SAPIENZA
UNIVERSITÀ DI ROMA

Development and Analysis of Graph Neural Network Algorithms to solve CS problems in the Statistical Mechanics framework

Faculty of Mathematical, Physical and Natural Sciences
Theoretical Physics

Marco Veith
ID number 1695398

Advisor
Prof. Stefano Giagu

Co-Advisor
Prof. Maria Chiara Angelini

[illegible]

Academic Year 22/23

Thesis not yet defended

Development and Analysis of Graph Neural Network Algorithms to solve CS problems in the Statistical Mechanics framework
Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: marco.veith@roma1.infn.it

To my future nibling,
for a joyful beginning of life
waiting to wish you the very first happy birthday

and

to myself,
for this enduring goal achieved,
for all the silent improvements
focusing on this new chapter of my life.

Contents

Introduction

Constraint satisfaction problems (CSPs) are a fundamental class of problems that involve the basic concepts of computational complexity theory and are encountered in a wide range of scientific disciplines, including computer science, physics, and engineering. Over the last decades, a wide range of diverse techniques hailing from various mathematical disciplines such as algebra, logic, mathematical programming, probability theory, graph theory, and combinatorics have been employed to study the computational complexity and approximate feasibility of algorithms linked to the constraint satisfaction problems. Concurrently, there has been a significant amount of research undertaken to examine the usefulness and restraints of algorithmic methods. This research avenue is progressing at an astounding rate and consistently disclosing highly robust and universal outcomes.

Today, CSPs are frequently employed in theoretical computer science, particularly because of their rich structure, which provides a good testing ground for classification and algorithmic techniques. CSPs are also vital in error-correcting codes when encoding transmitted information into a code word, satisfying a set of constraints to retrieve such knowledge after transmission through a noisy channel. In more applied fields of computer science, CSPs are considered a versatile and efficient method for modeling and solving a variety of real-world problems, such as scheduling and planning, natural language understanding, and software verification.

An instance of CSP comprises variables, values for variables, and constraints that restrict the possible combinations of values which those variables can take. The range of possible questions that can be asked include determining if there is an assignment of values to variables that satisfies all constraints or optimizing such assignments in a range of ways. It also includes counting satisfying assignments and finding assignments that fulfill as many constraints as feasible.
Constraint satisfaction has always held a crucial role in computational complexity theory, with various versions of CSPs being classic complete problems in most typical complexity classes. CSPs provide an excellent perspective on general computational phenomena by illuminating which mathematical properties make a computational problem solvable (in a broad sense, such as polynomial-time solvable or non-trivially approximable, fixed-parameter tractable or definable in a weak logic).

Through the introduction of a cost function, which depends on parameters that regulate the constraints defined by the problem, it is possible to study a CS problem as an optimization problem. Optimization is a widely employed concept in various fields of scientific activities. Generally, it involves a multitude of variables, such as particles, agents, cells, or nodes, and a cost function that depends on these variables, such as energy, expense, or risk measure. The problem lies in determining a state of variables that minimizes the cost function value. Many areas of science require optimization, whether it be in computer science, engineering, statistical physics, or even biology and social sciences. The process of optimization involves numerous variables and a cost function dependent on these variables, as mentioned before; it can often be difficult when encountering optimization problems in the NP-complete class. These particular problems are believed to require a vast number of operations to minimize the cost function, leading to an exponential increase in the system size.

By utilizing concepts from the statistical physics of disordered systems, specifically the cavity method typically used for describing glassy systems, it is possible to highlight new characteristics of the solutions space; the particular topic in which this thesis focuses is the vertex coloring problem of random graphs. Although the statistical description of these problems is manageable, the algorithmic aspect of solving them is difficult.
It is important to recognize the difficulty involved in solving real-world examples of constraint satisfaction problems; often, a solution can prove to be elusive despite best efforts. For many CSPs, overcoming this obstacle requires a combination of heuristics and combinatorial search approaches.
This topic is significant not only from a theoretical perspective, but also for practical reasons. Firstly, comprehending the origins of complexity can help improve the performance of CSP solvers. Secondly, identifying problematic instances can enable their avoidance if the situation permits. Thirdly, the discovery of incredibly challenging problems has potential applications in cryptography.

Extensive research has been conducted by the scientists involved in these studies, taking a crucial step towards those objectives by investigating the underlying causes of difficulty in random constraint satisfaction problems. Additionally, as these models resemble spin glasses, the scientific production applies techniques borrowed from the statistical physics of disordered systems.
In this thesis, the aim is to study the same problems, but with a new strategy: the employment of neural network algorithms, a branch of artificial intelligence.
Since 1950s the topic of artificial intelligence has become a matter of interest for the scientific community and has raised many philosophical discussions. As stated by A.M. Turing in [1], the focus on this subject should not deal with the question "Can machines think?" but instead it should be based on how well a machine can reproduce a cognitive human process (or how well a machine can play the so called by Turing in [1] *imitation game*).
Nowadays, in my opinion, the concept of artificial intelligence connected to the idea of consciousness, which is more closely related to science fiction, is far from the current situation. The perspective proposed by Turing focuses more on the abilities of the machine rather than on existential questions related to intelligence; based on this perspective, it is convenient to consider artificial intelligence as a powerful tool to leverage.

The application of artificial intelligence ranges many different fields nowadays. Several tasks that goes from image recognition, speech processing to decision making are covered by algorithms of artificial intelligence. Neural networks have become a great tool for solving various mathematical problems. These problems are complex and often require analyzing large amounts of data or making accurate predictions based on incomplete information.
Neural networks undoubtedly offer a significant advantage in the extreme variability of applications and the high potential demonstrated in various different fields. In such a manner arises the idea of applying neural networks to optimization problems related to CSPs. The objective is to explore the potential of such an approach and compare outcomes with those of traditional methods.

While a typical algorithm is based on the direct resolution of the problem, neural networks are based on learning the task through what is called training. Neural networks consist of interconnected nodes that process information and can learn from past experiences to improve their performance.
This completely different approach has both benefits and drawbacks, which will be investigated in this thesis. Although these algorithms have different applications and operations, their results can still be analyzed within the context of disordered systems, presenting a theoretical physics approach to the topic.

Chapter 1

The Vertex Coloring Problem

The picture will have charm when each color is very unlike the one next to it.

Leon Battista Alberti

Constraint Satisfaction Problems (CSPs) are a relevant issue, among the mathematical questions, in which the system is formed by a set of discrete variables and the problem refers to the search for the specific objects belonging to the set of combinations of variables where all the constraints are satisfied. Formally, for a set of N variables, each with domain D, a constraint (or clause) of $K \leq N$ variables, referred to as $\mathbf{x}$, is a function $\Gamma : D^K \longrightarrow \{0,1\}$ where if $\Gamma(\mathbf{x}) = 1$ the constraint is said to be satisfied (SAT), otherwise it is said to be not (UNSAT).
Typically, a CSP involves many different clauses and consists in the research of a configuration of the N variables in order to satisfy simultaneously a set of M constraints. Such assignment is a possible solution to the problem and the set of constraints in a CSP is called *instance*.
Constraint satisfaction problems may be studied introducing a cost function which give a quantitative measure of how much the system is not following the clauses. Using this, it is possible to map CSPs into discrete optimization problems, e.g. considering the cost function as the number of unsatisfied constraints. In this way, in CSPs the optimal solution is the one that satisfies particular conditions that are the characteristic which defines the mathematical question, i.e. the constraints themselves.
To give a more formal definition [2], given a set of discrete variables, the interest is to find $\min(\alpha(S))$ or $\max(\alpha(S))$ with $S \in F$ where F is the set of all possible arrangements, α is a cost function measured on the set S, depending on the imposed constraint. In this thesis, the vertex coloring problem is the main topic and it deals with combinatorics applied on graphs.
A graph is a simple mathematical object $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ defined by a specific number, i.e. the order of the graph, of points - called nodes or vertices, composing the set $\mathcal{V}$ - and connections (directed or undirected) between them, represented by lines that take the name of edges $e \in \mathcal{E}$. Graphs are useful to describe interconnections between each element of a system, but in our case we are concerned with *extremal graph theory*, a branch of combinatorics: *extremal* problems in graph the-

ory are mathematical problems that involve finding the optimal or maximum/minimum value of a property or characteristic of a graph or a family of graphs. In other words, they deal with determining the largest or smallest possible value of a certain parameter such as the number of edges, vertices, or partitions in a graph or the size of a particular substructure of a graph. These problems often involve finding structures or patterns that satisfy certain conditions while maximizing or minimizing a certain objective function. Despite the simplicity of the idea behind a graph, it is possible to take into account various tough questions [3] such as: at least how many edges are in a graph of order N if it is forced to contain a path of length ℓ? More generally, declaring the *forbidden subgraph* problem: given a graph F, which is the maximum number of edges in a graph of order N such that it does not contain F as its subgraph?
The vertex coloring problem is a CSP that consists in the following question: at least how many colors do we need to paint each vertex of a graph such that there are no two neighbouring vertices that share the same color? The constraint refers to the two vertices that cannot share the color and the optimal solution we look for is the one where we used the minimum number of colors, which takes the name of "chromatic number". An important theorem assures that there is always a solution for planar graphs using only four colors [4].

1.1 Random graphs

To deal with this problem, in order to better study the problem, it has been considered a particular class of graphs, known as "Erdős–Rényi" (ER) graphs. These are defined by a fixed number N of vertexes where E edges are chosen at random with uniform distribution - equivalently, using Gilbert definition, given N vertices, each possible pair is linked by an edge with a fixed probability p.
In order to describe the same graph, $p = \frac{E}{E_{tot}}$ where $E_{tot} = \binom{N}{2} = \frac{N(N-1)}{2}$ is the amount of all possible edges in the graph, i.e. how many pairs of nodes can be formed. Indeed, the Gilbert description let us write the distribution of edges $P(k)$ as a binomial distribution, for a given p:

$$P(k) = \binom{E_{tot}}{k} p^k (1-p)^{E_{tot}-k} \tag{1.1}$$

The expectation value is $p\frac{N(N-1)}{2}$ that is equal to E if we choose p defined as above. Since each node can be matched with the remaining $N-1$ nodes, the distribution of the number ℓ of edges a given node is linked to is again binomial, but for $p \ll 1$ it can be approximated as poissonian with parameter $c = p(N-1)$:

$$P_{deg}(\ell) \simeq \frac{c^\ell}{\ell!} e^{-c} \tag{1.2}$$

This leads to the mean value of connections per node, also called mean connectivity of the graph, obtained computing the expectation value of the latter distribution:

$$\langle \ell \rangle \equiv c = p(N-1) = \frac{E}{\frac{N(N-1)}{2}}(N-1) = \frac{2E}{N} \tag{1.3}$$

Graph's mean connectivity will turn out to be an important statistical parameter.

1.2 The statistical physics framework

The theory of computational complexity [5] seeks to determine the worst-case difficulty of decision problems. Random collections of CSP instances, which form a random constraint satisfaction problem, have become popular in efforts to examine not only the computational complexity of worst possible cases, but also of typical instances, and to test new algorithmic concepts. These random CSPs are formally connected to models studied in statistical physics of disordered systems, and utilizing methods from statistical mechanics has proven to be exceptionally prolific [6]. In general, problems from computer science can be settled as statistical mechanics models. Indeed, every CSP can be seen as a physics model where the number of unsatisfied constraints is equivalent to the energy function $\mathcal{H}$. The Boltzmann measure in the zero temperature limit will be the uniform measure over the solutions of the CSP, and in the same limit, the partition function will be the number of solutions themselves. As before, the satisfiable statements of the CSP will be the minimal (zero) energy configurations of the physics model, hence an instance of the CSP is satisfiable if and only if the ground-state of $\mathcal{H}$ is equal to zero.
Numerous CSPs are well known to be NP-hard; vertex coloring is a NP-complete problem [7]: it is simple to demonstrate that a solution - an assignment of colors for each vertex of a given graph - can be checked in polynomial time. Differently, finding a solution is a task that requires, in the worst case, a non-polynomial (exponential) time to be solved, with some exceptions for particular classes of graphs [8].
Having established the complexity of the problem, the description of Statistical Mechanics gives a different theoretical approach that can lead to more efficient solutions and strategies.

1.2.1 The Potts model

The Ising model is largely exploited in many cases of scientific production in statistical physics due to the existence of an analytical solution in two dimensions and a simple but powerful description of the system: a simple lattice full of spins which can be in two states $\sigma = \pm 1$. The Potts model is a generalization of the former where spins' state $\sigma \in \{s_1, \ldots, s_q\}$, hence the connection with graph coloring is straightforward, considering each spin state as a possible color.
Considering an ER graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a possible coloring - using q colors - $\vec{\sigma} \in \{s_1, \ldots, s_q\}^{|\mathcal{V}|}$ and the adjacency matrix $\mathcal{A}_{ij} = \{1 \Leftrightarrow (i,j) \in \mathcal{E}, 0 \ \textit{otherwise}\}$, the Hamiltonian can be written [9]:

$$\mathcal{H}(\mathcal{A}, \vec{\sigma}) = \sum_{\{i,j\}} \mathcal{A}_{ij} \delta_{\sigma_i \sigma_j} \quad \text{with} \quad \delta_{\sigma_i \sigma_j} = \{1 \Leftrightarrow \sigma_i = \sigma_j, 0 \ \textit{otherwise}\} \tag{1.4}$$

Here $\{i,j\}$ represents the set of all possible pairs of vertexes, thus $\mathcal{H}$ counts the number of edges in the graph which does not satisfy the condition. Hence the partition function, computed on the ensemble of all random ER graphs with fixed number of nodes and edges, can be written:

$$\mathcal{Z} = \sum_{\mathcal{G} \in ER} \sum_{\vec{\sigma}} e^{-\beta \mathcal{H}(\mathcal{A}(\mathcal{G}), \vec{\sigma})} \tag{1.5}$$

Here β is the inverse temperature, therefore the minimal state of energy corresponds to the limit $\beta \to \infty$.
In the class of notorious problems such as the resolution of Ising model (or Potts model, closer to the CSP studied here), statistical mechanics of critical phenomena addresses the comprehension of system's dynamics requiring to directly calculate the partition function.
Typically, the partition function is the key variable to compute all the other thermodynamic potentials, starting from the free energy. Using $\mathcal{Z}$, it is possible to directly compute the free energy density - free energy per unit of volume V of the system:

$$f_\beta = -\frac{1}{\beta V} \log\left(\mathcal{Z}(\beta)\right)$$

In the case of vertex coloring, given that the system is based on graphs, it is possible to write $f_\beta^{ER} = -\frac{1}{\beta|\mathcal{V}|}\log(\mathcal{Z}(\beta))$ where the volume is intended as the number of nodes. With the following identities it is possible to obtain the internal energy per node and the entropy per node:

$$u = \frac{\partial(\beta f_\beta)}{\partial\beta} \qquad s = \beta^2 \frac{\partial f_\beta}{\partial\beta}$$

In addition, it is possible to compute one quantity and get the other using the identity $s = \beta(u - f)$; to derive the expressions per edge from those per node, simply multiply by $\frac{c}{2}$ since each edge connects 2 vertexes and each node has in average c links, i.e. the mean connectivity. In this way, the internal energy density per node, at ground state, represents the fraction of edges that violate the constraints.
All these expressions displayed above are directly applied in simple cases where the partition function and then free energy are straightforward to obtain. Unfortunately, random CSPs and in particular vertex coloring need the application of more sophisticated methods coming from the field of statistical mechanics of disordered systems. To introduce the concepts addressed by this subject, it is useful to consider the simple example of the Random Field Ising Model (RFIM).

$$\mathcal{H}(\sigma) = \sum_{\{i,j\}} J_{ij}\sigma_i\sigma_j - \sum_{\{k\}} h_k\sigma_k$$

While the Hamiltonian of Ising model usually presents a term depending on external (local) magnetic field, which it is usually fixed (constant or null), in the case of RFIM h_k is a stochastic variable, whose distribution is considered to be known. With this additional term, the system becomes a bit more complicated; the assignment of spins which minimize the energy now deals both at the same time with local interaction between spins (contributing to align them in the same direction) and with each local random magnetic field suggesting to align in its direction.
Another example is the original case of spin glasses presented by Edwards and Anderson in [10] where the interaction J_{ij} between spin variables is randomly extracted for each lattice site. In both cases exposed, the resulting Hamiltonian is not explicitly known since it becomes a random variable itself: indeed it is interesting to ask which is its probability distribution.
The stochastic components added in the two examples are the source of the so-called

disorder of the system. Either it is the random interaction, the random magnetic fields or the random graphs themselves, whenever randomness is a feature of the system, disorder is present.
Coming back to the vertex coloring problem, it is rather difficult and not so useful to use the partition function $\mathcal{Z}$ in expression (1.5), without dealing with system's disorder. In order to do that, there are some distinct paths to follow, applying different methods such as Belief Propagation (BP) using Bethe approximation, Replica method and Replica Symmetric (RS) cavity method. Before moving to those topics, it is useful to discuss some further details on mentioned concepts involved in the theoretical framework.

1.3 Deal with disorder

One of the main ideas behind the field of statistical mechanics of disordered systems - and its applications to optimization problems as it is the case of CSPs - is the so called *self-averaging*: the randomness which causes disorder does not matter when taking the large size limit $N \to \infty$, at least in some cases, as the concept will now be better explained. Going in deeper details [11], recalling the example of RFIM, the partition function depends on the size of the system N and on the random fields $\mathbf{h}$. Therefore $\mathcal{Z}_N(\mathbf{h})$ is a stochastic variable and its fluctuations around the mean value are expected to be larger as N increases since $\mathcal{Z} \sim e^{-\beta\mathcal{H}}$ where $\mathcal{H}$ is extensive, i.e. it is directly proportional to the size of the system N.
Now, the physical interesting quantity is the ratio $-\frac{\log(\mathcal{Z}_N(\mathbf{h}))}{\beta N}$, which is the free energy density: since its fluctuations around the mean are in most cases $\propto \frac{1}{\sqrt{N}}$, it converges to a constant value as $N \to \infty$, which is the so called thermodynamic limit. Indeed, it is compelling to study the behavior of the system in this limit.
Whenever the probability distribution of an observable over the disorder concentrates around its mean, as the size of the system is increasing, it is the case of a *self-averaging* variable.
In this way, the challenge becomes to compute this mean over the disorder. There are mainly two ways to perform this evaluation: the correct but hard one and the easier but approximated (or sometimes wrong) other way.

1.3.1 The quenched ensemble

The first way is the direct computation of the mean of the observable. Recalling the free energy density and referring to the mean over the disorder with $\bar{\cdot}$ operator:

$$f_\beta^{mean} = \lim_{N\to\infty} -\frac{1}{\beta N}\overline{\log(\mathcal{Z}(\beta))} \tag{1.6}$$

The obtained quantity is the *quenched* average: for each integration over the spins, the source of disorder is fixed (hence quenched) therefore the thermal fluctuations of the spins and the disorder in the couplings or fields do not fluctuate together. In other words, for each configuration of h_k in RFIM or interaction couplings J_{ij} in vanilla spin glass Hamiltonian, free energy is computed and then it is averaged over the distribution of $\mathbf{h}$ or $\mathbf{J}$. Notice that free energy density is an example of thermodynamic potential, the same procedure may be applied to the other interesting potentials. Since it interesting to study the limit $\beta \to \infty$, it is useful to introduce the free entropy density Φ_β, which is obtained - starting from the free energy density - simply with a multiplication by $-\beta$, obtaining $\frac{\log(\mathcal{Z}_N)}{N}$; in this manner, the thermodynamic potential is well defined in the low temperature limit. In order to perform this computation, the useful identity behind the Replica method mentioned before may be employed. Supposing n to be small [11]:

$$\mathcal{Z}^n = e^{n \log \mathcal{Z}} = 1 + n \log \mathcal{Z} + o(n^2)$$

$$\frac{\mathcal{Z}^n}{n} - \frac{1}{n} + o(n) = log\mathcal{Z}$$

$$\implies \log \mathcal{Z} = \lim_{n \to 0} \frac{\mathcal{Z}^n - 1}{n} \tag{1.7}$$

Using the identity obtained in equation (1.7), since the mean operator commutes with the limit operator, it is possible to evaluate the quenched free energy density mean value directly computing $\mathcal{Z}^n$, taking the mean value and then computing the limit as in the identity (1.7).
This anyway could lead to long computations and the limit $n \to 0$ requires an analytic continuation which may not always be performed; despite it seems a not so polished approach, the replica method has proven to be effective in mathematics when adhering to the guidelines established by the physicists, following the pioneering work of G. Parisi, proposed over the last few decades [12]. Utilizing this method has consistently yielded accurate results, and can be relied upon with a high degree of confidence.

1.3.2 The annealed ensemble

The other way to deal with disordered systems is to consider the so called *annealed* ensemble. In this case, computations are much easier and the idea consists in taking the mean of the partition function and only then compute the thermodynamic potential taking the logarithm.

$$f_\beta^{ann.} = \lim_{N \to \infty} -\frac{1}{\beta N} \log(\overline{\mathcal{Z}(\beta)})$$

In order to obtain information about the phase space and infer phase transition thresholds, the above quantity cannot be used. Indeed, although it may be correct at high temperatures where thermal energy covers the disorder fluctuations, it is clear that the case of interest belongs to low temperatures; in CSP the aim is to study the system at $\beta \to \infty$.
The problem inside this computation is that the partition function, contrary to the free energy, is not an extensive quantity (indeed, it is exponential in N) therefore it is not in general *self-averaging*.
Anyway, the annealed average may be useful representing a good approximation in some case. It is also exploited in many demonstrations of several theorems in the physics of disordered systems, yielding fruitful results in computer science as well [13]; indeed, due to the concavity of the logarithm function, Jensen's inequality can be applied, therefore it is always true that $f_{annealed} \leq f_{quenched}$.

Furthermore, when disorder is quickly changing in the scale time of the case of interest, the correct physical ensemble to consider is the annealed one.
To conclude and go further with the algorithms employed to study the phase space, it is interesting to notice that the quenched free entropy density, if the typical instances of the random CSP are satisfiable, in the limit of $\beta \to \infty$ gives the number of solutions of the CSP as $\sim exp(N\Phi_{\infty}^{mean})$ which is the partition function itself.

1.4 The Bethe approximation

It is useful to introduce a model, called *graphical model*, which is very general and finds a direct application in the theoretical context of this thesis. Graphical models, also known as probabilistic graphical models or structured probabilistic models, are a mathematical framework for modeling complex systems using probability theory and graph theory. They are used to represent the dependencies and interactions between variables in a system.
Graphical models are composed of two elements: a graph and a set of probability distributions.

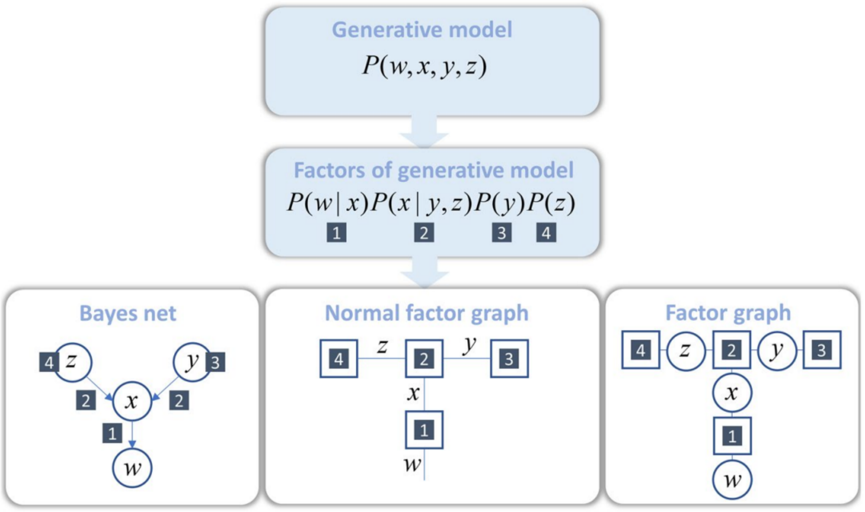


Figure 1.1. A diagram showing that the probabilities of a model can be exposed using a factor graph. For a generative model, this can be done by first breaking down the joint probability distribution into its individual factors (prior and conditional distributions) and assigning them square nodes. Nodes that share a random variable are connected. This creates a normal factor graph. Another way to represent the same model is through a Bayes net (left), which uses circular nodes for random variables connected by arrows that represent conditional distributions. An alternative factor graph (right) can also be used which combines the features of both the Bayes net and normal factor graph. Image credit: [14].

A particular class of probabilistic graphical models are normal factor graphs: the variables are the edges that link each distribution (factor) - represented with a square node - that depends on the same variable, as explained in figure 1.1. Anyway, in

this framework will be employed the (not normal) factor graph notation.
More formally, given a set of N variables $x_i \in \mathcal{V}$ which can assume values in a specific set Ω, called *alphabet*, and considering the set of neighbours ∂i corresponding to each of the M square nodes (factors in graphical models), the joint probability distribution over the variables:

$$P(\mathbf{x}) = \frac{1}{\mathcal{Z}} \prod_{i=1}^{N} P_i(x_i) \prod_{a=1}^{M} P_a(x_{\partial a}) \tag{1.8}$$

For example, Ising model in two dimensions may be described as a graphical model with spin variables $\Omega = \{+1, -1\}$, where $P_i(x_i) = e^{\beta h_i x_i}$ represent the magnetic field on every site i and $P_a(x_{\partial a}) = P_{ij}(x_i, x_j) = e^{-\beta J_{ij} x_i x_j}$ are the pairwise interactions between nearest neighbours of the square lattice.
Using this model, the aim is to obtain the marginal probabilities and then, using the distribution in equation (1.8) compute the free energy density in the thermodynamic limit, as in the quenched ensemble (see equation (1.6)). In particular, the distributions are found using the variational principle associated with the Gibbs free entropy functional:

$$\mathbb{F}[P] = -\beta\langle\mathcal{H}\rangle + \mathbb{S}[P] \tag{1.9}$$

The functional is defined over the space of all possible probability distributions P, $\mathcal{H}$ is the Hamiltonian, β is the inverse temperature, the operator $\langle\cdot\rangle$ is the expected value computed on the distribution P and $\mathbb{S}[P]$ is the Shannon entropy defined as $\langle \log(P)\rangle$. According to the variational principle, the optimal distribution corresponds to the one that minimize the functional in equation (1.9). Anyway, the basic method to obtain a first distribution as the equation (1.8) - since the direct calculation involves an exponential number of configurations - is using the mean-field approximation (MF). The baseline scheme is to simplify the dynamics ignoring all the correlations between variables; the name comes from the idea of replacing all spin couplings in Ising model with the interaction with a (constant and mean) field, generated by all the spins together. In this way, the distribution becomes directly the product of all the beliefs, one for each variable. This approach does not fit the case studied in the framework of this thesis.
Differently, the Bethe approximation uses a more sophisticated approximate posterior probability distribution which takes into account pairwise interaction. From a different point of view, the long range correlations are neglected. In order to apply the approximation, it is useful to introduce a set of possible local marginals. Although the marginals of a probability distribution $P(\mathbf{x})$ should be locally consistent, the opposite is not always true: locally consistent marginals do not necessarily correspond to any distribution. Therefore, to further highlight this point, these locally consistent marginals are also known as "beliefs".
Formally, the set is defined as $b_i(\cdot)$ over Ω for every $x_i \in \mathcal{V}$, and $b_a(\cdot)$ over $\Omega^{|\partial a|}$ for each factor a. In order to be consistent, this set of distributions is expected to follow the normalization conditions, i.e. summing over all variables gives 1, to be non-negative and to satisfy the marginalization condition:

$$\sum_{\{x_{\partial a \setminus i}\}} b_a(x_{\partial a}) = b_i(x_i) \qquad \forall i \in \partial a, \quad \forall a = 1, \dots M$$

In this way, it is possible to write the so called beliefs in the Bethe approximation:

$$b(\mathbf{x}) \simeq \prod_{i=1}^{N} b_i(x_i) \prod_{a=1}^{M} \frac{b_a(x_{\partial a})}{\prod_{r\in\partial a} b_r(x_r)} = \prod_{i=1}^{N} b_i(x_i)^{1-n_i} \prod_{a=1}^{M} b_a(x_{\partial a}) \tag{1.10}$$

In equation (1.10) the set of neighbours (in the sense of factor graph representation) of node i is expressed with ∂i and the cardinality of this set is $n_i = |\partial i|$, i.e. degree of node i. If the factor graph has a tree structure, the expression in equation (1.10) is exact. Using that distribution, it is possible to write the Bethe free entropy:

$$\mathbb{F}_{Bethe}[b] = -\beta\langle\mathcal{H}\rangle_b + \mathbb{S}[b]$$

In particular, the explicit expression [6] is:

$$\mathbb{F}_{Bethe}[b] = -\sum_{a=1}^{M} b_a(x_{\partial a}) \log\left(\frac{b_a(x_{\partial a})}{P_a(x_{\partial a})}\right) - \sum_{i=1}^{N}(1-n_i) b_i(x_i)\log(b_i(x_i))$$

In this manner, the free entropy of a tree-graphical model has a simple expression in terms of local marginals; it can be employed in graphs with loops with the hope that it provides a good estimate of the actual free entropy, in a similar way as it is done using the MF approximation, although it differs from it in several respects. When the factor graph has no loops, the Bethe free entropy has the same form of the Gibbs free entropy functional, thus in that case it is minimal for the correct marginals.
In order to find the stationary points of $\mathbb{F}_{Bethe}$, the Belief propagation method provides a useful algorithm; the stationary points of the Bethe free entropy $\mathbb{F}_{Bethe}$ are in one-to-one correspondence with the fixed points of the BP algorithm [6].

