Effectiveness of SDP rounding using Hopfield Networks

Éanna Curran
University College Dublin
eanna.curran@ucdconnect.ie

Saurabh Ray
New York University, Abu Dhabi
saurabh.ray@nyu.edu

Deepak Ajwani
University College Dublin
deepak.ajwani@ucd.ie

Abstract

We consider if the techniques used in the design of approximation algorithms can be leveraged to develop effective learning solutions for NP-hard graph problems. Specifically, we focus on semi-definite programs (SDPs), a powerful technique from operations research, that has been used in the design of many approximation algorithms. In these approximation algorithms, one typically solves an SDP relaxation of the optimization objective and then performs some problem-specific rounding of the SDP solution. In this paper, we present a learning framework that utilizes Hopfield networks to round the SDP solution for different problems. We show empirically that the approach performs well on benchmarking instances of three well-studied problems namely Max-Cut, Max-Clique and Graph Coloring. The solutions obtained are close to optimal and significantly better than those obtained by the corresponding approximation algorithms. The primary advantage of such a simple heuristic is that it can be applied to a large number of problems without much problem-specific engineering. Another advantage of our approach is that we only need a small number of tunable parameters in the rounding algorithm - this is because we start with an SDP solution which already contains useful global information. This in turn means that the parameters can be learnt efficiently with a small amount of training data. We also show that even approximate solutions to the SDP relaxation suffice - this makes our approach fast and practical.

1 Introduction

Semi-definite programming (SDP) is one of the most powerful techniques used in the design of approximation algorithms for combinatorial optimization problems. These approximation algorithms typically solve an SDP relaxation to the problem and round it to obtain a feasible solution (see the classical textbooks [1–3] for several examples of these approximation algorithms). The latter part is usually the non-trivial part and while very clever rounding algorithms have been designed for different optimization problems, no general techniques exist. This implies that every new variant of even well-studied problems typically requires designing a rounding algorithm from scratch. This, along with the fact that these algorithms are usually geared towards worst-case performance guarantees and do not exploit real-world instance distributions, has prevented the widespread adoption of SDP-based techniques to solve practical problems. One fact that stands out clearly from the existence of so many non-trivial approximation algorithms based on SDPs (see e.g., [4–10]) is that the SDP solution does contain useful information that captures the global structure of the problem. It is therefore desirable to find a generic rounding technique which can use the SDP solution to find good solutions for real-world instances. Such a rounding technique may not come with theoretical guarantees but it should be easy to use and work well for practical instances of a large number of problems.

In this paper, we present a simple rounding technique based on Hopfield networks [11], whose edge weights are determined from the SDP Gram matrix for the corresponding problem. The dependence of the edge weights on the SDP Gram matrix is problem-specific and can either be hand-designed or learnt from data. The latter approach allows for the rounding technique to be dependent on and exploit the instance distribution. Our technique is sufficiently generic that it can be used with any problem for which there is an SDP relaxation. We chose to use a Hopfield network for rounding since it is a simple way to make the outputs binary while respecting pairwise correlations among them. We demonstrate the efficacy of our approach on three well-studied problems: Max-Cut, Max-Clique, and Graph Coloring. In all three cases, our heuristic rounding algorithm finds solutions close to the optimal solution that are significantly better than the solutions found by the corresponding approximation

E. Curran et al., Effectiveness of SDP rounding using Hopfield Networks. *Proceedings of the Third Learning on Graphs Conference (LoG 2024)*, PMLR XXX, Virtual Event, November 26–29, 2024.

algorithms. Our work indicates that there is potential to design heuristics for SDP rounding that work well for practical instances.

Our experiments also indicate that our rounding algorithm is not very sensitive to noise in the SDP solution. This allows us to replace the exact computation of the SDP solution with an approximate computation, thus speeding up the algorithm while incurring very little loss in the quality of the solution. To give an example, our algorithm for Max-Cut finds a cut of size within 0.6% of the optimal solution in a graph with 10,000 nodes obtained from a benchmarking dataset in about one minute.

From a machine learning (ML) point of view, existing approaches for solving combinatorial optimization problems typically rely on end-to-end deep learning (e.g., Graph Neural Networks). Such techniques throw away the algorithmic techniques that have already proven useful and, as a result, also require a lot of time and data for training. These techniques also struggle to generalize well to instances of larger size (compared to training instances) and instances from a different input distribution. In light of these issues, Bengio et al. [12] opined in a survey of learning algorithms for combinatorial optimization that *"We believe end-to-end machine learning approaches to combinatorial optimization are not enough and advocate for using machine learning in combination with current combinatorial optimization algorithms to benefit from the theoretical guarantees and state-of-the-art algorithms already available."* In line with this, we directly integrate the SDP solution (which is theoretically known to be very useful) in the learning technique. Due to the simplicity of our rounding algorithm, it suffices to use shallow neural networks to compute the Hopfield network weights - this means that their parameters can be learnt efficiently with very little data. Our empirical results show that the learning generalizes across instances of different sizes, allowing us to learn from small instances and apply the model to larger instances. It also generalizes quite well across different instance distributions.

Embeddings obtained from Graph Neural Networks (GNNs) are known to be effective in capturing local topological features, but they often struggle to capture the global combinatorial structure in a problem instance (see e.g. [13],[14]). In contrast, linear programming (LP) and semi-definite programming (SDP) based relaxations have been very successful in the design of approximation algorithms, implying that "embeddings" obtained from them are good at capturing global combinatorial information, and this can be computed efficiently. It is therefore natural to consider machine learning architectures that leverage SDP-based embeddings. This is what motivates our current work.

