training, only slightly worse; we also see a margin of improvement with respect to WGAN and WGAN2 training methods.

feasibility ratio G			
train set	training method	training set	test set
A	stand-alone	1.00	0.60
B	stand-alone	1.00	0.55
A	WGAN	0.76	0.39
B	WGAN	0.77	0.45
A	WGAN2	0.94	0.60
B	WGAN2	0.86	0.40
A	SVM	0.99	0.59
B	SVM	0.99	0.51

Table 8.8: feas. ratio of G after training, using different training techniques

A second SVM is created (SVM-D) using Scikit-learn as well, with the goal of mimicking the behaviour of the Discriminator Neural Network. So, ultimately, the goal of SVM-D is to be able to distinguish between feasible and unfeasible solutions, given a partial solution as input. In table 8.9, we can see a comparison between SVM-D, the stand-alone trained discriminator NN, and the discriminator trained through GAN; we can not compare it with WGAN experiments since WGAN does not employ a proper discriminator.

accuracy D			
train set	training method	train set acc.	test set acc.
A	stand-alone	0.99	0.49
B	stand-alone	0.99	0.48
A	GAN	0.77	0.45
B	GAN	0.77	0.44
A	SVM	0.91	0.66
B	SVM	0.91	0.52

Table 8.9: accuracy of D after training, using different training techniques

Even though a proper comparison with GAN-trained D is not totally correct, since D in a GAN setting tends to stabilize its performance as G becomes better, we can see that SVM-D has an higher accuracy on the test sets than the other two instances, but lacks a little in accuracy on the training set if compared to the stand-alone case.

8.5 Stochastic generation evaluation

In this section, we report and evaluate the results obtained from the stochastic generation process, executed using various Generator networks.
The G models examined in this section were trained using different methods: each G model has been trained using a stand-alone approach, a *mixed* one (pre-train & GAN), through a GAN/WGAN only, or using SVMs.
We employed datasets of the A and B groups as the training sets, alternatively.

In table 8.10 we can see, for each training type of G, the percentage of G's generated solutions that satisfy each one of the three, previously described, criteria. Moreover, we report the amount of unique solutions produced, enabling us to assess the ability of G in producing not only valid solutions, but also solutions that are different from one another.
Each evaluation has been performed three times for each G model, taking the mean of each variable as its final value. The number of generated solutions has been set to 1000.

train type	train set	num. unique sol.	validity crit. 1	validity crit. 2	validity crit. 3
standalone	A	384	61.9%	61.9%	98.9%
standalone	B	391	61.1%	61.1%	98.7%
GAN	A	2	0.0%	0.0%	100.0%
GAN-pretrain	B	4	0.0%	0.0%	100.0%
WGAN	A	71	3.9%	3.9%	100.0%
WGAN	B	112	49.0%	49.0%	100.0%
WGAN2	A	319	43.8%	43.8%	96.6%
WGAN2	B	325	45.1%	45.1%	93.2%
SVM	A	756	26.5%	26.5%	100.0%
SVM	B	741	27.1%	27.1%	100.0%

Table 8.10: Evaluation of stochastic generated solutions, using different kinds of architectures for training. Number of solutions generated = 1000

As we can see, the standalone-trained G is the one that achieves the largest number of uniques solutions generated, and it also yields a validity on criteria 1 and 2 of over 60%.
By training G through the standard GAN model, it is possible to observe a typical problem of GANs architectures, called *mode collapsing*. The result is that G is unable to produce multiple

different solutions, and converges to 1 or 2 solutions used to fool the D. By training a pre-trained G through a GAN, using part of the original test set as the training set, the outcome is almost the same. In both instances, all the solutions produced do not satisfy the first two criteria, but they are perfectly valid for the third criterion (no queens in central position).

Using a WGAN to train G guarantees a much better outcome: the number of unique solution is significantly higher than the one seen by employing a standard GAN, furthermore, when using *B* dataset group as the training sets, almost half of the generated solutions satisfy the first two constraints.

By comparing the performance of G trained through WGAN, it is possible to observe a significant difference in the performance; indeed, by employing datasets coming from the *A* or *B* group as the training sets, we see a big difference in the amount of solutions that satisfy the first two constraints. This could be explained by the fact that when the training set *A* contains slightly more data than the *B* one, so the G training speed is not the same.