1.5 Belief Propagation

Belief Propagation (BP) is a procedure employed in Bayesian and Markov network for inference algorithms [15] and it is based on message propagation, aiming to obtain the marginal distribution of a specific variable.
In our case, it is applied to get the marginal probability that a given node takes a given color and it let us compute physical observables (e.g. internal energy, entropy, etc.).
The task at hand is to calculate the marginals of a graphical model. The naive algorithms that sum over all configurations takes a lengthy time of order Ω^N. A special structure in the underlying factor graph can be used to reduce the complexity, such as in the case of a tree factor graph where marginals can be calculated quickly through dynamic programming. Remarkably, this recursive process can also be done using the distributed message-passing belief propagation algorithm, which operates on messages associated with graph edges and updates them locally at graph vertices. BP has been discovered and utilized in various fields, including statistical physics and coding theory.
Although BP is proven to provide exact results on tree factor graphs, it can also be effective on graphs with loops as long as the underlying graph is locally a tree.

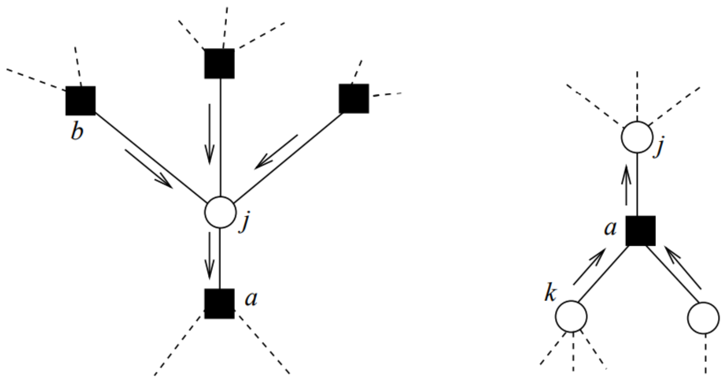


Figure 1.2. Left: the portion of the factor graph involved in the computation of $\psi_{j\to a}(x_j)$. This message is a function of the incoming messages $\chi_{b\to j}(x_j)$ with $b \neq a$. Right: the portion of the factor graph involved in the computation of $\chi_{a\to j}(x_j)$. This message is a function of the incoming messages $\psi_{k\to a}(x_k)$ with $k \neq j$. Image credit: [6]

However, its performance is limited when there are long-range correlations in the distribution, as typically happens at the occurrence of a phase transition.
The distribution of BP messages provides a large amount of information about CSP instances, helpful with the aim of this theoretical framework, which is to study the phase diagram.
Introducing a pair of new variables in the factor graph:

- $\psi_{j\to a}(x_j)$: marginal probability of x_j, considering the amputated factor graph in which only the edge connecting j and a has been removed.
- $\chi_{b\to j}(x_j)$: marginal probability of x_j, considering the amputated factor graph in which all edges around j have been cut, except for (j,b).

These may be visualized as the messages sent respectively from factor nodes to variables and viceversa. In this way, the BP equations, exact on tree factor graphs, are the following:

$$\begin{aligned} \chi_{b\to j}(x_j) &= \frac{1}{\mathcal{Z}^{b\to j}} \sum_{\{x_{\partial b\setminus j}\}} P_b(x_{\partial b}) \prod_{i\in\partial b\setminus j} \psi_{i\to b}(x_i) \\ \psi_{j\to a}(x_j) &= \frac{1}{\mathcal{Z}^{j\to a}} \prod_{b\in\partial j\setminus a} \chi_{b\to j}(x_j) \end{aligned} \tag{1.11}$$

Here, if $\partial b \setminus j$ is a set without elements, $\chi_{b\to j} = P_b(x_j)$; similarly, if $\partial j \setminus a$ is the empty set, $\psi_{j\to a}$ is a uniform distribution. In figure 1.2 are illustrated the role of these variables in BP on a factor graph.
The marginal probability of each variable x_i:

$$b_j(x_j) = \frac{1}{\mathcal{Z}^j} \prod_{b\in\partial j} \chi_{b\to j}(x_j)$$

If the factor graph is a tree, the prediction of the marginals b_i for the distribution $P(\mathbf{x})$ is exact. For graphs with loops, the approach is to look for a fixed point solution of equations (1.11). This can be done applying the algorithm of BP which consists in the employment of equations (1.11) themselves in a reiterative manner using time steps to update each variable. The iterative BP algorithm used to obtain marginal distributions is directly connected to the search of minima of the free energy. The BP equations may be gathered in a single group of equations inserting one equation in the other, obtaining:

$$\psi_{j\to a}(x_j) = \frac{1}{\mathcal{Z}^{j\to a}} \prod_{b\in\partial j\setminus a} \sum_{\{x_{\partial b\setminus j}\}} P_b(x_{\partial b}) \prod_{i\in\partial b\setminus j} \psi_{i\to b}(x_i) \tag{1.12}$$

1.5.1 Replica Symmetric cavity method

The replica-symmetric (RS) cavity method of statistical mechanics adopts a point of view which is very close to the previous one, but less algorithmic. This method is based on taking the average of the free energy over an ensemble of random graphs, in the thermodynamic limit.
The approach of this method is based on the procedure of computing the partition function recursively, by adding one variable node at a time. Equivalently, one may think of taking one variable node out of the system and computing the change in the partition function; for this reason, the name of this method comes from the idea of "digging" a cavity in the system [16].
The conceptual difference of the cavity method (respect to belief propagation) lies in some assumptions which may be summarized in the following way, namely the *replica symmetry assumption*; these consist in pretending that, for random graphical models, with ρ_a distribution of degree of nodes $|\partial i|$ and λ_j distribution of degree of factors $|\partial b|$, in the thermodynamic limit:

- There exists a solution to equations (1.11), or at least a quasi-solution, i.e. a set of messages such that the difference between right and left side of those equations is, in average, null in the large N limit.
- The solutions provide a good approximation of the marginals of the graphical model.
- The messages in this solution are distributed according to a density evolution fixed point.

The last condition assumes that the normalized variable to factor and factor to variable messages converge, in distribution, to variables $\{\psi, \chi\}$ solutions of the following distributional equations:

$$\psi = \Psi(\chi_1, \dots \chi_{a-1}) \qquad\qquad \chi = \mathcal{X}(\psi_1, \dots \psi_{j-1}) \tag{1.13}$$

where χ_i and ψ_i are respectively independent copies of χ and ψ, a and j are integers distributed according to ρ_a and λ_j.
In coding theory, the equations (1.13) are referred to as *density evolution* [6]; sometimes, this term is also applied to the sequence of random variables $\{\psi, \chi\}$. In probabilistic combinatorics, they are also called *recursive distributional equations.*
Generically, those equations cannot be solved in closed form; the message distribution is evaluated using numerical methods, in particular the algorithm of population

dynamics is largely employed. In the latter, the distribution of ψ or χ are approximated by a sample of N_s i.i.d. copies of ψ or χ, respectively. The error in the approximation goes to zero as $N_s \to \infty$ [6].
Now, applying the BP/cavity equations (1.12) to the vertex coloring problem, the probability measure coming from the Hamiltonian in equation (1.4) should be employed.
Denoting with $\psi_{s_p}^{i\to j}$ the probability that the node i takes the color $\sigma_i = s_p$ with $p \in \{1, \dots q\}$ in absence of the neighbouring node j, the belief propagation/cavity equations can be written [17]:

$$\psi_{s_p}^{i\to j} = \frac{1}{\mathcal{Z}_0^{i\to j}} \prod_{k\in\partial i\setminus j} \sum_{\{s_k\}} e^{-\beta\delta_{s_k,s_p}} \psi_{s_k}^{k\to i} \tag{1.14}$$

The quantity $\mathcal{Z}_0$ guarantees normalization of ψ_{s_p}. The equations (1.14) can be written in a simpler way using:

$$e^{-\beta\delta_{s_p,s_r}} = \delta_{s_p,s_r} e^{-\beta} + (1-\delta_{s_p,s_r}) = 1-(1-e^{-\beta})\delta_{s_p,s_r}$$

Solving the vertex coloring problem consists in finding groundstates of zero energy; this means we need to take the low temperature limit $\beta \to \infty$; considering that, equations (1.14) become:

$$\psi_{s_p}^{i\to j} = \frac{1}{\mathcal{Z}_0^{i\to j}} \prod_{k\in\partial i\setminus j} \sum_{\{s_r\}} (1-\delta_{s_p,s_r})\psi_{s_r}^{k\to i} = \frac{1}{\mathcal{Z}_0^{i\to j}} \prod_{k\in\partial i\setminus j} (1-\psi_{s_p}^{k\to i}) \tag{1.15}$$

where it has been used $\sum_{p=1}^{q} \psi_{s_p}^{i\to j} = 1$.
Applying the algorithm guided by the BP equations, if it converges, a fixed point is obtained. Therefore, it is possible to use the resulting messages to compute the Bethe free energy, yielding to the Bethe approximation of the free energy of the system. The algorithm fails in returning correct predictions in the presence of long range correlations (thus, entailing a wrong hypothesis in the application of Bethe approximation) and loopy factor graphs, in which the local structure is not equivalent nor similar to a tree graph. The expression in the equations (1.15) can be compressed in a shorthand notation: $\psi_{s_p}^{i\to j} = f\left(\psi_{s_p}^{k\to i}\right)$.
Using fixed point of equations (1.15), the entropy - the logarithm of the number of proper colorings - can be written:

$$S = \sum_{\{i\}} \log\Big[\sum_{p=1}^{q} \prod_{j\in\partial i} (1-\psi_{s_p}^{j\to i})\Big] - \sum_{\{ij\}} \log\Big(1-\sum_{p=1}^{q} \psi_{s_p}^{j\to i}\psi_{s_p}^{i\to j}\Big)$$

If the distribution of ψ is known, the entropy density or entropy per node $\frac{S}{N}$ can be computed; in the case of the random ER graphs ensemble, it is possible to write the cavity functional equation:

$$\mathcal{P}^{RS}(\psi_i) = \sum_{\ell=0}^{\infty} P_{deg}(\ell) \prod_{k=1}^{\ell} \int \mathcal{P}^{RS}(\psi_{k\to i})\delta(\psi_i - f(\{\psi_{k\to i}\}))d\psi_{k\to i} \tag{1.16}$$

Here $f(\cdot)$ is the shorthand notation introduced before, $P_{deg}(\ell)$ is the degree distribution expressed in equation (1.2). The prediction of the cavity method at the RS level for the quenched free energy (see equation (1.6)) is then obtained averaging the Bethe free entropy with respect to the message distribution $\mathcal{P}^{RS}(\psi_i)$ and the degree distribution $P_{deg}(\ell)$, taking into account the factor $-\beta$ that distinguishes the free energy from the free entropy. The equations of $\mathcal{P}^{RS}$ can be solved using population dynamics algorithm.

1.6 One Step Replica Symmetry Breaking method

Belief propagation's effectiveness hinges on the notion that the exclusion of a function node from a factor graph results in weakly correlated neighbouring variables with respect to the resulting distribution. However, this hypothesis may not hold either due to small cycles in the factor graph or distant correlations between variables. The latter scenario, even in factor graphs exhibiting a locally tree-like structure, leads to BP's failure. The emergence of long-range correlations marks a phase transition that divides a weakly correlated phase from a highly correlated one.
In addition to all the methods discussed so far, a technique named "one-step replica symmetry breaking" (1RSB) [16, 18] is used to get further description of the phase space. The study of the phase space is split in two ensembles: the random one, linked to ER graphs, and the so called *planted* ensemble; prior to delving into characteristics of the planted ensemble, which will be later discussed in detail, it's beneficial to outline the phase diagram of the corresponding purely random ensemble, associated to ER graphs. As the parameter of the phase space - in vertex coloring: the connectivity c as in equation (1.3) - fluctuates in random CSPs, various phase transitions occur.
When it is above the satisfiability threshold, there are no additional solutions; in the vertex coloring context, it is possible to say that above a certain connectivity, no random graph admits an acceptable configuration of colors which satisfy all the constraints. This threshold phenomenon is anticipated by a series of phase transitions that impact the solution space structure in the satisfiable phase. Statistical mechanics tools have discovered these structural phase transitions, which are comparable to those described in the mean field theory of structural glasses [19], and are primarily independent of specific random CSP criteria [6].
The specific analysis of the several phase transitions will be discussed later, but the important feature that should be presented here is the fact that, for some models, the set of solutions split in collections called clusters, which are pure Gibbs states. Zero energy paths can be found between solutions within the same cluster; for solutions belonging to different clusters, there exist barriers of physical potentials, whether energetic or entropic.
A formal definition of these clusters is exposed in [20].
It is necessary to notice that the structure and organization of pure states in such systems is far from being fully understood; any conclusion based on these assumptions should be interpreted with caution, and further research is needed to fully explore the nature of these kind of states in complex systems. However, in order to study those states, the 1RSB method has been introduced and exploited in statistical physics. About this method, there are also potential limitations and challenges: while it has been successful in certain cases, it is not a rigorous mathematical ap-

proach and can lead to incorrect predictions in certain situations. Additionally, there are still many unresolved questions and open problems in the study of Gibbs states on sparse random graphs.
In order to apply this method, the idea is to exploit the partition of the solution space into disjoint clusters $\mathcal{C}$, with three main assumptions:

- There exist exponentially many quasi-solutions of BP equations. The number of such solutions with free-entropy $\mathbb{F}(\psi) \approx N\Phi$ is (to leading exponential order) $e^{N\Sigma(\Phi)}$, where $\Sigma(\cdot)$ is the *complexity function.*

- The canonical measure $\mu(\mathbf{x})$, which usually takes the form of equation (1.8), can be written as a convex combination of Bethe measures that have short range correlations (SRBC), whose set is defined with $\mathcal{B}_{sr}$:

$$\sum_{n \in \mathcal{B}_{sr}} w_n \mu_n(\mathbf{x})$$

  where weights are related to Bethe free entropy $w_n = \frac{e^{\mathbb{F}_n}}{\mathcal{Z}_\mu}$ with $\mathcal{Z}_\mu = \sum_{n \in \mathcal{B}_{sr}} e^{\mathbb{F}_n}$.

- To leading exponential order, the number of SRBC-measures equals the number of quasi-solutions of BP equation: the number of SRBC-measures with free entropy $\approx N\Phi$ is also given by $e^{N\Sigma(\Phi)}$.

The complexity function $\Sigma(\Phi)$ is crucial in understanding how the measure μ breaks down into Bethe measures, given the three assumptions outlined. Introducing the *internal entropy* of clusters $\mathcal{C}$ as $s_i(\mathcal{C}) = \frac{\log \mathcal{Z}_\mathcal{C}}{N}$ where $\mathcal{Z}_\mathcal{C}$ is the number of solutions in $\mathcal{C}$, then $\Sigma(s_i)$ is defined as the logarithm of the average number of clusters with internal entropy s_i, divided by N.
As the amount of SRBC-measures with a particular free entropy density grows exponentially with the size of the system, addressing this issue through statistical physics is a good approach. Utilizing original problem's BP messages as new variables and the Bethe measures as new configurations is the principle behind 1RSB. In order to follow the statistical mechanics method, the distribution over SRBC-measures is defined through a parameter m named Parisi 1RSB parameter, which takes the role of an inverse temperature. In this manner, the partition function of this generalized framework is:

$$\mathcal{Z}_\mu(m) = \sum_{n \in \mathcal{B}_{sr}} e^{m\mathbb{F}_n} = \int e^{N[m\Phi + \Sigma(\Phi)]} d\Phi \tag{1.17}$$

In this way, each measure $n \in \mathcal{B}_{sr}$ is assigned to the probability $w(m) = \frac{e^{m\mathbb{F}_n}}{\mathcal{Z}_\mu(m)}$. Now, varying the parameter m it is possible to recover the full complexity function $\Sigma(\Phi)$. A saddle point evaluation of the integral in equation (1.17) lead to the *total free-entropy*:

$$\mathcal{E}_{tot}(m) = \lim_{N \to \infty} \frac{1}{N} \log \mathcal{Z}_\mu(m) = \sup_{\Phi} [m\Phi(m) + \Sigma(\Phi(m))] \tag{1.18}$$

The second equality is valid for $\Sigma(\Phi(m)) \geq 0$ since this condition ensures that clusters exist in the thermodynamic limit. The total entropy is determined by the

interplay between most numerous clusters, which have a large complexity, and bigger clusters with greater internal entropy.
The objective now is to calculate the 1RSB partition function by including contributions from different clusters. As it is possible to assume that long range correlations are neglectable in each cluster at RS level, then each cluster may be described with a set of messages, solution of the fixed point equations:

$$\psi^{\mathcal{C}}_{i\to k} = f(\{\psi^{\mathcal{C}}_{j\to i}\})$$

Hence, using the Bethe approximation, the partition function referred to clusters can be written:

$$\mathcal{Z}^{cl}_{\mu}(m) = \sum_{\{\mathcal{C}\}} e^{mN\Phi(\mathcal{C})}$$

where $\Phi(\mathcal{C}) = \Phi(\{\psi^{\mathcal{C}}_{i\to k}\})$ is the internal free entropy density of the cluster $\mathcal{C}$ computed in the Bethe approximation. In this way, the approach leads to an auxiliary graphical model which is very close to that of the original factor graph, speaking of topology of the graph itself. The 1RSB method consists in the computation of that Boltzmann distribution over Bethe measures, write it in the form of a graphical model, and use BP to study this model [6]: for random graphs, the distribution of 1RSB messages should follow the 1RSB functional equation.
Obtained a set of solutions (or approximated solutions) coming from BP algorithm, Bethe free-entropy of the auxiliary graphical model is obtained and in the thermodynamic limit, this leads to the 1RSB free-entropy density $\mathcal{L}(m)$, defined via $\mathcal{Z}_{\mu}(m) = e^{N\mathcal{L}(m)}$ using a saddle point evaluation. Finally, the complexity is computed through the Legendre transform of the free-entropy density in 1RSB $\mathcal{L}(m)$:

$$\mathcal{L}(m) = m\Phi + \Sigma(\Phi), \qquad \frac{\partial\Sigma}{\partial\Phi} = -m$$

As will be later explained in detail, the complexity function Σ is useful to characterize the phase space.
Overall, the 1RSB cavity method provides a useful tool for understanding the behavior of complex systems on sparse random graphs. By making certain assumptions about the pure state decomposition, physicists have been able to derive quantitative predictions about the satisfiability threshold and other properties of random constraint satisfaction problems. Although the method is not applicable without limitations, its successes have encouraged further exploration of the mathematical theory underlying sparse random graphs.

1.7 Phase transitions analysis

As it is now clear, CSPs of random nature like vertex coloring studied on ER graphs may undergo phase transitions the same way it happens in statistical mechanics of critical phenomena. The parameter to study the transitions is the mean connectivity and one of the transitions to discuss is the SAT-UNSAT one, i.e. the critical value of c above which the system does not present any solution inside the random ensemble. In the vertex coloring context, this thesis is focused on the problem of answering

whether a graph is colorable (SAT) using a given number q of colors (later on, q-coloring classification will refer to the latter definition). The hardness of this question depends of the value of connectivity, as will be made clear later on.
Starting with low values of connectivity c, as it is defined in equation (1.3), the type of solution is paramagnetic, a generic instance is satisfiable, the system can be studied using the BP approach [21] and the solution can be found even by a simple zero-temperature Metropolis algorithm [22]. The first transition is denoted as the "dynamic" one, it corresponds to a density of dynamical arrest [22] and above this c_d the space of solutions breaks up in many clusters. Here, local Monte Carlo Markov Chain strategies are useful at these connectivity values [20].
As explained before, the 1RSB method introduces the Parisi RSB parameter m which is useful to focus on clusters of a given dimension, weighting them by their free energy to the power of m [21].
When the space of solutions split in clusters, the total entropy is dominated by the contribution of the clusters of internal entropy $\mathcal{E}^*$ which satisfy equation (1.18). If $\Sigma(\mathcal{E}^*) > 0$, there is an exponential number of clusters contributing to the total entropy. On the contrary, if $\Sigma(\mathcal{E}^*) = 0$ there are two possible cases: either the mean connectivity $c < c_d$ resulting in only one giant cluster, or the connectivity $c > c_c$ referring to the next phase, which is called the "condensed phase" where many clusters disappear and only a sub-exponential number of clusters survives.

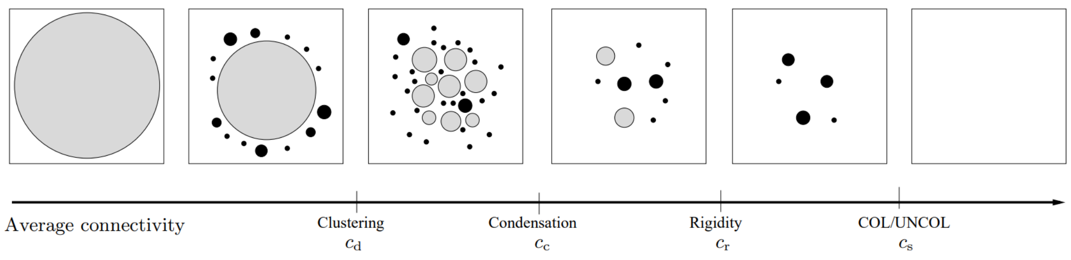


Figure 1.3. Representation of the phase space of states undergoing phase transitions. Black circles are the frozen clusters. Image credit: [17]

An interesting phenomenon called freezing appears during the clustering phase, where a fraction of variables are allowed only one color: these variables are called *frozen* variables. At some connectivity before or after c_c it is possible to identify the rigidity (*freezing*) transition at c_r where any cluster presents frozen variables, as shown in image 1.3 where the latter are painted black. Above the rigidity transition, when the number of nodes is large, the task of looking for a solution becomes algorithmically hard and solutions are found in non-polynomial time [23].
The last transition happens at c_s and no more random solutions are present for $c > c_s$. In particular, while the characterization of the phase space for thresholds c_d and c_c is done applying 1RSB method at $m = 1$, the satisfiability threshold is obtained from the computation of $\Sigma(m = 0) = \max_\Phi \Sigma(\Phi)$: this quantity gives the exponential rate of growth for the total number of clusters of solutions, independently of their size. When graph's mean connectivity increases, that number decreases: c_s is the highest c such that $\Sigma(m = 0) > 0$ holds.

1.7.1 The Planted set of solutions

To evaluate the performance of an algorithm, studying CSPs, it can be useful to have a set of hard-to-solve instances such that the constraint is satisfied. However, finding this set is difficult, due to the nature of the instances themself.

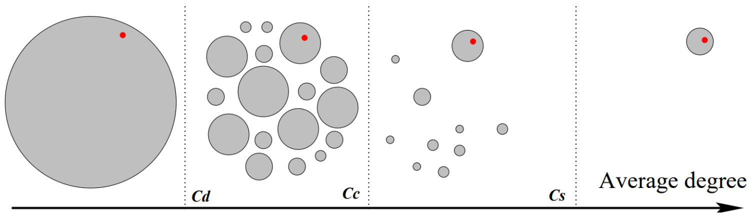


Figure 1.4. Representation of the phase diagram: the red dot refers to the planted cluster. Image credit: [21].

To solve this, there is a nice and straightforward method called "planting": instead of looking for a q-colorable (i.e. colorable in such a way that it satisfies the constraint using only q colors) random graph, a set of nodes are colored with q colors and only then all the edges are inserted, randomly but only between nodes of different color.
This is the "quiet" planted solution discussed in [21], in the sense that the planted ensemble shares the same properties of the purely random ensemble, but only for connectivity values $c < c_c$. Indeed, many numerical experiments indicate that both planted and random graphs present the same algorithmical difficulty; figure 1.6 shows results in such sense using an heuristic algorithm called Walk-COL [17].
This equivalence is rather important since this thesis aims to assay the computational hardness of q-coloring classification.
As shown in figure 1.4, the phase diagram of the planted ensemble is pretty similar to the random one. For connectivity values $c_d < c < c_c$ there is an exponential number of clusters, the planted one is disguised and equivalent to the random ones. Conversely, for $c > c_c$ the dimension of the planted cluster is larger than the dimension of the random clusters and it becomes larger than the total size of the residual ones.
At $c > c_s$ the planted cluster still remains: it is clear from the planting procedure that it is always possible to generate a q-colorable graph for any value of vertices and edges, thus for any connectivity.
This gives the opportunity to study the computational hardness of vertex coloring and q-coloring classification at any value of c.
In figure 1.5, the entropy of the planted ensemble for $q = 5$; the entropy of planted graphs is greater than the entropy of random ones for $c > c_c$ (it is the opposite, speaking of free energy), while it is the same for $c < c_c$.
To conclude, it is interesting to notice that the phase transitions, as it is generically in statistical mechanics of critical phenomena, are well defined in the thermodynamic limit.
Clearly, numerical simulations are done using a finite size (yet large) of the system; in this manner, the results obtained are valid in the regime where finite-size effects are negligible. For values of $N \geq 100$ - that is the regime at which, in this thesis,

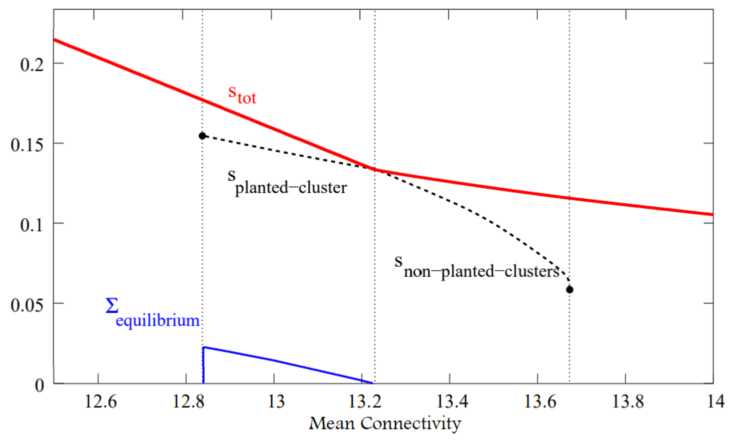


Figure 1.5. Entropy plotted as a function of connectivity c on the 5-coloring on the planted ensemble; total entropy (in red) with the subdominant part (dashed). $\Sigma_{equilibrium}$ is the logarithm of the number of dominant clusters. The dotted vertical lines represent in order: the clustering transition c_d, the condensation transition c_c and the satisfiability transition c_s. It is interesting to notice how the (quenched) entropy of random clusters goes to $-\infty$ at $c = c_s$, where the last non-planted cluster disappears. Image credit: [21].

numerical experiments are done - those effects can be considered negligible.