Information from an SDP solver comes in the form of a Gram matrix which indicates pairwise correlations. Hopfield networks are a simple way to do a rounding based on pairwise correlations. While one could certainly use more sophisticated techniques like GNNs, our main goal in the paper is to demonstrate the usefulness of SDP-based embeddings and for this, Hopfield networks suffice.

While we only considered three well-studied classical problems in this paper, SDPs have been successfully used in a wide variety of problems. Many combinatorial optimization problems of interest can be formulated as quadratic integer programs with binary variables and admit an SDP relaxation. Our approach is general enough to be applied to any such problem.

Note that heuristics like local search, which are known to work well in practical settings (especially 1-local search), can always be used on top of our framework. We do not study such additional tricks in this paper so that we can evaluate our framework in isolation.

Our Contribution. To summarize, our contribution in this work is as follows:

- We first show that there is a simple heuristic based on Hopfield networks which is able to round SDP solutions for practical instances of well-known combinatorial optimization problems, namely Max-Cut, Max-Clique and Graph Colouring.
- Next, we show that the functional relation between the SDP solution and the Hopfield network can be learnt from data. This allows the rounding technique to be adapted to new problems and also to the target input distributions for those problems. Furthermore, shallow neural networks suffice for this purpose, which means that the training process does not require much data or processing time.
- Finally, we show that our learnt heuristic is robust to noise in the SDP solution. This allows the replacement of an exact computation of the SDP solution by a fast approximate computation with little loss in the solution quality.

2 Preliminaries

In this section, we briefly describe the basic concepts of Semi-definite programming and Hopfield networks that we use in the rest of the paper.

2.1 Semi-definite Programming.

Let S_n denote the set of $n \times n$ real-symmetric matrices and $Tr(X)$ denote the trace of matrix X. Further, let $C, A_1, ..., A_m \in S_n$ and $B_1, ..., B_m$ be the input variables for a given problem and $X \succeq 0$ denotes that matrix X is positive semi-definite. A semi-definite program (SDP) is a convex optimization problem of the form:

$$\begin{aligned} \min_{X \in S_n} \quad & Tr(CX) \\ \text{s.t.} \quad & A_i X = B_i, i = 1, ..., m, \\ & X \succeq 0 \end{aligned}$$

Semi-definite programs are a special case of cone programming. Any integer quadratic program can be relaxed into the form of an SDP and solved in polynomial time using algorithms such as interior point method [15]. Throughout this paper, we transform several NP-Hard problems in their integer quadratic form into SDP versions of that problem. We consider the classical optimization problems of Max-Cut, Max-Clique and Graph coloring. We describe these problems and the SDP formulations that we use in Appendix A.

2.2 Hopfield Networks.

A Hopfield network [11] is a fully connected network with n neurons which we can number from 1 to n. Each neuron has one output which is initially set to some value (often randomly). Then we update the outputs in several rounds. Each neuron updates its output to a new value which is obtained by taking a linear combination of the outputs of all the other neurons and applying a non-linear activation function. Specifically, for each pair of neurons i and j, the corresponding edge in the complete graph has a weight $\boldsymbol{a}_{ij}$ called the *interaction* between the neurons i and j. In addition, for each neuron i there is an associated bias b_i. If we denote the output of neuron i at any point in time by z_i, then it is updated as follows:

$$z_i := A\left(\sum_{j \neq i} \boldsymbol{a}_{ij} z_j + b_i\right)$$

where A is a nonlinear activation function (typically the sigmoid or the tanh function). The output of the neurons can either be updated one by one (i.e., asynchronously) or together (synchronously). In the latter case, when updated z_i for any i, we use the old values of z_j for all $j \neq i$. Typically we continue updating the outputs until they have converged (i.e., they don't change significantly from one round to the next) or a certain threshold number of updates have been performed.

3 Related Work

3.1 Semi-definite Programs and Approximation Algorithms

Semidefinite programming is among the most powerful tools used in the design of approximation algorithms [1–3]. Goemans and Williamson's algorithm [16] for the MaxCut problem is the first approximation algorithm (from 1995) based on semi-definite programming, and it is still considered to be among the simplest and most impressive results in this area. It is also known that under the Unique Games Conjecture, this algorithm provides the best approximation possible in polynomial time [17]. SDP-based approximation algorithms are known (e.g., [4–10]) for a range of combinatorial optimization problems, such as graph colouring and maximum clique.

3.2 Rounding Algorithms for LP and SDP

Linear Programming and Semidefinite Programming relaxations are central to the design of many approximation algorithms. Several broad as well as problem-specific techniques have been devised for rounding the solutions to such relaxations. See for instance [18–23] and the references therein.

3.3 Machine learning for Combinatorial Optimization

In the last decade, a large number of machine learning techniques have been developed to solve combinatorial optimization problems (see [12] for a recent survey and the citations therein). These include graph neural networks (see [24] for a very recent survey and the citations therein), reinforcement learning (see [25] for a survey), neural symbolic computing (see [26] for a survey) and graph representation learning (see [27] for a survey). We focus on the last technique as it is closest to our work. In graph representation learning, the first stage embeds the graphs into low-dimension vectors, and the second stage uses machine learning to solve the combinatorial optimization problems using the embeddings of the graphs learned in the first stage. In contrast, we use embeddings derived from SDP formulations and use Hopfield networks to solve combinatorial optimization problems using the SDP embeddings. Furthermore, in graph embedding methods, the learning of the embeddings of the graphs has its own objective, which may not rely on the optimization problems to be solved. In contrast, the SDP embeddings are problem-dependent and capture global combinatorial information about the problem being solved.

