Master's Thesis

Neural network approach to the planted clique problem

Author:
Rudy Skerk

Supervisor:
Prof. Jean Barbier
Prof. Antonio Celani

A thesis submitted in fulfillment of the requirements for the international Master in Physics of Complex Systems

July 13, 2023

Abstract

This thesis investigates the computational performance of neural networks (NNs) in addressing a challenging problem known as the planted clique problem. This is a highly significant problem in average-case complexity theory due to its large statistical-computational gap. The primary aim of this thesis is to conduct a numerical analysis of NNs, comparing them with existing algorithms, and exploring potential similarities in problem-solving strategies employed by NNs and humans.

Firstly, the problem's setting is introduced, focusing on the detection of graphs with planted cliques. Different algorithms used to solve this task are described, among which Approximate Message Passing (AMP) is identified as the best-known algorithm. AMP, in fact, is able to recover the nodes of a planted clique in a random graph in polynomial time, up to a clique size of $k = \sqrt{n/e}$, where n is the number of nodes in the graph.

Convolutional Neural Networks (CNNs) are then employed to solve the same detection problem. CNNs are used for the visual analysis of adjacency matrices, specifically to identify distinctive patterns that indicate the presence of a planted clique. The theory and mechanisms behind CNNs are elucidated, and their performance in solving the problem is analyzed. Additionally, alternative neural networks capable of solving the same problem are briefly described and their performance is compared. Eventually, human participants will be incorporated into the experiments so that the strategies in problem-solving between CNNs and the human visual system will be compared.

This thesis contributes to the understanding of NNs' capabilities and limitations in solving the planted clique problem and other hard inference problems. It also provides some insights into the comparative performance of NNs, existing algorithms, and human participants. By investigating the similarities in problem-solving between NNs and humans, this research will offer a better understanding of the underlying mechanisms employed by NNs and their potential alignment with human cognitive processes.

Keywords— Planted clique problem, Neural networks, Approximate message passing algorithms

Contents

1 Introduction **3**
- 1.1 Random graphs, Cliques, and Planted Cliques . . . 3
  - 1.1.1 Adjacency matrix representation . . . 3
- 1.2 Historical overview . . . 4
  - 1.2.1 Algorithms . . . 4
  - 1.2.2 Fundamental limits: information and computation . . . 5
- 1.3 Neural network approach to hard inference problems . . . 6
  - 1.3.1 Humans and NNs . . . 7

2 Approximate Message passing and Belief Propagation **8**
- 2.1 Belief propagation . . . 8
  - 2.1.1 Inferring the clique using BP . . . 9
- 2.2 From BP to AMP . . . 10
  - 2.2.1 Inferring the clique using AMP . . . 11
- 2.3 Testing on real data . . . 12

3 Neural Network Approach **14**
- 3.1 Convolutional Neural Networks (CNNs) . . . 14
- 3.2 Training procedure . . . 16
  - 3.2.1 Dataset generation . . . 17
  - 3.2.2 Transfer learning method . . . 18
- 3.3 Testing on real data . . . 19
  - 3.3.1 CNN architecture . . . 20
  - 3.3.2 Phase transition . . . 21
  - 3.3.3 Is transfer learning useful? . . . 22
  - 3.3.4 Understanding the learned features . . . 23
  - 3.3.5 Comparison with human experiments . . . 24
- 3.4 Alternative Neural Networks and possible improvements . . . 25
  - 3.4.1 Counting algorithm . . . 25
  - 3.4.2 Are Neural networks dumb? . . . 27

4 Conclusions **28**

1 Introduction

1.1 Random graphs, Cliques, and Planted Cliques

A graph can be used to represent extremely various problems. For example, in biology, a cell can be thought of as a network of random reactions, in finance the relations between the traders can be represented in a graph, and many social structures are organized as networks. Graphs are extremely used also in physics. One can just think of the Ising model and its generalizations: these are all models based on graphs.

When some kind of stochasticity is present in the connections between individual parts of a problem, a random graph can be a useful tool to design a model. In such graphs, there can be present a set of atypical observations compared to the background noise, and often one is interested in identifying such anomaly. There are different ways in which one can find such anomalies. For instance, one could be just interested in saying whether there is or not a deviation from the typical behavior. In this case, we talk about detection. On the other hand, if one wants to identify completely all the properties of the anomaly, we talk about recovery. Both recovery and detection can be of two different kinds: weak or strong. Weak recovery (or detection) means to be able to recover (or detect) the anomaly better than chance, while for strong recovery (detection), the atypical property has to be recovered (detected) almost surely in the asymptotic limit.

One such anomaly is the presence of a large clique in a random graph. A clique is defined as a fully connected set of nodes. One says that a clique is of size k if the number of fully connected nodes is k. Most of the time, people are interested in finding the largest clique in the graph. In a random graph, cliques appear by chance, but in the asymptotic limit, the largest clique has a well-defined typical size, meaning that is almost impossible for larger cliques to appear in such a graph.

But, what if one "plants" a clique? To plant a clique means to insert a set of fully connected nodes in a random graph. If the clique is too small it will be well hidden among other cliques randomly present and it will be impossible to find. Contrarily, if the planted clique is large enough, one could be interested to identify this clique. Let us explain the difference between recovery and detection for this problem. Recovering the clique means finding the set of nodes that belong to this clique. Detecting the clique, on the other hand, consists in saying whether a random graph has a large planted clique or not.

The problem above is known as the Planted Clique Problem and it has several interesting properties. In the following thesis, we will be interested in the weak detection of planted cliques in specific random graphs. So our task is to tell whether there is or not a planted clique in a random graph. Since the detection is weak, we just want to detect graphs with a planted clique better than chance.

1.1.1 Adjacency matrix representation

Let's define formally what a graph is: a graph $G_n = ([n], E_n)$ is characterized by a set of nodes $[n] = \{1, 2, ..., n\}$ and of edges $E_n \subseteq \{(i, j) \in [n] \times [n]\}$, such that there is a connection between the node i and j only if $(i, j) \in E_n$. Clearly, by explicitly defining the set of edges E_n one has also completely defined the graph.

Often to define a graph one uses the adjacency matrix representation: it is easy to define a matrix A such that $A_{ij} = 1$ if $(i, j) \in E_n$ and $A_{ij} = 0$ otherwise. This matrix is symmetric since if $(i, j) \in E_N$ also $(j, i) \in E_n$ (in an undirected graph) and usually one avoids self-loops: $A_{ii} = 0, \forall i$. Notice also that any permutation of the indices in $[n]$ gives a completely equivalent graph since the indices are just dummy variables. This means that any permutation of the same rows and columns of the adjacency matrix represents an equivalent representation of a graph.

But since we are interested in the presence of a large hidden clique, how should this clique appear in an adjacency matrix? By defining the set of indices belonging to the clique as $C_n \subseteq [n]$, we have that since those are fully connected, then $A_{k,l} = 1$ if $k, l \in C_n$. One is then interested in recovering the set C_n or telling whether such set is present or not.

1.2 Historical overview

The problem of finding cliques in a graph has been studied for many years. A similar problem is the well-known Maximal clique problem, which is an NP-complete problem: in this case, the task is to find the largest clique present in the graph. Also, the Planted clique problem has been widely studied, however many properties of this problem are still unknown. In this part, we will briefly examine its known properties and present some of the open questions that we are interested in.

When the graphs we are considering have some statistical properties, it is possible to study them in the asymptotic limit, hence as the number of nodes goes to infinity. The canonical random graphs used in the setting of this problem are Erdős-Rényi graph. An Erdős-Rényi graph is defined in the following way: the edge between two arbitrary nodes is present with probability p. In terms of the entries of the adjacency matrix this means that the A_{ij} are independent and identically distributed random variables with a probability distribution, given by:

$$P(A_{ij} = A_{ji} = 1) = p \quad P(A_{ii} = 1) = 0$$

The properties of such graphs have been studied for many years [2]. An interesting property, useful in the Planted clique problem, is knowing the size of the largest clique present in such graphs. Our task will be to distinguish between a random graph that contains a planted clique and a pure Erdős-Rényi graph. If the size of the largest clique present by chance is larger than the planted clique, then it will not be possible to distinguish the two graphs. We will call this the Information theoretical threshold of the problem, since below this threshold it will be statistically impossible to detect the clique.

This problem has been studied first numerically by Matula [29] and then analytically by Erdős and Bollobas [7]. It can be proven that in the asymptotic limit, the size of the largest clique strongly concentrates at:

$$k_{IT} = 2\frac{\log(n)}{\log(1/p)}$$

where n is the number of nodes in the graph, while p is the edge probability defined above.

1.2.1 Algorithms

Now that we know which is the statistical threshold for the problem, we are interested to see if there exists any algorithm that can detect or recover planted cliques up to this threshold. We anticipate that there is no known polynomial time algorithm able to saturate this threshold. Here we will concentrate on recovery algorithms, meaning that those algorithms are designed to output the indices of the clique. The hardness of the recovery problem seems to be the same as for the detection (telling whether there is a planted clique or not) [21].

First of all, let us say that there exists a quasi-polynomial greedy algorithm that can find hidden cliques with high probability up to the information-theoretic threshold. This algorithm just loops through all the possible sets of nodes and checks whether the set of nodes is a clique. This algorithm is quasi-polynomial because near k_{IT} the

number of the sets one has to check grows quasi-polynomially with the number of nodes n. This ensures a time-complexity of $exp\big(O((\log(n))^2)\big)$.

On the other hand, according to the findings of Kučera in 1995, [18], it is noted that if the parameter k is of the order $O(\sqrt{n \log n})$, then it is highly probable that all the vertices within the planted clique will have higher degrees compared to the vertices outside the clique. This characteristic makes the clique considerably easier to detect. For smaller cliques though, this does not apply, so one has to find a different strategy.

A more refined approach to the problem looks at the spectral properties of the adjacency matrix. As it has been proven [1], the eigenvalues and more specifically the eigenvectors of the adjacency matrix contain relevant information about the clique nodes. In [1] they explain that one can find almost surely a large part of the clique nodes, just by looking at the second eigenvector of the adjacency matrix. Since several algorithms can compute the first eigenvectors of the matrix in polynomial time (e.g. the power method) the problem can be solved as well in polynomial time. This method can find the vertices of the hidden clique if $k > c\sqrt{n}$, where c is a problem-dependent constant. However, as it is shown in [17], this algorithm fails almost surely for $k < (1 - \epsilon)\sqrt{n}$, for any $\epsilon > 0$.

Currently, the best-known algorithm that solves the Planted clique problem in polynomial time is the Approximate message passing algorithm [8]. This is an algorithm that can be derived from belief propagation, which is a heuristic method used to evaluate the posterior distribution of a graphical model. Belief propagation (BP) and Approximate message passing (AMP) are different: BP is exact on locally tree-like graphs, while AMP is usually used on dense graphs.

It has been shown by Montanari and Deshpande [8] that AMP can solve the Planted clique problem in nearly linear time up to planted cliques of size $\sqrt{n/e}$. They generalized the result also to the sparse graph case, showing that no local algorithm can perform better than BP in solving this problem.

1.2.2 Fundamental limits: information and computation

As we have seen there are two fundamental limits in the problem. The first one is the information theoretical threshold, under which the problem is impossible to be solved, since the two graphs (with and without planted clique) are statistically indistinguishable. On the other hand, all the algorithms that have been studied so far fail much before the information theoretical threshold, more specifically the best-known algorithms cannot cross the so-called algorithmic threshold given by $k_{ALG} = \sqrt{n/e}$.