Later on, we will address this discrepancy by looking at the stochastic generation capabilities of G during different epochs of WGAN and WGAN2 training, not only after a fixed amount of epochs.

Another interesting observation, when using WGAN training, is that all the solutions generated by G are valid with respect to the third constraint, while in the standalone case a small amount of solutions were not valid (so they had queens in a central position). Overall the WGAN and the standalone training yield comparable results.

Finally, by training G through a WGAN2 (a WGAN in which the C uses both feasible and unfeasible data), we can see that the variability of generated solutions has improved once again, with more than 300 unique solutions produced. Almost half of the solutions satisfy the first two criteria, while there is a slightly decline in the percentage of solutions that do not have queens in a central position.

By looking at the SVM instance, it is possible to observe that this technique yields interesting results, with a very high number of unique solutions generated. On the other hand, the ratio of valid solutions for the first two criteria is lower than the stand-alone and WGAN instances; the third criterion is always respected, similarly to the GAN and WGAN instances.

As previously stated, we decide to analyze the apparent discrepancy in WGAN performance, by executing a new training process, and by eliminating the strong restriction on the number of epoch of training. We train the WGAN for 400 epochs using datasets of the *A* family, and every 20 epoch of training we will perform a stochastic generation evaluation for G.

We do the same thing for WGAN2; the results of these experiments can be seen in figure 8.12.

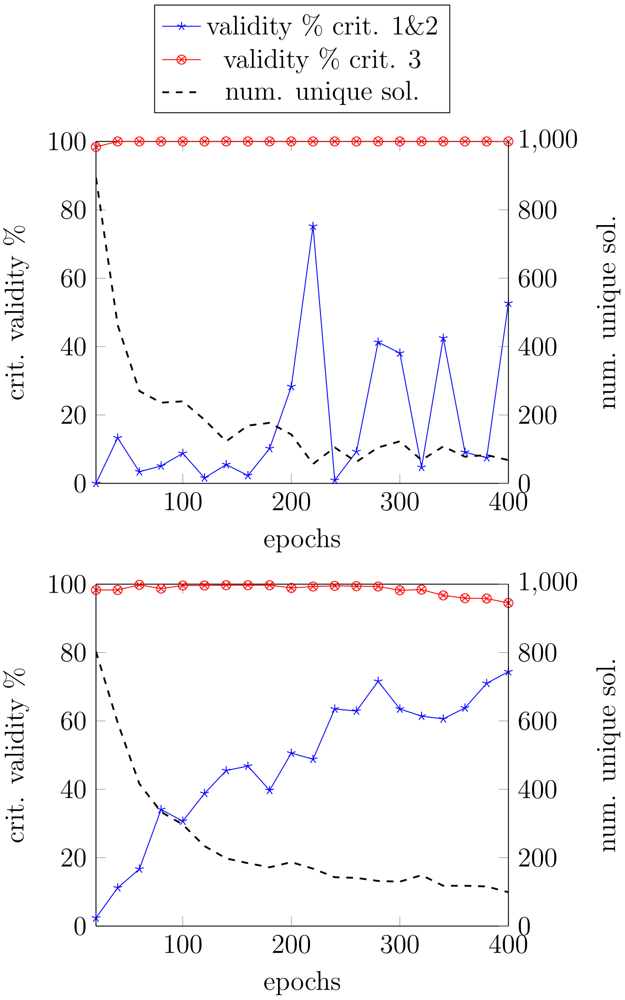


Figure 8.12: how WGAN (upper figure) and WGAN2 (lower figure) fare after 400 epochs of training, using datasets of the A group. The validity % for the first 2 criteria is plotted in blue; the validity % for the 3rd criterion is plotted in red; in a dashed black line we see the number of uniques solutions

On the left side we see the plot relative to the WGAN training. It is evident that the validity percentage for the first two criteria (blue) oscillates a lot during the epochs of training; this helps explaining the unexpected low percentage measured in the previous experiment, with respect to the B family dataset case. We also see that this percentage value has a positive trend, despite the ups and downs.
The validity % for the third criterion (red) is very high since the first epoch of training, an it stabilizes itself to 100% after 40 epochs.
The number of unique solutions generated (black, dashed line) decreases as the training progresses, more abruptly in the earlier epochs, much smoothly in the later ones.