3.4 Machine Learning and SDP Gram matrix

In recent years, neural networks have been used to approximate SDP Gram matrix computation (see e.g., [28–32]). In contrast, there is little work in using SDPs for learning to solve problems. Some examples in this direction are the use of SDPs for designing semi-supervised SVMs [33], the use of SDP as a lower bound in a branch and bound algorithm for an unsupervised minimum sum-of-squares clustering [34], the use of a low-rank SDP for probabilistic inference in pairwise Markov Random Fields [35] and community detection [36]. There are even fewer examples of work that uses SDPs in a learning framework for effectively solving combinatorial optimization problems. One example is the use of SDP to learn the Lovasz-Θ function and then use that to find planted cliques in random graphs [37]. In contrast, we are investigating a general machine-learning framework that can use SDPs to solve combinatorial optimization problems.

4 Rounding SDP solutions using Hopfield Networks

In each of the three problems we study in this paper, the corresponding SDP returns an n-dimensional embedding $\boldsymbol{\sigma}_{\mathbf{v}}$ for each vertex v in the graph. Most algorithms based on SDPs do not directly use these vectors [1] and instead use the Gram matrix of pairwise dot products of these vectors. This is what we also do in this paper. The pairwise nature of the information extracted from the SDP naturally suggests the use of Hopfield networks in which the interaction between any pair of neurons is a function of the corresponding Gram matrix entry. The actual function used depends on the problem. Similarly, the bias and initial output used for each neuron, as well as the activation function, is problem-dependent. We use the Hopfield network to find a rounded solution as follows. We start by setting the output of each neuron to its initial value and update the outputs of each of the neurons (synchronously/asynchronously) until either a threshold number of rounds is exceeded or the outputs of all of the neurons have converged. At this point, we round each output by setting it either to the closest rounded value or to one of the values probabilistically (the closer the rounded value, the higher the probability of rounding to that value).

To reiterate, we first obtain the correlation between two variables as a function of the corresponding Gram matrix entry in a problem-dependent way. After this, we use the Hopfield network as a mechanism to round the variables while trying to respect the pairwise correlations. This is somewhat akin to how graph neural networks are used for node classification except that in a Hopfield network, the underlying network is the complete graph. Our pipeline is illustrated in Figure 1.

In the following subsections, we describe simple functions for each of the three problems and show that they yield good empirical results. The particular hand-designed functions in this section are not the main point. The primary objective is to show that simple functions suffice and, therefore, an appropriate function can be efficiently learned from data. This is particularly useful for new problems where it may not be easy to hand-design good functions. We discuss how the function can be learnt from data in the next section.

[1] The solutions are often rotation invariant i.e., applying the same rotation to all the vectors in a solution yields an equally good solution to the SDP.

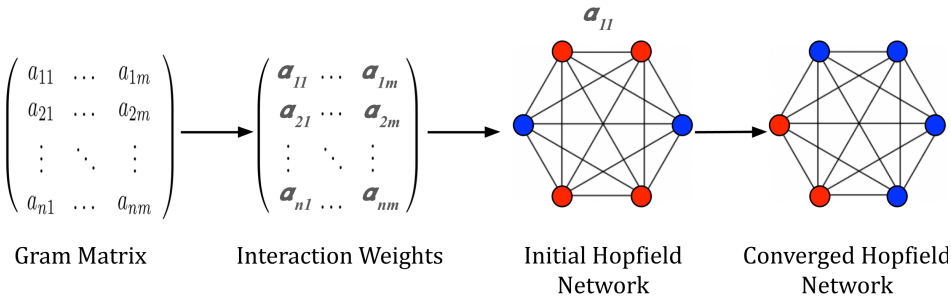

Figure 1: A schematic diagram showing our framework.

Remark. Note that updates in the Hopfield networks with all pairwise edges takes $\Theta(n^2)$ time in every round. This can be improved by doing approximate computations using matrix sketching [38]. We do not discuss this in this paper since our focus is on evaluating the quality of solutions obtained using our framework. For the problem sizes we consider, this is not the bottleneck since SDP computations dominate the running time. However, for scaling to larger instances, matrix sketching-based optimizations along with fast approximate SDP solvers will be necessary.

4.1 Hopfield Networks for Max-Cut.

In this problem, we would like each neuron of the Hopfield network to output a number in the range $[0, 1]$ from which we create a cut by taking one side to be the vertices whose corresponding neurons have output at most 0.5 and taking the other side to be the remaining vertices. Given this, we use the sigmoid function as the activation function for the neurons. The outputs of the Hopfield network are rounded to the nearest binary output to obtain a binary solution. Given the symmetry between the sides in the cut (flipping the sides yields the same cut), we set the bias for each neuron to 0.

In the formulation of the SDP for Max-Cut, the quantity $c_{uv} := (1 - \boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_v)/2$ is the coefficient of w_{uv} in the objective function. If c_{uv} is large, we would like u and v to be separated by the cut, and otherwise, we would like them to be on the same side of the cut. It is thus natural to use a decreasing function of c_{uv}, i.e., an increasing function of $\boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_v$ as the interaction between the neurons corresponding to the vertices u and v. One obvious option is to use $\boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_v$ as the interaction. Another option motivated by the Goemans-Williamson algorithm [16] is to use $2p_{uv} - 1$ as the interaction where $p_{uv} = \arccos(\boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_v)/\pi$ is the probability that $\boldsymbol{\sigma}_u$ and $\boldsymbol{\sigma}_v$ are separated by a random hyperplane through the origin. The reason for choosing $2p_{uv} - 1$ as the interaction is that we want the range to be $[-1, +1]$. The initial outputs of the neurons are chosen from $[0, 1]$ uniformly at random. Denoting the output of the neuron corresponding to vertex u by z_u, we update z_u to $A\left(\sum_{v \neq u} \boldsymbol{a}_{uv} z_v\right)$ where $\boldsymbol{a}_{uv}$ denotes the interaction between the neurons corresponding to vertices u and v and A is activation function - in this case, the sigmoid function. As mentioned before, we repeat the updates until the outputs converge or the number of rounds of updates exceeds a threshold.