Therefore the problem seems to have a hard phase, inside which the problem cannot be solved in polynomial time: only quasi-polynomial algorithms can solve it with high probability. Specifically, if the size of the clique planted in the graph is larger than k_{IT} and smaller than k_{ALG}, there is no known polynomial algorithm that can recover it significantly better than random chance. This became a conjecture: finding cliques in this region is assumed to be an NP-hard task and it has been used to show that also other problems are NP-hard by reductions [13, 3, 6].

One line of research that tries to prove the computational hardness of those kinds of inference problems is the one that studies the overlap gap properties. This property originates from spin glass theory and it suggests algorithmic hardness when it appears in a problem [24, 12, 11]. This field suggests that due to the presence of this property, it is not possible to enter the hard phase. However, this seems to be true only for local algorithms, meaning that an algorithm that works globally may be able to perform better than chance also in this hard region.

1.3 Neural network approach to hard inference problems

The main idea behind this thesis is trying to solve hard inference problems with neural networks. Neural networks have become extremely popular, since they can solve complex tasks, by learning extremely large sets of data. McCulloch and Pitts were the first that introduced the perceptron [22] as a model for the neuron: the perceptron represents just a unit that takes many inputs x_i (with $i = 1, 2, ..., n$) and outputs a single value $\hat{y} = f(\sum_{i=0}^{n} \omega_i x_i - \theta)$. Here the ω_i are the weights of each input, while θ is called the bias. The function f at the beginning was designed as a step function. The signal that one neuron receives from other neurons or from the environment is represented by the inputs x_i, while the weights ω_i can be thought of as the importance that the neuron gives to the signal x_i. Given that the weighted sum is larger than a threshold θ, the neuron emits a signal.

The perceptron quickly became an attractive tool for problem-solving. Computer scientists tried to use the same structure to solve complex tasks. Many problems can be in fact set as a simple inference task: given a set of inputs x_i predict the output $y = g(x_1, x_2, ..., x_n)$, that comes from an unknown function g. Since the function g is unknown, it is possible to solve the problem by learning the weights ω_i of a perceptron so that $\hat{y} = y$ for every set of inputs $(x_1, ..., x_n)$. The learning procedure is usually done by gradient descent (as we will see later).

The perceptron has a very limited computational power by itself, but if instead of a single neuron, one considers many of them, stacked in different layers, the computational power increases drastically. These structures are at the basis of deep learning; they are called feed-forward networks and are made of many layers of such "neurons". If each neuron of the i^{th} layer is connected to all the neurons of the $(i+1)^{\text{th}}$ layer, we say the network is a Fully connected cascade network (FCC) (see Figure 1). The name perceptron is still used if f is a step function, but nowadays different non-linear functions are more popular.

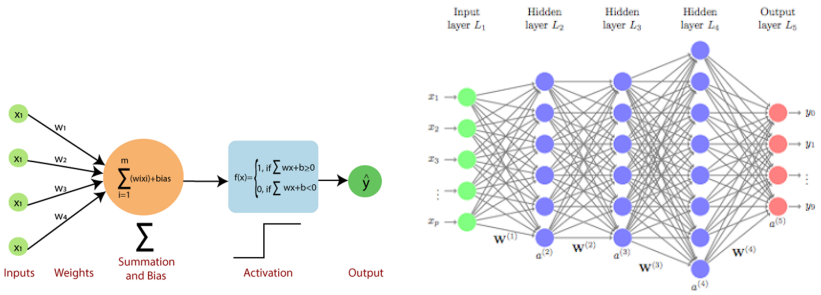

Figure 1: Graphical representation of a perceptron (left) and of an FCC network (right)

Neural networks became popular many years after the introduction of feed-forward networks. Those need to be "trained", as we will see later, but the training procedure is very computationally demanding and data-hungry. Only now it is possible to train those networks properly and use them to solve complex tasks. Nevertheless, the power of those tools is astonishing. Almost everyone that tried to use ChatGPT knows it. However, many other neural network-based AIs are currently able to solve extremely hard problems better than humans. AlphaFold is currently the best tool to predict the folding of proteins [15] and AlphaZero dominates the chess scene [27], just to cite two of them.

1.3.1 Humans and NNs

The idea for this project was not born only by seeing the astonishing results of the neural networks, but also from the much more humble world of neuroscience and cognition. Human vision remains one of the most complete tools for image recognition tasks, but not only. Different experiments have shown [20] that humans outperform very powerful algorithms at extremely hard inference tasks, for example, the Traveling Salesman Problem (TSP). This problem is very simple: finding the shortest route that passes through n points on a plane. This is an NP-complete problem, but still, humans perform extremely well at the task.

Also, the Planted Clique problem can be set as an image recognition problem. The adjacency matrix of the graph can be presented as a black-and-white image. By looking at this image it is then possible to say whether there is a planted clique in the graph or not. Some examples can be seen in Figure 2. There is present some structure in the case of a planted clique. This represents an easy instance, where the clique is relatively big: the main task of this project is to see where this method starts to fail.

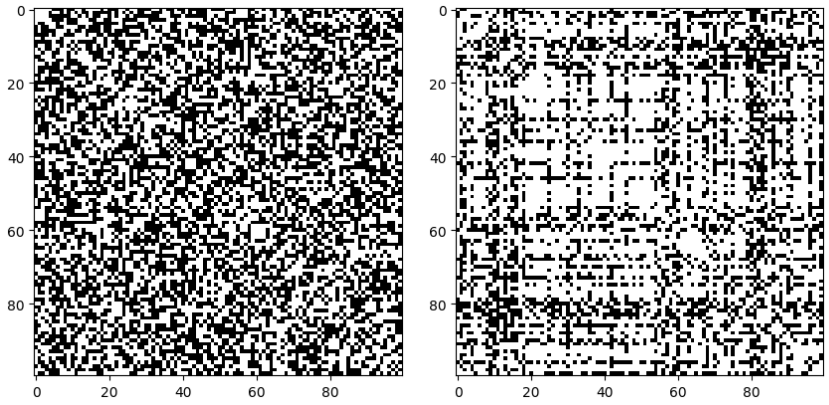

Figure 2: Left: Adjacency matrix of a pure Erdos-Renyi graph of size $n = 100$ Right: Adjacency matrix of a graph of size $n = 100$ with a planted clique of size $k = 60$

Even though the human visual system is extremely good, it is not easy to set an experiment to test where humans fail at distinguishing these images. That is why we are interested in using specific neural networks to see if they can solve the problem. Those neural networks are called Convolutional Neural Networks (CNN) and are mostly used for image recognition tasks.

The project has also a neuroscientific part: at SISSA we want to set the experiment and test also humans, to see if the two methods fail at the same point and if they share strategies when learning to distinguish the images. Then it will be necessary to compare the human performance against the best-known algorithms: we will concentrate on AMP since is the best-known algorithm that finds hidden cliques.

2 Approximate Message passing and Belief Propagation

Belief propagation (BP) and Approximate message-passing (AMP) are two physics-inspired message-passing algorithms designed to infer some hidden variables of a graphical model. Belief propagation was introduced to compute the marginal probabilities of graphical models, such as Bayesian networks or Markov random fields. It is a message-passing algorithm because it exchanges messages between connected nodes in the graph. Its accuracy depends mostly on the graph structure. Approximate message passing algorithms represent instead a generalization of belief propagation when the model we are considering is a linear estimation model, so basically when the relationship between the observed measurements and the underlying signal is linear.

In this part, we will mostly concentrate on AMP, since, as we will see, is the best choice when it comes to dense graphs. However, to give some intuition about how to design an approximate message-passing algorithm, we will see how it emerges from BP. Also in the planted clique problem AMP can be seen as a generalization of BP. Most of the results and definitions that we will describe can be found in much more detail in the original work of Montanari and Deshpande [8] and in other sources as [30, 25].

2.1 Belief propagation

Consider a graph, where we associate to each node a variable x_i, taking values in a finite alphabet χ. We are then interested to compute the marginal probability to observe x_i, given the other variables of the graph. As above $[n] = \{i\}_{i=1}^n$ are the indices of the nodes in the graph. We also define a family of subsets of variables $\{\partial a \subset [n] : a \in A\}$ and a family of positive $\Phi_a : \{x_i\}_{i\in\partial a} \to \mathbb{R}^+$. Let us also denote by ∂i the neighborhood of node i. The set A comes directly from the joint probability distribution of $x = (x_1, x_2, ..., x_n)$:

$$p(x) = \frac{1}{Z}\prod_{a\in A}\Phi_a(x_a)$$

where x_a is the subset of variables Φ_a depends on, while Z is the normalization. We can now define the "factor graph" as the bipartite graph $G = (V = [n] \cup A, E \in I \times A)$, with $(i, a) \in E$ if and only if $i \in \partial a$. Notice that we assume that the probability distribution can be written as above, which is the case for many graphical models, for example in the case of pairwise models the probability distribution can be written as follows: $p(x) = \frac{1}{Z}\prod_{i,j}\Phi_{ij}(x_i, x_j)\prod_i \Phi_i(x_i)$

Belief propagation is usually defined on this factor graph. In this case, the BP equations represent iterative equations that converge to the marginal probabilities of each node. Those equations can be written as follows:

$$\theta^{(t+1)}_{i\to a}(x_i) = \frac{1}{z^{(t)}_{i\to a}}\prod_{b\in\partial i\setminus a}\theta^{(t)}_{b\to i}(x_i)$$

$$\theta^{(t+1)}_{a\to i}(x_i) = \frac{1}{z^{(t)}_{a\to i}}\sum_{x_{a\setminus i}}\Phi_a(x_a)\prod_{j\in\partial a\setminus i}\theta^{(t)}_{j\to a}(x_i)$$

$$\theta^{(t+1)}_{i}(x_i) = \frac{1}{z^{(t)}_{i}}\prod_{a\in\partial i}\theta^{(t)}_{a\to i}(x_i)$$

$$\theta^{(t+1)}_{a}(x_a) = \frac{1}{z^{(t)}_{a}}\Phi_a(x_a)\prod_{i\in\partial a}\theta^{(t)}_{i\to a}(x_i)$$

The variables $\theta^{(t+1)}_{a\to i}(x_i)$, $\theta^{(t+1)}_{i\to a}(x_i)$ are usually called messages, while $\theta^{(t+1)}_{i}(x_i)$ and $\theta^{(t+1)}_{a}(x_a)$ are the marginal beliefs. It is possible to show (see [25]) that the marginal

beliefs converge to the actual marginals of x_i and x_a if the factor graph is a tree. However, this algorithm is practically useful also for graphs that are not trees. In this case, we call the algorithm Loopy belief propagation. In fact, it continues to work on locally tree-like graphs, which are graphs that locally maintain a tree-like structure [26]. This happens because in locally tree-like graphs loops are extremely long, and since the correlation decays with the length they are negligible.

2.1.1 Inferring the clique using BP

Now suppose to have a hidden binary signal $x = (x_1, x_2, ..., x_n) \in \{0, 1\}^n$ that we want to infer from a visible variable Y (notice that this variable could be also a vector or a matrix). For example, in the planted clique problem we can take that $x_i = 1$ if node i belongs to a planted clique, and $x_i = 0$ if it does not. The variable Y contains all the information we have about the nodes in the graph, hence $Y = A$, so Y is the adjacency matrix.