In the WGAN2 plot, we see a similar decrease in the number unique solutions generated as before, albeit with an overall higher number.
The validity % for the first two criteria is steadily rising during almost all the training process, resulting in higher and more stable values than the WGAN case.
Finally, regarding the validity % for the third criterion, we see that this value remains stable for almost all the training, with a small decrease in the latest epochs.

8.6 Analysis & Observations

8.6.1 Generator

If we compare the results obtained in this chapter with the ones obtained using the second family of datasets, we can see that, in this instance, both the test set accuracy and the feasibility ratio on the test set are higher. This can be explained by the fact that DS-G-A and DS-G-B datasets are fairly similar in terms of size and type of solutions contained, while the DS-G-C and DS-G-O contain only solutions with and without queens in a central position respectively. Other than the dataset motivation, a small boost in performance is also provided by the use of Leaky ReLU instead of standard ReLU, as the G activation function in residual blocks.

8.6.2 Discriminator

The results obtained in this chapter for D differ from the ones obtained in the previous experiments: the test set accuracy is lower than the one obtained during the first experiment (using the original datasets), but it is higher than the one obtained in the second experiment. As already stated for the G, this is an expected outcome, since DS-D-A and DS-D-B datasets are more similar than DS-D-O and DS-D-C.
Overall, D's accuracy is not high enough to invalidate a pre-training plus GAN scenario, so we were able to employ D in later experiments.

8.6.3 GAN

We evaluated GAN and WGAN performances after several different training configurations.
For the standard GAN, we performed the same two experiments seen in the previous chapter, using pre-trained and untrained G & D, obtaining very similar results. The G trained through the GAN seemed to reach a sort of upper value limit, in terms of feasibility ratio, similar to the previous experiments, strengthening the suspicion that the use of discrete data in the training set is the likely culprit. It is possible that, using a sort of smoothing method on the data in the training set, thus making the data continuous, the margin of improvement could be higher.

Using a WGAN, we experimented with an without unfeasible data; in the former case, we obtained comparable performances, for G, as the stand-alone training.

Finally, we examined the ability of G of generating full solutions, starting from empty ones, through the stochastic generation evaluation.
The third criterion, no queens in a central postion, was respected by G basically for all the solutions produced, and for all the different training situations. This means that G is able to learn the data bias correctly, understanding that, in order to produce valid solutions, they must not have central queens.
As far as the first and second criteria, the 8-queens problem constraints and the 8-queens problem constraints plus no queens in central positions respectively, the GAN-trained G exhibited the worse performance.
This is strongly correlated to the small amount of unique solutions that this training type shown; G is trained through a standard GAN and learns to produce few unique solution that trick D, but the problem is that, for that particular group of solutions, G is unable to place feasible queens when many queens are already on the board (eg. board with already 6 or 7 queens). It is also possible that D is unable to properly *direct* G towards the generation of feasible solution, especially for almost full solutions.

On the other hand, by training G through WGAN and WGAN2, we saw an high percentage of generated solutions that were valid for the first two criteria. Furthermore, by training both

models for more epoch, we saw that the amount of valid solutions even surpassed the one seen in the stand-alone case.
Finally, generators trained through WGAN and WGAN2 exhibited, especially in the latter case, a strong ability in producing unique solutions through the stochastic generation method, thus proving that WGANs are less prone to mode collapse, in comparison to standard GANs.

8.6.4 SVM

SVM-G and SVM-D proved to be capable models, even when compared to the stand-alone trained neural networks.
The surprising good performance of the two SVMs employed to tackle the the 8-queens problem, helped us understand that even a simpler technique could yield a meaningful insight on the problem at hand. This could indicate that the problem is not as complex as desired, and further experiments should be executed on more complex tasks.