1.7.2 The transition of algorithmic difficulty

Despite the ensemble does not present any characteristic mutation beyond c_s, it is clear - as it will be exhibited in the succeeding chapters - that for $c \gg c_s$ the q-coloring classification becomes trivial and easy to solve [26].
As mentioned in the preceding paragraph and based on what is known from scientific literature, no existing algorithm is able to find solutions in the frozen clusters in polynomial time [17, 21, 23, 27], and since in the region near to the colorability threshold, i.e. rigidity phase, all clusters are frozen this provides a lower bound for the algorithmically hard phase.
These observations lead to consider a new phase transition, above c_s and below those high connectivity values for which the task becomes easy, where the algorithms switch from the hard phase to the numerically easy one.
As usual, this transition is identified in the thermodynamic limit.
As it can be checked in figure 1.6, BP algorithms, for $q = 5$, find this transition value at connectivity $c = 16$.
Interestingly, in [21] arises that both the planted and the purely random ensembles admit the so-called liquid solution whose stability undergoes a transition at $c_l = (q-1)^2$ and this value appears to coincide with the computational hard/easy transition. Another algorithm that has been employed in this thesis, which is called simulated annealing, finds numerically that this algorithmic phase transition happens - even here for $q = 5$ - at $c = 18$.
An interesting result of this thesis, which will be discussed in the last chapter together with the simulated annealing, is about the investigation of this algorithmic

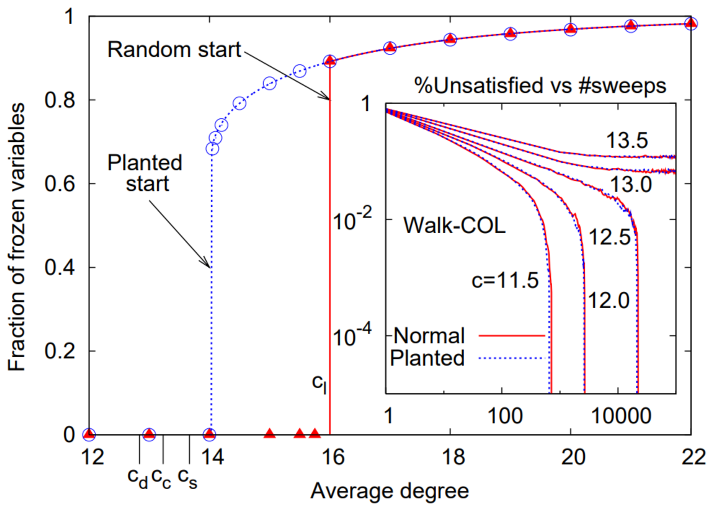


Figure 1.6. Fraction of variables frozen in their planted colors in 5-coloring of a graph with $N = 10^5$. Data obtained from the BP fixed point when initialized randomly (full triangles) and in the planted configuration (also called the whitening [24], empty circles), compared to the theoretical predictions [17, 25] (full and dashed lines). For $c > c_l = 16$, BP converges spontaneously to the planted fixed point. For $c < 14.04$ [25] there are no frozen variables in the planted cluster. Inset: Fraction of monochromatic edges versus the number of sweeps of the Walk-COL algorithm [17] in 5-coloring of a purely random and a planted graph, both $N = 10^5$. Quiet planting does not seem to affect the computational hardness in the region $c < c_s$. Image credit: [21].

transition for neural networks.
Since for $c > c_s$ there is no satisfiable (i.e. colorable) random graph and planted graphs can be easily generated, the neural network may be trained using a combination of known SAT and UNSAT instances.

1.8 The high dimensional inference limit

In the field of statistical inference, the process of deriving properties of a distribution from observed data, there is an interesting concept which may be linked to the transition addressed in this thesis. To lay out an inference problem in a structured manner, a given group of random variables $\mathbf{x} = \{x_1, \dots x_N\}$ can be analyzed via some limited observations or measurements $\mathbf{y} = \{y_1, \dots y_M\}$. Discussing about inference, the objective is to determine the values of the variables $\mathbf{x}$ using the potentially uncertain and imprecise details included in the observed data $\mathbf{y}$. Generally, statistical theory focuses on the scenario where there is a finite number N of variables or parameters to be estimated and a diverging number of observations M. However, it is important to extract as much useful information as possible, since part of data coming from the set $\mathbf{y}$ contribute as noise. Even here, it is interesting to study the system in the limit as $N \to \infty$. In this case, it is much more challenging to separate the useful information from noise. This is referred to as *high-dimensional statistical theory*, which requires attention to both solvability and computational efficiency [28]. In this framework, the link with statistical physics is particularly striking as it is designed to describe large assemblies of small elements. Focusing on the analogous to the thermodynamic limit in physics, which is $N \to \infty$, the fixed ratio (since M diverges too) $\alpha = \frac{M}{N}$, which may be called *signal-to-noise* ratio, becomes the useful parameter to study the behaviour of the system.

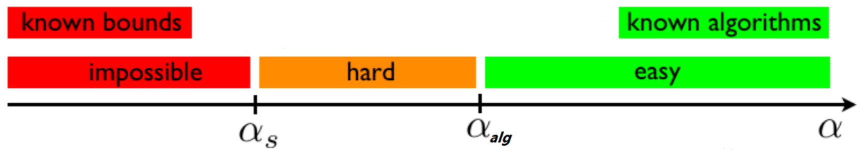


Figure 1.7. Schematic representation of a typical high-dimensional inference problem. Image credit: [28].

When α is less than α_s, successful inference of the variables using any algorithm is impossible due to inadequate observational information, therefore it is the solvability transition. In contrast, for $\alpha > \alpha_{alg}$, computationally efficient inference algorithms are available. In the intermediate range, successful inference is technically feasible but considerably more challenging.
In the context of vertex coloring, this statistical inference standpoint consists in obtaining the planted solution, which is possible for $c > c_c$, and here M is the number of edges, N is the number of nodes. In a general inference problem parametrized by α, dealing with random instances of a CSP, it is possible to consider the parameter itself as the ratio between the number of constraints and the number of variables. In a graph, that ratio is linked to the number of edges divided by the number of nodes; looking at equation (1.3) it is evident the relationship between the mean connectivity and the mentioned parameter.
Lastly, α_{alg} shares the role of c_l as a threshold representing the algorithmic hard/easy transition.

Chapter 2

Artificial Neural Networks

A computer would deserve to be called intelligent if it could deceive a human into believing that it was human.

Alan Mathison Turing

The first idea behind neural networks, in the field of machine learning, was conceived thinking about the operating of the nervous system. In a net of biological neurons, information is passed using electrical impulses following the rule known in physiology as “all-or-none”: this means that a single cell will communicate sending the message (always with the same amplitude) only if it received enough input signal to exceed a threshold.
The connection between this rule with boolean variables led a neurophysiologist and a logician to consider the activity of neurons as something that can be represented with a logical proposition [29]; they proposed in 1943 the first mathematical model - which took authors’ names, being called McCulloch-Pitts neuron - describing in a simplified way the logic behind the functionality of a brain cell.
This model consists in a logic gate that gathers information using a weighted sum of input variables and it gives a binary output. In the same way, the biological neuron determines whether or not to fire the signal through the axon, based on the totality of impulses coming from overall dendrites - the branched extensions that nerve cells use to connect each other.

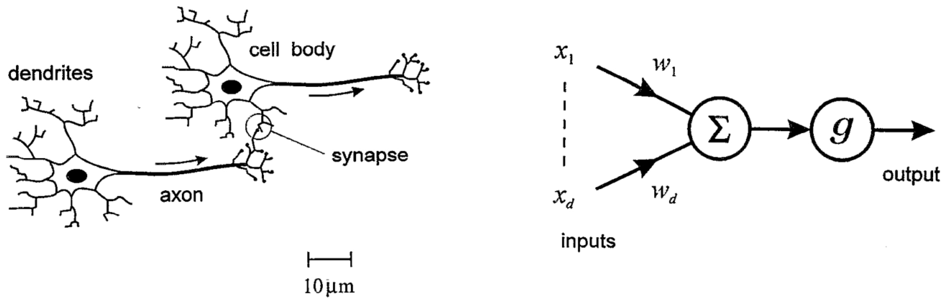


Figure 2.1. Comparison between the biological neuron structure and the schematic representation of McCulloch-Pitts neuron. Images credit: [30]

A simple task in which McCulloch-Pitts neuron could be directly employed is binary classification: considering a set of labeled variables that can be distinguished in two groups, based on the way the weighted sum of input values is performed, the neuron could assign a new variable to one of the two groups looking at its binary output.
Starting from McCulloch-Pitts model, the formal definition of an artificial neuron in the context of binary classification is pretty simple [31]: denoting the input values as a vector $\vec{x} \in \mathbb{R}^n$ and given a single scalar weight w_i for each input value, $\vec{w} \in \mathbb{R}^n$, it is possible to define the net input z as the scalar product of these two vectors.
Then, the artificial neuron evaluate if the net input is sufficient to reach the threshold using the - so called - "activation function" g depending on z, where in typical binary classification context $g(z)$ gives an output that easily suggests the prediction of the model. As shown at right in figure 2.1:

$$z = \sum_{k=1}^{n} w_k x_k \qquad \hat{y} = g(z) \longrightarrow \text{ prediction based on } z \text{ value}$$

Following this definition, a neuron is just capable to decide how to classify the input based on initial values of $\vec{w}$ and the shape of the activation function. The actual faculty to learn is not implicit in this definition; Rosenblatt in 1957 proposed a model based on McCulloch-Pitts neuron with the aim to fill this lack. Calling "perceptron" a system which learns to recognize similarities or identities between 'optical, electrical or tonal informations' [32], he presented an algorithm that computes the optimal weight coefficients in order to let the neuron better decide whether to fire the signal.

2.1 How a machine learns

There are three main kind of machine learning: supervised learning, unsupervised learning, and reinforcement learning. As it will be clear, supervised learning is based on a labelled dataset and the evaluation of a cost function using those labels. Unsupervised learning is done on dataset without any prior extra information so it is useful to learn and recognize hidden structures on data.
Finally, reinforcement learning is structured defining the neural network as an agent interacting with an environment. The latter gives to the agent the information about the present state and the agent answers with an action, modifying the state.
The learning procedure is guided by a reward system that is activated when the final state of the agent corresponds to the sought one - an example of reinforcement learning is a chess program, where the environment is the chessboard, states are the possible moves and the reward is given if the game is won (check mate). From now on, only the first type will be discussed so that referring to learning will imply supervised learning.
For an artificial neural network, the process of learning consists in the determination of weights, in a similar way as it happens with parameters of the polynomial best fit calculated on a set of data. In the latter case, data represents the ground truth pattern points and the aim is to reproduce it using a polynomial function; the problem is to find the best tuning of parameters of such a function in order to minimize the distance between the curve and the data points.

To deal with this problem quantitatively, it is introduced an error function - typically the mean squared deviation - that represents the aforementioned distance.
About neural networks, the particular choice of the loss function is based on the task it is needed to solve. In this case, the loss is computed between the prediction of the neural net model and the target data, considered ground truth like data points in the example of best fitting.
Neural networks can be applied in many different tasks and the dataset could be composed of any sort of data, from images to written text. It is clear that the mentioned dataset must be represented in such a way it is possible for the neural network to take it as an input: generically collections of data are represented by a tensor $\mathbf{x} \in \mathbb{R}^{n_1 \times \cdots \times n_k}$ and labels $\mathbf{y} \in \mathbb{R}^{n_1 \times \cdots \times n_t}$ that must have the same shape of neural network's output.

2.1.1 Training procedure

The main phase in which the neural network learns, gathering information from the input dataset, is called training.

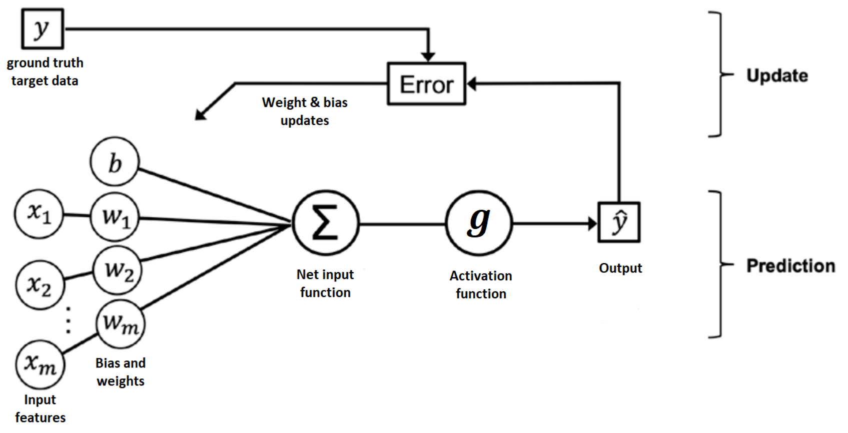


Figure 2.2. Schematic training procedure applied using a perceptron. Image credit: [31].

This procedure starts taking input values $\vec{x}$ and initializing weights $\vec{w}$ - the way it is done will be discussed later. Then the net input is computed, adding an extra trainable parameter b (just like the weights) called bias. The activation function g is applied to the net input and this gives the first prediction of the model; the training consists in the minimization of the loss function computed on the latter prediction $\hat{\mathbf{y}}$ and the ground truth target data $\mathbf{y}$. Weights are then updated using a method called "backpropagation" and they are then utilized again to evaluate the new net input on the same dataset. Every time the neural network gives a new prediction, it is utilized to computed the loss and the loss is utilized to calculate new weights which will give a better prediction. In this way, the neural network is analyzing the same training dataset many times, each of those gives a new evaluation of weight parameters. This approach is directly applied with a for loop: every time the input dataset has been entirely passed to the model, one loop cycle is completed - this single loop is often referred to with the name of *epoch*. Thus, the number of epochs is an hyperparameter for the program and represents how many times the neural network is supposed to repeat the learning procedure on the whole dataset. The optimal value depends on the details of the problem and depends on the architecture of the model itself. A summary of the training procedure is displayed in figure 2.2.

2.1.2 Loss functions

There are many different loss functions that can be adopted in the framework of neural networks. Recalling the comparison of the training procedure with the task of finding the best fit of a parametrized curve on a set of points, a first example of loss function is the mean squared error (MSE) even in this case. The expression is the same:

$$\mathcal{L}_{MSE} = \frac{1}{M}\sum_{m=1}^{M}(\hat{y}_m - y_m)^2$$

This is the most common choice when the output model $\hat{\mathbf{y}}$ and target values $\mathbf{y}$ are vectors in $\mathbb{R}^M$. Sometimes this can lead to deal with problematically high values of the loss functions, contributing to complicate the learning procedure during the computation of weight parameters. Therefore, to mitigate this behaviour, it is possible to use the mean absolute error (MAE):

$$\mathcal{L}_{MAE} = \frac{1}{M}\sum_{m=1}^{M}|\hat{y}_m - y_m|$$

Dealing with classification problems (as in the case of the vertex coloring problem, where a graph is either colorable or not), the output of the model must be interpreted as the probability that the input belongs to a certain category.
Indeed, the dataset should contain both data - used to compute the net input - and labels - used to compute the loss and quantify how much the prediction is far from the correct answer.
In this case, ground truth labels can be expressed using the representation called "one-hot". This vector $\vec{h}$ is defined in the following way, if the target data is assigned to a specific k category between all the L possible ones:

$$\vec{h} \in \{0,1\}^L \qquad \text{such that} \qquad h_i = \delta_{i,k} \qquad \text{for} \qquad i = 1, \dots L$$

In this representation, the prediction of the neural network will be a vector $\vec{y} \in [0,1]^L \subset \mathbb{R}^L$ in which every component i corresponds to the probability (obviously expressed as a number between 0 and 1) that input belongs to category i.
That being said, due to the fact that each computation in the loss function deals with numbers in $[0,1]$ interval, MSE/MAE are not useful. Instead, there is a loss function that maps $[0,1] \longrightarrow \mathbb{R}$ so it is a better choice for this case. That function is the cross entropy and the expression is the following, for a dataset with M different input data and labels:

$$\mathcal{L}_{CE} = -\frac{1}{M}\sum_{i=1}^{M}\sum_{k=1}^{L} h_k^i log(\hat{y}_k^i)$$

In the particular case of two categories - as in the vertex coloring - the one-hot representation becomes obsolete: using a scalar $h \in \{0,1\}$ is sufficient, referring to a category with either 0 or 1. This function is called binary cross entropy (BCE):

$$\mathcal{L}_{BCE} = -\frac{1}{M}\sum_{i=1}^{M}\Big[h^i log(\hat{y}^i) + (1-h^i)log(1-\hat{y}^i)\Big]$$

In this way, the loss becomes a sum of logarithms which is a function that maps $[0,1] \longrightarrow (-\infty, 0)$ and that is the purpose of the minus sign inside the definition.

2.1.3 Backpropagation

The key ingredient of a supervised learning is in the minimization of the loss function aimed to update the weights. This computation starts from the first output of the neural network, performed using the initial weights. This is called forward pass and consists in the simple application of the model to input data.
Given the output, the procedure to minimize the loss is called backward pass and will be now presented. The analytical computation of the minimum using calculus methods would become rather complex, considering the non-linearity of activation functions and the presence of more than one neuron in the network.
Therefore, it is better to build an algorithm that can be implemented numerically, a method composed of steps in which the weights and biases are updated with values such that $\mathcal{L}(\mathbf{w}_{n+1}, \mathbf{b}_{n+1}) < \mathcal{L}(\mathbf{w}_n, \mathbf{b}_n)$.
The gradient of the loss function (with opposite sign) gives the right solution and the amplitude of each step is monitored with a constant η usually referred to as "learning rate":

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{w}}$$

$$\mathbf{b}_{n+1} = \mathbf{b}_n - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{b}}$$

The details of the computations are inside the implicit expression of the partial derivatives.

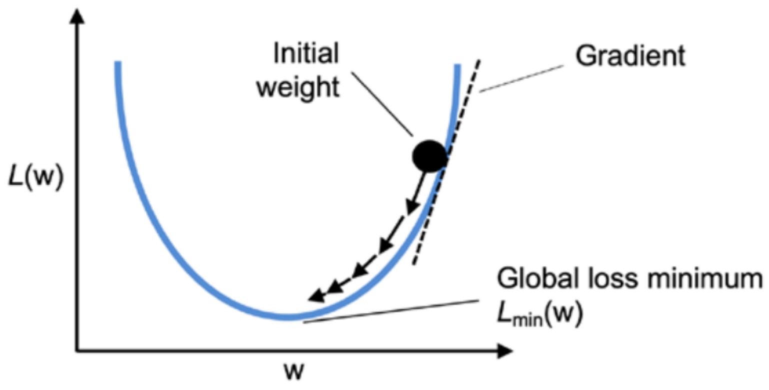


Figure 2.3. Backward steps for a parabolic loss function. Image credit: [31].

2.1.4 Optimizers

The expression used in the last two equations refers to the simplest way to update weights. If the dataset is divided in smaller parts, usually called batches, then at each iteration the model is applied on a single batch and the loss function is computed on that smaller amount of data.
However, ideally the backpropagation should be computed on the total loss over the whole dataset. This is why the fragmentation of dataset leads to treat the loss evaluated as a random variable. In this context, the gradient function in the update expression becomes a random variable too; the optimizer here takes the name of Stochastic Gradient Descent (SGD) and it is considered to be the basic method (therefore seldom) chosen to update weights. This stochastic feature presents itself as a source of noise which contribute to speed up the learning procedure: the descent path followed along the shape of the loss expressed as a function of parameters towards its global minimum is disturbed by fluctuations but converges faster. The simple gradient descent computed on the whole dataset for each step employs a higher number of epochs to reach the minimum. Furthermore, the deterministic algorithm is more likely to get stuck in a local minima than the stochastic one.
A different approach is SGD with momentum, firstly introduced in [33] where the gradient is used to update the velocity of the point in weight space instead of its position:

$$\Delta \mathbf{w}(t) = \alpha \Delta \mathbf{w}(t-1) - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}}(t)$$

In Pytorch's optimizers library, the class *torch.optim.SGD* offers the option to add momentum (set to zero by default).
Another method is to modify the learning rate for each parameter based on the historical gradients that have been observed during training. Specifically, in AdaGrad (that stands for *adaptive gradient algorithm*), the sum of the squares of the gradients with respect to each parameter is accumulated over time, and this sum is used to scale the learning rate for that parameter. Parameters that receive large gradients will have a smaller learning rate, and parameters that receive small gradients will have a larger learning rate. RMSProp (for Root Mean Square Propagation) is also a method in which the learning rate is adapted, in a way to prevent it from becoming too small - which is a potential drawback of AdaGrad. One of the most common optimizers nowadays is Adam, short for *adaptive moment estimation*, since it incorporates techniques from both AdaGrad and RMSProp.
Introduced in [34], remembering that the loss function is considered as a random variable, the interest is to minimize its expected value $\mathbb{E}[\mathcal{L}(\mathbf{w})]$ with respect to parameters $\mathbf{w}$. As explained in the original article, the algorithm updates exponential moving averages of the gradient $\mathbf{m}_t$ - i.e. momentum - and the squared gradient $\mathbf{v}_t$ (such as in AdaGrad) where the hyper-parameters $\beta_1, \beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages.
The moving averages themselves are estimates of the first moment (the mean) and the second raw moment (the uncentered variance) of the gradient. These first and second moments are then corrected removing bias; the unbiased estimates are expressed with $\hat{\mathbf{m}}_t, \hat{\mathbf{v}}_t$. Adam is the algorithm employed in every code written during this thesis project; it is implemented in Tensorflow and Pytorch in the same way

with the following equations, inserted in a while loop (cycling on t) that ends up when parameters converge to the updating value:

$$
\begin{aligned}
&\mathbf{g}_t = \frac{\partial \mathcal{L}_t}{\partial \mathbf{w}_{t-1}} \\
&\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \\
&\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \\
&\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t} \\
&\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t} \\
&\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}
\end{aligned}
$$

The small parameter ϵ is inserted to avoid division by zero to happen. Notice that in the expression of $\hat{\mathbf{m}}$ and $\hat{\mathbf{v}}$, the corrections made with β_1 and β_2 have more influence at first steps. For large t we have β_1^t, $\beta_2^t \simeq 0$, so it is clear that after many time steps, biased and unbiased estimators coincide. The profound influence of this algorithm inspired multiple newer, less well-known momentum-based optimization schemes.

2.1.5 Activation functions

There are many possible activation functions to be inserted in the training algorithm, just like loss functions, the most suitable depends on the task. The activation function, as well as a stack of many neurons interacting, gives non-linearity to the neural network, in order to better adapt to data. The expression of activation functions can be rather simple; the all-or-none rule that biological neurons follow can be described using the step function, with threshold set to zero:

$$
g(z) = \begin{cases} 1 & for\ z \geq 0 \\ 0 & for\ z < 0 \end{cases}
$$

Another simple function is the linear one: $g(z) = kz$, where the constant k is independent of the input and weights - for $k = 1$ the function becomes the identity and it has no effect. Generally, the latter function is suitable only for simple tasks; derivative in the linear case is a constant and this may cause hindrance in backpropagation [35].
About the step function, the derivative is always zero (except for $z = 0$) and this implies that the modern backpropagation algorithm cannot be applied. Indeed, this activation function was initially used in the rudimental neural networks composed of single perceptrons. In those cases, weights were updated using the - so called - perceptron learning algorithms introduced by Rosenblatt in [36].
Still simple but quite more useful is the Rectified Linear Unit function (ReLU):

$$
ReLU(z) = \begin{cases} z & for\ z \geq 0 \\ 0 & for\ z < 0 \end{cases}
$$

It can also be expressed as $ReLU(z) = \max(0, z)$ and despite it is a combination of the linear and step functions, this feature leads to have a network with some neurons activated and some others deactivated.
This helps the convergence of gradient descent towards the global minimum of the loss function and it is widely used in many neural networks. The limitation of this function is that it may contribute to get null gradient; this can lead to turn off some neurons and leave them deactivated during training. If this problem is present, a solution is given by the leaky ReLU function defined as $g(z) = max(\epsilon z, z)$ where $\epsilon \sim 10^{-1}, 10^{-2}$ in order to restrict the behaviour caused by ReLU when net input is negative.
Whenever the output of the model should represent a probability, as in the case of one-hot or binary classification, a good activation function outputs a value (or list of values for multi-class) in $[0, 1]$ interval. To obtain this, let's consider the log-odds function:

$$f(p) = \log\left(\frac{p}{1-p}\right)$$

where the ratio $\frac{p}{1-p}$ represents the odds related to probability $p \in [0, 1]$, in this way $f(p) : [0, 1] \subset \mathbb{R} \longrightarrow \mathbb{R}$. Considering that we need a function that maps net input $z \in \mathbb{R}$ into the probability interval, it is useful to look for the inverse of log-odds. This is the case of the sigmoid:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

That output can be concatenated with a threshold function which assigns the input to one of the two classes.
A similar solution that may replace the sigmoid is the hyperbolic tangent function:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

It has a shape pretty similar to the sigmoid, as displayed in figure 2.5.
The main difference is that it gives values in range [-1,1], instead of [0,1], and that its derivative is more peaked near $z = 0$ (see figure 2.4), therefore it is useful to obtain higher gradient values and increase the learning steps.
For a classification problem with more than two classes, as the label vector $\vec{h}$ has length equal to the number of classes in one-hot representation, equivalently the output of the net should be a vector of the same length with probabilities for each element, acting as the actual probability to belong to a given class.
More formally, for each $i = 1, \dots L$, as the label $h_i = \delta_{ik}$ of a given data in class k, the output will be $\sigma_i = p(i)$ where with $p(i)$ it is intended the probability of given data to belong to class i. This is done using the function called softmax:

$$\sigma_i = \frac{e^{z_i}}{\sum_{j=1}^{L} e^{z_j}}$$

Here, the net input (given a neural network shaped to output L different net inputs z_i - e.g. using L neurons) is inserted in an exponential in order to have small numbers for $z < 0$ - similarly to ReLU/Leaky ReLU - and a rapid growth to mark

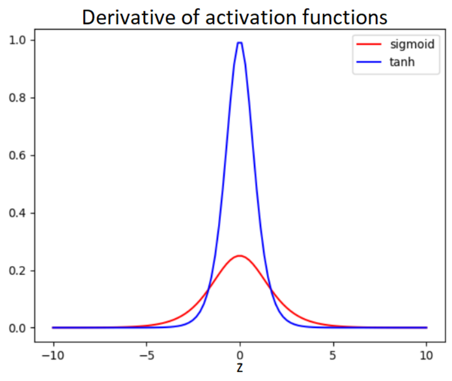


Figure 2.4. Comparison between derivatives of the Sigmoid and Tanh.

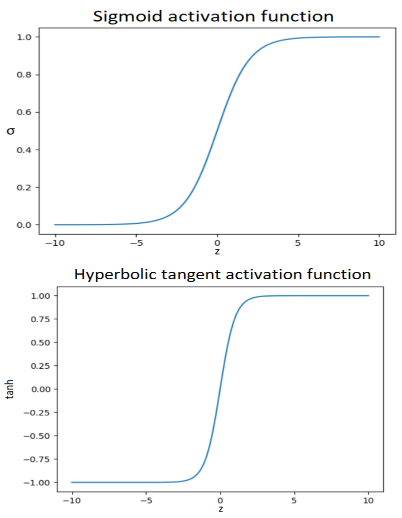


Figure 2.5. Graphs of the two activation functions: Sigmoid and Tanh.

the difference between probability predictions of different classes. The denominator is simply a constant function that guarantees normalization, so that the output value $\sum_i \sigma_i = 1$, i.e. $\sigma_i = p(i)$ is a probability.

2.1.6 Weight initialization

The values of weights at first step of training influence the performance of the learning procedure due to the shape of loss function and activation function. There is no standard prescription about the best initialization of weights, it depends on the architecture of the neural network. For a simple structure, initialization is secondary to the role of tuning parameters such as the learning rate; in the first models, it has been used the random uniform or random normal weight initialization.
Lately, it has been shown that these choices leaded to poor model performance. In [37], Glorot and Bengio studied empirically the effect of initialization with the objective of maintaining balanced activation functions and variances of back-propagated gradients during training steps and through the space of network parameters, in order to obtain an overall steadier network.
They proposed the following expression to adopt, in case of uniform distribution ($\mathcal{U}$) of weights:

$$\mathbf{w} \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right)$$

Considering the network as a stack of many neurons connected to each other, here n_{in} is the number of input neurons multiplied by the weights while n_{out} is the number of output neurons that feed into the next group of neurons. Using the normal distribution, it is suggested to use zero mean value $\mu = 0$ and a standard deviation equal to:

$$\sigma = \frac{\sqrt{2}}{\sqrt{n_{in} + n_{out}}}$$

This normalization takes Glorot's first name Xavier and is widely used in deep neural networks, although there may still be some situations where other choices are preferable [38].