Since the value of A depends only on the hidden variable x, its probability distribution can be written as:

$$P(A|x) = \prod_{(i,j)\in E_n} Q_{x_i x_j}(A_{ij})$$

Here the subscript $x_i x_j$ represents the product between the two indices x_i and x_j. The planted clique problem can be set as follows. There are two types of nodes, identified by the variable x. If both nodes i and j belong to the clique (therefore $x_i = 1$ and $x_j = 1$), the probability to have an edge between them is $Q_1(A_{ij}) = \delta(1, A_{ij})$. On the other hand, if one of the two nodes does not belong to the clique, the edge probability is the same as in an Erdős-Rényi graph with $p = 1/2$: $Q_0(A_{ij}) = (1/2)\delta(1, A_{ij}) + (1/2)\delta(0, A_{ij})$.

However, we are interested in the posterior distribution, namely $P(x|A, k)$ (supposing that we also know the size of the planted clique). Using the Bayesian rule it is easy to get:

$$P(x|A, k) = \frac{1}{Z(A, k)} \prod_{(i,j)\in E_n} Q_{x_i x_j}(A_{ij}) \prod_{i\in[n]} P_{x_i}(k)$$

where $Z(A, k)$ is the normalization, while $P_{x_i}(k)$ represents the probability that node i belongs to the clique.

Notice that this is a simple pairwise model, therefore now the factor graph is very simple. In fact, this model can be now thought of as a simple Ising model with a field and pairwise coupling. Let us show this connection by explicitly writing the probability distribution in the Boltzmann-Gibbs form. To do it, we first need to explicitly define the probability distribution $P_{x_i}(k)$. We can relax the constraint of having exactly k nodes in the clique and set $P(x_i = 1) = \frac{k}{n}$. Then the number of nodes belonging to the clique will concentrate sharply around k as n goes to infinity. Anyway, the probability distribution we have defined above, can be now written as:

$$P(x|A, k) = \frac{1}{Z(A, k)} \prod_{(i,j)\in E_n} \exp\Big(\log(A_{ij}) x_i x_j + \log(2)(x_i x_j - 1)\Big) \prod_{i\in[n]} \left(\frac{k}{n}\right)^{x_i} \left(1 - \frac{k}{n}\right)^{1-x_i} \quad (1)$$

$$P(x|A, k) = \frac{1}{Z(A, k)} \exp\Big(\sum_{(i,j)\in E_n} \log(A_{ij}) x_i x_j + \log(2)(x_i x_j - 1)\Big) \exp\Big(\sum_{i\in[n]} \log\left(\frac{k}{n}\right) x_i + \log\left(1 - \frac{k}{n}\right)(1 - x_i)\Big) \quad (2)$$

if we put all the constant terms that do not depend on x_i in the normalization factor $Z(A,k)$, we can rewrite the probability distribution as an Ising model with fields and couplings:

$$h_i = \log \frac{k}{n}\left(1-\frac{k}{n}\right)^{-1}$$
$$J_{ij} = \log(2A_{ij})$$

A factor graph when considering pairwise models is much simpler, since the factors can be identified by the edges $(i,j) \in E_n$ and nodes $i \in [n]$. In this case, it is not necessary to define factors, but the messages can be defined directly on the original graph, meaning that we can define $\theta^{(t)}_{i\to j}(x_i) \equiv \theta^{(t)}_{i\to(i,j)}(x_i)$. In this case, the BP equations become:

$$\eta^{(t+1)}_{i\to j} = h_i + \sum_{l\in\partial i\setminus j} \tanh^{-1}(\tanh J_{ij} \tanh \eta^{(t)}_{l\to i})$$

where $\eta^{(t)}_{i\to j} = \frac{1}{2}\log\left(\frac{\theta^{(t)}_{i\to j}(1)}{\theta^{(t)}_{i\to j}(0)}\right)$. Therefore it is possible to iterate those equations until convergence and in this way compute the marginals.

2.2 From BP to AMP

But let us emphasize again, BP converges to the correct marginals, only if the factor graph is a tree or at least a locally tree-like graph. In the planted clique problem this is not the case. We are working with a fully connected graph of size n and eventually among all those connections we are taking the two probability distributions Q_0 and Q_1. But let us take the last equation we have found and let us generalize it a bit:

$$\eta^{(t+1)}_{i\to j} = h_i + \sum_{l\in\partial i\setminus j} f(\eta^{(t)}_{l\to i})$$

the BP equations usually appear in this way in pairwise models, where the function f is a non-linear function. But let us rewrite it using the adjacency matrix instead:

$$\eta^{(t+1)}_{i\to j} = h_i + \sum_{l\in[n]\setminus j} A_{il} f(\eta^{(t)}_{l\to i})$$

We are ready to define AMP. Since we are not anymore on a tree-like graph, BP should not yield good results. Still, in many cases, it has been shown [9, 5, 14] that by adding a correction term the algorithm still works well. This term is known as the Onsager reaction term, introduced for the first time in [28] as a correction term to compute the magnetization of random energy models.

The iterative equations in AMP are known as the state evolution, and in their most general form are defined as follows:

$$\eta^{(t+1)}_i = \sum_l A_{il} f(\eta^{(t)}_l; t) - b_t f(\eta^{(t-1)}_i; t-1)$$

where $b_t = \frac{1}{n}\sum_l f'(\eta^{(t)}_l)$. The second term is the Onsager reaction term, while the matrix A_{il} is just the adjacency matrix, but in this case $A_{ij} = 1/\sqrt{n}$ if $(i,j) \in E_n$ and $A_{ij} = -1/\sqrt{n}$ otherwise. This change in the definition is usually done just for mathematical convenience. It is now clear why it is called Approximate message passing: we start from BP, therefore a message passing algorithm and we do a few approximations.

On the other hand, if one does not want to start from BP, it is possible to see the state evolution as a generalization of the power method. Suppose to take the function f to be the identity function. In this case, the first step of the state evolution will be:

$$\eta_i^{(1)} = \sum_l A_{il}\eta_l^{(0)}$$

Since the adjacency matrix is a random matrix, if we take $\eta_l^{(0)} = 1$ for $l = 1, ..., n$, then due to the central limit theorem $\eta_i^{(1)}$ will be a Gaussian random variable. But in the next step of the state evolution, if we do not apply the Onsager reaction term, the central limit theorem will not be applicable because $\eta_i^{(1)}$ depends on A_{il}. In this sense, the Onsager reaction term is also called a memory term: only by removing it we will get at each time step a Gaussian vector.

2.2.1 Inferring the clique using AMP

As we have seen, in the case of dense graphs the best algorithm is Approximate message passing. However, due to the many approximations, one makes to get the right equations, it is necessary to find for every problem the right non-linearity. In the case of the planted clique, a good strategy was described in [8]. Let us briefly overview their method.

In this article, the aim is to find the indices of the hidden clique. To do so they apply the state evolution:

$$\eta_{i\to j}^{(t+1)} = \sum_{l\in[n]\setminus i,j} A_{li} f(\eta_{l\to i}^{(t)}, t), \quad \forall j \neq i$$

$$\eta_i^{(t+1)} = \sum_{l\in[n]\setminus i} A_{li} f(\eta_{l\to i}^{(t)}, t)$$

which are extremely similar to the one above, but in this case, the Onsager reaction term is not necessary (since the term $\eta_{i\to j}$ makes its work). The non-linear function $f(x,t)$ though, cannot be any function. We see that in Lemma 2.2 from [8], this function has to be a finite-degree polynomial for each $t \in \mathbb{N}$. Then let us take A to be the adjacency matrix of a graph with a planted clique as above (i.e. $A_{il} \in \{-1/\sqrt{n}, +1/\sqrt{n}\})$, $n >> 1$, $k \propto \sqrt{n}$ and $C_n \subset [n]$ to be the subset of nodes belonging to the clique ($|C_n| = k$). By setting the initial values of the messages to $\eta_{i\to j}^{(0)} = 1, \quad \forall i, j$ the following limits hold in probability:

$$\lim_{n\to\infty} \frac{1}{k} \sum_{i\in C_n} g(\eta_i^{(t)}) = \mathbb{E}[g(\mu_t + \tau_t Z)]$$

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i\in[n]\setminus C_n} g(\eta_i^{(t)}) = \mathbb{E}[g(\tau_t Z)]$$

where $g : \mathbb{R} \to \mathbb{R}$ can be any bounded Lipschitz function, $Z \sim N(0,1)$ and μ_t, τ_t are defined by the following recursion:

$$\mu_{t+1} = \mathbb{E}[f(\mu_t + \tau_t Z, t)]$$

$$\tau_{t+1} = \mathbb{E}[f(\tau_t Z, t)^2]$$

Practically, we can say that through this iterative state evolution, we get a vector $\eta = (\eta_1, ..., \eta_n)$ such that all its entries are Gaussian random variables, but with different mean. Namely, if $i \in C_n$, then η_i will have mean μ_t, and if $i \notin C_n$ its mean will be 0. In this way, it will be extremely easy to separate the two sets and determine whether the node i belongs to the clique or not. The procedure one can use is to take as $\hat{C}_n = \{i \in [n] : \eta_i^{(t)} > \mu_t/2\}$ and it is possible to show that in the asymptotic limit and for $k > \sqrt{n/e}$, $\hat{C}_n = C_n$ with high probability. Here we skipped many details, such as the definition of the non-linear function $f(x,t)$. All the calculations and proofs can be found in [8].

2.3 Testing on real data

Finally, we can analyze the performance of the AMP algorithm described above and test it on real data. Let us say that the algorithm was designed to recover the nodes of the clique planted in the graph. However, this algorithm can be used also as a detection algorithm in the following way. Say you are given a graph and you want to determine whether there is or not a planted clique of size k. By applying AMP you would get almost surely for $k > \sqrt{n/e}$ all the indices belonging to the clique if the graph has a planted clique. If the graph, instead, is purely random, then the output nodes will be just random nodes. Therefore from the output nodes one can say whether there is or not a planted clique in the graph.

Since the aim of the experiment, is not to design a perfect AMP algorithm for the detection task, we decided to choose a simple criterion to say whether the graph we are considering has a planted clique or not, based on the output of the algorithm described above. Our scope is just to have a coarse comparison between the NNs we will design and the AMP algorithm.

Since the output of the AMP algorithm is a set of k indices that with high probability belong to the clique, we can always compute the adjacency matrix restricted to those k indices, which we call $A^{(k)}$. Then we say that the graph has a planted clique only if the sum of all entries of the restricted adjacency matrix $A^{(k)}$ is larger than a given threshold. Clearly, if the adjacency matrix is purely random, the restricted one will be random as well, therefore statistically indistinguishable from the adjacency matrix of an Erdős-Rényi graph with k nodes. On the other hand, if there is a planted clique of size k, in the best case scenario all the indices will belong to the clique and therefore $A^{(k)} = 1$ (all entries of the matrix are 1).

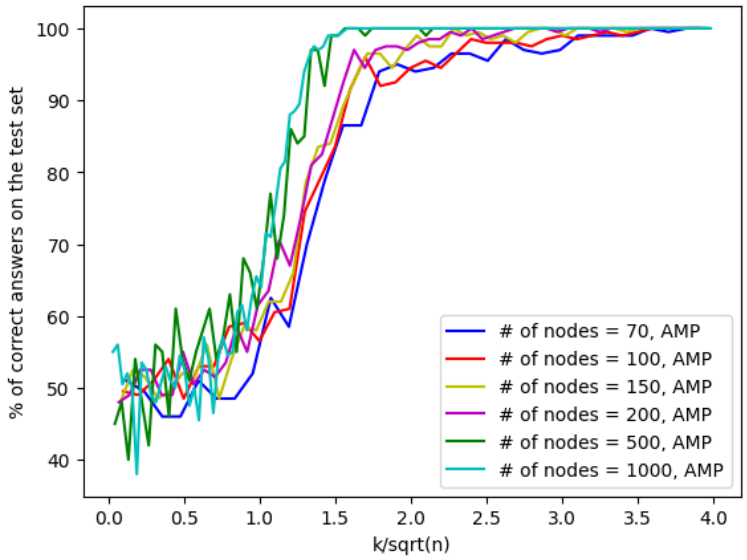

Figure 3: Performance of the AMP algorithm in recovering the Planted clique in graphs of different sizes

This algorithm is a simple hypothesis test based on the sum $S_k = \sum_{i,j} A^{(k)}_{ij}$: if the sum is above a given threshold Λ one says that there is a clique in the graph. Since the average degree of the graph without planted clique will be peaked at $k(k-1)/2$ in the asymptotic limit, one can simply take the threshold to be $\Lambda = k(k-1)/2 + \epsilon$ where $\epsilon > 0$ and of order $\epsilon = O(k)$, which is the order of the variance. This strategy is

not optimal, since we do not know the statistical properties of $A^{(k)}$ when there is a planted clique in the graph. A very similar strategy will be presented later on in the last chapter.