Chapter 9

Conclusions

In this thesis work, different experiments were performed, using two distinct Deep Neural Networks: a Generator (G) that, given a partial solution, is trained to perform a single, globally consistent assignment for a Constraint Satisfaction Problem; a Discriminator (D), that is trained to distinguish between feasible and unfeasible (possibly partial) solutions.
During the various experiments, we saw that both networks were able to become better in working and understanding the 8-queens completion problem, and we can say that these networks have successfully learned something about the problem's structure. This is a remarkable fact, since the models were agnostic to the problem's constraints, thus they needed to automatically extract knowledge about the problem directly from the unstructured, binary data provided as input.
While both networks were unable to significantly enhance their pre-existing abilities in a GAN setting, this configuration proved to yield interesting improvements, even if less impactful as the ones obtained through stand-alone training.
In the last experiment, we intentionally discarded solutions with certain characteristics from the training set, thus introducing a bias. Furthermore, we designed three criteria to properly evaluate the ability of G in learning constraints and preferences: the first one concerns the three constraints of the 8-queens problem (row, column, diagonal); the second one is the same as the previous one, with the addition of the bias; the third one considers only the bias. During this experiment, the G trained through a GAN displayed an interesting ability in learning data biases and preferences, while lacking in production of unique solutions; on the other hand, the G trained through a WGAN demonstrated a tremendous amount of improvement both in the variability of solutions produced, and in the solutions validity with respect to the three, previously defined, criteria.
By performing the same experiments using a simpler machine learning technique, such as SVMs, we obtained interesting results: SVMs exhibited a remarkable ability both in generator and discriminator tasks, thus achieving comparable performances to the stand-alone trained Neural Networks and WGAN-trained generator. In particular, the generator SVM was able to produce a huge amount of unique solutions during the stochastic generation process, an higher number than the ones obtained using other techniques. Despite the high variability, the generator SVM produced solutions that, for the most part, were not valid with respect to the first two criteria; on the other hand, the first two criteria are respected much more frequently in solutions generated using neural network versions of G, showing that the networks learned more about the problem's characteristics and properties than the SVM instances.
All things considered, this thesis work proves that Deep Learning and GANs methods, applied to CSPs, can yield to interesting results, even without relying on customized, carefully hand-crafted data.

9.1 Future Developments

Due to the time constraints of this thesis work, there are several techniques and evaluation methods that could not be tested, but they could be explored more in detail in subsequent researches and studies.
One possible alternative, to the models proposed in this thesis work, is to employ Convolutional Neural Networks instead of Fully-Connected ones; even if it means being a little less agnostic to the problem structure as we would like to be, this could help improving the performance on previously unseen data.
Another interesting path is to examine more in detail the WGAN model, and try to experiment with different hyper-parameters from the standard ones.

In this thesis, we applied Deep Learning techniques only to the 8-queens completion problem, so an important task is to extend the analysis to different, hopefully more complex, CSPs. Some examples of CSPs to tackle could be the Partial Latin Square problem, fairly similar to the 8-queen problem, or more realistic, practical issues, such as scheduling and timetabling problems.

In terms of evaluation techniques, it could be useful to perform a local feasibility test, by checking whether a partial solution is actually feasible, instead of a global one, where we test if a partial solution can be extended to a full feasible solution.
Furthermore, it could be possible to gain more insight about the solutions produced by the Generator network by measuring the *cost* of each solution. This could be done by counting how many queens in a solution violate the constraints, and also by counting the number of queen to move in another position to make the solution feasible.
Finally, it could be useful to evaluate the number of violations for each category of constraint (row, column, diagonal), distinguishing between standard and binary violations. In the latter, each queen should be checked with respect to another queen.

Bibliography

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

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN: 0136042597, 9780136042594.

[3] Milano M. and Lombardi M. *Intelligent Systems M - CSP, AA 2018-2019*. URL: http://ai.unibo.it/node/606.

[4] Kiziltan Z. and Gabbrielli M. *Constraint Propagation: The Heart of Constraint Programming*. URL: http://www.cs.unibo.it/gabbri/MaterialeCorsi/CP@CS.pdf.

[5] Mello P. and Chesani F. *Fondamenti di intelligenza artificiale M - Vincoli, AA 2018-2019*. URL: http://lia.deis.unibo.it/Courses/AI/fundamentalsAI2018-19/.

[6] URL: https://ktiml.mff.cuni.cz/~bartak/constraints/propagation.html.

[7] Andrea Galassi. "Symbolic versus sub-symbolic approaches: a case study on training Deep Networks to play Nine Men's Morris game". 2016.