Experimental results. We compared the performance of our approach with that of the Goemans-Williamson algorithm on SF-295 [39], a collection of graphs representing small molecules recording cancer screening (4026 instances with 31 nodes and 33 edges on average), and Twitter Snap [40] graph datasets (97 instances with 130 nodes and 1421 edges on average) along with a custom dataset. The custom dataset consists of 1000 Erdős-Rényi random graphs, each containing 128 nodes with a probability of edge existing between a node pair of 0.05. In addition, we inserted a Max-Cut between a random, even partition of nodes in the graph. To do this, we consider all node pairs with a node on either side of the partition and insert an edge between them with a probability of 0.15. We carefully remove random edges from both partitions to ensure that the degree of nodes has a similar distribution after the planted cuts.

Table 1 presents the result of this comparison. For the Goemans-Williamson algorithm (referred to as the GW Algorithm in the table), we use 5 random planes to generate cuts and take the best result from the 5 cuts. We also run our Hopfield Network based approach 5 times and take the best cut. Table 1 shows the mean optimality ratio (i.e., size of generated cut/optimal cut) and the standard deviation over all graphs in the different datasets. Throughout these experiments, the Hopfield network always

converged to a stable state in a handful of iterations and didn't require to be terminated after a fixed number of steps. As can be seen in Table 1, our approach based on the Hopfield network produces cuts that are significantly better than generating random cuts for each graph, implying that Hopfield networks can decode the cut information contained in SDP vectors. Surprisingly, we find that it even returns better cuts than the classical Goemans-Williamson approximation algorithm that has provable guarantees on the cut size. Outperforming the well-studied Goemans-Williamson approximation algorithm on various graph datasets clearly outlines the potential of our Hopfield network-based approach. We also show in Table 1 that the mean time to pre-process the Hopfield network edges, computing the weights (excluding the SDP computation time) and for the Hopfield network to converge to a solution is very small (less than a second).

In addition, we investigate how our framework performs as the size of the instances increases. To do this, we generated an additional 25 Erdős-Rényi graphs in the same manner as before, except that we generated them for sizes varying from 16 nodes to 512 nodes. We refer to them as our custom-cut instances. The same experiment as before was carried out with results displayed in Table 2. Once again, we observe that the cuts produced by the Hopfield network have a better mean optimality ratio compared to the Goemans-Williamson approximation algorithm. We also compare it with taking a random cut and note that random cuts have a considerably poor mean optimality ratio and the usage of SDP vectors is indeed crucial to obtaining good cuts on these problem instances.

	SF-295	Twitter	Custom Cut
Hopfield Network	0.998($\pm$0.005)	0.993($\pm$0.006)	0.998($\pm$0.002)
GW Algorithm	0.943($\pm$0.073)	0.937(0.041)	0.897($\pm$0.051)
Random	0.621($\pm$0.071)	0.797($\pm$0.044)	0.686($\pm$0.019)
SDP Runtime	0.745($\pm$2.173)	9.001($\pm$8.709)	3.961($\pm$0.823)
Hopfield Network Runtime	0.014($\pm$0.015)	0.256($\pm$0.192)	0.203($\pm$0.006)
GW Algorithm Runtime	0.0002($\pm$0.0001)	0.002($\pm$0.001)	0.0017($\pm$0.0028)

Table 1: Mean optimality ratio for Max-Cut and runtimes in seconds

Cut # Nodes	Hopfield	Hopfield Runtime	GW	Random
16	0.998($\pm$0.006)	0.002($\pm$0.001)	0.951($\pm$0.045)	0.782($\pm$0.060)
32	0.998($\pm$0.005)	0.008($\pm$0.0006)	0.935($\pm$0.047)	0.679($\pm$0.065)
64	0.993($\pm$0.007)	0.036(0.001)	0.908($\pm$0.049)	0.678($\pm$0.029)
128	0.998($\pm$0.002)	0.203($\pm$0.006)	0.894($\pm$0.047)	0.683($\pm$0.020)
256	1.000($\pm$0.000)	0.587($\pm$0.032)	0.923($\pm$0.060)	0.674($\pm$0.009)
512	1.000($\pm$0.000)	2.356($\pm$0.112)	0.918($\pm$0.098)	0.672($\pm$0.003)

Table 2: Mean optimality ratio for varying size Erdős-Rényi Max-Cut instances

Table 2 also shows that for these graph sizes, the running time remains quite small (around 2.4 seconds for 512 node Erdős-Rényi graphs). The running time scales roughly quadratically with the input size (Around 1024 times increase as the number of nodes increases by a multiplicative factor of 32 from 16 to 512). This is in line with what we would expect theoretically – Hopfield networks with all pairwise edges take $\Theta(n^2)$ time in every round. As discussed before, this can be improved by doing approximate computations using matrix sketching [38].