Finally, in Figure 3 we see that the transition does not occur at the theoretic threshold of $k = \sqrt{n/e}$, but as we increase the graph size, the transition shifts more and more to the left, signaling that in the asymptotic limit, this algorithm could reach the optimal performance. One may argue that the detection and recovery problems are different. However, much research suggests that the hardness of the detection problem is the same as that of the recovery problem [21]. In any case, it is clear that our algorithm was not designed specifically for the detection task, therefore the optimal performance could be hard to reach in practice.

As a final remark we add that in [8] they managed to solve the problem also in the sparse setting (see the article for more details). In this setting, they also proved that there is no local algorithm that can outperform AMP (in the sparse case one can use BP). An algorithm is said to be t-local if its output is a function of the neighborhood $Ball_{G_n}(i;t)$ of each node i, where $Ball_{G_n}(i;t) \subset G_n$ is the sub-graph containing all nodes and edges distant less then t from node i. A local algorithm is then a t-local algorithm, for some t independent of n. Therefore this theorem suggests that to perform better than AMP and maybe enter the hard phase, it is necessary to design a non-local algorithm. This was also one of the main ideas that made us study the problem. The hope was to design a neural network that exploits non-local features of the graph.

3 Neural Network Approach

As we have anticipated, the problem we want to solve is somehow different from the one analyzed by the message-passing algorithm above. In fact, rather than a recovery problem we will focus on the detection problem. Instead of trying to find with high probability the set of nodes that form the hidden clique, we will try to distinguish a graph with a planted clique of size k from one that is purely random.

Let us define properly the setting of the problem: let $\mathcal{D} = \{(X_i, y_i)\}_{i=1}^{m}$, where X_i is the adjacency matrix of the i^{th} graph. This graph has a label y_i that indicates if in the graph there has been planted a clique or not. Therefore, if the label $y_i = 0$ there is no clique planted in the graph so the adjacency matrix X_i is just a pure Erdős-Rényi one. On the other hand, if $y_i = 1$, there is a planted clique in the graph.

There is a very simple way to generate those random matrices with a planted clique. We have already seen that to generate a pure Erdős-Rényi graph with n nodes and edge probability p, that we will call $\mathcal{G}(n, p)$, we just have to draw independently each entry of the adjacency matrix. The entry is 1 with probability p and 0 with probability $1 - p$. The diagonal is instead set to zero since we do not want self-loops. The random graphs that contain a planted clique are generated in a very similar way. First of all, we draw the adjacency matrix A of $\mathcal{G}(n, p)$ and then we set the $A_{ij} = 1$ if $i, j \leq k$ and $i \neq j$. In this way, the first k indices will be part of a clique of size k. Obviously, in this way, we would generate only graphs where the first k indices are in the clique. Hence, due to the invariance for permutations of rows and columns, we can permute them and get a different matrix, that represents the same graph with permuted indices.

The problem can be therefore set as a visual task; the adjacency matrix of the problem can be represented as a grayscale image since it is just a matrix of zeros and ones. As we have seen in Figure 2, the adjacency matrices with a large planted clique have clear structures that do not appear in the purely random case. Therefore after one has generated the dataset $\mathcal{D}$, this can be used as the training set for a convolutional neural network.

3.1 Convolutional Neural Networks (CNNs)

Traditional models, like fully connected linear networks, are extremely popular networks, that work for many problems and they are mostly used in problems with tabular data. On the other hand, these become unwieldy for high-dimensional perceptual data like images. These networks do not exploit the symmetries and structure of the data, which makes them inefficient for learning image representations. Convolutional neural networks (CNNs) instead leverage the structure in images and are widely used for image recognition tasks.

If we want to detect an object in an image we should not heavily rely on its precise location. In fact, the object we aim to detect could be present in every position of the image. In [32] this property is explained using as an example the game "Where's Waldo,". This is a kid's game where Waldo appears in different images in unlikely locations. Despite the chaotic scenes, recognizing Waldo does not depend on his location (see Figure 4). This represents simply a translational invariance property, hence if we want to design a good image detection algorithm it is necessary to exploit this symmetry. CNNs are designed to exploit this concept of spatial invariance and in this way, learn efficiently and with fewer parameters.

CNNs exploit the translational invariance in the following way. Instead of connecting each pixel of the image to a neuron we create a kernel. The kernel is a very simple object that when applied to an image works as a cross-correlation operator. A cross-correlation operation can be formally defined as follows. Say that A is an image in the form of an $n \times n$ matrix of numbers and K is a $l \times l$ matrix that we call the kernel. If we apply

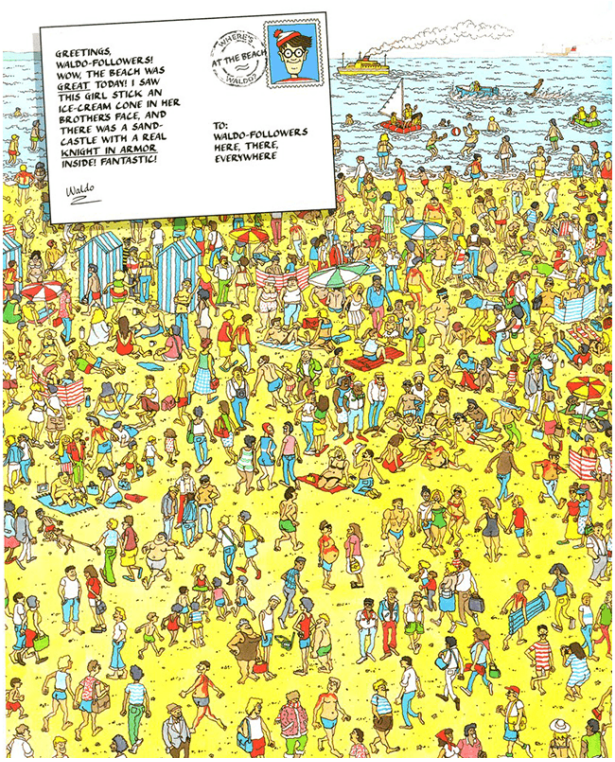

Figure 4: Image from the game "Where's Waldo?"

this kernel on A the output B of the operation is the following:

$$B_{k,l} = \sum_{i,j=1}^{k} K_{i,j} A_{k+i,l+j}$$

One simple example of this operation can be seen in the following Figure 5 (again borrowed from [32]). As we can notice from the simple example in Figure 5, the size of the image gets shrieked by the cross-correlation operation so that the dimensionality of the initial input is smaller.

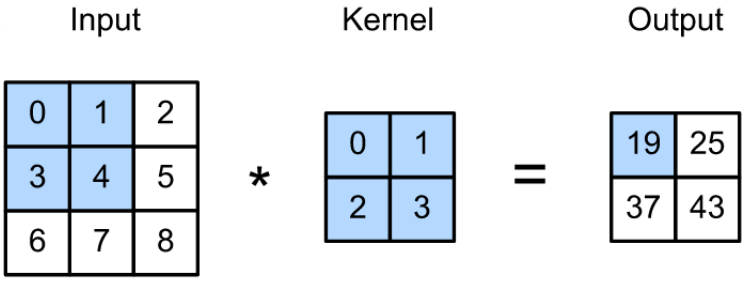

Figure 5: Application of a Kernel on a matrix

A CNN is usually constructed by many convolutional layers, where each layer contains many different kernels that are applied to the input image. Therefore, the

output of a convolutional layer will be a collection of dimensionally reduced images. The dimension of the output depends on the kernel size and on other variables that we will not use (strides and padding [32]). So that if we apply a kernel of size $l \times l$ on an image of size $n \times n$, the output will then be of size $(n - l + 1) \times (n - l + 1)$. Since a convolutional layer contains many different kernels, each will output a different shrunk image.

But convolutions are not the only operations present in a CNN. Usually after a convolutional layer (or more than one) one has a pooling layer. The pooling layer is almost identical to a convolutional layer, but in this case, the kernel is substituted by a pooling operator. Pooling operators, like cross-correlations, use a fixed-shape window that slides over the input. For each location visited by the window, a single output is computed, but in this case, pooling layers have no parameters or kernels. Instead, pooling operators are deterministic and calculate either the maximum value (max-pooling) or the average value (average pooling) of the elements within the pooling window.

Formally one can define the output image C^{max} of the max-pooling operator in the following way:

$$C^{max}_{k,l} = \max_{i,j=1}^{k} A_{k+i,l+j}$$

and C^{avg} for the average pool:

$$C^{avg}_{k,l} = \operatorname*{avg}_{i,j=1}^{k} A_{k+i,l+j}$$

Modern CNNs use max-pooling rather than average pool operators. One can see these operators as some sort of coarse-graining operators in the renormalization group [23]. In fact, also in this case the size of the input shrinks, so that after the application of many pooling layers one gets an image that is similar to the original one but reduced in the dimensionality.

After we have applied different convolutional and pooling layers the original image is much smaller and can be treated with a simple fully connected layer (or more than one). The output of this CNN, since we have a binary classification task, will be a string of binary variables.

3.2 Training procedure

As always, if we want to solve a binary classification task with a neural network, we first need to train the weights of the network. The weights in a CNN are represented by the entries of each kernel and by the weights of the fully connected layer. The training of a deep neural network is generally done as follows.

First of all, we need a dataset from which the network can learn; we will call it $\mathcal{D}$. Since our task is a supervised learning task, the dataset will contain two separate objects: input data and labels. In fact, as in our case, to each image X_i we assign a label y_i, where $i = 1, 2, ..., m$ and m is the number of examples in the dataset. Since our task is a binary classification of images $y_i = \{0, 1\}$.

Once we have our dataset, we can give the set of inputs X_i to the network. Each input gets transformed by the network that eventually will return a binary output $\hat{y}_i$. We desire the output of the network to be identical to the labels: that means that the network learned how to solve the task. At this point, it is therefore necessary to introduce a loss function. This loss function is a measure of the correctness of the output. Usually, it is simply a distance between y_i and $\hat{y}_i$. Let us define a generic loss function $l : \{0, 1\}^m \times \{0, 1\}^m \to \mathbb{R}$, that given the input it outputs a real number. This is usually taken to be an l_1 or l_2 norm (or a cross-entropy).

Now that we defined the dataset and the loss function, we want to minimize the loss function over our dataset. This procedure is done by the backpropagation algorithm

and by gradient descent. We will just briefly describe those two algorithms that are crucial for all deep learning networks. A much more detailed explanation can be found in [23].