[8] *Constraint satisfaction techniques in planning and scheduling - Scientific Figure on ResearchGate*. URL: https://www.researchgate.net/figure/Map-Graph-Coloring-Problem_fig2_226427912%20[accessed%2026%20Jun,%202019].

[9] Charles J. Colbourn. "The complexity of completing partial Latin squares". In: *Discrete Applied Mathematics* 8 (Apr. 1984), pp. 25–30. DOI: 10.1016/0166-218X(84)90075-1.

[10] L. D. Andersen. "Chapter on The history of latin squares". In: *Research Report Series, No. R-2007-32* (2007).

[11] Jaromy Kuhl and Michael W. Schroeder. "Completing Partial Latin Squares with One Nonempty Row, Column, and Symbol". In: *Electr. J. Comb.* 23 (2016), P2.23.

[12] James R. Bitner and Edward Reingold. "Backtrack Programming Techniques." In: *Commun. ACM* 18 (Nov. 1975), pp. 651–656. DOI: 10.1145/361219.361224.

[13] Alan K. Mackworth and Eugene C. Freuder. "The complexity of some polynomial network consistency algorithms for constraint satisfaction problems". In: *Artificial Intelligence* 25.1 (1985), pp. 65–74. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(85)90041-4. URL: http://www.sciencedirect.com/science/article/pii/0004370285900414.

[14] James Crawford et al. "Symmetry-Breaking Predicates for Search Problems". In: (Mar. 1997).

[15] Jordan Bell and Brett Stevens. "A survey of known results and research areas for n-queens". In: *Discrete Mathematics* 309.1 (2009), pp. 1–31. ISSN: 0012-365X. DOI: https://doi.org/10.1016/j.disc.2007.12.043. URL: http://www.sciencedirect.com/science/article/pii/S0012365X07010394.

[16] Ian Philip Gent, Christopher Anthony Jefferson, and Peter William Nightingale. "Complexity of n-Queens Completion". In: *Journal of Artificial Intelligence Research (JAIR)* (Aug. 2017).

[17] Xindong Wu et al. "Top 10 algorithms in data mining". In: *Knowledge and Information Systems* 14 (Dec. 2007). DOI: 10.1007/s10115-007-0114-2.

[18] Esperanza García-Gonzalo et al. "Hard-Rock Stability Analysis for Span Design in Entry-Type Excavations with Learning Classifiers". In: *Materials* 9 (June 2016), p. 531. DOI: 10.3390/ma9070531.

[19] Sartori C. *Data Mining M - Linear Classification with the Perceptron, AA 2018-2019.*

[20] W. S. McCulloch and P. Walter. *A logical calculus of the ideas immanent in nervous activity.* The bulletin of mathematical biophysics. 1943.

[21] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para.* Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957. URL: https://books.google.it/books?id=P%5C_XGPgAACAAJ.

[22] Fei-Fei Li, Justin Johnson, and Serena Yeung. *Stanford University CS231n: Convolutional Neural Networks for Visual Recognition.* 2017.

[23] Chris Manning. *Stanford University CS224n: Natural Language Processing with Deep Learning.* 2017.

[24] Xavier Glorot and Y Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Journal of Machine Learning Research - Proceedings Track* 9 (Jan. 2010), pp. 249–256.

[25] URL: https://www.learnopencv.com/understanding-activation-functions-in-deep-learning/.

[26] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[27] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[28] Léon Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010).* Ed. by Yves Lechevallier and Gilbert Saporta. Paris, France: Springer, Aug. 2010, pp. 177–187. URL: http://leon.bottou.org/papers/bottou-2010.

[29] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (Dec. 2014).

[30] URL: https://stats.stackexchange.com/questions/131233/neural-network-over-fitting.

[31] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[32] Mohammad Sadegh Ebrahimi and Hossein Karkeh Abadi. "Study of Residual Networks for Image Recognition". In: *CoRR* abs/1805.00325 (2018). arXiv: 1805.00325. URL: http://arxiv.org/abs/1805.00325.

[33] Kaiming He et al. "Identity Mappings in Deep Residual Networks". In: *CoRR* abs/1603.05027 (2016). arXiv: 1603.05027. URL: http://arxiv.org/abs/1603.05027.