4.2 Hopfield Networks for Max-Clique and Graph Coloring.

We present the details of the Hopfield networks for the Max-Clique and Graph coloring problems in Appendix B and C, respectively. Our experimental results on various benchmark instances show that the SDP rounding based on Hopfield network finds near-optimal solutions in very little runtime. Our approach compares favourably to the Erdős Goes Neural [41] GNN architecture as well as a greedy heuristic for the Max-Clique problem.

5 Learning the Hopfield Network parameters

In this section, we show that instead of hand-crafting the weights of the Hopfield network, we can express the interaction between a pair of neurons as a parameterized function of the corresponding

Gram matrix entry and learn the parameters from the data (as illustrated in Figure 2). For instance, we can train a neural network that learns this function separately for each combinatorial optimization problem. We experimentally demonstrate the efficacy of the idea for the Max-Cut problem. A similar approach can be used for other problems.

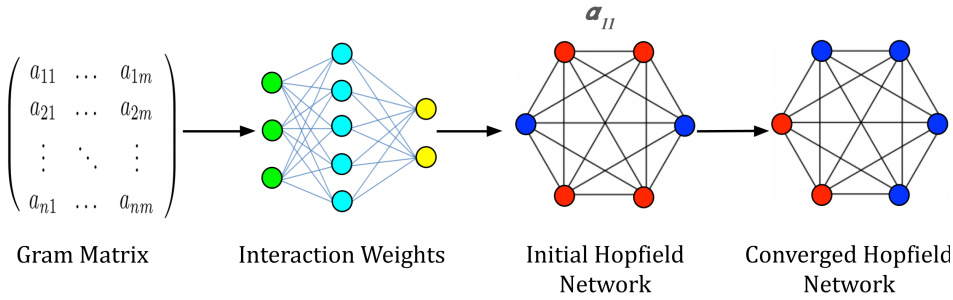

Figure 2: A schematic diagram showing our framework where a small neural network is used to learn the interaction weights of the Hopfield network.

While the input graph $G = (V, E)$ can be sparse, the Hopfield network works on a complete graph $G' = (V, E')$ with $E \subseteq E'$. We need to learn the function governing the interaction weights $\boldsymbol{a}_{uv}$ for all edges (u, v) in the Hopfield network. The input to the neural network that learns the weight function consists of the corresponding Gram matrix value X_{uv} together with some polynomial terms X_{uv}^2, X_{uv}^3 etc. In addition, we also input an indicator variable corresponding to whether or not an edge in the Hopfield network is in the input graph or not.

We train a small, dense neural network to compute an interaction weight for the Hopfield network edges. For this, we need to design an appropriate loss function. For each vertex u, let z_u be the output corresponding to u. If the $z_u's$ were assumed to be in $\{0, 1\}$, finding the max-cut is equivalent to maximizing $\sum_{(u,v)\in E} w_{uv}(z_u - z_v)^2$. Accordingly, we use the negative of the above sum as the loss function. Note that here we are using the "raw" outputs of the nodes in the Hopfield network and not rounding them since we want the loss function to be differentiable.

Remark. The max-cut problem is particularly simple since all solutions are feasible. In problems with constraints, one needs to construct the loss function carefully. This is in general problem dependent but a generic method is to use a Lagrangian relaxation. For instance, in the max-clique problem, the loss function that we used is $\sum_{v\in V} z_v - \lambda \sum_{(u,v)\notin E} z_u z_v$ where λ is a parameter which in our experiments was set to 1. We backpropagate the loss through each iteration of the Hopfield network - this is similar to backpropagation through time in Recurrent Neural Networks - yielding the gradient with respect to each edge weight in our Hopfield network. Since these edge weights are themselves the output of a neural network, we further backpropagate through that network to obtain the gradient of the loss with respect to the parameters in that network.

Experimental Results. To evaluate this approach, we consider the Max-Cut Problem and select a subset of 1000 graphs from the SF-295 dataset. We perform a 70/30 train/test split on this dataset and train a small neural network for 50 epochs with a learning rate of 10^{-3} using the ADAM optimiser with the default parameters in Pytorch. The model consists of three dense layers, with a single hidden layer with a width of 6 neurons and $tanh$ activation functions.

Similarly, for the Max-Clique problem, we perform a 70/30 train/test split on a subset of 1000 graphs from the IMDB-Binary graph dataset and train a small neural network for 50 epochs. The only difference is that the learning rate of 10^{-4} is used in ADAM optimiser instead of 10^{-3}. The learning rate needs to be carefully tuned as there is a trade-off between the number of instances for which the Hopfield network returns cliques of sub-optimal size and the number of instances where the Hopfield network produces a superset of the optimal clique (which itself is not a clique and therefore an invalid solution). A large learning rate tends to cause the optimizer to jump between the two extremes. The hidden layer, in this case, had 10 neurons.

Figure 3 shows that for both these problems, after only a few epochs, the optimality ratio associated with the learnt weight function on the Hopfield network edges reaches above 0.9. Thus, a good weight function for this instance distribution could be learnt in only a few iterations.

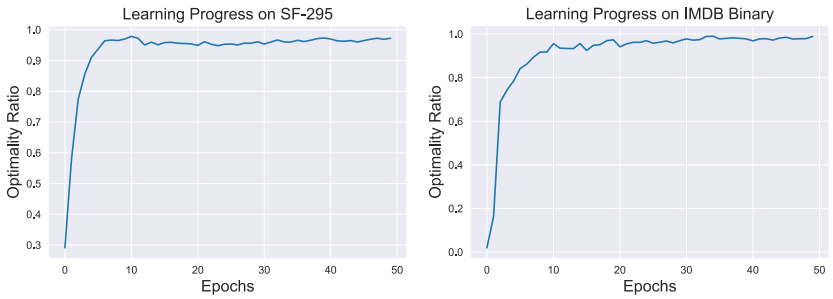

Figure 3: Optimality ratio associated with learning the weight function of the Hopfield network using SDP during training. The left plot shows the progress for the Max-Cut problem on the SF-295 training dataset, and the right plot shows the progress for the Max-Clique problem on the IMDB-Binary training dataset.