If we want to minimize the loss function, which is a very complicated function of the weights, the most computationally efficient method we can use is gradient descent. To apply gradient descent one needs to compute the gradient of the loss with respect to the weights of each layer. Once the gradient is computed, one has to update the weights in the following way. Say $\nabla_\omega l(y_i, \hat{y}_i; \omega)$ is the gradient of the loss with respect to all weights ω. Then one step of the gradient descent is just:

$$\omega \leftarrow \omega - \eta \nabla_\omega l(y_i, \hat{y}_i; \omega)$$

where η is also called the learning rate. The learning rate is usually taken to be a small value so that at each step, the weights change in such a way that the loss function decreases and eventually reaches a minimum.

Clearly, since a deep neural network is an extremely complex object, also the gradient with respect to all the weights is an object that requires to be analyzed separately. Indeed, there exists a simple iterative algorithm, backpropagation, that given the loss function, is able to compute the gradients extremely efficiently. This is basically an application of the chain rule for derivatives to those networks (see [23]). This algorithm is contained by default in the packages of Pytorch: once one has calculated the loss function, it is just necessary to call the gradient, and all the calculations are automatically done. The training procedure described above works well for most of the deep neural network architectures, including in particular CNNs and fully connected layers.

3.2.1 Dataset generation

Now let us describe more in detail the training procedure we are using in the problem. Usually, in deep learning one has to generate a dataset with many examples and associated labels, that will be used to train the network. But in most cases, we do not use the whole dataset at once but we rather split it into many subsets. The main split happens at the beginning: we divide the whole dataset into a training, validation, and test set. Then the training set gets split again into many small subsets, that we call mini-batches.

Let us explain first the train-validation-test split. In almost every machine learning task we should divide the whole dataset in these three sets. The training set is the largest one and it is the one on which we perform the training, meaning that we give the examples in this set to the network and use gradient descent and backpropagation until we reach a satisfying performance of the network. But, what about examples that are not present in the train set? Will the network be able to classify them? That is why we use also the validation set: during the training, we check the performance of classification on the validation set, in this way, we avoid overfitting. The term overfitting, in the machine learning literature, means that the network has learned well the examples in the train set, but it is not able to make predictions about an unseen sample. In the end, if we want to test the performance, we need an untouched set of examples on which we could test the network: the test set.

The second split is much less intuitive. There are two reasons why it is important to divide the training set into mini-batches. The first is computational efficiency, the second is to optimize the gradient descent. Let us explain these two reasons separately.

Computational efficiency is crucial in this problem. One just has to think that to generate one graph with n nodes it is necessary to sample $O(n^2)$ random numbers. But since we are interested in large graphs and large datasets, the sole generation of such sets would require a lot of memory and time. That is why we rather implement an online generation of mini-batches. We generate a small set on which we train our

network. Once the network has learned this sample, we generate another one and train again. We iterate this procedure until the error on the validation set is small enough.

Additionally, it has been shown that mini-batches optimize learning by gradient descent [23]. Actually, in deep learning simple gradient descent is never used; a much better alternative is instead stochastic gradient descent (SGD). In fact, by using gradient descent to minimize a generic function, one will find a relative minimum of the function. Unfortunately, the problems in deep learning are high dimensional and the loss function that one has to minimize is extremely complex ad full of saddles and relative minima. Gradient descent does not allow us to find a suitable solution. That is why one has to add stochasticity to the algorithm.

There is an intuitive correspondence with physical phenomena: at zero temperature a physical system is at the minimum of the energy. But if we decrease the temperature extremely fast, the system can get stuck in relative minima. On the other hand, a slow decrease in temperature will lead to the absolute minimum. A non-zero temperature induces fluctuations, that can lead the system to escape from local minima: this method is known as annealing in metallurgy. The same happens in SGD, by adding stochasticity to the gradient descent one reaches much better results [19]. These optimized algorithms are present in the Pytorch packages [19, 16]

3.2.2 Transfer learning method

As we have seen in the introduction, the hardness of the problem can be easily represented in the parameter k, the size of the planted clique. Easy instances have very large planted cliques and detecting them is very easy. That is why one could decide to train the neural network with the transfer learning method. Transfer learning aims at improving the training procedure by transferring the features learned in easier instances to harder ones [33].

In this problem transfer learning appears naturally if we think of it as an image recognition task. In easy instances, the adjacency matrix image representation has an evident structure, that becomes more and more subtle as we decrease the size of the planted clique (see Figure 6). Still, the task remains the same, therefore learning the first very easy instances could be useful to understand the problem.

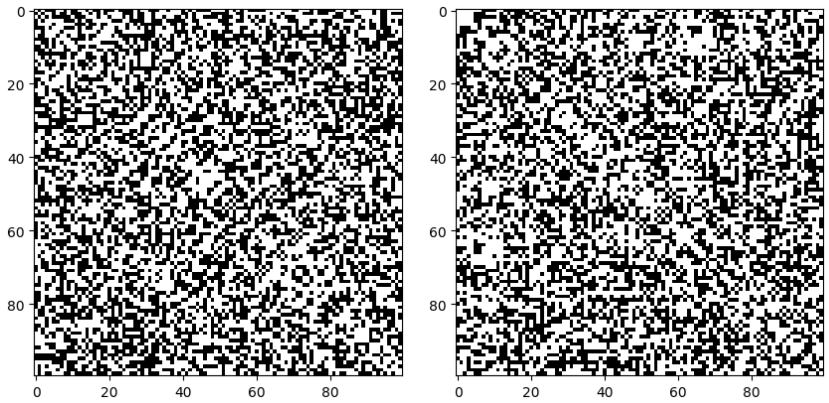

Figure 6: Left: Adjacency matrix of a pure Erdos-Renyi graph of size $n = 100$ Right: Adjacency matrix of a graph of size $n = 100$ with a planted clique of size $k = 30$

That is how we would teach humans the task, but is it a good approach in general? One could see misleading structures in easier instances and be fooled later on. In some

sense, one can think about the transfer learning method as follows. Our task is to find the global minima of the loss function: easier instances will have a very deep global minimum that will be easy to find by SGD. On the other hand, as the problem gets harder, the number of relative minima will increase and the global minima will start to be well mixed with those. Other properties could arise, for example, overlap gap properties [12], which makes the problem even harder to be learned.

Transfer learning could be a good or bad strategy in these cases. Maybe the global minimum mix with the other relative ones, but stays always present as the deepest one: in this case by learning first the problem in the easy setting, one will find the right strategy to solve the problem also in the harder setting. On the other hand, if the strategy that one learns in the easy case is misleading, meaning that new very different global minima could appear in the hard setting so that all the features learned in the easier instances are useless.

3.3 Testing on real data

We proceed in describing now the results of the neural network approach to the planted clique problem as it was presented above. But before diving into the results there are a few details that we should add.

Firstly, as one can see from Figure 2 it is simply possible to count the number of white pixels in the image to distinguish a planted clique example from a non-planted one. Since we are interested in the presence of patterns in the image and the connection between those and the human visual system, we would like to have two images with the same average degree. In order to achieve this, when generating the two different classes of graphs, we should use different probabilities. Let us denote p as the probability to generate an edge in the pure Erdős-Rényi graph, while for the graph with the planted clique, we add an edge with probability p_0. We desire the average degree to be the same in both graphs. One can define p as a function of p_0 so that eventually is possible to set $p_0 = 0.5$ and change p accordingly to get the same average degree.

It is very simple to compute the average degree in both cases since we are dealing with binomial variables. When adding a clique, we take a set of k nodes and make it into a fully-connected subgraph (while preserving the connections that go from the nodes of this set to other nodes in the larger graph). This can be seen as simply adding a term to the average degree that accounts for a subset that is fully connected. This can be written simply as:

$$pn(n-1) = p_0 n(n-1) + (1-p_0)k(k-1)$$

where the left-hand side is the average degree of the pure random graph, while the right-hand side is the average degree of the graph with a planted clique of size k. In this way one can compute $p(p_0)$ as:

$$p(p_0) = p_0 + (1-p_0)\frac{k(k-1)}{n(n-1)}$$

which indeed corrects the number of white pixels in the image (as we can see from Figure 7)

Actually, we will keep this correction only to compare the CNN performance with the human performance. The correction changes the setting of the original problem, but it is still useful to avoid teaching to the network (or to humans) simple tasks such as counting the number of white pixels. Let us mention that we will analyze also the simple algorithm that counts the average degree, with surprising results, that we will describe in the final sections

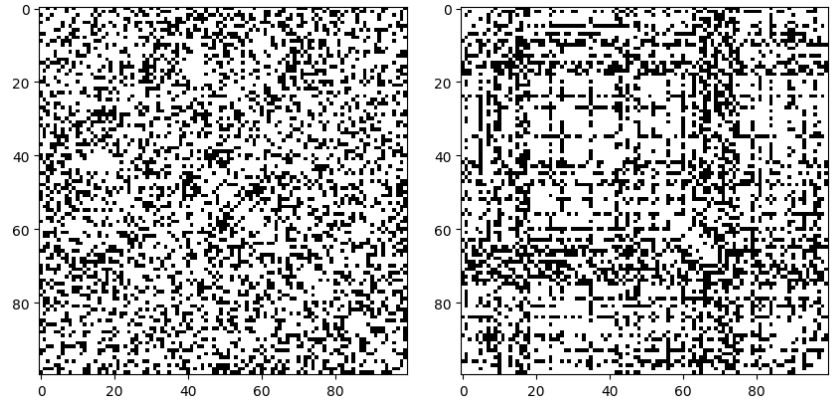

Figure 7: Left: Adjacency matrix of a pure Erdos-Renyi graph of size $n = 100$ with a corrected edge probability $p(p_0)$
Right: Adjacency matrix of a graph of size $n = 100$ with a planted clique of size $k = 60$ and edge probability $p_0 = 1/2$

3.3.1 CNN architecture

Let us shortly describe the network architecture that we used for the experiment. In the last few years, many new architectures were introduced in the field of image recognition tasks. Yet, we decided to keep the architecture pretty simple to have the possibility, later on, to study the learned features. Many modern CNNs have extremely complex structures and sometimes they are designed with a trial-and-error procedure. The decision to keep the network simpler is also useful for computational efficiency reasons: fewer parameters correspond to faster training.

Since depending on the size of the graph n, the image will have a different size, one has to slightly change the architecture depending on n. However, the basic structure stays the same, meaning that the blocks that constitute the network remain unchanged. If the image has a small size fewer blocks are needed since as we have seen before, convolutional and pooling layers reduce the dimensionality of the image. When the image is small enough, one can connect each pixel to a neuron and design a final linear layer that classifies the data. For a larger image, instead, it is necessary to apply many blocks to get a dimensionally reduced representation of the input.

Let us describe now the blocks that we used: these were very simple combinations of convolutional and pooling layers. More precisely, the first three operations were convolutional layers with c different kernels each, so that the output of each layer were c dimensionally reduced images. Then the last operation was a max-pooling operation on each of those c images. Usually before the max-pool, we applied a dropout, which is a technique that typically improves the generalization of the network [23]. The dropout just sets to zero some outputs in a layer of neurons: if y_i is the output of the i^{th} neuron, it will be set to zero with probability p.

This is the basic block we used in the experiment. We used almost always 3×3 kernels and a dropout probability $p = 0.2$: these were the ones that worked the best in terms of performance on the test set. The output of the last block is a series of c_{last} dimensionally reduced images of size $n_{last} \times n_{last}$: all the pixels of each image are then connected into a fully connected layer so that the input size is $c_{last} n_{last}^2$. As we said, the number of blocks depended on the size of the image. We used as many blocks as needed to feed into the fully connected layer an input of order 10^2. Usually, we used a series of 2 fully connected layers to classify the graph with ~ 100 neurons per layer.