2.2 Training implementation

In the last paragraphs it has been exposed the core algorithm behind the learning procedure for a neural network. Usually libraries adopted in the implementation of training programs have automatized schemes to perform backpropagation (the actual application of all the backward passes) in the optimal way, given that this undertaking is computationally intensive, differently from the forward pass. In Pytorch, the whole process takes into account only two lines of code - lines 17 and 18 in the code displayed below.

```
 1  import torch
 2  import mymodel from ModelClass_file
 3
 4  #num_epochs defined with an integer
 5  #Dataset imported as an iterable
 6
 7  model = mymodel() #it could take inputs
 8  optimizer = torch.optim.Adam(model.parameters())
 9  loss = torch.nn.BCELoss()
10
11  for epoch in range(num_epochs):
12      for batch in Dataset:
13          output = model(batch.feature, batch.edge_index)
14          loss_value = loss(output, batch.label)
15
16          #backpropagation
17          loss_value.backward()
18          optimizer.step()
19          optimizer.zero_grad()
20
21  torch.save(model.state_dict(), 'model_trained.pt')
```

Listing 2.1. Training code implementation

In listing 2.1, optimizer and loss function are chosen as the same adopted in every code run. Even the input of *model* in line 13 is simulated to be a graph, given that it is expressed as a *feature* and an *edge_index*. The last one is the data structure of a graph: a pair of lists in which every element corresponds to the number of a node; each couple, composed of the n^{th} element in both lists, is a pair of nodes linked with one edge. That is the only information needed to construct the graph, other than the number of nodes (which is the maximum value, representing the node label, inside the two lists) and the number of edges, equal to the length of those lists, indeed.
Those lines of code are not directly executable since are written with the aim to briefly expose the structure and the logic behind the usual training implementation. At line 17, listing 2.1 presents *loss_value.backward()* that takes the loss value and compute gradients with respect to weight parameters, including bias - as expected in every GD algorithm. The gradient is computed and saved in each tensor involved

during backpropagation: the values are added up and stored in *torch.tensor.grad* attribute.
To do that in Pytorch, tensors should have the attribute *torch.tensor.grad = True.* The next line of code updates parameters depending on the defined optimizer; once that it is done, gradients of the loss computed on that batch, in that epoch, are no more useful and must be deleted in order to make space for next iteration.
Not deleting cumulated gradients with the call *optimizer.zero_grad()* will lead to a bad computation during training procedure. Finally, in line 21 the command *torch.save()* stores all the final weight values in a *.pt* file. In this way, the model has learned to achieve the task or give a prediction on a dataset that has never seen before; this means that the neural network is ready to be employed, it is only needed to load weights from that file on the same model.

2.3 A simple neural network: multilayer perceptron

So far it has been explained the functioning of a single neuron. This can be summarized with the definition of net input $z(\vec{x}, \vec{w})$ and the application of a differentiable (in order to apply backpropagation) activation function $g(z)$. The next step is to consider many neurons deployed to follow the learning procedure: the same input $\vec{x}$ is given to each neuron, but each neuron has its own weights and will output a particular net input. In this way, this collection of neurons forms the structure called layer:

$$z_k = \sum_{j=0}^{N} W_{kj} x_j \qquad \longrightarrow \qquad \hat{y}_k = g(z_k)$$

In such a manner, the input of N data is processed by the layer which produces M output values $\vec{\hat{y}} \in \mathbb{R}^M$. Note that in this way, as the neuron is equipped with a vector of weights, here layer's learnable parameters are collected in a matrix $\mathbf{W} \in \mathbb{R}^{M \times N}$. Setting $x_0 = 1$ adds bias to weight parameters.
Now consider the M output values as new inputs for another layer. In this way, it is designed the simplest neural network known as multilayer perceptron (MLP) as a stack of many layers. To express more formally the computations involved in a MLP with two layers, starting from raw data $\vec{x} \in \mathbb{R}^N$ and calling the overall output of the network $\vec{u} \in \mathbb{R}^M$:

$$u_k = f\left(\sum_{j=0}^{H} W_{kj} t_j\right) \qquad \text{where} \qquad t_j = g\left(\sum_{r=0}^{N} \tilde{W}_{jr} x_r\right)$$

The vector $\vec{t}$ is composed of variables that are involved only in intermediate calculations: that is why this is called the "hidden" layer. The two activation functions g and f are not necessarily the same; weight matrices have dimensionalities connected to the shape of the respective layer: first layer has N inputs and H output, the second layer has H inputs and M outputs therefore the matrices W, $\tilde{W}$ are respectively $M \times H$ and $H \times N$ - to be intended as *rows* $\times$ *columns*.
The stack composition of layers remarkably enlarges the capability of the network to approximate a function: it has been shown that, provided a sufficient number of hidden units, the model can reproduce any continuous function, for a finite set of

input variables reaching an arbitrarily high accuracy [30, 39].
Indeed, an MLP can be composed of a generic number of layers and a generic quantity of hidden variables for each layer.

2.4 Correlation in time: Recurrent Neural Network

The field of application of machine learning is wide, based on what kind of data are employed or the actual answer we expect from the machine, the simple structure of MLPs may be not adequate or sufficient.
For language modeling, translation, speech recognition and similar jobs, recurrent neural networks (RNNs) are designed to give a solution.
Generically, a RNN is written to take time series or sequential data as input and the main feature is the internal structure which correlates data along time or the sequence itself; to obtain this, the network is based on hidden states and a structure named cell is repeated along network layers. While for usual feedforward neural networks, output is directly processed from every element of the input, the RNN returns data dependent on the prior elements within the input sequence and updates weights using back propagation through time.
The standard RNN definition [40] can be expressed considering the dataset as a sequence where each element is given respectively to every cell as input, the cell considered as a model that takes both output of the previous cell and its element input, multiplies those by learning parameters adding the bias. Activation functions are inserted in the model, too.
A popular model of recurrent neural network is the long short-term memory (LSTM) network. Introduced in [41], it has all the features of the RNN with the addition of control gates that can be trained to avoid vanishing gradients during backpropagation. These controls let the cell store single sequence step information, therefore called *short-term*, that can last (the network has memory of these using input, output and forget gates) for many time steps - that justifies *long* in LSTM.

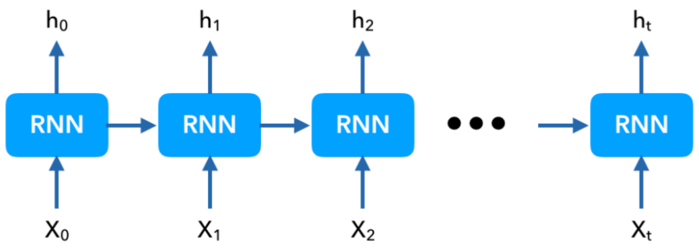


Figure 2.6. Simple graphic exposition of recurrent neural network structure. Image credit: [42].

In a LSTM (and generically for a RNN) forward and backward propagation steps are sequential. In back propagation through time (BPTT), introduced in [43], gradients are computed from current time-steps to past time-steps. That is, at time-step T, gradients from time-steps $T-1, T-2, \ldots t$ are added to the gradient at time-step T. This is why in the simpler structure of RNN, gradients may vanish or explode. Going into more details [31], in BPTT the loss is computed through each time step, in a network composed of hidden units h_t:

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t(\mathbf{h}_t, \mathbf{h}_{t-1} \ldots, \mathbf{h}_{t_0})$$

The output of the network $\mathbf{r}_T$ - or the output at time t - comes after the last hidden layer, weights for each layer are collected in the matrix $\mathbf{W}_{hh}$:

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{W}_{hh}} = \frac{\partial \mathcal{L}_t}{\partial \mathbf{r}_t} \frac{\partial \mathbf{r}_t}{\partial \mathbf{h}_t} \left(\sum_{k=1}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}} \right) \qquad \text{where} \qquad \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{s=k+1}^{t} \frac{\partial \mathbf{h}_s}{\partial \mathbf{h}_{s-1}}$$

The sequential composition of the algorithm requires that all computations are made successionally; this trait typical of RNNs makes them unable to benefit from parallelization, a computational procedure used in graphics processing unit (GPU) - hardware largely employed in machine learning - where many calculations or processes are carried out simultaneously. This fact leads recurrent neural networks to be slow during training procedure.

2.5 Convolutional layers

There are particular tasks in which the learnable information is hidden in the structure of data and it is not possible for an MLP to detect it. For example, given a set of pictures as dataset, image recognition requires to deal with groups of pixels taken together representing the recognizable object in picture, it cannot be performed considering each pixel as a single independent data.

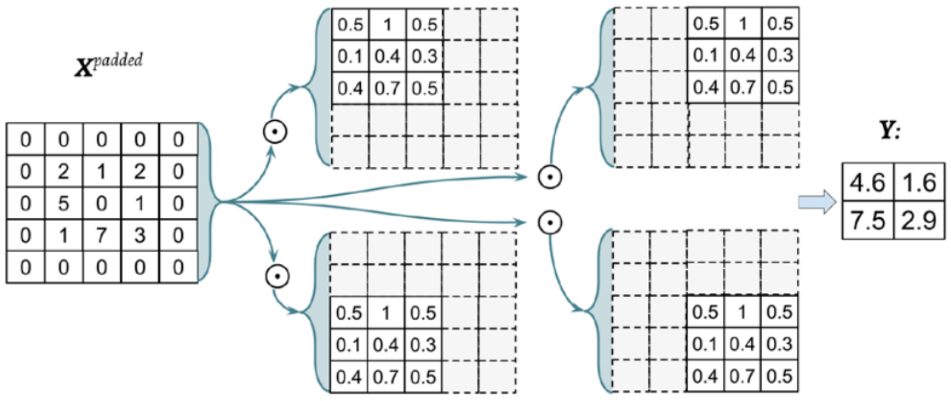


Figure 2.7. Procedure involved in a convolution for a padded input matrix. Image credit: [31].

The solution is to implement a technique that collects data and preserve or condense information concerning near data: in image recognition, a convolution consists in the application of a sliding filter over the image, the filter carries weights which multiply each pixel data input. Therefore, learning parameters are shared over image data in such a way the algorithm result is invariant under translation transformation - figure objects are recognizable for each spatial disposition inside the image.

Then, products of input data and weight parameters are summed for each filter application and the output will be a smaller matrix. Data output may be then passed to a pooling layer to reduce its dimension: no learning parameters are involved in this step, pooling procedure simply consists in taking the mean or max value of numbers in the given matrix.
Sometimes it may be useful to adapt the shape of input matrix to the filter dimension; in order to get this, the input matrix is filled with zeros following the acting called padding.

2.5.1 Graph convolution

In a similar way, convolutions can be performed over graphs. The field of machine learning dealing with graphs is named graph neural networks (GNN).
Given a graph with N vertexes and N feature vectors of length f_{in}, these vectors are input data for the convolutional layer while the output is composed of other N vectors of arbitrary (the desired shape of the layer depends on this choice) length f_{out}.

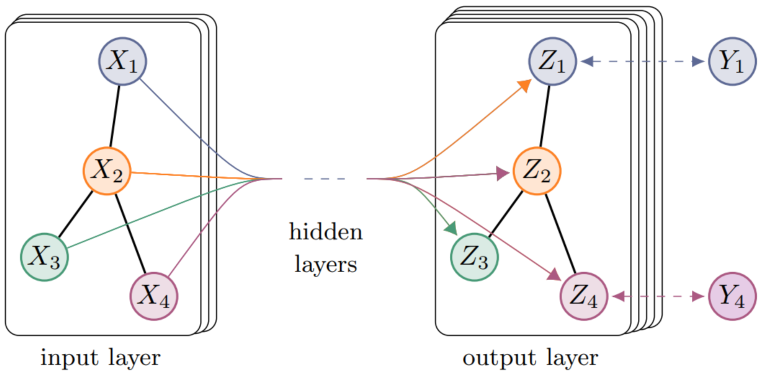


Figure 2.8. Graph convolutional layers. Image credit: [44].

The filter here is a weight matrix, input data is typically collected in a (feature) matrix $\mathbf{X}$. As well as in the case of MLP layers, rows and columns of $\mathbf{W}$ matrix are linked to input-output sizes of the layer: $f_{in} \times f_{out}$. $\mathbf{X}$ has dimensions $N \times f_{in}$.
Indeed, graphs are used to represent connections between elements of a system, the interesting information is therefore contained in its structure and list of edges. Defining the adjacency matrix $\mathcal{A}$ and calling $\mathcal{I}$ the identity, the filter convolution consists in the matrix multiplication $\mathbf{Y} = (\mathcal{A} + \mathcal{I})\mathbf{XW}$. The addition of identity corresponds to consider the graph with the supplementary characteristic of self loops for each node; this is done only to simplify the notation considering the contribution of node's embedding on which the convolution is computed. Writing explicitly the expression:

$$Y_{kl} = \sum_{m=1}^{N} (A_{km} + \delta_{km}) Z_{ml} = Z_{kl} + \sum_{m=1}^{N} \mathcal{A}_{km} Z_{ml} \quad \text{where} \quad Z_{ml} = \sum_{p=1}^{f_{in}} X_{mp} W_{pl}$$

The size of $\mathbf{Y}$ is $N \times f_{out}$, again a set of N vectors, one for each vertex. This means that convolutional layers can be simply stacked since the output and input shapes

are similar; the only constraint is to equalize $f_{out} = f_{in}^{\text{next layer}}$, just like it is done for linear layers in a MLP.
The product $\mathbf{XW}$ gives a matrix $\mathbf{Z}$ of size $N \times f_{out}$ where each *m-row* vector has, for every element, net inputs z_l. Adjacency matrix is used to substitute each embedding - for each vertex - with the sum of all embeddings of the neighbouring vertices, plus the vertex itself as self loops have been added. Obviously, the structure of the graph is left unchanged, as it is clear looking at figure 2.8; graph convolution collects local and topological information and stores it in each vertex embedding.
The general role of convolutional layers is to implement the technique commonly named message passing. This term is used, in graph neural networks, every time there is an elaboration of information for each graph's vertex - the message - and a procedure of aggregation of this information collected by the neighbouring vertices, i.e. passing the message. The procedure is often employed in GNN and it may also be applied without implementing a convolution.

2.6 Pooling layers

In neural networks, pooling layers are used to reduce the spatial dimensionality (width and height) of the input data, while preserving important features. They work by applying a function to a small window or "pool" of adjacent input values and outputting a single value, which represents a summary or abstraction of the information contained in that window. There are many different types of pooling layers: max pooling, which outputs the maximum value within each window, is relatively common; average pooling is used in this thesis work, which outputs the average value within each window.
Indeed, for GNN the windows are filters that correspond to the whole graph or to the set of neighbouring vertices. The idea expressed in convolutional layers is the same, where features does not undergo a message passing procedure but are collected and replaced in order to decrease feature dimensionality, e.g. with global average pooling it is possible to get a graph-level feature starting from per-node embeddings. Other types of pooling include L2-norm pooling, which outputs the L2-norm (Euclidean length) of the values within each window or pool, and many others depending on the given task.

2.7 Overfitting and regularization

Overfitting is a common problem in machine learning, where a model learns to fit the training data too closely and loses its ability to generalize to new, unseen data. In other words, it occurs when the network learns to memorize the training data instead of learning the underlying patterns. Sometimes it takes place when a model is trained with a large number of parameters relative to the size of the training dataset.
As the model becomes more complex, it can fit the training data more closely, but it also becomes more sensitive to noise and irrelevant features in the data. This causes the model to perform well on the training data but poorly on the test data, as it fails to give predictions on a new dataset that has not been submitted during

training. Overfitting can also arise when the dataset is too small or when the model is trained for too long. The opposit problem is underfitting, but it relies on the simplicity of the model, thus it is not interesting to be discussed here.

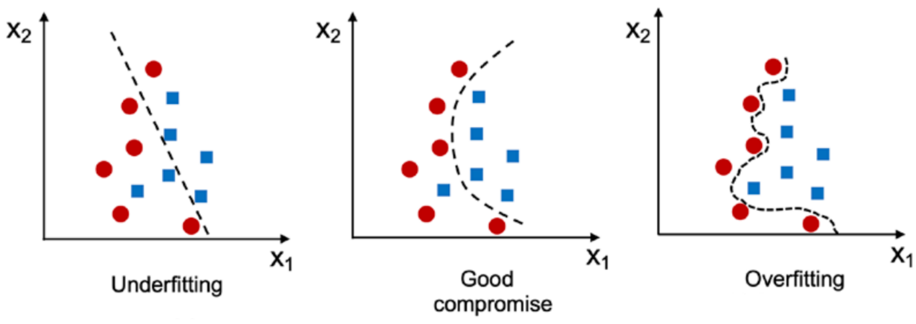


Figure 2.9. Examples of trained models expressed in data space, dashed lines represent the model itself and the shape depends on learned parameters. Image credit: [31].

To detect overfitting, it is important to evaluate the performance of the model on a separate validation dataset. If the performance on the validation dataset is significantly worse than the performance on the training dataset, it is a sign that the model is overfitting.
Generally, in this case, it may be necessary to adjust the model architecture or regularization techniques - which are the methods applied to properly constrast overfitting - to improve its ability to generalize to new data. During this thesis project, the problem had arised and in the next chapter will be exposed how it showed up and how it has been dealt with.

2.7.1 Dropout layers

Dropout is a regularization technique used in neural networks to prevent overfitting. Dropout works by randomly setting to zero (dropping out) a certain percentage of neurons in a layer during training.
The dropout layer is typically placed between fully connected layers in a neural network. During training, each neuron in the layer is dropped out with a probability p and it is kept with a probability $1 - p$. This means that the contribution of each neuron to the output is temporarily removed, and the remaining ones have to learn to compensate for their absence. This process effectively reduces the interdependence of the neurons in the layer, making the network more robust and less likely to overfit.
During testing, the dropout layer is turned off, and all nodes are used for prediction. However, the weights of the nodes are scaled by a factor of p to account for the dropout during training. Otherwise, for example in Pytorch, the outputs are scaled by a factor of $\frac{1}{1-p}$ during training; in this manner, during evaluation or testing the module simply computes an identity function. This ensures that the expected output of the layer remains the same, even when all neurons are active. Dropout is the most common regularization technique used nowadays and it has been shown to improve the performance of a wide range of models.

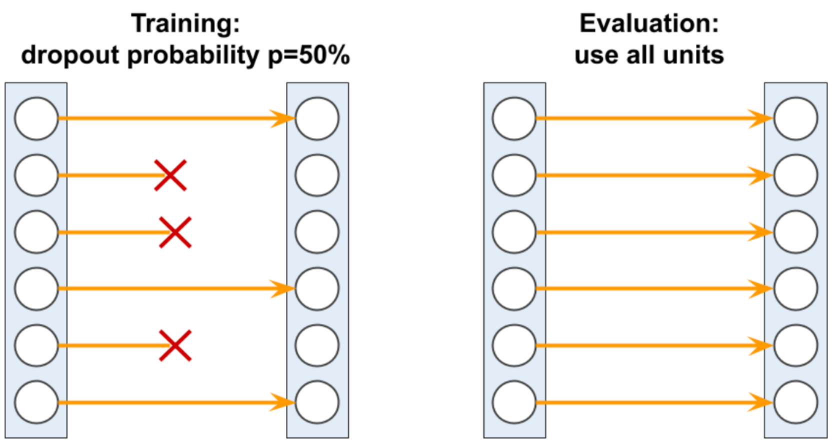


Figure 2.10. Execution of a Dropout layer with p = 0.5. Image credit: [31].

2.7.2 Weight decay

L2 regularization, also called weight decay or L2 shrinkage, is another regularization method commonly used in deep learning to prevent overfitting of the model. During training, the objective is to minimize the loss function, which measures the difference between the predicted output and the actual output. To prevent overfitting, weight decay adds a penalty term to the loss function $\mathcal{L}$ that encourages the model to have smaller weights; this is achieved by adding a term that penalizes large weights, obtaining the final *wd*-loss function $\mathcal{L}_{wd}$. Considering a simple case with n data, one layer with $\vec{w} \in \mathbb{R}^m$ parameters:

$$\mathcal{L}_{wd} = \mathcal{L} + \frac{\lambda}{n} \sum_{j=1}^{m} w_j^2$$

The role of weight decay applied to the model is controlled by a hyperparameter λ called regularization parameter. This hyperparameter determines the relative importance of the weight decay term in the loss function compared to the original loss function. A higher value of the regularization parameter will result in a greater emphasis on weight decay, and smaller weights in the model; putting $\lambda = 0$ is equivalent to suppress weight decay.
To implement weight decay in Pytorch, some optimizers - Adam being one of those - offers λ (set to zero by default) as an input parameter to be initialized when calling an instance of the optimizer class.

2.7.3 Learning rate scheduler

A learning rate scheduler is a function that monitorates the training and decreases or adjust the learning rate in those occasions where parameters are frozen around a minimum local point of the loss function or are becoming divergent.
It is not a technique that helps to contrast overfitting, but in general it can improve the performance and stability of the neural network during the training process. In Pytorch, it is implemented in many ways inside the class *torch.optim.lr_scheduler*

and in particular it has been used the one called *ReduceLROnPlateau* that waits until the loss function is giving constant value, then reduces η by a factor chosen by the user.

Chapter 3

Development of the Network

The aim of this thesis is to investigate the potential of graph neural networks in solving the vertex coloring problem. The topic of this chapter is about the attempt to develop a neural network able to classify a given input graph as colorable - using only a determined number of colors - or not.
The task is not at all trivial since there is no rule or explicit feature present in the graph that could be learnt by the neural network: the algorithm is written in order to grasp information on the topological structure using message passing, with the purpose to answer whether the graph can satisfy the vertex coloring constraint.
In literature are present conventional (e.g. not based on machine learning) algorithms, designed to solve the same task, which are based on the direct resolution, looking for a configuration where each pair of linked vertexes has different colors. In the next chapter, we compare the results obtained with the graph neural networks with those yielded by one of these conventional algorithms.
The first attempt consisted in a graph neural network written to test the code libraries and the computational framework: a simple network with one convolutional layer followed by a pair of linear layers (as in a MLP).
Since the aim was to test the code rather than to obtain a good result, the dataset was composed of small graphs with tens of nodes and a random quantity of links. Each graph was paired to a label representing if it was colorable with 2 colors. To get this label, it has been written a simple program that directly tries to assign colors to each graph's vertex, in the context of the 2-color problem, which is pretty simple since a graph is not 2-colorable only when it contains a ring (in the sense of a circular closed connection between nodes) made with an odd number of vertices. This program, named “chained coloring”, may be consulted on my personal GitHub page[*].
The initial attempt did not yield any valuable insights into the vertex coloring problem, but it established the framework in which a better concocted neural network could run.
There are two main neural network models employed in this thesis project, details are exposed in the next paragraphs: “Graph neural network article reference” and “Convolutional Graph Neural Network”, one for each model.

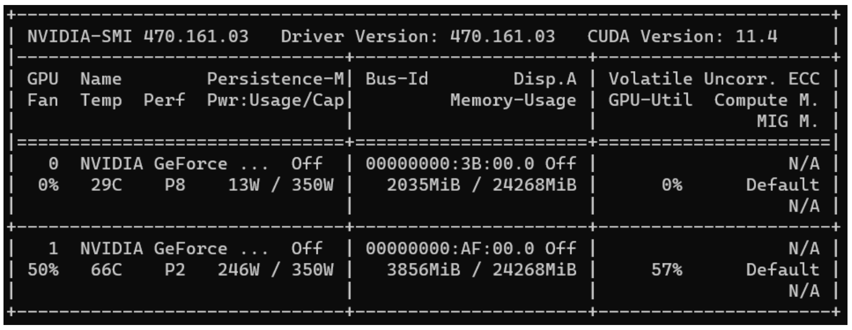
[*]https://github.com/vmarc0/Master-Thesis.git

3.1 Computational environment

The algorithm utilized in a neural network can often involve intricate calculations, and in order to optimize the program's processing capabilities, it is recommended to utilize a GPU instead of a regular central processing unit (CPU).
This is especially effective since the calculations in question are already optimized through the use of specialized libraries. The advantage consists in the capability of processing many operations simultaneously; GPUs are widely used in video editing and gaming too.
To employ a graphics card, a specific environment and drivers are needed: the thesis includes work completed through the use of Anaconda, a distribution platform that incorporates the Spyder IDE (version 5.4) for running the code. To enable the computing power of the Nvidia graphics card, the Cuda toolkit (version 11.6) was also utilized as a driver.
The programming language utilized is Python (version 3.10) and the main library used to build up the neural network is Pytorch (version 1.11), in particular for graph neural networks the libraries of Pytorch Geometric (version 2.2). About the hardware, my laptop on which this framework was installed is equipped with a Nvidia GeForce MX130.

3.1.1 INFN server

Throughout this thesis project some programs required more computational resources. To tackle this, my advisor has given access, through an INFN account and using SSH protocol, to a group server on which are installed two GPUs much more powerful: Nvidia RTX3090 with 24GB of RAM each, compared to MX130 that has only 4 GB. On this server, the environment is installed using docker.



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.161.03   Driver Version: 470.161.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:3B:00.0 Off |                  N/A |
|  0%   29C    P8    13W / 350W |   2035MiB / 24268MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce ...  Off  | 00000000:AF:00.0 Off |                  N/A |
| 50%   66C    P2   246W / 350W |   3856MiB / 24268MiB |     57%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Figure 3.1. Hardware information displayed using terminal command "nvidia-smi" on the group server.

3.2 Graph neural network article reference

Due to the lack of results with the initial neural network, an article [45] presenting a graph neural network based on a LSTM was used as a reference. The objective related to the purpose of this model is the same as the one in this thesis - to create a neural network classifier that determines if a graph can be colored or not. The

model is then used to assign a chromatic number to the input graph by starting with the lowest possible number of colors and gradually increasing the minimum constraint on colors until the network gives a positive answer. The number of colors at which the network answers positively is then assigned as the chromatic number of the graph in accordance with its definition.
The original code of this algorithm has been written using an old version of Tensorflow (Python) libraries; the algorithm has been rewritten in Pytorch (modern Python libraries used to implement DL algorithms) in order to make the logical steps of the program much more clear and to leverage the higher efficiency of these libraries.
It has been used the Stochastic Gradient Descent algorithm implemented via Adam optimizer [34], present in Tensorflow as well as in Pytorch.
For both implementations, binary cross entropy loss has been employed since the task involved a classification. The loss is computed between the ground truth label which belongs to the dataset - a boolean variable expressed as 0 or 1 - and the prediction of the model, a number in $[0, 1] \subset \mathbb{R}$ that represents the probability - according to the model - of the graph beeing colorable or not, given the fixed value of Q.