After training the network, we compute the mean optimality ratio and standard deviation in a similar manner as our experiments in Section 4.1. Even with learnt edge weights (instead of manually selected edge weight in Section 4.1) using a small neural network, for Max-Cut our approach returned a cut with the mean optimality ratio of $0.979(\pm 0.105)$ on the 300 test graphs of the SF-295 dataset. For Max-Clique, our approach returned a clique with a mean optimality ratio of $1.0(\pm 0.0)$ for valid cliques, and for only two instances, the network failed to produce a valid clique. This shows that the neural network converged towards learning a function that produced edge weights with comparable performance to our manually selected edge weight function. This implies that the design of the transformation function can be automated through the use of a small dense neural network.

In Appendix D, we show that our learning models for Max-Cut and Max-Clique generalizes well across distributions and larger size instances. Our models achieves near optimal solutions on graph classes that are very different in size and density from the classes on which they were trained.

We observed that the learnt model for Max-Clique is quite different from the carefully selected function presented in Appendix B. Specifically, it puts less emphasis on the connection to the dummy vertex. The fact that the learning technique has converged to a different function for Max-Clique on this graph class and that this function was learnt from a space that generalizes the carefully designed weight function suggests that the learnt function minimizes the loss on the training instances more than the handcrafted weight function.

6 Using approximate Low Rank SDP solutions

A potential criticism of our approach is that it relies on a computationally expensive step of calculating SDP vectors. We ran experiments to check the sensitivity of the rounding algorithm to the accuracy of the SDP solutions and found that the algorithm is quite robust to noise. The results are presented in Appendix E. Motivated by these results, we tried our approach on large instances of Max-Cut in which instead of computing the optimal SDP solution (which would be prohibitive), we used low-rank SDP solutions obtained using the mixing method [42] - a simple and fast algorithm based on coordinate descent. Since for such large instances, we cannot compute the optimal solution in a reasonable amount of time, we compare our results with Breakout Local Search (BLS) [43], one of the top heuristics for this problem. The rank of the solutions is a tunable parameter that provides a trade-off between the quality of the solution and the speed of computation. We used $\sqrt{2n}$ as the rank in line with the recommendation by Wang et al. [42]. We use the default hyperparameters for BLS from Benlic and Hao [43] since they use the same benchmarking dataset and had used those hyperparameters for that dataset.

Our Max-Cut instances were from the Gset graph dataset [44], with graphs having between 800 and 10,000 nodes. Table 3 shows that our approach based on Hopfield Networks obtains solutions that are nearly as good as those obtained by BLS [43] but is significantly faster, especially for large instances. For instance, for the graph G70 with 10,000 nodes, our approach finishes in around a minute with a Max-Cut size that is only 0.6% less than that of BLS, whereas BLS takes over 3 hours.

To avoid the $O(n^2)$ storage requirement of the Gram matrix, we can approximately store the entries via a low-rank approximation. We know that there exist vectors $\boldsymbol{v}_1, ..., \boldsymbol{v}_n \in R^n$ s.t. the entry G_{ij} of the Gram matrix is $\boldsymbol{v}_i \cdot \boldsymbol{v}_j$. Given these vectors, we could use dimension reduction to reduce their dimension to $O(\log n)$ while approximately preserving dot products. We could also use SDP solvers that can return approximate solutions where the vectors are of dimension $d \ll n$.

	$\|V\|$	SDP Time	Hopfield Obj.	Hopfield Time	BLS Obj.	BLS Time
G1	800	1.601	11450	1.553	11624	13
G14	800	0.641	2970	0.91	3064	119
G15	800	0.610	2977	0.92	3050	43
G22	2000	2.203	13000	4.605	13359	560
G23	2000	2.376	12973	4.698	13354	278
G24	2000	2.161	13024	4.669	13337	311
G35	2000	1.641	7403	4.276	7684	442
G36	2000	1.561	7401	4.254	7677	604
G37	2000	1.582	7407	4.21	7689	444
G45	1000	1.016	6482	1.486	6554	104
G53	1000	0.829	3732	1.308	3850	117
G54	1000	0.813	3740	1.285	3852	131
G55	5000	3.082	9903	4.34	10294	842
G58	5000	4.317	18539	22.577	19263	1354
G60	7000	4.471	13631	40.916	14176	2822
G70	10000	4.572	9485	61.805	9541	11365

Table 3: Comparison of the Max-Cut size and runtimes in seconds of the low-rank SDP solution rounded using our Hopfield network approach with Breakout Local Search (referred BLS)

7 Discussion

We have shown that a simple Hopfield network with appropriate interaction weights based on SDP embeddings can obtain near-optimal solutions for practical instances of the classical combinatorial optimization problems of Max-Cut, Max-Clique and Graph Colouring obtained from various benchmark datasets. Furthermore, we show that the appropriate problem-dependent interaction weights can be learnt efficiently using a small neural network.