Also, let us add that it is necessary to choose a non-linear function f to use for each layer. Many modern deep neural networks use ReLU activation functions, defined as follows:

$$ReLU(x) = \max(0, x)$$

We also used this activation function, which works extremely well if combined with batch normalization (the output of a mini-batch gets normalized so that it has zero mean and unit variance). In fact, those non-linear activation functions work well when the argument is centered around zero [23].

3.3.2 Phase transition

Finally, we can describe the outcome of the experiment. At a fixed size of the graph n, we set a value for the clique size k and started to train the network. As we have said above, we used the transfer learning method to train the network, meaning that we started to train it with high values of k, which represent easy instances, and eventually decrease the value of k up to the information-theoretic limit. For each value of k, we first generate a mini-batch and train the network with it. The training on the mini-batch stops in two cases: if we reach a maximum arbitrary number of steps of the gradient descent, or if we see that the error on the validation set starts to increase too much. This signals that the network has learned the instances present in the mini-batch, therefore we discard it and generate a fresh new sample.

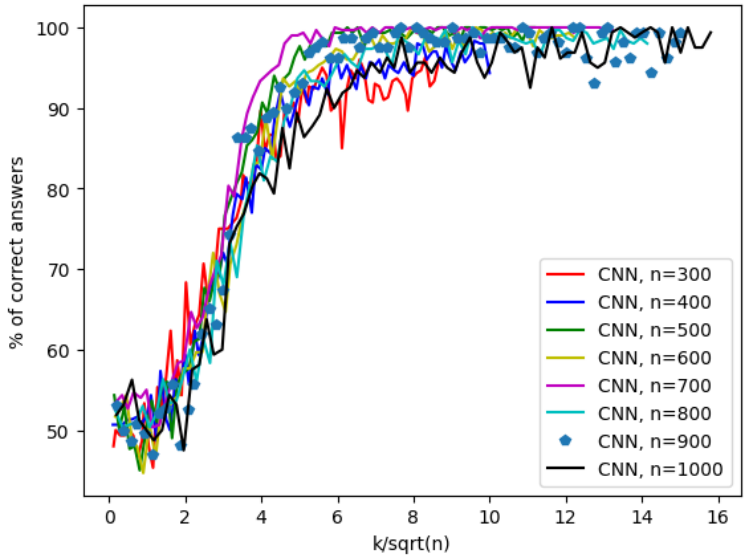

Figure 8: Performance of the CNN in detecting the Planted clique in graphs of different sizes

We repeat this iterative procedure up to the point when the error on the validation set is so low that it means that the network can predict almost perfectly all the instances for a given k. At this point, we do not reinitialize the weights of the network and we repeat the same training procedure, but this time with a smaller value for k. Usually, the value of k, with which we start is rather high compared to $\sqrt{n}$, since in this setting the problem is easier. As we start to decrease the value of k the problem gets harder and also the training procedure takes longer. For large clique sizes, it takes less than 100 examples in the training set to learn almost perfectly to detect the graphs with a

planted clique. As we start to approach the sizes of k that scale as $\sqrt{n}$, the training starts to get longer, and larger training samples are required to learn the problem perfectly.

Eventually, around the value of $k \simeq 3\sqrt{n}$, there is a transition. The network cannot learn how to solve the problem: both the training error and the validation error start to become higher and even if the training is extremely long with a very high number of samples (around 10^5), still the performance decreases as the clique size decreases. After the transition the network cannot perform better than chance, meaning that the performance on the test set oscillates around 50%. In Figure 8 one can see that for different values of n, the transition occurs basically at the same point (notice that we look at $k/\sqrt{n}$ and not just at k).

3.3.3 Is transfer learning useful?

But is transfer learning really useful? For instance, one could be able to learn just some features of the problem in the easy phase, which are enough to solve the problem in this phase. Then later on, as the problem gets harder, the network remains stuck and it is not able to learn other relevant features of the problem that could be useful in the hard phase. However, this does not seem to be the case we have tried both methods: with and without transfer learning.

Without using the transfer learning method, the procedure that we described above remains the same as the one described above. The only difference is that as we finish the training with a selected value of k, we first reinitialize the weights of the CNN and then train the network with a different clique size. As we can see from Figure 9, the two transitions are indistinguishable.

On the other hand, it seems that the transfer learning method accelerates learning. It is very easy for the network to get stuck in non-optimal solutions if we train it without transfer learning.

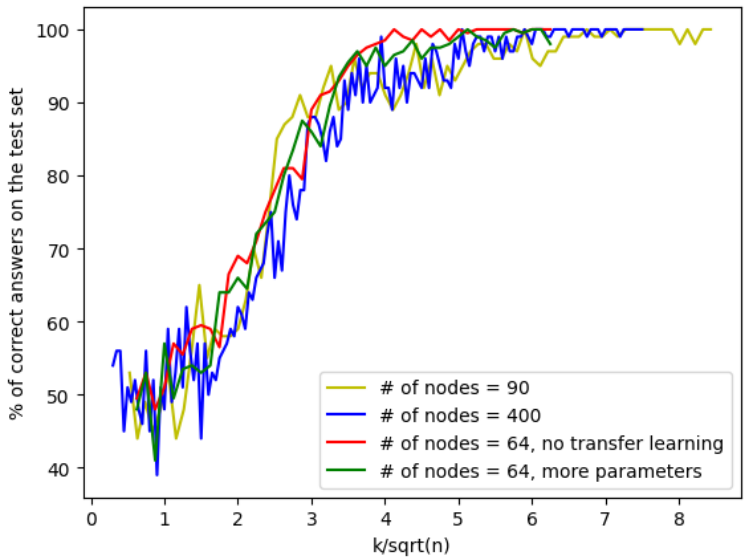

Figure 9: Compared performance of various CNN architectures (with and without transfer learning) in detecting the Planted clique in graphs of different sizes

In Figure 9 we show also the transition of networks with many more parameters and trained on larger datasets. Surprisingly there is no way to perform better: the

transition seems to be robust and independent of the complexity of the network.

3.3.4 Understanding the learned features

Since we are interested to confront the outcome with the results of human performance, we are also interested in explaining which are the features that the CNN is looking at when making a prediction. This problem is very general in the field of image recognition since it is not clear how these networks capture the relevant features of an image.

A popular way of explaining the learned features of a network is reversing the training [31]. Once the network has been trained, one can look for images that maximize the activation of some neurons. This means that instead of modifying the weights to get the desired output, we rather start with a random image and modify it to maximize the activation of some neurons in the network. If the activation of a neuron is maximized this signals that the neuron has learned to detect such inputs.

To perform this reversed training, we give a random image to the network and we look at the activation of a single neuron in the last linear layer of the CNN. Once the activation is computed, we can compute also the gradients with respect to the input. So if the input is the matrix A and the activation is given by x, we should compute $\frac{\partial x}{\partial A_{i,j}}$ for each $i, j \in [n]$. After computing this quantity we can perform the usual stochastic gradient descent on the input, by changing the various pixels.

The main issue with this procedure in our task is that the images that we are using have some special properties, that we would like to maintain. These come from the fact that our image is an adjacency matrix, thus is symmetric and each pixel can take only $\{0, 1\}$ values. To enforce those constraints it is necessary to use some regularizations. Regularizations are additional terms to the gradient descent that allows us to satisfy some constraints. Two very popular regularizations are L1 (LASSO) and L2 (Ridge regression) regularizations [23]. The first one penalizes very small non-zero parameters, while the second one penalizes parameters with large values.

In our case the regularizations we had to add penalized non-symmetric images with pixels different from $\{0, 1\}$. Practically, this was achieved by adding some terms to the usual gradient descent. In fact, the gradient descent can be thought of as a discrete dynamical system, since the weights, or as in this case the input, is modified at each iteration and it will therefore converge eventually to a stationary point. Regularizations ensure that this stationary point satisfies the necessary constraints.

Hence, to ensure the symmetry, at each step of the gradient descent we reduced the anti-symmetric part of the new input, by a small value: in this way, after a couple of iterations, we will get a symmetric input. On the other hand, we added a penalization to each input with non $\{0, 1\}$ valued pixels. Eventually, we also added some stochasticity to the gradient descent. The full gradient descent step is the following (applied $\forall i, j \in [n]$):

$$A_{i,j}^{(t+1)} \leftarrow A_{i,j}^{(t)} - \eta \frac{\partial x}{\partial A_{i,j}^{(t)}} - \beta A_{i,j\,asymm}^{(t)} - \gamma(2A_{i,j}^{(t)} - 1) + \xi$$

where $A^{(t)}$ represents the input at the iteration number t, $A_{asymm}^{(t)}$ is the anti-symmetric part of the input, ξ is a random noise, while η, γ and β represent small constants that are chosen arbitrarily. The initial input $A^{(0)}$ is a random image with pixel values in the interval $(0, 1)$.

Once the gradient descent is defined it is possible to select a single neuron, or more than one, and find the inputs that maximize its activation. Some examples are presented below in Figure 10. It is possible to see that many neurons have an enhanced activation when the image is full of white "blobs"; in some inputs that we found, we can also see clear patterns, similar to those that appear in the adjacency matrices with a planted clique. This confirms the supposition that the CNN is actually detecting those patterns.

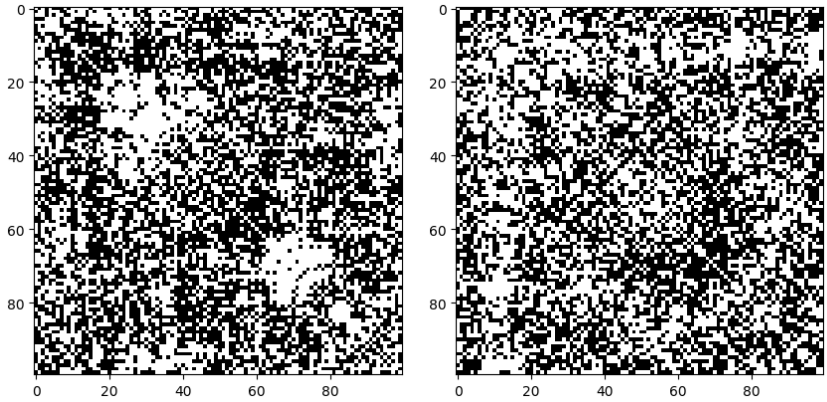

Figure 10: Two inputs that maximize the activation of the neurons in the last layer

3.3.5 Comparison with human experiments

The following section is dedicated to the comparison of our results with the human visual system. This part of the research was made mainly by Daniele Tirinnanzi and Eugenio Piasini, that are part of the computational neuroscience group at SISSA. Since this is not part of our work we will just briefly describe the set-up of the experiment. The experiment is still a pilot, but very interesting results are emerging.

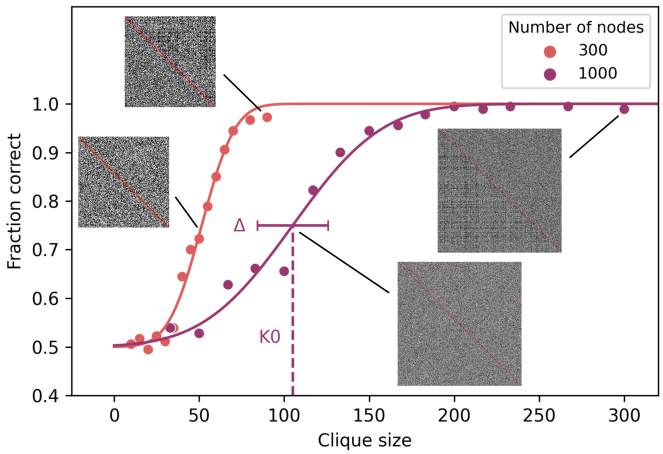

Figure 11: Performance of humans in detecting the Planted clique in graphs of different sizes. The images near the plots represent the stimulus given to the participants

The participants in the experiment are given an exhaustive explanation of the problem. After the explanation, they are presented with a set of adjacency matrices that are presented in pairs: on one side there is an image with a planted clique, while on the other side, the adjacency matrix is purely Erdős-Rényi (see Figure 11). The participant is then asked which is the side with a planted clique. This experiment is

set up as a "game"; the first few instances are easy to detect, but the clique size slowly decreases so that the decision becomes harder and harder. In the end, the results are saved in a database.