3.2.1 Neural Network Architecture

This neural network uses a recurrent neural network, the local information is gathered throughout a given number of message passing iterations, just like usually happens with a convolutional layer. This information is then aggregated and given to the recurrent neural network; this procedure is inserted in a loop cycle: for each round, the output of the RNN is condensed following the adjacency structure and then given as input to the same RNN again. After a number of cycles, the graph presents new embeddings that contain all the processed information.
Now let's describe the model in more detail, starting from the input. The neural network is expected to receive a graph and return a boolean variable that answers the question "does the graph accept a Q-coloration?".
The whole graph information corresponds to the list of links between pairs of nodes; this data will be utilized during the message passing phase and may also be recorded in the form of an adjacency matrix, which is calculated at the outset.
Another input that must be given to the model is the Q-color value: the algorithm does not look for the chromatic number χ but it is written only to tell if $Q \geq \chi$.
Thinking about the problem to solve, there must be an embedding space representing colors; if the embedding of each node is directly acting as the color of the node itself, then we would deal with the problem of finding the right initial configuration of colors that satisfy the constraint.
It is evident that the configuration in question is not known and presents a more challenging problem compared to the classification one that we are currently addressing using the neural network.
Otherwise, the other option is to give only a random initial setting of colors and expect the neural network to learn that each neighbouring embedding must be different. This strategy is not promising of good results since the network uses the graph-level ground truth label telling only if the graph is colorable; a single node

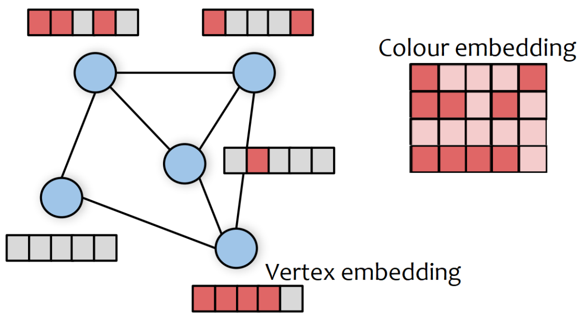


Figure 3.2. Representation of starting features, stored in the vertex embedding matrix **X**, where each row is displayed near the respecting vertex, and color embedding matrix **C**. Diagram elaborated from image in [46]

embedding gives not enough space to elaborate the local information and infer the satisfaction condition. In other words, construct a model where the constraint request is explicitly stated is a difficult job, keeping in mind that the loss is computed using the final answer and knowing how the learning method works.
To tackle this problem, the initial embedding is composed of vertex features - appropriate to process the message passing - and color features. In this manner, the input is composed of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $|\mathcal{V}| = N$ and a given value for Q, a vector of vertex embeddings and another one for color embeddings, both of dimension d.
Those are collected in two different matrices, as shown in figure 3.2, $\mathbf{X}$ of size $N \times d$ and $\mathbf{C}$ of size $Q \times d$. These values are initially taken uniformly distributed between zero and one, but then become a learnable parameter. To assign a color to each vertex, it is used the so called "vertex to color" adjacency matrix $\mathcal{M}_{VC} \in \{1\}^{N \times Q}$, i.e. a matrix full of ones in order to connect every node to each possible color, giving no prior information to the model.
Here message passing is split in two phases: computation of the messages using a three-layered MLP and the application of the adjacency matrices. Adopting the same notation of the original article [45], $V_{msg}, C_{msg} : \mathbb{R}^d \longrightarrow \mathbb{R}^d$ are the *message-computing* layers for both vertices and colors, where messages coming from vertexes are passed to both color-LSTM and vertex-LSTM, the same is done with those from color embeddings.
The messages are transferred to neighbours using $\mathcal{M}_{VC}$ and $\mathcal{M}_{VV}$ (the usual adjacency matrix describing the structure of the graph).
Note that this operation is pretty simple and easy to be computationally implemented, but the details inside matrix multiplication comprehend the sum of all neighbouring features directly assigned to the right node corresponding to the row in the output matrix; this is done all at once for every vertex of the graph, with only one matrix product.
For each LSTM, there is a hidden state that stores information to be correlated along time cycles (see RNN paragraph); in this model, the output of each LSTM is rejected and it is took only the final hidden state.
At each time step t, the two hidden states are referred to as V_h^t and C_h^t respectively

for vertexes and colors. Before entering the cycle, these variables are set equal to vertex and color embeddings while cell states are set to zero.
The variables obtained will be joined in a longer along-rows stacked matrix given as first input to the LSTM, naming those matrices V_{LSTM} and C_{LSTM}:

$$V_{LSTM}^{t=0} = \Big(V_h,\ \ \mathcal{M}_{VV}\mathbf{X},\ \ \mathcal{M}_{VC}\,C_{msg}(\mathbf{C})\Big)$$

$$C_{LSTM}^{t=0} = \Big(C_h,\ \ \mathcal{M}_{VC}^T V_{msg}(\mathbf{X})\Big)$$

Here $V_h = V_h^{t=0} = \mathbf{X}$ that is the vertex embedding matrix and similarly $C_h = \mathbf{C}$, as stated above. The application of V_{msg} and C_{msg} are intended to be on each row vector of input matrices. The superscript T on $\mathcal{M}_{VC}$ inside the second equation refers to the traspose operator, useful since the output of $V_{msg}(\mathbf{X})$ is a matrix of size $N \times d$ while the matrix $\mathcal{M}_{VC}^T$ has size $Q \times N$ so that the product gives a matrix of dimension $Q \times d$, a proper shape for the color-LSTM (C_h has the same shape).
The update of each feature is done using $\mathcal{V}_{LSTM} : \mathbb{R}^{3d} \longrightarrow \mathbb{R}^d$ and $\mathcal{C}_{LSTM} : \mathbb{R}^{2d} \longrightarrow \mathbb{R}^d$ where it has been considered hidden state as input too.
In this way, the core structure of the algorithm is expressed in the following cycle. Starting from $V_h^{t=1} = V_{LSTM}^{t=0}$ and $C_h^{t=1} = C_{LSTM}^{t=0}$, for every t in the sequence $1, \dots T$:

$$V_h^{t+1} = \mathcal{V}_{LSTM}\Big(V_h^t,\ \ \mathcal{M}_{VV}V_h^t,\ \ \mathcal{M}_{VC}\,C_{msg}(C_h^t)\Big)$$

$$C_h^{t+1} = \mathcal{C}_{LSTM}\Big(C_h^t,\ \ \mathcal{M}_{VC}^T V_{msg}(V_h^t)\Big)$$

At the end of the cycle, the interesting variable is V_h^T. It is a matrix with a row vector for each node of the graph. The information contained in each vector is now crucial to answer the initial question.
Each vector is passed to a final $\text{MLP}_{vote} : \mathbb{R}^d \longrightarrow \mathbb{R}$ which translate embeddings in logits.

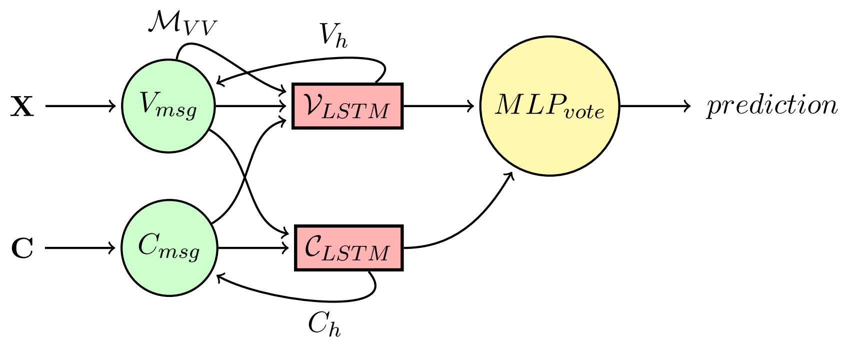


Figure 3.3. A schematic diagram of the model. Backward arrows involve a loop cycle. For each loop cycle, $\mathcal{V}_{LSTM}$ takes 3 inputs and returns V_h, input of the next iteration. $\mathcal{C}_{LSTM}$ does the same, taking 2 inputs. At the end of the loop, the outcomes of the LSTMs is passed to MLP_{vote} layers.

The last steps of the algorithm consist in a global mean pooling layer and a sigmoid as final activation function. In this way it is obtained the graph-level prediction, output of the model. The algorithm is presented in figure 3.3 schematically.

3.2.2 Computations behind layers

The layers involved in this model are MLPs, where there is no additional characteristic beyond the classic structure explained in the previous chapter, and LSTM layers. The details behind the latter are expressed in the following equations and represent all the computations involved in a *torch.nn.LSTM* layer:

Input gate:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

Forget gate:

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

Cell gate:

$$g_t = \tanh(W_{gx}x_t + W_{gh}h_{t-1} + b_g)$$

Output gate:

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$$

Cell state update:

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

Hidden state update:

$$h_t = o_t \odot \tanh(c_t)$$

where:

- i_t, f_t, o_t are the input gate, forget gate, and output gate activations at time step t, respectively.
- g_t is the cell gate at time step t.
- c_t is the cell state at time step t.
- h_t is the hidden state at time step t.
- x_t is the input at time step t.
- W terms denote weight matrices between gates, e.g. W_{ix} is the matrix of weights from the input to the input gate.
- b_i, b_f, b_g, and b_o are bias vectors for the input gate, forget gate, cell gate, and output gate, respectively.
- σ is the sigmoid function, and $\odot$ is the element-wise product (Hadamard product).

A schematic summary is displayed in figure 3.4. LSTMs were originally constructed with memory blocks in the recurrent hidden layer. These blocks contained memory cells responsible for retaining the temporal state of the network, along with special units called gates that controlled the flow of information. Each memory block housed an *input gate* for regulating the flow of input activations into the memory cell, as well as an *output gate* for directing the activation outputs to the network.

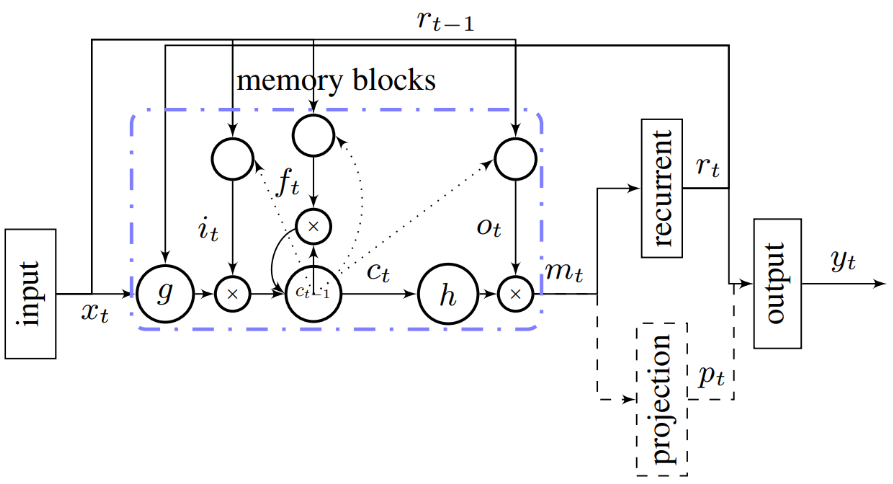


Figure 3.4. LSTM based RNN architectures where a single memory block is shown. Image credit: [47].

However, the original LSTM models had a weakness: they struggled to process continuous input streams that lacked segmentation into subsequences, which caused issues when resetting cell states. To solve these problems, a *forget gate* was added to the memory blocks, which scaled the internal cell state and allowed for adaptive forgetting or resetting of the cell's memory.
In addition, modern LSTM architecture includes peephole connections between the internal cells and the gates in the same cell, allowing for more precise timing of the outputs. The peephole connections allow all gates to inspect the current cell state even when the output gate is closed [47].

3.2.3 Dataset

The details of how the dataset is composed are crucial to the proper functioning of the model in the sense of obtaining the expected results.
The article [45] from which the neural network is taken, based the research on a dataset generated with the so called "adversarial" method. The idea was to work on hard instances, building the training set of graphs on the verge of satisfiability. To do so, the adversarial method consists in the generation of a graph choosing the chromatic number χ between 3 and 8, then the adjacency matrix is populated with a connectivity adjusted to the selected chromatic number. The result is passed to a solver provided by Google [48] which ensures if the given graph has the chosen chromatic number. This is the first graph of the adversarial instance (composed of a couple of graphs), paired to the True or 1 label. Then, edges between nodes are added until the solver is no longer able to solve the graph at the given χ value; this is done in order to obtain the second graph, completing the instance, this time with a 0 or False label. The last single edge added is called frozen edge and it is the one responsible for the non-colorability of the second graph.
For both the two graphs of the adversarial instance, $Q = \chi$ since the first satisfy the constraint with that number of colors, the second one does not, with the same

number of colors.
This dataset can be generated using the programs available at authors' Github page [49] or directly downloaded from the same webpage. Graphs are available in form of *.graph* files, essentially equivalent to text files where each line contains the pair of vertex labels corresponding to an edge: in other words the usual edge index is vertically printed in a file, therefore the dataset was easy to be read writing a simple program dealing with input stream coming from a file. At the end, two other important information are present: chromatic number (from 3 to 8) and the list of frozen edges - typically composed of just one edge. For each graph, the number of nodes is uniformly extracted between $N = 40$ and $N = 60$.
The article dataset is composed of 2×2^{15} instances, the dimension of embeddings $d = 64$, MLPs are three-layered and between layers, ReLU activation function is exploited. The last hyperparameter is the cycle length $T = 32$. In order to generate graphs with true labels, it has been used a Python function called "MakePlanted" whose algorithm will be explained in detail in the next paragraph.
On the other hand, to generate graphs with the other label - i.e. not colorable graphs - "MakeRandGraph" is adopted, discussed in the next paragraph, too. Then the output is passed to the other function "chained coloring" to ensure that there is no satisfiable configuration of colors. If there would be, the graph is simply rejected and the procedure starts again calling the first function. This is done for cases with $Q = 2$ since it is the only case for which the "chained coloring" code is written.
In the replication of the algorithm written in Pytorch, the adversarial method has been applied using a different strategy, without making use of an external program. As explained in the first chapter and exposed in [21], there is a technique named "planting" used to construct a vertex coloring satisfiable graph (in general, any SAT proposition), at any value of connectivity, even above the SAT/UNSAT phase transition.

3.3 Dataset generation

There are two main algorithms to generate graphs. Fixed an initial chromatic number χ, in order to generate SAT propositions, i.e. graphs that present a satisfiable configuration of colors, the scheme is based on the mentioned *planting* method. The idea is to firstly choose Q different colors, with $Q = \chi$ chromatic number, then generate N nodes and paint each of them with different colors following a random uniform distribution.
Only at this point, connect nodes which have been assigned different colors until the number of edges is the desired one (in particular, until the total number of edges satisfy equation (1.3)), with c as the desired mean connectivity of the graph).
The other main algorithm generates random graphs: in this case, the logic follows the definition of ER graphs, discussed in the first chapter. In the following Python listing, it is possible to consult the code written to generate random graphs:

```python
import torch
import random

def MakeRandGraph (N, M):

 6      degree = torch.zeros(N, dtype = torch.int)
 7      adj = torch.zeros(N,N, dtype = torch.int)
 8
 9      neigh = [] #lists of neighbouring nodes, one per node
10      for k in range(N):
11          neigh.append([])
12
13      graph = torch.zeros(2, M, dtype=torch.int64)
14
15      for j in range(M):
16          var1 = random.random()
17          var1 *= N
18          var1 = int(var1)
19          var2 = var1
20          while (var2 == var1):
21              var2 = random.random()
22              var2 *= N
23              var2 = int(var2)
24          graph[0][j] = var1
25          graph[1][j] = var2
26
27          neigh[var1].append(var2)
28          neigh[var2].append(var1)
29
30          adj[var1][var2] = 1
31          adj[var2][var1] = 1
32
33          degree[var1] += 1
34          degree[var2] += 1
35
36      return graph, neigh, degree, adj
```

Listing 3.1. Code used to generate random graphs

Here, a graph is generated with N vertices and M edges. The first variable *degree* is a list containing the degree of each vertex, i.e. the number of neighbouring nodes for each node; similarly, *neigh* is a list where instead of the node degree, it is stored another list containing all the labels of neighbouring nodes - in this manner, it is a list of lists.
Each pair of values in every column of *graph* tensor corresponds to an edge linking those vertexes numbered with those values. This is the usual tensor also referred to with the expression *edge_index* and it is one of the structures used to define a graph; in particular it is the sparse representation, against the dense representation expressed by the adjacency matrix, which is returned by the generating "MakeRandGraph" function too, with the variable name *adj* and the shape of a $N \times N$ matrix.
About the algorithm used to construct the graph, it is shown starting from a *for* loop in line 15 and it is rather simple: a pair of nodes are randomly and uniformly extracted until they are different (the same node is extracted twice with a probability of $\frac{1}{N}$). The process ends when M connections are made; there is the possibility of double edges (even triple and more), in the meaning of a second extraction of a pair already linked, but these events happen with a negligible probability which is equal to $\frac{1}{N^2-N} \approx 10^{-4}$ for $N = 100$.
The one just exposed is the algorithm employed to generate ER random graphs;

the following code follows a similar scheme in order to generate planted graphs:

```python
import torch
import random

def MakePlanted (N, M, Q):
    NoverQ = N/Q
    degree = torch.zeros(N, dtype=torch.int)
    adj = torch.zeros(N,N, dtype = torch.int)

    mapp = list(range(n))
    random.shuffle(mapp)

    neigh = [] #lists of neighbouring nodes, one per node
    for k in range(n):
        neigh.append([])

    graph = torch.zeros(2, M, dtype=torch.int64)

    for i in range(M):
        var1 = random.random()
        var1 *= N
        var1 = int(var1)
        var2 = var1
        while (int(var1/NoverQ) == int(var2/NoverQ)):
            var2 = random.random()
            var2 *= N
            var2 = int(var2)
        var1 = mapp[var1]
        var2 = mapp[var2]
        graph[0][i] = var1
        graph[1][i] = var2

        neigh[var1].append(var2)
        neigh[var2].append(var1)

        degree[var1] += 1
        degree[var2] += 1

    return graph, mapp, neigh, degree
```

Listing 3.2. Code used to generate planted graphs

Many variables and tensors have the same name as in the previous function; those have the same role and are defined in the same way. The main changes are the presence of Q as input value, *mapp* list and *NoverQ* variable and some variations in the algorithm, indeed. The planted method let generate a graph with any possible chromatic number: that is why there is a new input variable (Q).
Setting Q value will lead to a graph with chromatic number $\chi = Q$; this is ensured by the algorithm behind the planting procedure. The idea is simple, a graph with N vertices that requires to be Q-colorable is generated dividing the first $\frac{N}{Q}$ nodes from the others, and same for the seconds and so on. The division into groups is virtual since no operation is done, but to better understand the algorithm, let's imagine that each group is painted with a different color. In this way, the graph should present N nodes where every multiple of *NoverQ* is the node label that corresponds

to the first vertex carrying a new color, for a total of Q different colors.
Then, pairs of nodes are extracted again uniformly, this time until the two values are belonging to different colors, i.e. numbers are continuously extracted until $\lfloor var1/NoverQ \rfloor == \lfloor var2/NoverQ \rfloor$ is a no longer valid condition, at this point one edge is fixed between those vertices. Notice that the ratio $\frac{var}{NoverQ}$, truncated to the lower integer (here done with floor function $\lfloor \cdot \rfloor$), is the number representing the color of the *var* node.
Since the dataset is written to be a training dataset for a neural network, to avoid that the network is going to learn only the pattern - for which all first *NoverQ* nodes have the same color (and so on, for successive groups and colors) - it has been used a relatively simple operation which will now be explained.
Considering that the structure of connections is all the information which represents a graph, if those edges are preserved, the graph will be invariant under node label reciprocal interchange.
The list *mapp* in line 9 of listing 3.2 is defined and utilized on this purpose: after the shuffle, *mapp* will be a list working as a function that maps old labels into new labels. That is why the values for vertex labeling are being transformed by *mapp* function (technically, a list) in lines 27 and 28.
About the first model, dataset is generated with $Q = 2$. For graphs with true labels, the function "MakePlanted" is sufficient. On the other hand, to generate graphs with the other label - i.e. not colorable graphs - "MakeRandGraph" is adopted, then the output is passed to the other function "chained coloring" to ensure that there is no satisfiable configuration of colors.
If there would be, the graph is simply rejected and the procedure starts again calling the first function. This is done for cases with $Q = 2$ since it is the only case for which the "chained coloring" code is written.
In those cases where datasets has been generated on my own - not downloaded from article repository - in some code files they were generated as the first thing at the start of the program. In the last code projects, dealing with many different runs on many datasets, data has been produced once using *torch_geometric.data.Dataset* class.
Using inheritance from that class and overriding some few methods, it has been written a code - *Dataset_creator.py*[*] - that generates a determined number of graphs, with a specific number of nodes, edges, features, a graph-level label required during supervised training, and then saves those in a *.pt* file. Once the program is run a second time, if the same dataset is needed (in that case, the file name will be the same), then the generating functions are skipped and the program loads data from the present file.
In other words, if there is no dataset in the given path, it is generated; if there is one, matching the name of the file, then data are loaded without spending further time and resources. The functions inside this class which generates graphs are those explained above.
Part of the experiments on the first model are done using the dataset provided by article's authors, both for the original code in Tensorflow and the one written in Pytorch. Other studies are carried out generating the dataset with the adversarial

[*] Available at https://github.com/vmarc0/Master-Thesis/tree/main/Dataset

method. In this respect, it has been used the planting method to generate graphs with the same distribution of vertices and edges in article dataset.
In the last model, presented in the next paragraph, the dataset has been generated adopting the two Python functions, defined previously, gathered in a Python class, as it will be made clear later.

3.4 Convolutional Graph Neural Network

Convolutional Graph Neural Networks (CGNNs) offer a promising approach for tackling the vertex coloring problem and other graph classification tasks, by leveraging the power of deep learning and graph convolution to capture the complex relationships between graph structures and vertex features. The first structure of this model has been written keeping the network simple. Since that architecture gave good results in its simplicity, there has been no need to insert additional layers.

3.4.1 Neural Network Architecture

The structure of this model is lighter than the previous one. Instead of using embeddings for colors and vertexes, connected by vertex-to-colors matrices and LSTM crossing layers, here the starting point is the graph itself and a scalar random feature for every node.
The model is designed with the great help of Pytorch Geometric, Python libraries built on top of PyTorch that provide a set of efficient and easy-to-use modules for designing and training graph neural networks. Those include a wide range of state-of-the-art GNN models, including Graph Convolutional Networks, Graph Attention Networks, Graph Isomorphism Networks, among others; for this project, it has been used GraphConv, a simple class for message passing and convolutions.
The network consists of two graph convolutional layers, which are used to learn representations of the nodes in the input graph. The first layer has an input dimension of 1 (since the input features are one-dimensional), and an output dimension of 64. The second layer has an input dimension of 64 and an output dimension of 32. In this way the two layers can be concatenated without data shape transformations.
The peculiar choice of those input-output dimension values for each layer is motivated by the fact that, starting from a single scalar value per vertex, the model needs a virtual space to store the information collected through convolutions. The idea behind those numbers is to let this virtual space take place in the dimension of weights parameters. Indeed, it is needed a number great enough in order to satisfy those requirements. On the other side, too many parameters could lead to a model which is excessively complex: in those cases, a foreseen result is a model that overfits on every dataset.
After each Graph Convolutional Layer, the output is passed through a ReLU activation function and a Dropout layer to introduce non-linearity and prevent overfitting. Then, the output is passed through a Multi-Layer Perceptron (MLP) with three fully connected layers.
The MLP consists of two hidden layers with 16 and 8 units respectively, followed by a final output layer with a single unit that predicts the binary label. Each hidden layer is followed by a ReLU activation function and a Dropout layer. The role

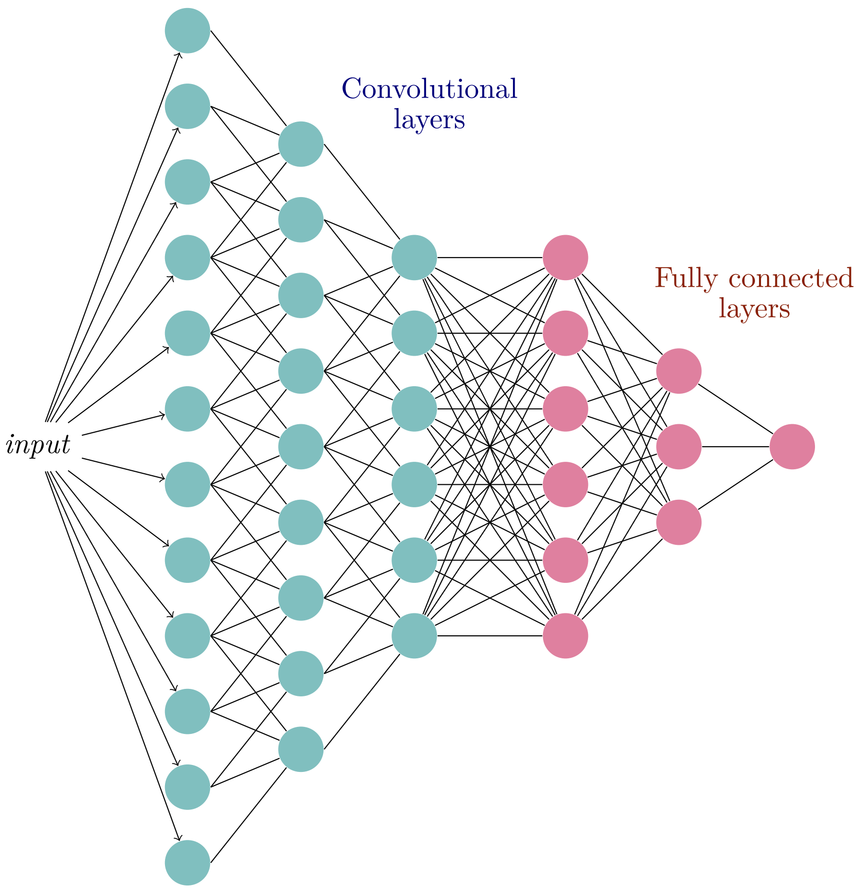


Figure 3.5. Illustration of the scale model: input is the node feature, two convolutions followed by 3 MLP layers; output is then passed to a graph pooling layer and exits with a sigmoid, here not displayed.

of these last MLP layers is to add elaboration to the information collected and to reduce the output to the shape of the original input.
Finally, since the model has produced a scalar for every node, the output from the MLP is aggregated by applying a global mean pooling operation to obtain a single scalar value to represent the whole graph, which is then passed through a sigmoid activation function to obtain the final binary prediction.
Overall, this network is designed to learn a hierarchical representation of the input graph by applying Graph Convolutional Layers and an MLP, and to use this representation to make a binary classification prediction. Details are displayed in the following code listing, which is the implementation of the neural network model:

```python
import torch
import torch.nn as nn
from torch_geometric.nn import GraphConv
from torch_geometric.nn import global_mean_pool

dev = torch.device('cuda')

class colored_Net(nn.Module):
    def __init__(self):
        super().__init__()

        self.Convol_layer1 = GraphConv(1,64).to(dev)
        self.Convol_layer2 = GraphConv(64,32).to(dev)

        self.relu = nn.ReLU()
        self.drop = nn.Dropout(0.2)
        self.MLP = nn.Sequential(nn.Linear(32, 16, device=dev),
                                 nn.ReLU(),
                                 nn.Dropout(0.15),
                                 nn.Linear(16, 8, device=dev),
                                 nn.ReLU(),
                                 nn.Dropout(0.2),
                                 nn.Linear(8, 1, device=dev))


    def forward(self, feat, graph, batch):
        conv = self.Convol_layer1(feat, graph)
        conv = self.relu(conv)
        conv = self.drop(conv)
        conv = self.Convol_layer2(conv, graph)
        conv = self.relu(conv)
        conv = self.drop(conv)
        last_layer = self.MLP(conv)
        last_layer = global_mean_pool(last_layer, batch)
        last_layer = last_layer.squeeze()
        output = torch.sigmoid(last_layer)

        return output
```

Listing 3.3. Code of the Convolutional Graph Neural Network model

Inside the other file that launches the training - the scheme is exposed in Listing 2.1 - the *forward* function is called using the object of the *colored_Net* class as a function itself.
The training implementation of the model follows the same structure of listing 2.1. Specifically, the line code written in the training program that replace line 13 in listing 2.1 is here written in line 9 of listing 3.4:

```python
1  import torch
2  from Network_file import colored_Net
3  [...]
4  model = colored_Net() #the class takes no input arguments
5  [...]
6  for epoch in range(num_epochs):
7      for data in Dataset:
8  
9          output = model(data.x, data.edge_index, data.batch)
10 
11         # Accuracy
12         pred = output > 0.5
13         for k,p in enumerate(pred):
14             if p.item():
15                 pred[k] = 1
16             else:
17                 pred[k] = 0
18         correct += int((pred == data.y).sum())
```

```
19
20         loss_b = loss_fn(output, data.y)
21         loss += loss_b
22
23         # Backward and optimize
24         loss_b.backward()
25         optimizer.step()
26         optimizer.zero_grad()
27 [...]
```

Listing 3.4. Implementation of the forward pass during training

Starting to analyze line 9, the input passed to the model is composed of three items: *data.x* is a torch tensor of size $N \cdot batch \times 1$ corresponding to a matrix with batch size times N rows and one column. Each row is corresponding to every vertex in every graph inside the batch; the single value in the row is node's scalar random feature.
When training deep learning models on graph data, it's common to train on mini-batches, smaller parts of the original dataset where each graph can have a different size and structure (in this case, all graphs have the same properties), since the whole dataset may be too heavy to be read by the neural network all at once, in addition to give a stochastic component during backpropagation. In particular, the user defines the *batch_size* quantity, a value between 1 and the total number of graphs *n_data* in the whole dataset. Then, methods and functions in those classes in charge of dealing with data - *torch_geometric.loader.DataLoader* in Pytorch Geometric - split the dataset in $\frac{n\_data}{batch\_size}$ parts. Clearly, those parts are called batches and contain *batch_size* graphs; in listing 3.4 each batch is called *data*.
Usually, in Pytorch, data is grouped in batches concatenating data along a new dimension (the first one) of the tensor. The first dimension of *data.x* contains a stack of all vertices along graphs in the batch: Pytorch Geometric uses a very clever way to organize data, allowing to group multiple graphs (that may have different sizes) into a single data batch creating a giant graph that holds multiple isolated subgraphs, and node and target features are simply concatenated in the node dimension; then some additional attributes are added, useful to work with data, such as *data.batch* - the third input item. The latter is a vector of length $batch\_size \cdot N$ (i.e. the total number of vertexes in a batch) that maps each node to its respective graph; it is useful to recognize and match vertices with proper graphs for global pooling, indeed the vector is passed as input to that layer in line 34 of listing 3.3.
The second input element is *data.edge_index* where all graphs are disconnected: in this manner, message passing or convolutional layers work in the same way as if the input was a single graph since messages cannot be exchanged between two nodes that belong to different graphs. This batching procedure works completely without any padding of node or edge features, and there is no memory overhead because graphs are represented with the sparse notation (using an edge index instead of a dense adjacency matrix).
Coming back to the *colored_Net* class, at line 35 in listing 3.3 the output of the pooling layer is squeezed: the operation consists in removing a dimension of size 1 from a tensor, e.g. transforming a matrix $(N \times 1)$ into a vector of length N. In this way, the output of the model is a vector of dimension *batch_size* containing all the

predictions for every graph inside the starting data batch.
To compute the accuracy of the model, these predictions expressing the probability of the graph to be colorable are transformed in boolean variables, giving a ultimate projection about the satisfiability of the graph in the vertex coloring context. Then, output is again expressed with numbers in order to compare the results with target labels, which are expressed with numbers since are employed in loss evaluation too. Finally, to compute the loss between the ground truth label *data.y* and the final prediction of the model, is used the first output of the model, i.e. predicted probability values in $[0, 1] \subset \mathbb{R}$. In line 21 of listing 3.4 the loss of each batch is summed in *loss* variable in order to obtain an estimate of the actual loss during that epoch: at the end of the first for loop, the variable *loss* is divided by *len(Dataset)* that is the total number of batches in which dataset had been split (this operation is not displayed as code in listing 3.4) .