Our approach has two parts – computing a low-rank SDP solution and then rounding the solution using a Hopfield network. Yu et al. [28] have recently shown that a low-rank SDP vector can be computed using a graph neural network (GNN). Similarly, for the second part, while we have used a Hopfield network, a more scalable and general approach would be to use GNNs initialized with the SDP solution vectors of a fixed rank. The final embedding obtained by the GNN can then be processed by another neural network to output a decision for each node (e.g., whether it is part of the solution, the colour of the node, etc.). While this approach sounds natural and appealing, it does not seem easy to make it work in practice. An interesting open problem is to design a flexible architecture based on GNNs that can integrate existing algorithmic tools like SDPs.

Code availability: Our code is available at https://anonymous.4open.science/r/SDP-Hopfield-645D/

Acknowledgement: This publication has emanated from research supported in part by a grant from Science Foundation Ireland under Grant number 18/CRT/6183. For the purpose of Open Access, the author has applied a CC BY public copyright licence to any author accepted manuscript version arising from this submission.

References

[1] Bernd Gärtner and Jirí Matousek. *Approximation Algorithms and Semidefinite Programming*. Springer, 2013. ISBN 978-3-642-22014-2. doi: 10.1007/978-3-642-22015-9. 1, 3, 16

[2] Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001. ISBN 978-3-540-65367-7. URL http://www.springer.com/computer/theoretical+computer+science/book/978-3-540-65367-7.

[3] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011. 1, 3

[4] D. R. Karger, R. Motwani, and M. Sudan. Approximate graph coloring by semidefinite programming. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 2–13. IEEE Computer Society, 1994. 1, 3

[5] A. L. Blum. *Algorithms for approximate graph coloring*. Phd thesis, Cambridge, MA, USA, 1992.

[6] E. Chlamtac. Approximation algorithms using hierarchies of semidefinite programming relaxations. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 691–701. IEEE Computer Society, 2007.

[7] V. Guruswami and S. Khanna. On the hardness of 4-coloring a 3-colorable graph. In *IEEE Conference on Computational Complexity*, pages 188—-197, 2000.

[8] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, 1972.

[9] S. Khanna, N. Linial, and S. Safra. On the hardness of approximating the chromatic number. In *Israel Symposium on Theory of Computing Systems*, pages 250–260, 1993.

[10] A. Wigderson. Improving the performance guarantee for approximate graph coloring. *Journal of ACM*, 30(4):729–735, 1983. 1, 3

[11] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982. 1, 3

[12] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2), 2021. 2, 4

[13] Mattia Silvestri, Michele Lombardi, and Michela Milano. Injecting domain knowledge in neural networks: A controlled experiment on a constrained problem. In *Proceedings of the 18th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research CPAIOR*, volume 12735 of *Lecture Notes in Computer Science*, pages 266–282. Springer, 2021. 2

[14] Alessandro Daniele, Emile van Krieken, Luciano Serafini, and Frank van Harmelen. Refining neural network predictions using background knowledge. *Machine Learning*, 112(9):3293–3331, 2023. 2

[15] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Review*, 38(1): 49–95, 1996. 3

[16] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42 (6):1115–1145, 1995. 3, 5, 13, 16

[17] Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. Optimal inapproximability results for max-cut and other 2-variable csps? *SIAM Journal on Computing*, 37(1):319–357, 2007. 3

[18] Nikhil Bansal. On a generalization of iterated and randomized rounding. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, page 1125–1135. Association for Computing Machinery, 2019. ISBN 9781450367059. 3

[19] David G. Harris. Dependent rounding with strong negative-correlation, and scheduling on unrelated machines to minimize completion time. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2275–2304. SIAM, 2024.

[20] Adi Avidor and Uri Zwick. Rounding two and three dimensional solutions of the SDP relaxation of MAX CUT. In *Proceedings of Approximation, Randomization and Combinatorial Optimization, Algorithms and Techniques (Approx-RANDOM)*, volume 3624 of *Lecture Notes in Computer Science*, pages 14–25. Springer, 2005.

[21] Boaz Barak, Prasad Raghavendra, and David Steurer. Rounding semidefinite programming hierarchies via global correlation. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS*, pages 472–481. IEEE Computer Society, 2011.

[22] Heng Yang, Ling Liang, Luca Carlone, and Kim-Chuan Toh. An inexact projected gradient method with rounding and lifting by nonlinear programming for solving rank-one semidefinite relaxation of polynomial optimization. *Math. Program.*, 201(1):409–472, 2023.

[23] Kiarash Shaloudegi, András György, Csaba Szepesvári, and Wilsun Xu. SDP relaxation with randomized rounding for energy disaggregation. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems*, pages 4979–4987, 2016. 3

[24] Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Velickovic. Combinatorial optimization and reasoning with graph neural networks. *J. Mach. Learn. Res.*, 24:130:1–130:61, 2023. 4

[25] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers and Operations Research*, 134:105400, 2021. 4

[26] Luís C. Lamb, Artur S. d'Avila Garcez, Marco Gori, Marcelo O. R. Prates, Pedro H. C. Avelar, and Moshe Y. Vardi. Graph neural networks meet neural-symbolic computing: A survey and perspective. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI*, pages 4877–4884, 2020. 4

[27] Yun Peng, Byron Choi, and Jianliang Xu. Graph learning for combinatorial optimization: A survey of state-of-the-art. *Data Sci. Eng.*, 6(2):119–141, 2021. 4

[28] Morris Yau, Eric Lu, Nikolaos Karalias, Jessica Xu, and Stefanie Jegelka. Are graph neural networks optimal approximation algorithms? *arXiv (CoRR)*, abs/2310.00526, 2023. 4, 9, 16

[29] Tamás Kriváchy, Yu Cai, Joseph Bowles, Daniel Cavalcanti, and Nicolas Brunner. High-speed batch processing of semidefinite programs with feedforward neural networks. *New Journal of Physics*, 23, 10 2021.