Clearly, the procedure is not identical to the one we use when training and testing the CNNs, still is very similar. The experiment still needs to be run on a larger set of participants, but from the small pilot, it is evident that the performance of the CNNs and the humans is similar. A transition occurs in the vicinity of $k = 3\sqrt{n}$ (see Figure 11). The meaning of such a transition has to be investigated further.

3.4 Alternative Neural Networks and possible improvements

Finally, we describe and briefly analyze some alternative NNs that we developed during the project. The main question we were focusing on was: is there any neural network that could perform better than AMP and, more specifically, enter the hard region of the problem? The desire to try a neural network approach was born from the Theorem 2 in [8], saying that no local algorithm can perform better than AMP in the task of recovering cliques in a random graph. Intuitively, an algorithm is said to be local if its output is based on the local neighborhood of each node. Therefore it neglects long-range interactions between nodes.

Our first attempt consisted in using the understanding we got from the visual task, to design "filters" able to detect non-local patterns. What we are able to see in the adjacency matrix are large white stripes, representing a large number of fully connected nodes, that appear as close indices in the matrix. Therefore, our idea was to design filters that select some of those rows (or columns) and then combine them to predict the presence of a clique. Each filter has weights that should be learned. We call this network the Rows/Columns NN.

Very quickly we found out that the network we were using was not very smart, due to permutational invariance. The best strategy would be to use equal weights since there is no preferential ordering. In this case, the filters represented just a sum of rows (or columns), which means that this NN is just looking at the degree of the network. In this sense it is local, therefore it would not be able to perform better than AMP.

3.4.1 Counting algorithm

Subsequently, we tried to design again different filters that would work in a smarter way. After a while, we found out that even if we were able to improve the performance of the CNN we were still very far from entering the hard phase. Oddly, the best algorithm that we were able to design was an algorithm counting the total number of edges and making a decision based on that count. Notice that in this case, we avoid the shift in probability described above and consider the two graphs with $p_0 = p$.

This algorithm is the easiest hypothesis test one can think of, yet it seems to be able to perform better than all the NNs we designed. Therefore we decided to study this algorithm analytically. The counting argument is simple, but it should be understood in the asymptotic limit for $n >> 1$ and k proportional to $\sqrt{n}$. The case in which the clique size grows linearly with the graph size is trivial, so we do not need to analyze it.

We want to make a hypothesis test based on the total number of edges in the graph we are presented with. A graph can be purely Erdős-Rényi, so $G(0.5, n)$ or with a planted clique $G(0.5, n, k)$. Since the probability to have an edge between two nodes is simply given by $p = 0.5$, when summing all the entries of the adjacency matrix we will get in both cases a new random variable that will be distributed as a Gaussian (due to the central limit theorem). Let me call $S = \sum_{i,j} A_{ij}$, the sum of all the entries of the adjacency matrix A, and let me denote by $X = 1$ the graph with a clique and $X = 0$

the pure random case. Finally, we have that:

$$P(S|X=0) \sim N(n(n-1)/2, n(n-1)/2)$$

$$P(S|X=1) \sim N(n(n-1)/2 + k(k-1)/2, (n(n-1) - k(k-1))/2)$$

where $N(\mu, \sigma^2)$ is a Gaussian with mean μ and variance σ^2.

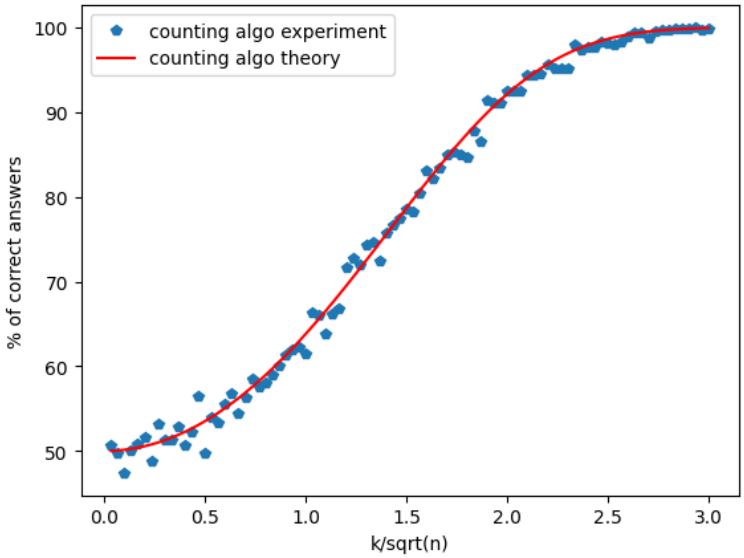

Figure 12: Performance of the "Counting algorithm" in detecting the Planted clique in graphs of size n=900

At this point, we have to choose a threshold to say whether there is or not a planted clique in the graph, by looking at the random variable S. We set this threshold to be the point in between the two mean values: $tr = n(n-1)/2 + k(k-1)/4$. This value is not important in the asymptotic limit, since the two Gaussians are extremely peaked in the asymptotic limit, so any point in between the two average values would work. Therefore to get the performance of this test we need to compute the following probabilities: $P(S < tr|X=0)$ and $P(S > tr|X=1)$. Let me compute just the first one (the second can be computed in the same way).

$$\begin{aligned} P(S < tr|X=0) &= P(S - n(n-1)/2 < k(k-1)/4|X=0) \\ &= P\left(\frac{S - n(n-1)/2}{\sqrt{n(n-1)/2}} < \frac{k(k-1)/4}{\sqrt{n(n-1)/2}}\middle|X=0\right) \end{aligned} \tag{3}$$

but

$$P\left(\frac{S - n(n-1)/2}{\sqrt{n(n-1)/2}} < \frac{k(k-1)/4}{\sqrt{n(n-1)/2}}\middle|X=0\right) = P\left(\frac{S-\mu}{\sigma} < \frac{k(k-1)/4}{\sqrt{n(n-1)/2}}\middle|X=0\right)$$

and since

$$Z = \frac{S-\mu}{\sigma}$$

is a Gaussian with zero mean and unit variance, we get:

$$P(S < tr|X=0) = 0.5\left(1 + erf(\frac{k(k-1)}{4n(n-1)})\right)$$

In the same way, one can compute the other desired probability:

$$P(S > tr|X = 1) = 0.5\left(1 + erf(\frac{k(k-1)}{4(n(n-1) - k(k-1))})\right)$$

that is equal to the one above in the asymptotic limit, therefore is an optimal test in this sense.

The red curve you can see in the Figure 12 is the function above and, as you see, it fits perfectly the observed data even at finite sizes.

3.4.2 Are Neural networks dumb?

Moved by some recent papers [4, 10], we wanted to see if the performance of this algorithm could represent a limit for the NNs we have investigated so far. Indeed, all the experiments we made so far were not able to perform better than this simple statistical test (see Figure 13).

This represents a very interesting point since there are two reasons for this behavior. The problem could be really hard, meaning that there is nothing to be learned in the hard phase. This is linked to the question P vs NP, but in this problem, it is much more subtle. The planted clique conjecture says that solving the planted clique problem in the hard phase is NP. Still, there is no proof for that yet, and our experimental results in this sense confirm empirically the hardness of the problem. On the other hand, we should analyze the performance of many other NNs to confirm this result.

The second reason could be that NNs are not smart enough to learn a better statistical test than the counting algorithm. This is in line with the papers cited above. Without exploiting the symmetries and using a smart way to develop an algorithm it's impossible to use NNs efficiently.

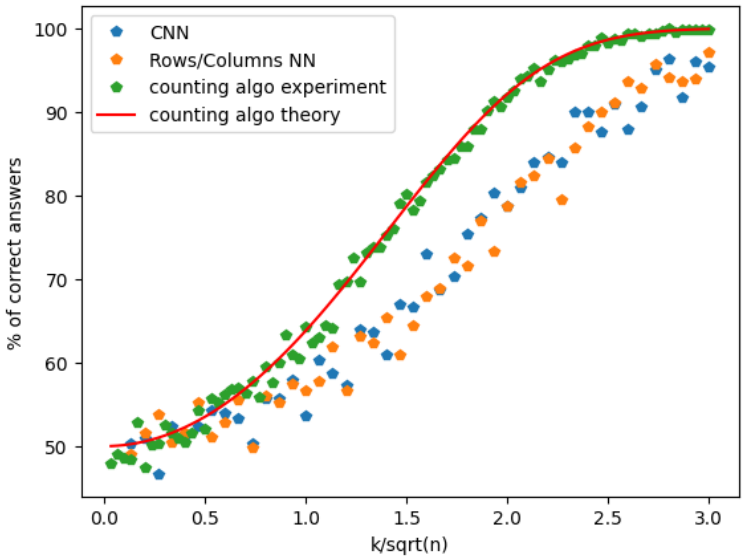

Figure 13: Comparison between the performance of the "Counting algorithm" and other NNs in detecting the Planted clique in graphs of size n=900

4 Conclusions

In this thesis, we have focused on the Planted clique problem from different perspectives. We described the best-known algorithm that solves the problem in polynomial time, hence AMP. Then we focused on a neural network approach, hoping to improve the performance of AMP. On the other hand, the aim of the study was also to compute the performance of both NNs and humans in solving the task, in order to see if they share strategies.

The AMP algorithm defined in [8] was designed to recover a planted clique random graphs and it has some well-defined limits. Inspired by the fact that no local algorithm can outperform AMP in this task, we decided to exploit neural networks, and more precisely CNNs, to detect planted cliques in random Erdős-Rényi graphs.

Since we wanted to give the same detection task to humans, we decided to modify slightly the problem. A shift in the edge probability ensured that the average degree of the two classes of graphs was the same. Due to this modification and the fact that we considered the detection, rather than the recovery task, we plot in Figure 14, the comparison between the AMP performance and the results from the CNN.

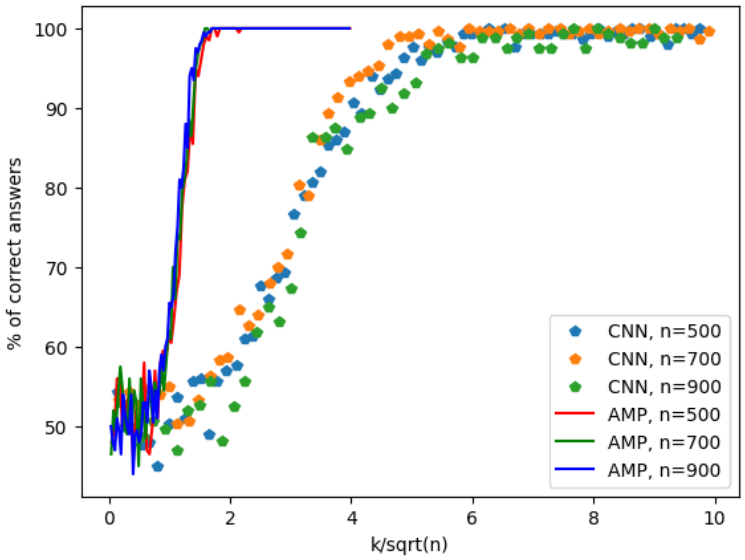

Figure 14: Comparison between the performance of CNNs and AMP algorithm for different graph sizes. The performance is tested on graphs with the shift in probability defined above

The shift in probability does not affect much the performance of AMP, so it is noticeable that the two transitions occur at different positions. The curves of the CNNs performance collapse as we scale the clique size with the square root of the graph size. This is not the case for the AMP algorithm: the performance improves as the graph size increases, signaling that in the asymptotic limit, the algorithm could reach a limiting performance. However, it is clear that the CNNs are not able to perform better than AMP, yet the transitions seem to occur at a clique size close to $k_{CNN} \simeq 3\sqrt{n}$.