3.4.2 Computations behind layers

As already stated, convolutional layers are implemented using *GraphConv* which is an operator took from the paper [50]. The algorithm involved is relatively simple and consists in the standard definition of convolution operated on graphs, as explained in the previous chapter:

$$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \sum_{j \in \partial i} \mathbf{x}_j$$

where the sum is made on the neighbours of node i. The layer offers also weights for edges, but are all set to 1 by default. MLP layers are exactly a composition in succession of linear layers, therefore no further explanation is needed. About global mean pooling layer, the computation implicated in order to get scalar output r over N vertices, is the following:

$$r = \frac{1}{N} \sum_{k=1}^{N} x_k$$

Notice that this may result to be a vector or generically a tensor; in the case of this model, the dimension is reduced to a simple scalar - since the expression is referring to the whole graph, starting from N vertices.

3.4.3 Dataset

About the dataset generated for the second model, the problem was studied fixing the value $Q = 5$ so it has been applied a different strategy, dismissing the adversarial method. Since above the connectivity value corresponding to SAT/UNSAT phase transition all random graphs are always not satisfiable (UNSAT), to get the data with false label it has been employed directly the function “MakeRandGraph” without the need to carry further checks.
Again, “MakePlanted” produced graphs always with true labels, although the connectivity corresponds to UNSAT configurations. Indeed, this method is relatively simple and let us study the problem for any chromatic number, but it binds us to the higher connectivity region.
For this dataset, data has been generated using *Dataset_creator.py* where input

arguments are: size of the graphs N, the number of graphs *n_data* that is set to 10^4 by default - i.e. the size of every training dataset generated, *root* variable which is the path where dataset file will be searched or saved after generation, a boolean variable named *painted*. The latter is a characteristic introduced during the evaluation of the convolutional model (the second one); proper motivations and the purpose of this variable will be exposed in the next chapter. The effect of this variable is a singular feature in the structure of generated graphs, but the results of training on that dataset is decisive.
Details of how the dataset is made emerge analyzing the generating code in the following lines - the entire file being accessible on GitHub:

```
import torch
from random import random
from torch_geometric.utils import to_undirected
from torch_geometric.data import Data
from Library import MakePlanted
from Library import MakeRandGraph

[...] #self.N and self.conn are defined by class constructor:
#respectively number of nodes and connectivity

data_list = []
Q = 5

if painted:

    for k in range(n_data):
        nodef = torch.zeros(self.N, dtype=torch.float32)

        #random returns a casual value between 0 and 1
        if (random() > 0.5):
            Y = torch.tensor([1], dtype=torch.float32)
            grafo, mapp, _, _ = MakePlanted(self.N,
                                    int(self.conn*self.N/2), Q)

            #Graph's nodes get painted
            for i in range(self.N):
                nodef[mapp[i]] = int(i*Q/self.N)

        else:
            Y = torch.tensor([0], dtype=torch.float32)
            grafo, _, _, _ = MakeRandGraph(self.N,
                                int(self.conn*self.N/2))

            #Since the graph is not colorable, colors are random
            for i in range(self.N):
                nodef[i] = int(random()*Q)

        grafo = to_undirected(grafo)
        nodef = nodef.view(-1, 1)
        graph_obj = Data(nodef, edge_index = grafo, y = Y)
        data_list.append(graph_obj)

else:

    for k in range(n_data):

        if (random() > 0.5):
            Y = torch.tensor([1], dtype=torch.float32)
            grafo, _, _, nodef = MakePlanted(self.N,
                                  int(self.conn*self.N/2), Q)

        else:
            Y = torch.tensor([0], dtype=torch.float32)
            grafo, _, nodef, _ = MakeRandGraph(self.N,
                                  int(self.conn*self.N/2))

        grafo = to_undirected(grafo)
        nodef = nodef.view(-1, 1)
        graph_obj = Data(nodef, edge_index = grafo, y = Y)
        data_list.append(graph_obj)
[...]
```

Listing 3.5. Class employed to create or load datasets.

The dataset in Pytorch Geometric is stored in a list using *torch_geometric.data.Data* objects. Therefore, the aim is to create instances and store them in *data_list*.
In any case, independently from *painted*, every graph is generated extracting a random number between 0 and 1: if it is higher than $\frac{1}{2}$, a planted graph is generated, otherwise a random graph. In this way, quantity of SAT and UNSAT graph instances are balanced on average.
The idea behind *painted*, when it is equal to True, is to initialize a graph with one-dimensional vertex feature representing a color. In particular, in the algorithm of MakePlanted, every amount of $\frac{N}{Q}$ node labels (counting from 0 to N) are all assigned to the same color: using the ratio in line 25, node i is assigned to color $\lfloor \frac{node_i Q}{N} \rfloor$, but recalling that graph's label vertices are shuffled, the feature is assigned applying *mapp* function.
In the other case, where the extracted random number is below 0.5 and the graph generated is random, colors are filled randomly since there is no configuration that satisfies the vertex coloring constraint. This is due to the fact that the number of edges is selected in order to obtain the specific mean connectivity value above UNSAT threshold - in the context of this second model, all data is generated in this connectivity region.
It it interesting to notice that in both MakePlanted and MakeRandGraph arguments, the number of edges is passed using the mean connectivity value: the expression $\frac{c \cdot N}{2}$ is the number of edges in a graph with N vertices and mean node's degree equal to c, as it is possible to verify looking at equation (1.3).
When *painted* is set to False, graphs are produced with the usual properties, with degree of each vertex as feature: *degree* output of MakePlanted or MakeRandGraph is directly saved in the node features vector *nodef*.
In the ending lines, the function *to_undirected* is employed to automatically double each edge in the graph, since a single connection is meant to represent directed graphs. Then, feature vector is shaped in order to constitute a vertex feature matrix composed of one column.
Utilizing a Python class to generate a dataset offers a significant advantage consisting in the fact that the dataset is only generated once during the initial run of the

neural network program, and subsequently saved in a file for any future run. Without this approach, each run would require the program to regenerate the dataset, yielding in a waste of resources.

Chapter 4

Analysis of Results and comparison with conventional Algorithms

As mentioned in the introduction of the preceding chapter, the initial attempts in implementing models available in early literature on the subject did not produce any significant results that were worth discussing. During the initial phases of the thesis, various neural network programs that were documented in literature were tested in the experimentation process. However, these programs posed compatibility issues with modern frameworks since the codes were outdated and obsolete.
Consequently, this approach failed to deliver the expected outcomes. Therefore, as an alternative, the article [45] and its model were used as a reference to develop a similar neural network algorithm. The code was written using modern libraries, which proved to be more effective.
This approach provided valuable insights on the topic, ultimately leading to the final model discussed in the preceding chapter, namely the Graph Convolutional Neural Network. In the following paragraphs will be analyzed the two models, starting from the one taken from the article [45].

4.1 Performance of the first network

To evaluate the performance of a neural network, two key performance metrics are commonly used: loss and accuracy. In classification tasks, accuracy is calculated as the ratio of correctly predicted samples to the total number of samples.
As the training proceeds through the stochastic gradient descent steps, it is expected that the accuracy of the model should increase, while the loss should decrease. This means that as the model learns facing repeatedly the same data, it should become better at making accurate predictions and therefore produce a higher accuracy score. At the same time, the loss should decrease because the model is becoming better at minimizing the difference between its predictions and the actual values.
It is important to notice that the rate at which accuracy and loss improve during training can vary depending on the complexity of the problem being solved, the amount of data available, and the architecture of the neural network. Therefore,

it is essential to monitor both accuracy and loss during training to ensure that the model is learning effectively and not overfitting or underfitting the data.
In order to do that and to check the real performance, during training the model has been evaluated both on a test dataset and on the training dataset, which is the only one used to update weights and apply backpropagation. In this way, for each epoch, it has been possible to observe the accuracy on training and test: a model presenting a good value for both, proves to have capability to learn and that it can classify independent and identically distributed data unseen by the model during training.
The graph neural network in the article [45] itself did not work as initially expected. The code written in Tensorflow (the first version) was hard to be completely translated in Pytorch. The first objective was to obtain a program able to take article's dataset and reproduce the same results; this has not been achieved but interesting analysis emerged from this work.

4.1.1 Batching procedure

A difference in the implementation with Pytorch was about how data was managed. The original program built batches collecting graphs in a single giant block adjacency matrix, wasting memory in the process. As explained in the preceding chapter, Pytorch Geometric libraries use optimized batching applying the sparse representation (list of nodes presenting an edge). This strategy is not useful in our case, since in the algorithm of the model, the adjacency matrix is required for message passing. The adopted solution is to take each of those blocks and stack them in a new dimension of the adjacency tensor.
This is done using two functions that can be consulted on my GitHub page in `Batching/Article_to_PyG.py`. Furthermore, the shape of tensors following the algorithm needs to be changed: functions defined for that purpose are available inside the file containing the network. The relative code, along with the completing details, are available in "Batching" and "Blocking" folders on GitHub page[*].

4.1.2 First attempt of hyperparameter tuning

A large number of executions of the program were conducted to find the optimal configuration of hyperparameters for the first model. This section will provide a brief summary of the results, presenting only a few key plot figures, showing the achieved accuracy.
Those program executions, presented in figures 4.1 and 4.2 are done using the hardware equipped on my laptop.
Observing those plots, it is clear that since the test accuracy is almost flat in figure 4.1 and ranging between 0.4 and 0.6 for 4.2, the model is not adapting to new data hence it is not learning. About the first run in figure 4.1, the hyperparameters were: batch size = 20, i.e. the model read 20 graphs per iteration over the dataset, learning rate $\eta = 10^{-5}$, embedding dimension set to 16 and internal LSTM cycles $T = 8$. The dataset employed (both train and test) was provided by article [45] authors, available at [49]; the size of each graph is random, the number of nodes is

[*] https://github.com/vmarc0/Master-Thesis.git

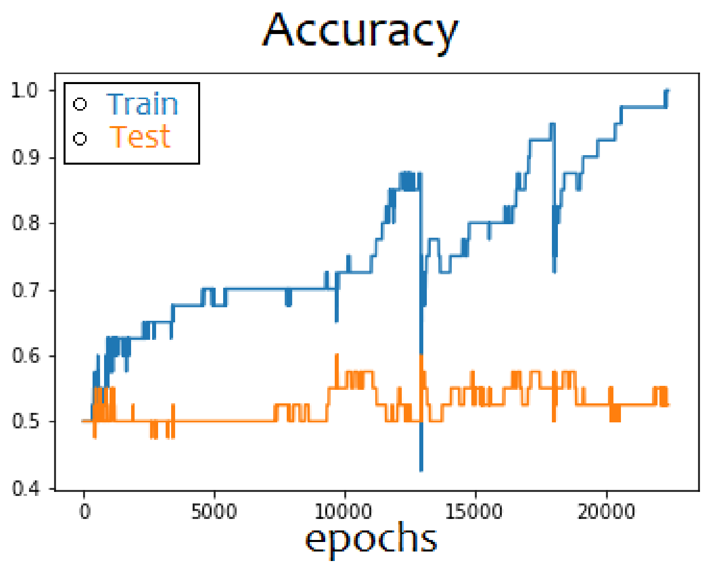


Figure 4.1. Accuracy of the first model, neural network run on article [45] dataset.

uniformly extracted between $N = 40$ and $N = 60$.
For the experiment in figure 4.2, batch size = 10, same learning rate but T and embedding dimensions equal 32. These particular combination of hyperparameters was identified through an extensive series of experiments; while many runs with similar settings produced consistent results, other configurations failed to produce significant improvements. At times, after hours of training, the model's accuracy and loss would remain stagnant, as illustrated in the accompanying figure 4.3.

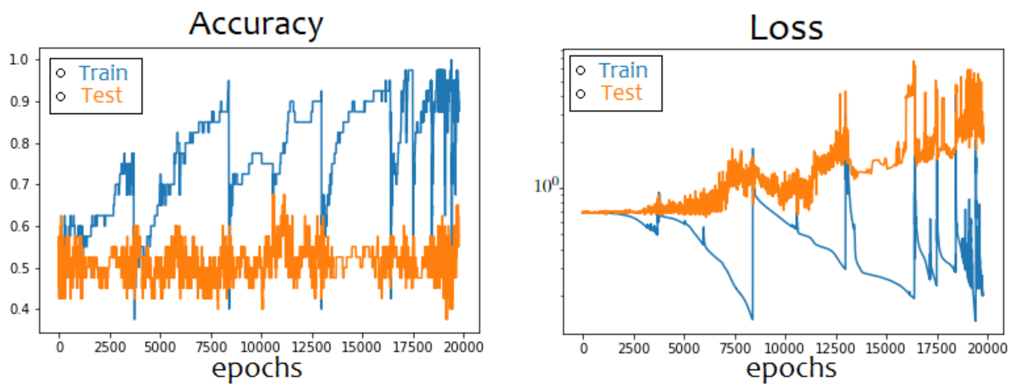


Figure 4.2. Accuracy and Loss of one example run. Dataset from [45].

It is worth to notice that the number of epochs is extremely high: this has been set in order to let the model increase in accuracy. Another feature that draws attention in those plots is the particular increasing trend in the training accuracy (decreasing for the loss) with sudden drops, followed by a quick recovery, increasing again.
This phenomenon may happen due to several reasons: in this case, the neural net-

work is probably overfitting the training dataset. Further reasons could be due to convergence problems: during training, the model may have difficulty converging towards an optimal solution.
In the general case, this may be caused by various factors, such as using a learning rate that is too high or too low, or having noisy or incorrectly labeled data. With regard to our case of interest, those are not possible causes since the dataset was the same employed in the original experiment and the hyperparameters were tuned to be the optimal ones, as explained before. Therefore, the main cause could be choosing an inappropriate neural network architecture.
Article's neural network reveals itself to be not a reliable nor effective model, independently from the set of hyperparameters. Indeed, after a huge quantity of attempts - in which, for every failure, the model was refined with small changes in the algorithm, in order to try plentiful possible different implementations - with the aim to reproduce the results claimed in the original article [45], it is conceivable to interpret those results as a circumstance of incomplete evidence: the model works only employing the original code on the original dataset, a setting different from that leads to the outcomes exposed so far.

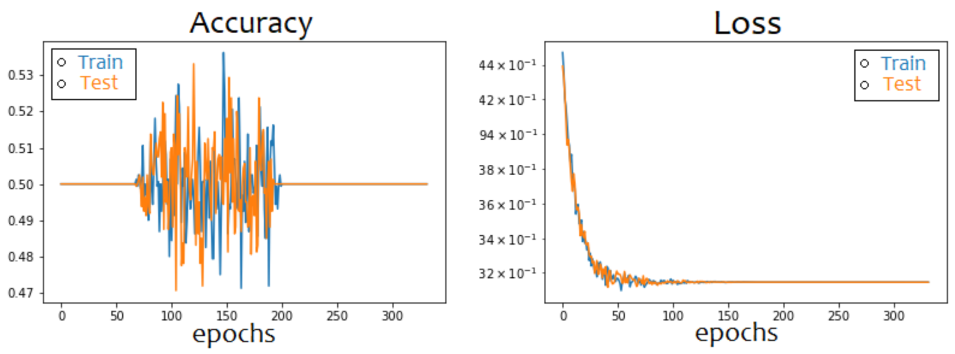


Figure 4.3. Accuracy and loss of a bad run.

4.1.3 Check for overfitting during the model training

To further investigate the reason of the unsuccess in reproducing the results from the original paper [45], a more powerful HPC system for deep learning has been used. The subsequent results are displayed in Figure 4.4 where the training and test loss have symmetrical but opposing trends. This indicates that the network was overfitting the training dataset.
In response, regularization techniques such as dropout and weight decay were applied, but with small initialization parameters, the network continued to overfit.
Indeed, these methods are inserted in the model using some dedicated parameters; if those are set to zero, regularization is ineffective, while increasing those variables, the effects are more pronounced.
Interestingly, as soon as the regularization was strengthened beyond a certain threshold, the network stopped learning, as depicted in figure 4.5.
This indicates that after a certain regularization threshold, the expressive power of the model during the training stage becomes too small to learn the complex

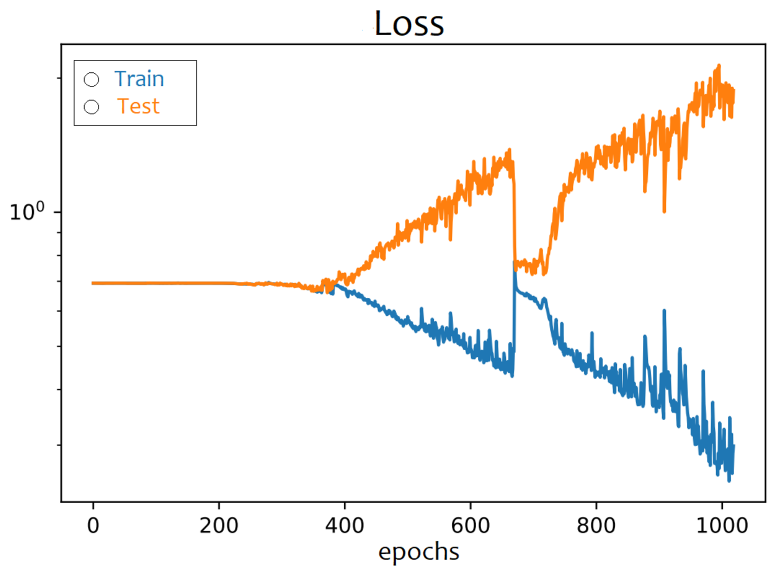


Figure 4.4. Typical symmetric but opposite trends of loss functions when the model is overfitting.

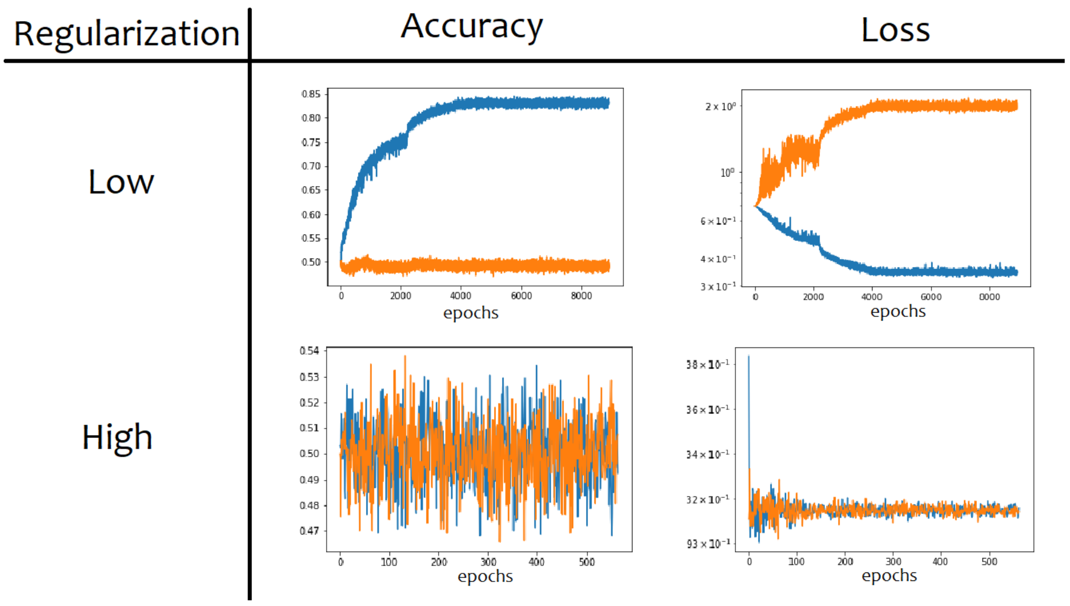


Figure 4.5. Results using regularization techniques with lower and stronger parameters.

structures in the input data.
For this reason, it has been decided to discard this approach and to move to a more modern implementation of a neural network based on graph convolutional layers.
Another reason to move to the graph convolutional implementation is in the higher computational efficiency with respect to the RNN approach.
The latter in fact is intrinsically non parallelizable and so unable to leverage the acceleration provided by the HPC system GPUs.

4.1.4 Attempt to execute original paper's code

As a final check, the original code, provided by article [45] authors, have been executed. The original dataset is composed of graphs with different chromatic numbers: indeed, in article's algorithm, the neural network solves the classification task for the lower possible Q_{min} (which is equal to 3) and, if the answer is negative - i.e. the model predicts that the graph is not colorable with 3 colors - the model is again applied to classify the same graph with $Q_1 = Q_{min} + 1$. This procedure is replicated until the model answers positively: in that case, Q_k is considered to be the predicted chromatic number χ. If the neural network classifies the graph as not colorable even for $Q_{max} = 8$, the algorithm continues proposing the next graph to be classified.
In this manner, the accuracy is computed considering, among all samples, only those graphs whose predicted chromatic number coincides with the real one, information that is available for each graph in the dataset. This is why, following that procedure, accuracy may be below 50%.
Running the model using the original code on the dataset provided by article's authors, the results obtained in this attempt are the following, where accuracy is computed on the testing dataset:

- At 1/5 of training, Accuracy: 58%
- At 4/5 of training, Accuracy: 73.7%
- Training concluded, Accuracy: 74.8%

The training lasted a total of 15 days (around 360 hours). This result means that the model is capable of learning and somehow solve the problem, but with low accuracy and only using the provided dataset. To evaluate the performance of the same code on a different dataset, the program had been executed on a dataset generated with my code based on the adversarial method: for each colorable graph produced with the planting procedure, the other one not colorable was obtained adding a frozen edge to the first one - similarly to the original dataset. Therefore, still using the original code and calling *new dataset* that dataset generated with my code:

- Training done on the original dataset, Accuracy on *new dataset*: 17.7%
- Training done on *new dataset*, Accuracy on *new dataset*: 15.6%

The obtained results strongly point in the direction of a choice of model's parameters made by the authors of the original article that works only with the specific dataset used in the study and that does not generalize well. For this reason, it has been decided to change strategy implementing a new model.

4.2 Transition to a new model

In order to move on, a couple of new models were implemented, one more similar to the one used so far, but without LSTM, and another simpler one - with less learning parameters - to avoid the often encountered overfitting; both model written with convolutional layers.

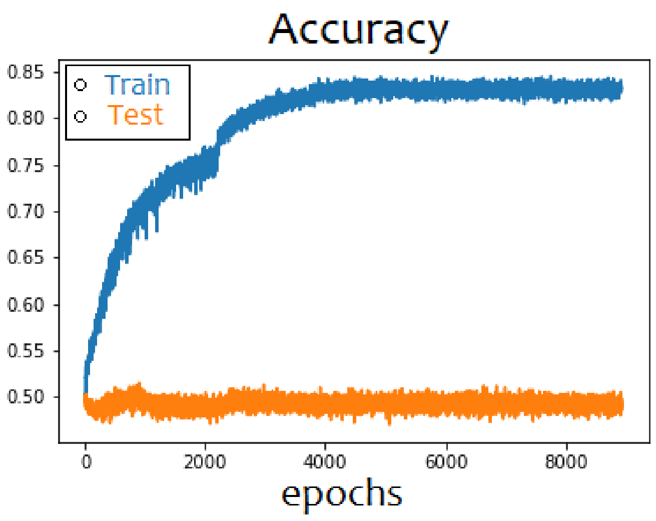


Figure 4.6. Last attempt with a new model on the regular dataset.

Unfortunately, these new models did not show any evidence of effective learning, as can be observed from Figure 4.6. Subsequently, following the suggestion of my supervisor, the goal was simplified, aiming to obtain a network capable of classifying graphs based on a dataset where nodes' features, when the graph was colorable with Q colors, followed the right color distribution corresponding to the problem solution. To achieve this new goal, the dataset had been regenerated, adding 10% more edges between nodes of the same color, for non-colorable graphs.

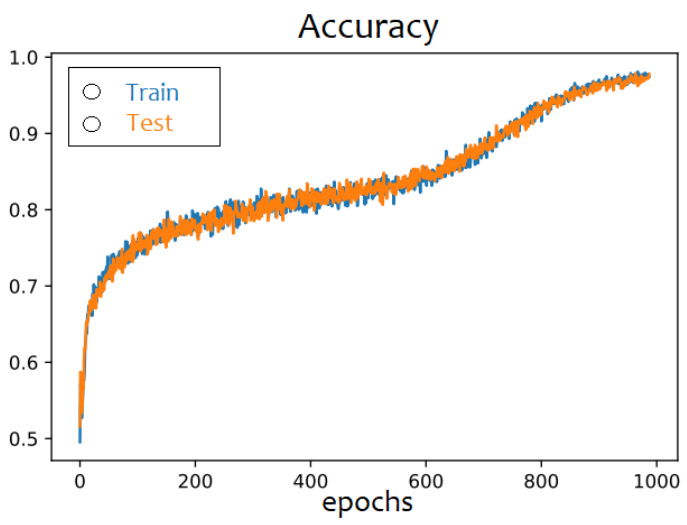


Figure 4.7. A first promising result of new model's accuracy, using the new dataset.

This modification led to finally obtain a model that performed highly accurately on both the train and test datasets, as shown in figure 4.7. In those cases, the dataset presented graphs with equal size, each of them containing $N = 100$ vertices.
These results confirmed the validity of the last model written for this thesis project,

but they also underlined the importance of how the dataset is structured. Indeed, it is worth to notice that employment of graph convolutions was a strategy already been attempted; the challenge was posed by the complex and demanding structure of the dataset described in the article, where graphs were differentiated between satisfiable and non-satisfiable using the adversarial method, and solely by the presence or absence of a single edge, referred to as the *frozen edge*.
From now on, all presented results are obtained employing the new dataset where the chromatic number $\chi = 5$ is fixed.

4.3 Performance of Convolutional Graph Neural Network

The idea of incorporating convolutional layers into the architecture of the graph neural network was present from the outset of this thesis project. Once the path to be followed was understood, the model was refined, resulting in the one presented in the previous chapter.
This final neural network was applied to the new dataset, where UNSAT graphs had 10% more edges, and SAT graphs were equipped with vertex features corresponding to the satisfiable color distribution; from now on, graphs presenting those features are called *painted* graphs.
This explains the purpose of variable *painted* in listing 3.5, used to generated the proper dataset.
For UNSAT graphs, edges were 10% more populated in order to help the model with a dataset presenting clearer attributes.

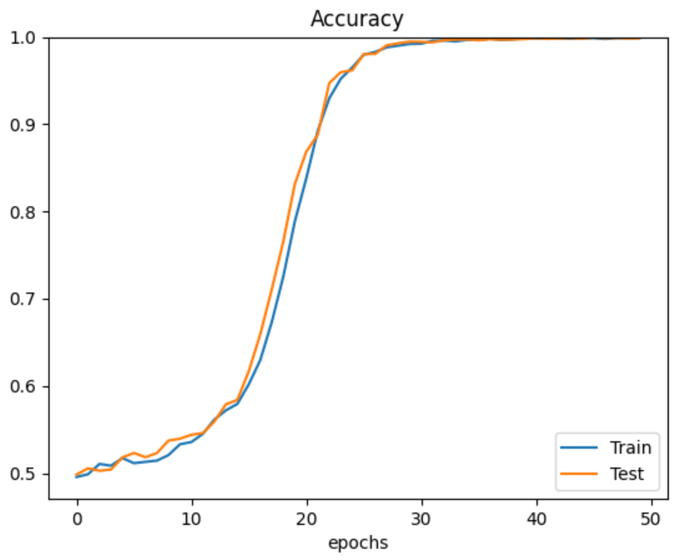


Figure 4.8. Accuracy of the model on painted graphs. Connectivity $c = 40$ and number of nodes $N = 100$.

Although the outcome of the neural network applied on this dataset was adequate,

since the objective was to study the performance (intended as accuracy) as a function of connectivity, comparing those results could lead to some difficulties, because of the difference in the total number of edges.
Furthermore, as suggested by my co-advisor, a probable interpretation of the outcome could be that the neural network learns to classify graphs only gaining information about the connectivity.
In order to avoid those problems, the dataset generating function has been modified to create both SAT and UNSAT graphs with the same connectivity.
Therefore, considering that high connectivity values - $c \gg 20$ - correspond to the computationally easy range, in order to begin with the first run of the new model on this new dataset, it had been chosen to set $c = 40$, following the hint given by my co-advisor.