[34] Andrea Galassi, Marco Lippi, and Paolo Torroni. "Attention, please! A Critical Review of Neural Attention Models in Natural Language Processing". In: *CoRR* abs/1902.02181 (2019). arXiv: 1902.02181. URL: http://arxiv.org/abs/1902.02181.

[35] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *CoRR* abs/1409.0473 (2015).

[36] John Pavlopoulos, Prodromos Malakasiotis, and Ion Androutsopoulos. "Deeper Attention to Abusive User Content Moderation". In: Jan. 2017, pp. 1125–1135. DOI: 10.18653/v1/D17-1117.

[37] URL: http://kvfrans.com/variational-autoencoders-explained/.

[38] Jamie Hayes et al. "LOGAN: Evaluating Privacy Leakage of Generative Models Using Generative Adversarial Networks". In: (May 2017).

[39] Ian J. Goodfellow et al. "Generative Adversarial Networks". In: (June 2014). URL: https://arxiv.org/abs/1406.2661 (visited on 01/08/2017).

[40] Soumith Chintala et al. *How to train a GAN - Tips and tricks to make GANs work.*

[41] Tim Salimans et al. "Improved Techniques for Training GANs". In: *CoRR* abs/1606.03498 (2016). arXiv: 1606.03498. URL: http://arxiv.org/abs/1606.03498.

[42] Ka Chan, C T. Lenard, and Terence Mills. "An Introduction to Markov Chains". In: Dec. 2012. DOI: 10.13140/2.1.1833.8248.

[43] Martín Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein GAN". In: *CoRR* abs/1701.07875 (2017).

[44] Keras Team. *Improved WGAN - Official Keras Documentation.*

[45] Ishaan Gulrajani et al. "Improved Training of Wasserstein GANs". In: *CoRR* abs/1704.00028 (2017). arXiv: 1704.00028. URL: http://arxiv.org/abs/1704.00028.

[46] Andrea Galassi et al. "Model Agnostic Solution of CSPs via Deep Learning: A Preliminary Study". In: (2018). URL: https://doi.org/10.1007/978-3-319-93031-2_18.

[47] Hong Xu, Sven Koenig, and T K. Satish Kumar. "Towards Effective Deep Learning for Constraint Satisfaction Problems". In: Aug. 2018. DOI: 10.1007/978-3-319-98334-9_38.

[48] Yexiang Xue and Willem-Jan van Hoeve. "Embedding Decision Diagrams into Generative Adversarial Networks". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research.* Ed. by Louis-Martin Rousseau and Kostas Stergiou. Cham: Springer International Publishing, 2019, pp. 616–632.

[49] Wei Shao et al. "Approximating Optimisation Solutions for Travelling Officer Problem with Customised Deep Learning Network". In: *CoRR* abs/1903.03348 (2019). arXiv: 1903.03348. URL: http://arxiv.org/abs/1903.03348.

[50] André Hottung, Shunji Tanaka, and Kevin Tierney. "Deep Learning Assisted Heuristic Tree Search for the Container Pre-marshalling Problem". In: *CoRR* abs/1709.09972 (2017). arXiv: 1709.09972. URL: http://arxiv.org/abs/1709.09972.

[51] Hao-Wen Dong and Yi-Hsuan Yang. "Training Generative Adversarial Networks with Binary Neurons by End-to-end Backpropagation". In: *CoRR* abs/1810.04714 (2018). arXiv: 1810.04714. URL: http://arxiv.org/abs/1810.04714.

[52] Keras Team. *Mnist ACGan - Official Keras Documentation.*

[53] URL: https://scikit-learn.org/stable/modules/svm.html.

Acknowledgements

I would like to thank Prof. Michela Milano, along with Michele Lombardi and Andrea Galassi, for the opportunity to work on this thesis, and for their support during these months.

I would like to also thank all the professors and students that I encountered during my studies in Bologna, I certainly learned a lot from each one of them.

I want to express my gratitude to my parents, all my family and my friends, for their costant support and encouragement during these years. In particular, I want to thank my grandfather Ermanno, because he gave me confidence when I needed it the most.

Finally, I am extremely thankful to Laura for helping and supporting me in any given situation, and for all the good moments that we have spent together.