We can approximate the CNNs transitions with a sigmoid function and determine the point K_0 at which the performance reaches 75%. By plotting those points we see that in this regime of n values, the relationship is well fitted also by a linear function (see Figure 15). Therefore, further work has to be done on this point. In order to understand whether the relationship is better fitted by a square root law or a linear

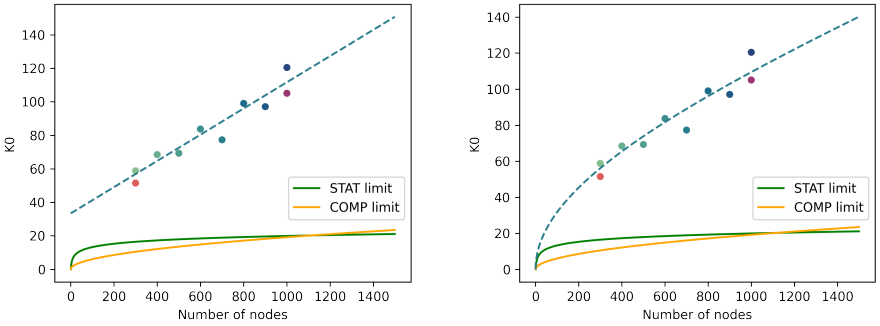

Figure 15: Fit of the transition point as a linear relationship (left) or a square root relationship (right) with the graph size (number of nodes). The blue dots represent the CNNs' transition point, while the red ones are the transitions of humans. The curves below represent the statistical/information-theoretical limit and the algorithmic/computational limit

law, it is necessary to test the CNN with a wider range of graph sizes. Additionally, finding an algorithm that displays the same behavior and can be treated analytically could give a better understanding of the strategies involved.

From Figure 15 we also see that both the linear and the square root relationships fit extremely well with the human performance on the task, signaling that the strategies involved to solve the task are extremely similar. The work on human data is still at the beginning, yet we do not expect to see major differences when looking at a larger set of subjects.

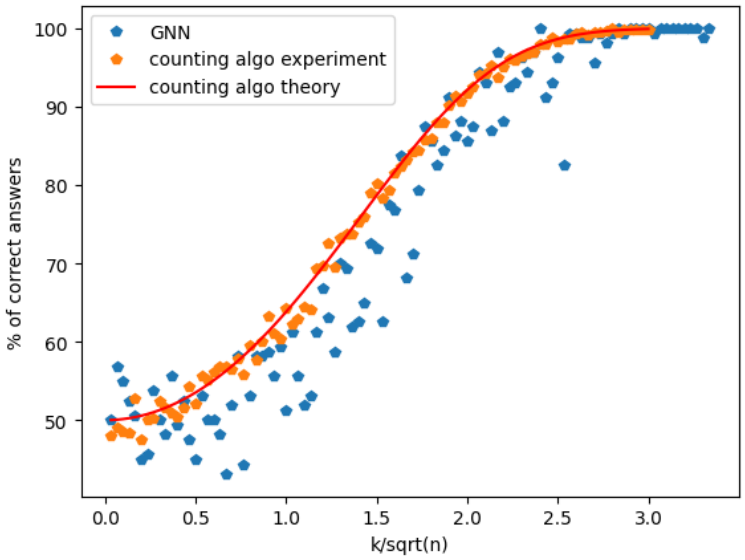

Figure 16: Comparison between the performance of the "Counting algorithm" and a GNN in detecting the Planted clique in graphs of size n=900

Finally, after our failed attempts at beating the performance of AMP using different

neural networks, and moved by recent papers suggesting that NNs cannot do better than simple greedy algorithms, we decided to test a wide variety of NNs in order to see if they are able to outperform the simple counting method. In this setting, we clearly avoid the shift in probability described before. The results suggest that all the networks that we exploited so far are not able to beat this simple algorithm.

Obviously, we just tested a very small number of NNs, but let us anticipate that our last results show that neither Graph neural networks (GNNs) are able to beat the counting method (see 16). Graph neural networks are very recent networks that leverage the structural symmetries of graphs. However, those results have to be analyzed further.

The results from this project opened many questions. Why do CNNs and humans perform in such a similar way, and what is the relationship between the graph size and the transition point? Is the performance of all NNs limited by the simple counting algorithm or will more complex architectures allow us to outperform this method? We hope to answer those questions by looking to a wider variety of algorithms and neural networks, and by understanding better the theory behind the Planted clique problem.

References

[1] Noga Alon, Michael Krivelevich, and Benny Sudakov. "Finding a large hidden clique in a random graph". In: *Random Struct. Algorithms* 13 (1998), pp. 457–466.

[2] Noga Alon and Joel H. Spencer. "The Probabilistic Method". In: *ACM-SIAM Symposium on Discrete Algorithms*. 1992.

[3] Noga Alon et al. "Testing k-wise and almost k-wise independence". In: *Symposium on the Theory of Computing*. 2007.

[4] Maria Chiara Angelini and Federico Ricci-Tersenghi. "Modern graph neural networks do worse than classical greedy algorithms in solving combinatorial optimization problems like maximum independent set". In: *Nature Machine Intelligence* 5.1 (Dec. 2022), pp. 29–31. DOI: 10.1038/s42256-022-00589-y. URL: https://doi.org/10.1038%5C%2Fs42256-022-00589-y.

[5] Mohsen Bayati and Andrea Montanari. "The Dynamics of Message Passing on Dense Graphs, with Applications to Compressed Sensing". In: *IEEE Transactions on Information Theory* 57.2 (Feb. 2011), pp. 764–785. DOI: 10.1109/tit.2010.2094817. URL: https://doi.org/10.1109%2Ftit.2010.2094817.

[6] Quentin Berthet and Philippe Rigollet. "Complexity Theoretic Lower Bounds for Sparse Principal Component Detection". In: *Annual Conference Computational Learning Theory*. 2013.

[7] Béla Bollobás and Paul Erdös. "Cliques in random graphs". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 80 (1976), pp. 419–427.

[8] Yash Deshpande and Andrea Montanari. *Finding Hidden Cliques of Size $\sqrt{N/e}$ in Nearly Linear Time*. 2013. arXiv: 1304.7047 [math.PR].

[9] David L. Donoho, Arian Maleki, and Andrea Montanari. "Message-passing algorithms for compressed sensing". In: *Proceedings of the National Academy of Sciences* 106.45 (Nov. 2009), pp. 18914–18919. DOI: 10.1073/pnas.0909892106. URL: https://doi.org/10.1073%2Fpnas.0909892106.

[10] O. Duranthon and L. Zdeborová. *Optimal Inference in Contextual Stochastic Block Models*. 2023. arXiv: 2306.07948 [cs.SI].

[11] David Gamarnik. "The overlap gap property: A topological barrier to optimizing over random structures". In: *Proceedings of the National Academy of Sciences* 118.41 (Oct. 2021). DOI: 10.1073/pnas.2108492118. URL: https://doi.org/10.1073%2Fpnas.2108492118.

[12] David Gamarnik and Ilias Zadik. *The Landscape of the Planted Clique Problem: Dense subgraphs and the Overlap Gap Property*. 2019. arXiv: 1904.07174 [math.ST].

[13] Elad Hazan and Robert Krauthgamer. "How Hard Is It to Approximate the Best Nash Equilibrium?" In: *SIAM J. Comput.* 40.1 (2011), pp. 79–91. DOI: 10.1137/090766991. URL: https://doi.org/10.1137/090766991.

[14] Adel Javanmard and Andrea Montanari. *State Evolution for General Approximate Message Passing Algorithms, with Applications to Spatial Coupling*. 2012. arXiv: 1211.5164 [math.PR].

[15] John M. Jumper et al. “Highly accurate protein structure prediction with AlphaFold”. In: *Nature* 596 (2021), pp. 583–589.

[16] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[17] Antti Knowles and Jun Yin. *The Isotropic Semicircle Law and Deformation of Wigner Matrices*. 2012. arXiv: 1110.6449 [math.PR].

[18] Ludek Kucera. “Expected Complexity of Graph Partitioning Problems”. In: *Discret. Appl. Math.* 57 (1995), pp. 193–212.

[19] Siyuan Ma, Raef Bassily, and Mikhail Belkin. “The Power of Interpolation: Understanding the Effectiveness of SGD in Modern Over-parametrized Learning”. In: *International Conference on Machine Learning*. 2017.

[20] James N. MacGregor and Yun Chu. “Human Performance on the Traveling Salesman and Related Problems: A Review”. In: *J. Probl. Solving* 3 (2011).

[21] Jay Mardia. *Is the space complexity of planted clique recovery the same as that of detection?* 2020. arXiv: 2008.12825 [cs.CC].

[22] Warren S. McCulloch and Walter Pitts. *A logical calculus of the ideas immanent in nervous activity. Bulletin of mathematical biophysics, vol. 5 (1943), pp. 115–133.*

[23] Pankaj Mehta et al. “A high-bias, low-variance introduction to Machine Learning for physicists”. In: *Physics Reports* 810 (May 2019), pp. 1–124. DOI: 10.1016/j.physrep.2019.03.001. URL: https://doi.org/10.1016%2Fj.physrep.2019.03.001.

[24] William Merrill and Nikolaos Tsilivis. *Review: Planted Clique as a Case Study for Statistical-Computational Gaps*.

[25] Marc Mézard and Andrea Montanari. “Information, Physics, and Computation”. In: 2009.

[26] Andrea Montanari and Tommaso Rizzo. “How to compute loop corrections to the Bethe approximation”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2005.10 (Oct. 2005), P10011–P10011. DOI: 10.1088/1742-5468/2005/10/p10011. URL: https://doi.org/10.1088%2F1742-5468%2F2005%2F10%2Fp10011.

[27] Nick Petosa and Tucker R. Balch. “Multiplayer AlphaZero”. In: *ArXiv* abs/1910.13012 (2019).

[28] David J. Thouless, Philip W. Anderson, and Richard G. Palmer. “Solution of ’Solvable model of a spin glass’”. In: *Philosophical Magazine* 35 (1977), pp. 593–601.

[29] Matula D. W. “On the complete subgraphs of a random graph”. In: *Combinatory Mathematics and its Applications* (Chapel Hill, N .C ., 1970) (1970), pp. 456–469.

[30] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. “Understanding belief propagation and its generalizations”. In: 2003.

[31] Jason Yosinski et al. “Understanding Neural Networks Through Deep Visualization”. In: *ArXiv* abs/1506.06579 (2015).

[32] Aston Zhang et al. *Dive into Deep Learning*. 2023. arXiv: 2106.11342 [cs.LG].

[33] Fuzhen Zhuang et al. *A Comprehensive Survey on Transfer Learning*. 2020. arXiv: 1911.02685 [cs.LG].