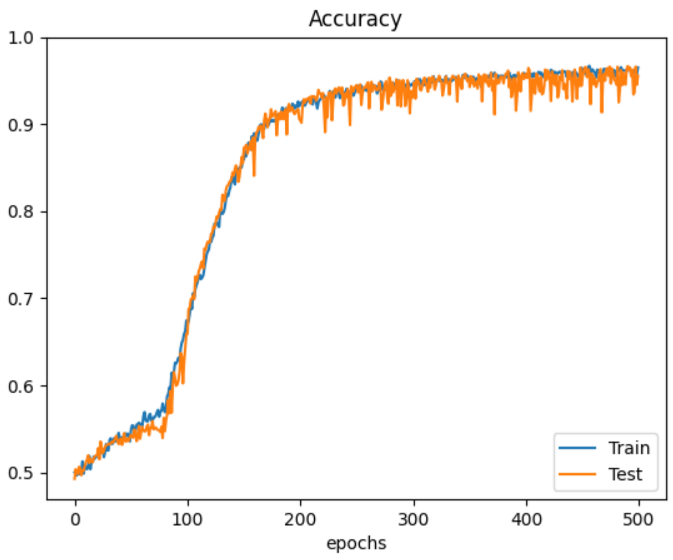


Figure 4.9. Accuracy of the model on standard graphs with no prior color distribution. Connectivity $c = 40$ and number of nodes $N = 100$.

This is valid for chromatic number $\chi = 5$ which is now fixed for every graph, to analyze vertex coloring.
Observing figure 4.8, it is displayed the accuracy performed during training where the connectivity is $c = 40$, size of each graph in the dataset being $N = 100$. Since the model proved to be valid, a test was carried out removing the prior distribution of colors and providing the model only graphs with node degree as node features.
Surprisingly, the network was able to classify new graphs after training on this new dataset, returning to the original task, i.e. to classify graphs focusing only on the topological structure. The ability to solve this much difficult task makes the model more interesting with respect to the one able to classify graphs that are already colored.
The results - available looking at the figure 4.9, where the connectivity was now the same as in the painted dataset $c = 40$ - have improved since the model was refined, surpassing those in the previous figure 4.7, comparing the total number of epochs

required to achieve those accuracy values.
It is clear, looking at the figure 4.9 that the strategy, employed to simplify the task introducing the *painted* variable was not required in order to obtain an effective model. Hence, the final version of the Convolutional Graph Neural Network model is expected to be executed only on those graphs which are not equipped with additional helpful information, in order to recall the original task aimed to be the main goal of this thesis work.
Nevertheless, presenting the results computed on both datasets (painted and standard) is an additional instrument to evaluate the performance of the neural network.

4.3.1 Performances with different connectivities

In this paragraph will be reported a collection of results for different values of connectivity. Having exposed the case of $c = 40$, it is interesting to continue with connectivity values below $c = 20$, getting near the computationally hard phase. It is worth to notice that the mentioned hardness is experienced using traditional algorithms written to solve the problem in a direct way.

Below c = 20

The range of connectivities between $c = 13$ and $c = 20$ has been completely explored with $N = 100$ fixed, running the program with those values set as hyperparameters. It is not interesting to share the details of each run since outcomes relative to adjacent c values were rather similar. Thus, only the significant part of the obtained results will now be presented, starting with c = 19.

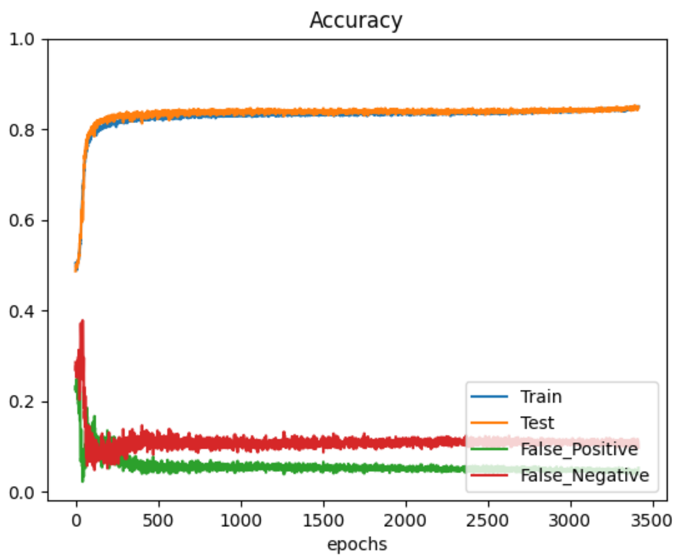


Figure 4.10. Accuracy for $c = 19$ computed on a standard (not painted) dataset with $N = 100$.

The total number of epochs involved is 3500, a rather high number; that choice was motivated by the objective to let the neural network have all the required number

of epochs to reach the maximum possible accuracy.
In figure 4.10 it is possible to notice two extra curves; those are the false positive (FP) and false negative (FN) values whenever the model failed to correctly classify the sample. Clearly, those curves are related to the accuracy Acc with the relation: $Acc + FP + FN = 1$ as can be checked observing the plot.
Another similar result is the one obtained for $c = 17$. Since there are many plots to be shown, the next ones are collected in appendix A. In figure A.1 the trend is equivalent but the difference is the time required to reach the plateau accuracy value, other than the upper limit value achieved.
Indeed, for $c = 17$ it is around $Acc = 82.8\%$ while for $c = 19$ data shows $Acc = 85\%$. Equivalently, for $c \in [13, 16]$ the results show the same attributes; as the connectivity decreases, the plateau achieved decreases too, arriving to $Acc = 75\%$ for the last $c = 13$ displayed in figure A.2.
Another interesting characteristic is the fact that *test* accuracy, at the end of the training, is usually a little higher than *train* accuracy. This could sound rather strange, but considering that between layers of the neural network model is inserted a strong dropout, *train* neurons had been hampered while during *test* they had full capacity thus gave a slightly better prediction, increasing the overall accuracy.
As it is clear observing those plots, the neural network presents an interesting quality: the performance is similar and it appears to not suffer the same computational difficulty as the standard algorithms, although it is important to notice that numerical experiments are done with small values of N, which is below $\sim 10^3$.

Above c = 20

In this different region of connectivities, the situation presents some changes: the model reaches the upper limit accuracy with far less epochs.
For this reason, it has been decided to compare learning results forcing to stop the training as soon as the *train* accuracy achieves a predetermined threshold, in this case set to $Acc_{lim} = 85\%$. Starting from $c = 20$, the first result is displayed in figure 4.11.
The interesting information to focus on, in figure 4.11, is the fact that training ended in 700 epochs. This is the number to be compared with next trainings at higher connectivities. Moreover, the neural network reaches the plateau in (around) 150 epochs, a little less than trainings performed on $c < 20$. Continuing with next connectivity values, it has been studied the values $c \in [20, 30]$ since those from 30 to 40 are really similar and there is no peculiar further attribute to analyze.
As before, outcomes for adjacent c values are similar, thus the analysis continues skipping the odd connectivities, in order to discuss half of the results. In figure A.3 it is possible to notice that, in order to achieve the external limit imposed of $Acc_{lim} = 85\%$, the total number of epochs is exactly the half of those employed in 4.11.
With only two steps in the connectivity space, the algorithm showed a considerable gain in algorithmic efficiency.
Looking at data behind figure A.4, the total number of epochs is 120. Data for the run with $c = 26$ displayed in figure A.5 report a total number of epochs equal to 110. To conclude, data behind the run depicted in figure A.6 show a value of 90 as

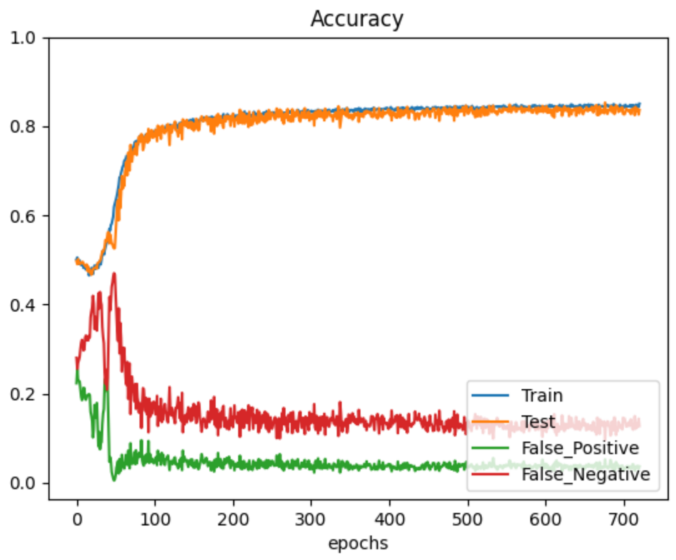


Figure 4.11. Accuracy for $c = 20$ and $N = 100$, computed on a standard dataset and stopped at $Acc = 0.85$.

the last epoch that conclude the training.
All those numbers are collected in table 4.1 where can be directly compared. It emerges that the growth in efficiency is faster in the first range near $c = 20$, then it slows approaching a limit value corresponding to the algorithmically easy connectivity area.

Connectivity	Total epochs
20	700
22	350
24	120
26	110
28	90

Table 4.1. Comparison between training duration values expressed in terms of total epochs to reach plateau value in accuracy, for different values of connectivity.

4.4 Generalization

In various executions of the program, the model underwent extensive training for a significant number of epochs; the strong dropout helped to mitigate the expected overfitting. It is reasonable to consider those dropout layers as the reason why, after all those epochs, the accuracy training reaches a limit value without further improvement even after numerous epochs.
Traditionally, regularization methods, such as dropout, are believed to be necessary for neural networks to generalize their learning. A well-trained model should be

capable of making accurate predictions on unseen data, even if the data has different characteristics, while maintaining the overall structure that assigns the correct labels. It is commonly thought that small generalization errors are a result of either the properties of the model architecture or the regularization techniques used during training.
In a relatively recent paper [51] through extensive systematic experiments, authors show how these traditional approaches fail to explain why large neural networks generalize well in practice. Specifically, their experiments establish that modern convolutional networks for image classification trained with stochastic gradient methods easily fit a random labeling of the training data. This phenomenon is qualitatively unaffected by explicit regularization, and occurs even if they replace the true images by completely unstructured random noise.
The authors argue that current theories of generalization in deep learning do not adequately explain the empirical results seen in practice, and that a new theory is needed to fully understand how these models work. This paper is significant because it provides a new perspective on deep learning generalization and it is reasonable to suppose that deep neural networks, at least partially, work mainly by memorization, more properly by interpolation between the memorized examples.
From this article, it emerges that explicit regularization may improve generalization performance, but is neither necessary nor by itself sufficient for controlling generalization error.
Considering all those insights, it has been investigated the generalization property of the Convolutional Graph Neural Network model not necessarily expecting a sufficient outcome, despite the strong regularization techniques employed.
Performing the training on a dataset with specific parameters, namely the number of nodes N in each graph and the connectivity c, the trained model has been applied to classify graphs from a dataset with slightly different parameters.
In particular, it has been tested the network trained with c = 15 on a dataset with c = 16 and viceversa; trained on c = 19 and tested with c = 20, and viceversa. For those cases, N was fixed to 100. The only notable value is the case of $c_{train} = 20$ and $c_{test} = 19$ with a computed accuracy $Acc = 85\%$ while for all the other cases, the accuracy fluctuates around 50%, i.e. the network worked as if there were no training phase.
The ability to generalize has been tested also on the size of the network, with the following results:

# For c = 14, trained on N = 100:

- Test on N = 110, Accuracy: 74.0%
- Test on N = 120, Accuracy: 63.7%
- Test on N = 130, Accuracy: 56.1%
- Test on N = 150, Accuracy: 51.3%

# For c = 18, trained on N = 100:

- Test on N = 110, Accuracy: 69.7%
- Test on N = 120, Accuracy: 56.6%

- Test on N = 130, Accuracy: 51.1%
- Test on N = 150, Accuracy: 50.4%

# For c = 20, trained on N = 100:

- Test on N = 110, Accuracy: 67.9%
- Test on N = 120, Accuracy: 54.3%
- Test on N = 130, Accuracy: 49.9%
- Test on N = 150, Accuracy: 50.0%

# For c = 40, trained on N = 100:

- Test on N = 110, Accuracy: 82.8%
- Test on N = 120, Accuracy: 57.8%
- Test on N = 130, Accuracy: 51.4%
- Test on N = 150, Accuracy: 50.0%

Based on these accuracy scores, specifically on the first experiment, it can be concluded that the neural network is unable to generalize in extrapolation. This is not entirely surprising, given the new considerations about the limitations of generalization.
It is interesting to note that the model struggles to generalize even on graphs that only differ in size, as indicated by the varying values of N, except for really small variations as 10%. Indeed, this test has been executed also with the following values: c = 17 training, tested on a c = 40 and viceversa; fixing c = 15, training on N = 100 and testing on N = 200, 500, 1000. For all these last results, the accuracy scored 50%. It has also been tried to increase the size of weight parameters space, redefining the model oversizing the dimension of **W** matrices, but the outcome did not change at all.

4.5 Comparison for size and time scaling

The Convolutional Graph Neural Network model was employed to study the connectivity phase space. To further develop the analysis of the model properties, a study was conducted to observe the performance of the model when scaling the sizes of each graph. Since there is a limit of accuracy for each connectivity value, this study allows us to observe whether the network achieves the same accuracy for different system sizes, as well as analyze the time it takes to reach a certain value. By collecting timing data, a comparison can be made with classical algorithms. The study was conducted in the hard/easy transition zone, and representative connectivity values of c = 15, c = 17, and c = 20 were chosen.
Starting with the first value, an execution of the program was carried out with c = 15 and N = 200, 500 (the total number of graphs in the dataset was kept fixed at 10,000), and the results are shown in figures A.7 and A.8. A particular trend is found in the case with N = 500 displayed in figure A.8.
For values of N above 500, in the case of c = 15, the neural network did not show

Size (N)	Accuracy
200	78.5%
300	76.2%
500	73.3%
800	68.3%
1000	66.5%

Table 4.2. Accuracy scores yielded for c = 17, scaling the size of each graph in datasets.

Size (N)	Accuracy
200	83.5%
300	81.2%
500	77.4%
800	74.1%

Table 4.3. Accuracy scores yielded for c = 20, scaling the size of each graph in datasets.

any capability to learn; for this reason there is no plot for those parameters.
Going beyond, in appendix are available plots for c = 17. Further experiments have been done with other values of N, collected in table 4.2. Concluding with the experiments made with c = 20 - figures A.11 and A.12 - in this case it is interesting to notice how the system presents peculiar fluctuations, even if the connectivity value is quite distant from the hard/easy transition. Values are collected in table 4.3.

4.5.1 Simulated Annealing

All the results presented about efficiency of neural network algorithms, intended as the accuracy score followed by the time spent to reach it, to be meaningful, must be compared to the traditional algorithms applied in research of this field.
In order to do this, my co-advisor provided a program based on a technique called simulated annealing, a method for solving unconstrained and bound-constrained optimization problems. The method models the physical process of heating a system, it slowly lowers the temperature to decrease defects, thus minimizing the system energy.
The algorithm starts with an initial configuration and iteratively explores neighboring configurations by making small changes to the current one. The algorithm uses a "temperature" parameter that controls the amount of randomness in the search process. Higher temperatures allow for more exploration and observation of configurations, considering also those that are worse than the current one, while lower temperatures reduce randomness and tend to focus only on exploitation of states with lower energy.

Table 4.4. Performances obtained with the simulated annealing algorithm (left) compared to neural network (right).
In the first column, c and N are connectivities and number of vertices per graph, respectively.
Time performances (run time) for the neural network refer to the time employed only to assign predictions over the dataset; the time spent for training is listed alongside.
Values computed for SA consider accuracy scores obtained for both planted and random graph datasets.

c, N	Accuracy$_{SA}$	Run time	Accuracy$_{NN}$	Training time (total)	Run time
15, 100	96.5%	2h	75% (79.6%)	30 min (4h)	3.5 sec
15, 200	91.63%	4h 20 min	70.8%	1h 30 min	3.9 sec
15, 500	70.0%	12h 10 min	70.0% (78.6%)	5h 50 min (9h 40 min)	6.0 sec
17, 100	99.9%	1h 45 min	80.2% (82.9%)	40 min (2h 40 min)	2.8 sec
17, 200	99.9%	4h	78.5%	3h 10 min	3.3 sec
17, 500	99.8%	10h 30 min	73.3%	3h 20 min	3.5 sec
20, 100	100%	1h 50 min	85.1% (86%)	30 min (2h 30 min)	2.6 sec
20, 200	100%	4h	83.5%	3h 30 min	3.4 sec
20, 500	100%	11h	77.4%	3h 25 min	3.5 sec

As explained in the first chapter, describing the graph using Potts model, the configurations for which the Hamiltonian (see equation (1.4)) equals zero, correspond to satisfiable states, i.e. when all nodes in the graph are correctly colored following the constraint.
As the algorithm progresses, the temperature is decreased gradually over time, reducing the randomness and converging towards a more optimal solution. The process continues until a stopping criterion is met, either temperature $T = 0$, energy $\mathcal{H} = 0$, or when the total number of steps is accomplished, which is fixed at the beginning. Overall, simulated annealing is a versatile and widely used optimization algorithm that is particularly well-suited for solving complex optimization problems with large search spaces and multiple local optima.
The program used is written in C language, the dataset used had the same exact properties of those passed to the neural network - even in this case, each dataset contained 10^5 graphs.
The initial conditions set for each simulated annealing run are an initial temperature of $T_0 = 1.5$ and a total number of steps that is equal to 10^5.
With the C program, simulated annealing has been executed on both random and planted datasets. Considering that SA directly searches the configurations that satisfy the vertex coloring problem, it is clear why the accuracy scored on every random (UNSAT) dataset, not reported in table 4.4, is always 100% for every connectivity value. The numbers reported in table 4.4 are indeed the overall accuracies which comprehend the performance on both SAT and UNSAT graphs.
Observing the compared values in table 4.4, it is possible to conclude that Simulated Annealing give an overall better performance, studying the problem of vertex coloring, scoring higher accuracies. It is interesting to notice that the neural network is faster in order to reach lower values of accuracy.

4.6 Accuracy phase transition

One of the most interesting results of this thesis pertains to the study of the algorithmic phase transition that involves the shift from the hard phase to the easy phase, computationally speaking. As mentioned in the first chapter, BP algorithms have found that for $\chi = 5$, this transition corresponds to the critical value of mean connectivity $c = 16$. It is not guaranteed that the same will emerge from the analysis with neural networks; for example, in literature [52] can be found that - still for Q = 5 - SA has a critical value of $c_{SA} = 18$.
Before examining the overall results, it is useful to review the theory in order to understand what can be experimentally expected from numerical simulations. An illustrative example is provided by the theory of magnetic systems, a topic with a wide range of applications in statistical mechanics of critical phenomena.
Taking a ferromagnet of volume V at (inverse) temperature β^{-1}, subjected to an external uniform magnetic field h, it can be briefly described using a toy model in which the partition function is known to be $\mathcal{Z}(h) = 2\text{cosh}(\beta h V)$. With few computations, using the free energy density f_β, the magnetization m may be calculated:

$$f_\beta(h) = -\frac{1}{\beta V}\log\Big(2\text{cosh}(\beta h V)\Big)$$
$$m(h) = -\frac{\partial f_\beta}{\partial h} = \tanh(\beta h V)$$

From this, an important theoretical aspect emerges: phase transitions are defined in the thermodynamic limit. Indeed, studying the limit $V \to \infty$ this simple model reveals a phase transition described by the parameter h.
Using increasingly larger values for volume, the behavior of magnetization, as shown in figure 4.12, can be observed and in the limit of infinite volume that function becomes $m(h) = \text{sign}(h)$.

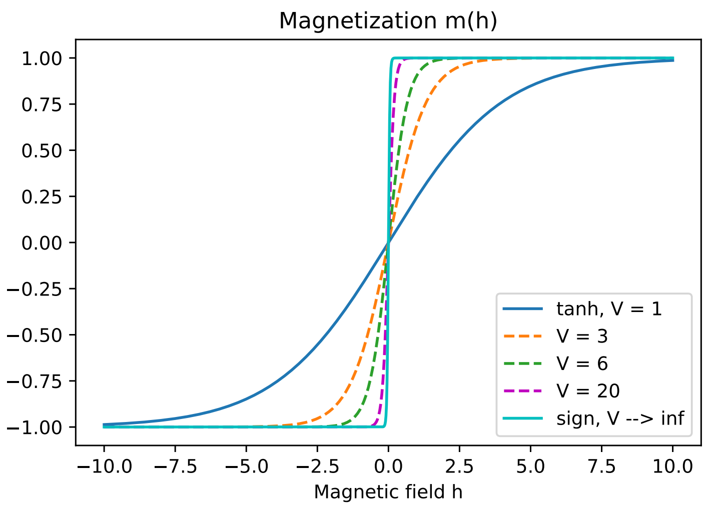


Figure 4.12. Magnetization m as a function of the magnetic field h. In this simulation, $\beta = 0.25$ is fixed while the volume of the system increases, as reported in legend.

In particular, the parameter value at which all the curves intersect themselves is the critical value of the phase transition. In this case, it is clearly $h_c = 0$: an infinitesimal magnetic field is enough to produce a non-zero magnetization.
This may appear to be a purely theoretical argument, since real physical systems of infinite volume do not actually exist, just as it is not possible to draw a graph with an infinite number of nodes. However, a real macroscopic system with finite volume exhibits finite variations of internal energy (in units of k_B, in the proper unit system), near a critical point, for temperature changes of $\sim 10^{-23}$ Kelvin [53]. That is the case where in practice an observable, that is always a continuous function of physical quantities, is stretched in such a way to present a discontinuity.
Similarly, in our case the phase transition should become visible when the system is studied for increasingly larger values of N. Unfortunately, a complete investigation with $N \to \infty$ was not possible as the neural network model did not always provide valid data for both $N > 500$ and for all mean node degree values.
Nonetheless, for initial increasing values such as $N = 100, 200, 500$, the behavior of the accuracy found for neural network and SA algorithms, may be observed in comparison looking at figure 4.13.

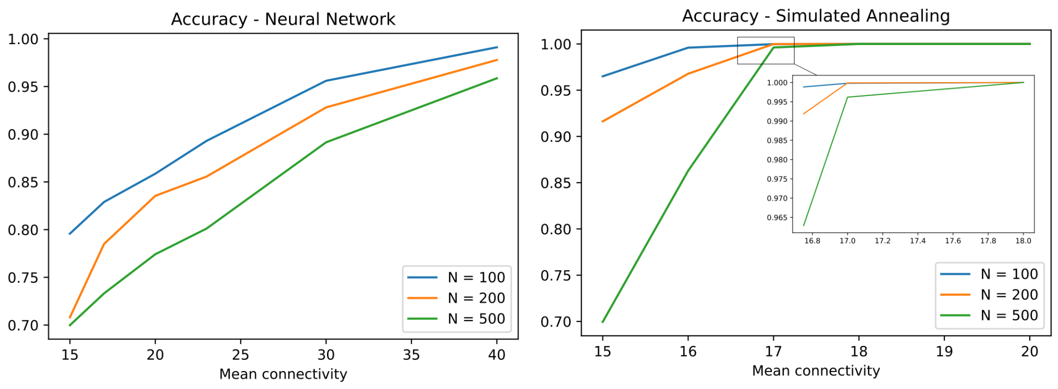


Figure 4.13. Comparison between the results obtained for the neural network (left) and for simulated annealing (right). The right figure presents a third inset figure which is a zoom of the connectivity range for which the three curves intersect.

Since the method adopted to acquire data is based on the fact that above SAT/UNSAT connectivity transition all random graphs are not colorable, the only possible range to investigate is $c \geq 15$.
About the simulated annealing results, the hard/easy transition is not clearly recognizable at first glance, however considering that the three curves meet when the accuracy equals almost 1.0 (which is a possible case), the transition parameter is confirmed to be $c_{SA} = 18$; the inset zoom in (right) figure 4.13 shows the crossing point.
Contrarily, the accuracies of Convolutional Graph Neural Network do not cross for those connectivity values. Since neural networks are basically different in the algorithmic functioning, with this result a possible interpretation is that neural networks do not follow the same computational constraints which regular algorithms undergo.
Indeed, although the outcome is partial due to the lack of data for N above 500, it is

evident that even with more data, the curves do not intersect within the considered connectivity range. Anyway, the trend in left figure 4.13 suggests that at higher connectivity values, the three curves could meet in a common point, probably in a similar way as it happened for the simulated annealing algorithm.

Conclusion

The first objective of this thesis was to obtain an operating model of neural networks able to classify graphs in the context of the vertex coloring problem. Taking into account the overall attempts made during the first part of this thesis work, in general that objective may be considered a not trivial task.
The first model, taken from the article [45] as a reference, did not brought the expected results, leading to an initial production of numerical outcomes without further application to the other aim of this thesis.
The second objective of this thesis was to apply the obtained model to a standard dataset made of graphs generated with specific characteristics: employing the random ensemble of ER graphs, the theory of disordered systems in statistical mechanics could be applied, in order to get a theoretical prediction of the possible outcomes.
Therefore, using the particular methods discussed in the first chapter, the phase space has been analyzed; the mean connectivity of the graphs (or mean node degree) turned out to be the parameter which let the system undergo a phase transition. This also means that the set of random graphs should be studied in the limit of infinite size, i.e. when the number of nodes (consequently the number of edges) goes to infinity.

The vertex coloring problem is a NP-complete problem. In spite of that, the aim in this thesis is not to study the worst cases of that problem.
Therefore the phase space has been explored at mean node degree values corresponding to graphs which are known to be not colorable. In order to train the neural network model, this has been useful to generate the dataset, exploiting the procedure known as *planting* in order to get also colorable graphs.
In this way, the last model, which proved to be valid in order to classify graphs, has been employed.

The main result is the accuracy performance focused on the scaling size examination; the same research has been conducted using standard algorithms, in order to make a comparison. In this respect, the neural network presented peculiar features which distinguish it from any other conventional algorithm. In fact, the neural network model achieved slightly lower accuracies, but in much less time: once the model is trained, the time spent to give a prediction over the whole dataset is about few seconds while other algorithms, like the simulated annealing, require hours. Furthermore, the neural network is faster even presenting training times rather smaller than the time employed by SA to give the outcome, especially for

large size graphs.
On the other hand, although the neural network is much faster on large size graphs, it did not perform well (intended as accuracy score) in the examination of the algorithm in the thermodynamic limit: the model was not able to learn above a threshold of number of nodes. However, the few available data let us conclude that neural networks do not follow the same hard/easy algorithmic phase transition which standard algorithms present, at least not in the range of connectivity examined.

These interesting results are obtained using a neural network model with a simple architecture; further research utilizing more sophisticated models could lead to a deeper comprehension of the underlying reasons connected to these outcomes. Ideally, it would also be compelling to develop a model which is able to return, in those cases where it exists, a solution of the CSP, i.e. a coloring of the graph that properly follows the constraint given by the vertex coloring problem.

Appendix A

Appendix

Here are listed all the plots and figures discussed in the last chapter.

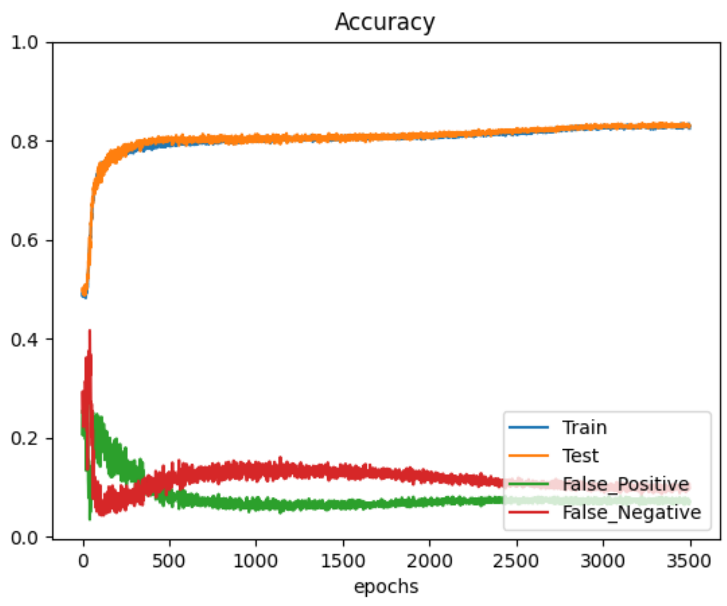


Figure A.1. Accuracy for $c = 17$ and $N = 100$, computed on a standard (not painted) dataset.

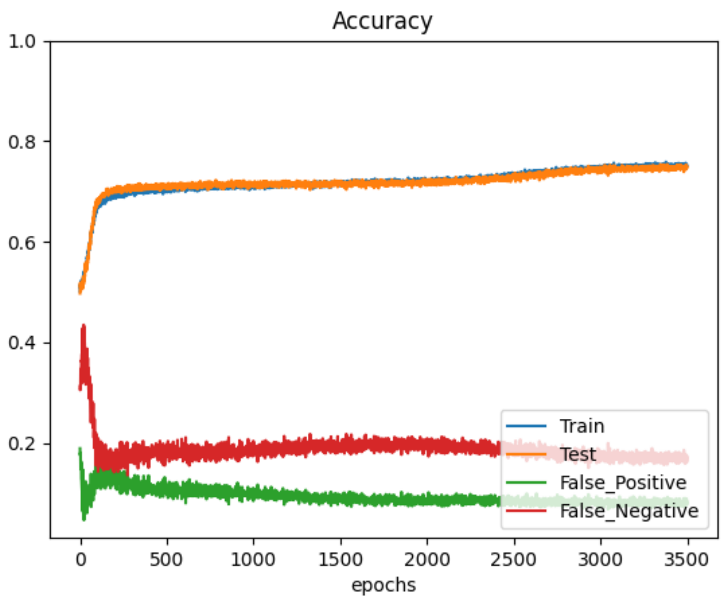


Figure A.2. Accuracy for $c = 13$ and $N = 100$, computed on a standard (not painted) dataset.

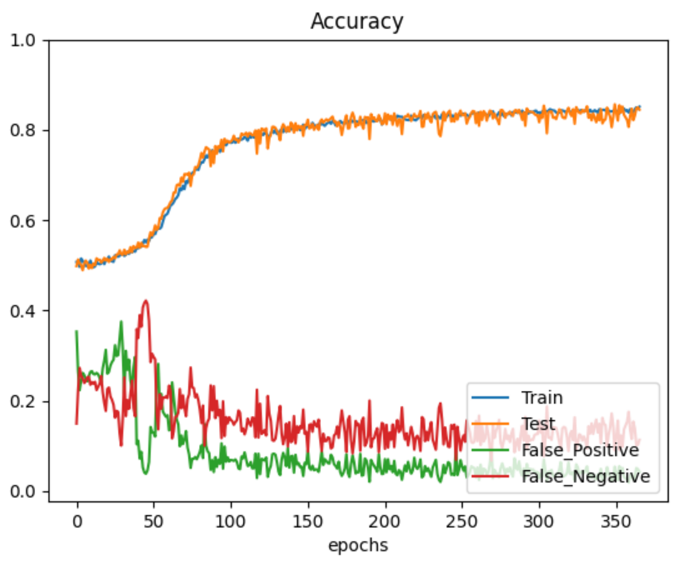


Figure A.3. Accuracy for $c = 22$ and $N = 100$, computed on a standard dataset and stopped at $Acc = 0.85$. Here the total number of epochs is 350.

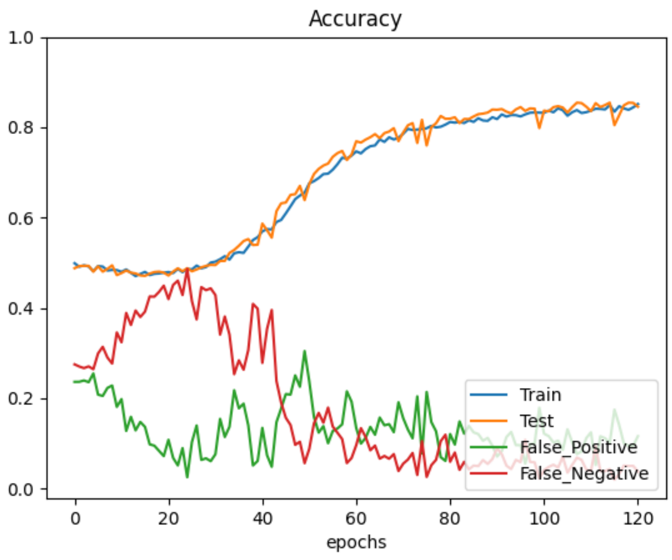


Figure A.4. Accuracy for $c = 24$ and $N = 100$, computed on a standard dataset and stopped at $Acc = 0.85$. Here the total number of epochs is 120.

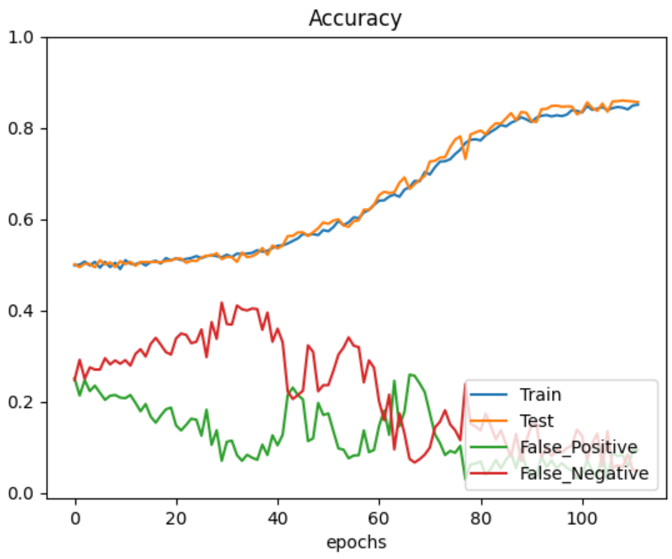


Figure A.5. Accuracy for $c = 26$ and $N = 100$, computed on a standard dataset and stopped at $Acc = 0.85$. Here the total number of epochs is 110.

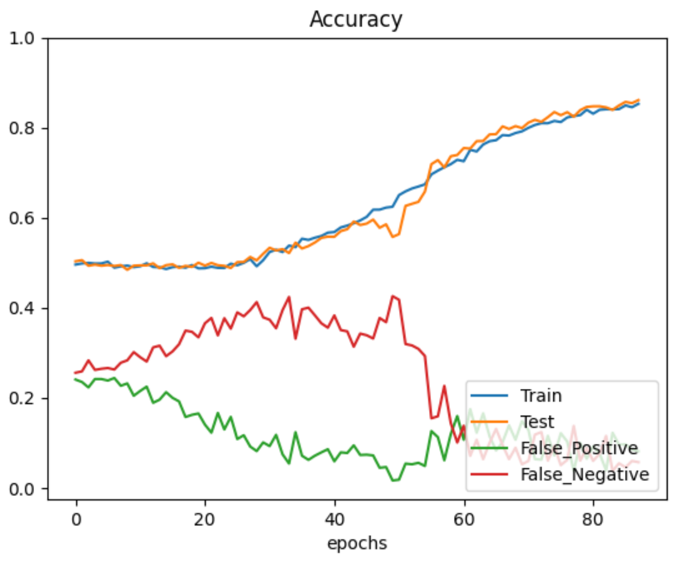


Figure A.6. Accuracy for $c = 28$ and $N = 100$, computed on a standard dataset and stopped at $Acc = 0.85$. Here the total number of epochs is 90.

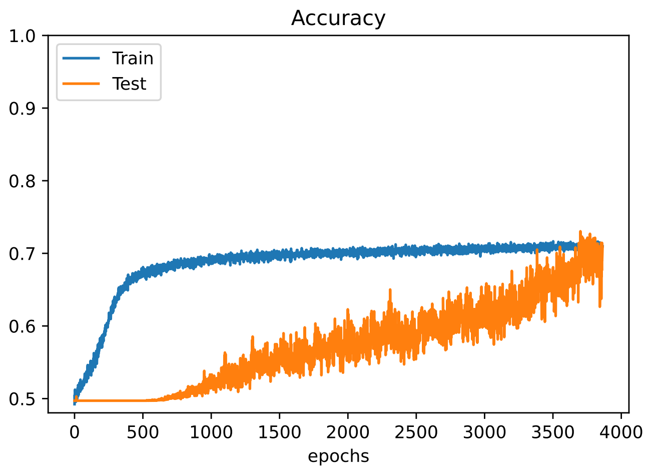


Figure A.7. Run for $c = 15$ and $N = 200$.

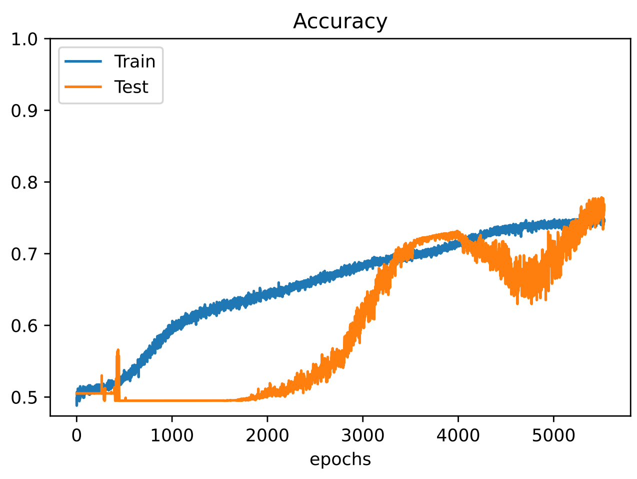


Figure A.8. Run for $c = 15$ and $N = 500$.

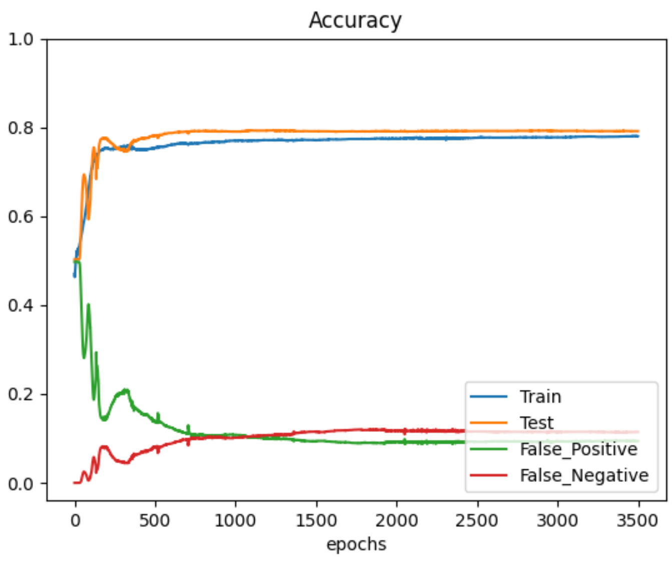


Figure A.9. Run for $c = 17$ and $N = 200$.

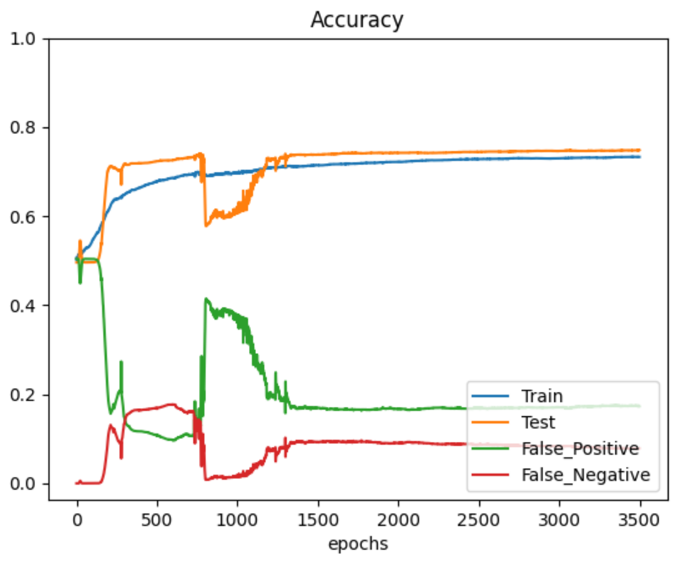


Figure A.10. Run for $c = 17$ and $N = 500$.

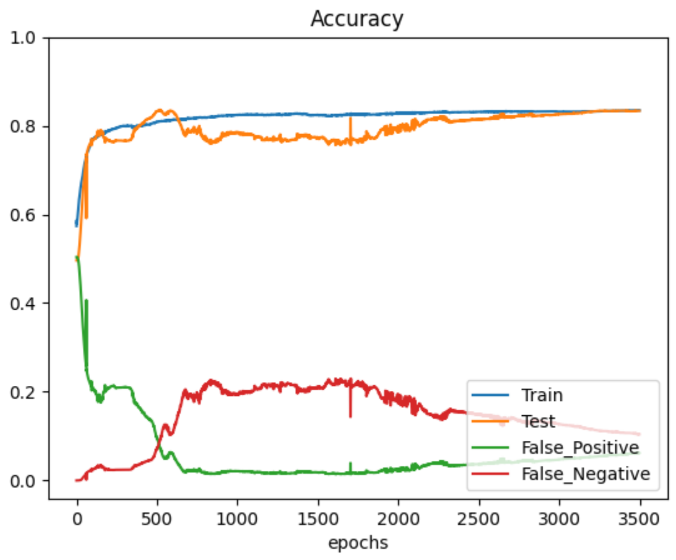


Figure A.11. Run for $c = 20$ and $N = 200$.

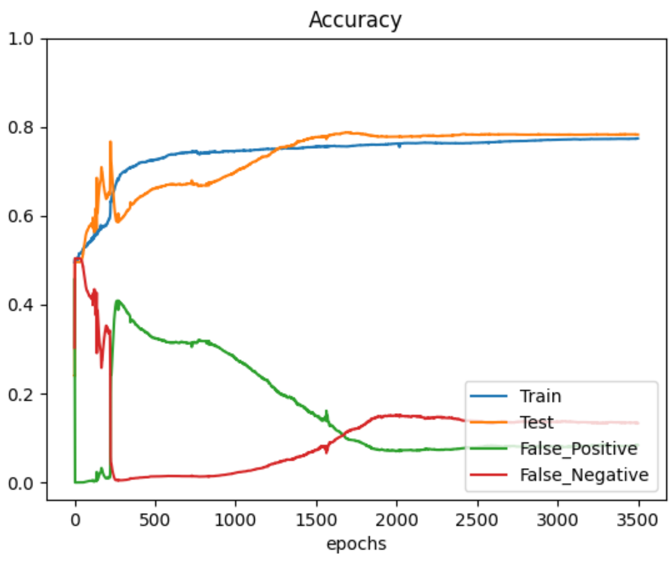


Figure A.12. Run for $c = 20$ and $N = 500$.

Acknowledgements

These pages are addressed to all those people who deserve a special thank. If you can not read the following text, i am afraid it is not written for you, sorry not sorry.

Per cominciare, un ringraziamento al Professor Giagu e alla Professoressa Angelini, i miei due relatori, per la disponibilità nel rispondere ad ogni mio dubbio e richiesta di aiuto e per la cura riposta nella correzione di questa tesi.
Un doveroso ringraziamento a Matteo, conosciuto per caso ad una festa di laurea, per tutto l'aiuto, i consigli e il tempo speso in numerosi appuntamenti, a partire dallo scorso ottobre.

Grazie ai miei genitori per avermi permesso di raggiungere questo traguardo, a lungo atteso ma senza mai avermelo fatto pesare. Un pensiero e un abbraccio a mia sorella Irina, in questo periodo occupata da ben più importanti cambiamenti; spesso distanti fisicamente, ma mai distanti col cuore.
Un ringraziamento speciale ad Enrico, detto Orso e da poco anche cognato, per avermi chiesto puntualmente a fine marzo, a pochi giorni dalla scadenza, se avessi fatto la domanda di laurea per maggio, scongiurando il rischio di rimandare ulteriormente alla seduta successiva.

In merito alla burocrazia, un ringraziamento anche al nostro attuale governo per il decreto milleproroghe, grazie al quale in questa particolare seduta, mi viene concesso di laurearmi tecnicamente nell'anno accademico precedente a questo. Devo ancora capirne il senso, ma nel dubbio grazie. In particolare, un ringraziamento al nostro presidente Giorgia Meloni, adesso possiamo dirlo: ha fatto anche cose buone.

Un caloroso grazie a tutti i Bubboni, fenomenale gruppo composto dalle più disparate personalità, formando nella sua unità un mix esplosivo di simpatia, gioco, sarcasmo, empatia, talvolta cultura e ovviamente una certa dose di geometria, anch'essa essenziale. Questo pensiero include anche Giulio, compagno di crescita personale, che se anche ha intrapreso una strada diversa, non può certo essere escluso da tale descrizione.
Un particolare pensiero va a Nicolò: ci siamo conosciuti all'inizio di questa magistrale e siamo cresciuti assieme lungo questo percorso, costellato di brutte e belle esperienze, nelle quali infine ci siamo sempre riusciti a ritrovare, fra nuove consapevolezze e cambiamenti.

Terminato questo percorso universitario, ci tengo a ringraziare i vari colleghi e amici che mi hanno accompagnato, sia nell'aspetto accademico con dei confronti utili, sia in tutto ciò che fa parte della vita universitaria ma che non riguarda studio e apprendimento. In particolare, penso a Marco Tosoni, Antonio, Vigilante e Raoul; meno recenti ma non meno degni di nota, un affettuoso grazie a tutti i miei vecchi amici conosciuti nel percorso della triennale, in particolar modo Danilo e Marco Radiconcini, con cui il legame è rimasto vivo nel corso degli anni.

Per concludere, un ringraziamento a tutte le persone che in questi anni hanno talvolta riempito significativamente le mie giornate, donandomi preziose idee, spunti, riflessioni che mi hanno aiutato a crescere o regalandomi esperienze a cui ripenso con felicità.

Bibliography

[1] A. M. Turing. “Computing machinery and intelligence”. In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. URL: `https://doi.org/10.1093/mind/LIX.236.433`.

[2] Ronald L. Rardin R. Gary Parker. *Discrete optimization*. Academic press, inc., 2014, pp. 1–2.

[3] Béla Bollobás. *Modern Graph Theory*. Springer, 2014, pp. 67–68.

[4] K. Appel and W. Haken. “Every planar map is four colorable”. In: *Illinois Journal of Mathematics* 21 (1977), pp. 421–491.

[5] Dexter C. Kozen. *The Design and Analysis of Algorithms*. New York, NY: Springer New York, 1992. ISBN: 9781461287575 9781461244004. URL: `http://link.springer.com/10.1007/978-1-4612-4400-4` (visited on 04/12/2023).

[6] Marc Mezard and Andrea Montanari. *Information, physics, and computation*. OCLC: ocn234430714. Oxford ; New York: Oxford University Press, 2009. ISBN: 9780198570837.

[7] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. URL: `https://doi.org/10.1007/978-1-4684-2001-2_9`.

[8] I. Dagan, M. C. Golumbic, and R. Y. Pinter. “Trapezoid graphs and their coloring”. In: *Discrete Applied Mathematics* 21.1 (1988), pp. 35–46. ISSN: 0166-218X. URL: `https://www.sciencedirect.com/science/article/pii/0166218X88900327`.

[9] Jort van Mourik and David Saad. “Random graph coloring: Statistical physics approach”. English. In: *Physical Review E* 66.5 (Nov. 2002). ISSN: 1539-3755.

[10] S F Edwards and P W Anderson. “Theory of spin glasses”. In: *Journal of Physics F: Metal Physics* 5.5 (May 1975), pp. 965–974. ISSN: 0305-4608. URL: `https://iopscience.iop.org/article/10.1088/0305-4608/5/5/017` (visited on 04/13/2023).

[11] Florent Krzakala and Lenka Zdeborová. *Statistical Physics Methods in Optimization and Machine Learning*. Dec. 2022. URL: `https://sphinxteam.github.io/EPFLDoctoralLecture2021/Notes.pdf`.

[12] M. Mezard, G. Parisi, and M.A. Virasoro. *Spin Glass Theory And Beyond: An Introduction To The Replica Method And Its Applications*. World Scientific Publishing Company, 1987. ISBN: 9789813103917. URL: https://books.google.it/books?id=DwY8DQAAQBAJ.

[13] M.C. Angelini, F. Caltagirone, and F. Krzakala. *Statistical Physics of Inference problems*. URL: https://www.ipht.fr/Docspht//articles/t14/045/public/notes.pdf.

[14] Thomas Parr, Dimitrije Markovic, Stefan J. Kiebel, and Karl J. Friston. "Neuronal message passing using Mean-field, Bethe, and Marginal approximations". In: *Scientific Reports* 9.1 (Feb. 2019), p. 1889. ISSN: 2045-2322. URL: https://doi.org/10.1038/s41598-018-38246-3.

[15] Koller D. and Friedman N. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[16] M. Mézard and G. Parisi. "The Bethe lattice spin glass revisited". In: *The European Physical Journal B - Condensed Matter and Complex Systems* 20.2 (Mar. 2001), pp. 217–233. ISSN: 1434-6036. URL: https://doi.org/10.1007/PL00011099.

[17] Lenka Zdeborová and Florent Krzął̧kała. "Phase transitions in the coloring of random graphs". In: *Phys. Rev. E* 76 (3 Sept. 2007), p. 031131. URL: https://link.aps.org/doi/10.1103/PhysRevE.76.031131.

[18] Marc Mézard and Riccardo Zecchina. "The random K-satisfiability problem: from an analytic solution to an efficient algorithm". In: *Physical Review E : Statistical, Nonlinear, and Soft Matter Physics* 66 (2002). 38 pages, 13 figures; corrected typos, p. 056126. URL: https://hal.science/hal-00002362.

[19] Ludovic Berthier and Giulio Biroli. "Theoretical perspective on the glass transition and amorphous materials". en. In: *Reviews of Modern Physics* 83.2 (June 2011), pp. 587–645. ISSN: 0034-6861, 1539-0756. URL: https://link.aps.org/doi/10.1103/RevModPhys.83.587 (visited on 04/21/2023).

[20] F. Krzakala, A. Montanari, F. Ricci-Tersenghi, G. Semerjian, and L. Zdeborová. "Gibbs states and the set of solutions of random constraint satisfaction problems". In: *Proceedings of the National Academy of Sciences* 104.25 (2007), pp. 10318–10323. URL: https://www.pnas.org/doi/abs/10.1073/pnas.0703685104.

[21] Florent Krzakala and Lenka Zdeborová. "Hiding Quiet Solutions in Random Constraint Satisfaction Problems". In: *Phys. Rev. Lett.* 102 (23 June 2009). URL: https://link.aps.org/doi/10.1103/PhysRevLett.102.238701.

[22] Marc Mezard, Giorgio Parisi, and Riccardo Zecchina. "Analytic and Algorithmic Solution of Random Satisfiability Problems". In: *Science (New York, N.Y.)* 297 (Sept. 2002), pp. 812–5.

[23] Luca Dall'Asta, Abolfazl Ramezanpour, and Riccardo Zecchina. "Entropy landscape and non-Gibbs solutions in constraint satisfaction problems". In: *Physical review. E, Statistical, nonlinear, and soft matter physics* 77 (Apr. 2008), p. 031118.

[24] Giorgio Parisi. 2005. URL: https://arxiv.org/abs/cs/0212047.

[25] Guilhem Semerjian. “On the Freezing of Variables in Random Constraint Satisfaction Problems”. In: *Journal of Statistical Physics* 130.2 (Jan. 2008), pp. 251–293. ISSN: 1572-9613. URL: https://doi.org/10.1007/s10955-007-9417-7.

[26] Fabrizio Altarelli, Rémi Monasson, and Francesco Zamponi. “Can rare SAT formulae be easily recognized? On the efficiency of message-passing algorithms for K-SAT at large clause-to-variable ratios”. In: *Journal of Physics A: Mathematical and Theoretical* 40.5 (Jan. 2007), p. 867. URL: https://dx.doi.org/10.1088/1751-8113/40/5/001.

[27] Lenka Zdeborová and Marc Mézard. “Locked Constraint Satisfaction Problems”. In: *Phys. Rev. Lett.* 101 (7 Aug. 2008), p. 078702. URL: https://link.aps.org/doi/10.1103/PhysRevLett.101.078702.

[28] Lenka Zdeborová and Florent Krzakala. “Statistical physics of inference: thresholds and algorithms”. en. In: *Advances in Physics* 65.5 (Sept. 2016), pp. 453–552. ISSN: 0001-8732, 1460-6976. URL: https://www.tandfonline.com/doi/full/10.1080/00018732.2016.1211393 (visited on 04/12/2023).

[29] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. en. In: *The Bulletin of Mathematical Biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 0007-4985, 1522-9602. URL: http://link.springer.com/10.1007/BF02478259 (visited on 02/04/2023).

[30] Chris M. Bishop. “Neural networks and their applications”. en. In: *Review of Scientific Instruments* 65.6 (June 1994), pp. 1803–1832. ISSN: 0034-6748, 1089-7623. URL: http://aip.scitation.org/doi/10.1063/1.1144830 (visited on 02/04/2023).

[31] Sebastian Raschka, Yuxi Liu, Vahid Mirjalili, and Dmytro Dzhulgakov. *Machine learning with PyTorch and Scikit-Learn: develop machine learning and deep learning models with Python*. eng. Birmingham: Packt Publishing, 2022. ISBN: 9781801819312.

[32] F. Rosenblatt. *The perceptron - A perceiving and recognizing automaton*. Tech. rep. 85-460-1. Ithaca, New York: Cornell Aeronautical Laboratory, Jan. 1957.

[33] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. URL: https://doi.org/10.1038/323533a0.

[34] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[35] Anidhya Athaiya. “Activation functions in neural networks”. In: *IJEAST* 4 (2020), pp. 310–316.

[36] F. Rosenblatt. *Principles of Neurodynamics*. Spartan Books, 1959.

[37] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[38] Kumar Siddharth. “On weight initialization in deep neural networks”. In: (Apr. 2017). URL: https://arxiv.org/abs/1704.08863.

[39] Ken-Ichi Funahashi. “On the approximate realization of continuous mappings by neural networks”. en. In: *Neural Networks* 2.3 (Jan. 1989), pp. 183–192. ISSN: 08936080. URL: https://linkinghub.elsevier.com/retrieve/pii/0893608089900038 (visited on 02/23/2023).

[40] Alex Sherstinsky. “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network”. In: *Physica D: Nonlinear Phenomena* 404 (2020), p. 132306. ISSN: 0167-2789. URL: https://www.sciencedirect.com/science/article/pii/S0167278919305974.

[41] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[42] Goku Mohandas. *Recurrent Neural Networks - Made With ML*. URL: https://madewithml.com/courses/foundations/recurrent-neural-networks/.

[43] P.J. Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560.

[44] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: Palais des Congrès Neptune, Toulon, France, 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

[45] Henrique Lemos, Marcelo Prates, Pedro Avelar, and Luís Lamb. “Graph Colouring Meets Deep Learning: Effective Graph Neural Network Models for Combinatorial Problems”. In: Nov. 2019, pp. 879–885.

[46] Emanuele Rossi, Henry Kenlay, Maria I. Gorinova, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M. Bronstein. *On the Unreasonable Effectiveness of Feature Propagation in Learning on Graphs with Missing Node Features*. 2022. URL: https://openreview.net/forum?id=tx4qfdJSFvG.

[47] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. “Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition”. In: *CoRR* abs/1402.1128 (2014). URL: http://arxiv.org/abs/1402.1128.

[48] Google. *CP-SAT Solver*. URL: https://developers.google.com/optimization/cp/cp_solver.

[49] P. Avelar H. Lemos M. Prates and L. Lamb. *GNN-GCP*. URL: https://github.com/machine-reasoning-ufrgs/GNN-GCP.

[50] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. “Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks”. In: Honolulu, Hawaii, USA: AAAI Press, 2019. ISBN: 978-1-57735-809-1. URL: `https://doi.org/10.1609/aaai.v33i01.33014602`.

[51] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. “Understanding Deep Learning (Still) Requires Rethinking Generalization”. In: *Commun. ACM* 64.3 (Feb. 2021), pp. 107–115. ISSN: 0001-0782. URL: `https://doi.org/10.1145/3446776`.

[52] Maria Chiara Angelini and Federico Ricci-Tersenghi. “Limits and Performances of Algorithms Based on Simulated Annealing in Solving Sparse Hard Inference Problems”. In: *Phys. Rev. X* 13 (2 Apr. 2023), p. 021011. URL: `https://link.aps.org/doi/10.1103/PhysRevX.13.021011`.

[53] G. Parisi. *Statistical Field Theory*. Avalon Publishing, 1998. ISBN: 9780738200514. URL: `https://books.google.it/books?id=bivTswEACAAJ`.