[30] Po-Wei Wang, Priya L. Donti, Bryan Wilder, and J. Zico Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *Proceedings of the 36th International Conference on Machine Learning, ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 6545–6554, 2019.

[31] Radu Baltean-Lugojan, Pierre Bonami, Ruth Misener, and Andrea Tramontani. Scoring positive semidefinite cutting planes for quadratic optimization via trained neural networks. In *Preprint: https://optimization-online.org/2018/11/6943/*, 2019.

[32] Long Cheng, Zeng-Guang Hou, Noriyasu Homma, Min Tan, and Madam M. Gupta. Solving convex optimization problems using recurrent neural networks in finite time. In *2009 International Joint Conference on Neural Networks*, pages 538–543, 2009. 4

[33] Liming Yang, Laisheng Wang, Yongping Gao, Qun Sun, and Tengyang Zhao. A convex relaxation framework for a class of semi-supervised learning methods and its application in pattern recognition. *Engineering Applications of Artificial Intelligence*, 35:335–344, 2014. ISSN 0952-1976. 4

[34] Veronica Piccialli, Anna Russo Russo, and Antonio M. Sudoso. An exact algorithm for semi-supervised minimum sum-of-squares clustering. *Computers & Operations Research*, 147: 105958, 2022. 4

[35] Chirag Pabbaraju, Po-Wei Wang, and J. Zico Kolter. Efficient semidefinite-programming-based inference for binary and multi-class mrfs. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Proceedings of the 33rd annual conference on Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 4

[36] Po-Wei Wang and J. Zico Kolter. Community detection using fast low-cardinality semidefinite programming. In *Proceedings of the 33rd annual conference on Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 4

[37] Vinay Jethava, Anders Martinsson, Chiranjib Bhattacharyya, and Devdatt P. Dubhashi. The lovasz theta function, svms and finding large dense subgraphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1169–1177, 2012. 4

[38] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast monte carlo algorithms for matrices I: approximating matrix multiplication. *SIAM J. Comput.*, 36(1):132–157, 2006. 5, 6

[39] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S. Yu. Mining significant graph patterns by leap search. In *SIGMOD Conference*, 2008. 5

[40] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014. 5, 14

[41] Nikolaos Karalias and Andreas Loukas. Erdos goes neural: an unsupervised learning framework for combinatorial optimization on graphs. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 6, 14, 15

[42] Po-Wei Wang, Wei-Cheng Chang, and J. Zico Kolter. The mixing method: low-rank coordinate descent for semidefinite programming with diagonal constraints. *CoRR*, 2018. URL `https://arxiv.org/abs/1706.00476`. 8

[43] Una Benlic and Jin-Kao Hao. Breakout local search for the max-cutproblem. *Engineering Applications of Artificial Intelligence*, 26:1162–1173, 03 2013. doi: 10.1016/j.engappai.2012.09.001. 8

[44] J. & Krevl A. Leskovec. Gset dataset. https://web.stanford.edu/ yyye/yyye/Gset/, 2003. 8

[45] Pinar Yanardag and S. V. N. Vishwanathan. Deep graph kernels. *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015. 14

[46] Wei Li, Ruxuan Li, Yuzhe Ma, Siu On Chan, David Z. Pan, and Bei Yu. Rethinking graph neural networks for the graph coloring problem. *CoRR*, abs/2208.06975, 2022. 15

[47] Alp Yurtsever, Joel A. Tropp, Olivier Fercoq, Madeleine Udell, and Volkan Cevher. Scalable semidefinite programming. *SIAM Journal on Mathematics of Data Science*, 3(1):171–200, 2021. 16

[48] Anirudha Majumdar, Georgina Hall, and Amir Ali Ahmadi. Recent scalability improvements for semidefinite programming with applications in machine learning, control, and robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 3(1):331–360, 2020. 16

[49] Valentin Durante, George Katsirelos, and Thomas Schiex. Efficient low rank convex bounds for pairwise discrete graphical models. In *International Conference on Machine Learning, ICML*, volume 162 of *Proceedings of Machine Learning Research*, pages 5726–5741, 2022. 16

Appendix

A Problems Considered and their SDP

In this appendix, we describe the problems considered in this paper and the ILP formulations that we use for SDP relaxation.

Max-Cut: In the Max-Cut problem, we are given a weighted graph $G = (V, E)$ in which the edge $(u, v) \in E$ has weight w_{uv} and the goal is to partition the vertex set V into two sets $S \subset V$ and $V \setminus S$ so that we maximize the number of edges in E which have one end-point in both sets. This problem is NP-hard and we can use the following integer programming formulation for this problem.

$$\max \sum_{u,v \in E} w_{uv} \left(\frac{1 - x_u \cdot x_v}{2} \right) \text{ s.t. } x_u \in \{-1, 1\} \tag{1}$$

Here, $x_u = 1$ if $u \in S$ and $x_u = -1$ otherwise. The value of the cut equals $\sum_{u \in S, v \in V/S} w_{uv} = \sum_{u,v \in E} w_{uv} \left(\frac{1 - x_u \cdot x_v}{2} \right)$. The integer constraint $x_u \in \{-1, 1\}$ can also be written as $x_u^2 = 1$.

The best approximation ratio for this problem is obtained by a celebrated algorithm due to Goemans and Williamson [16] using semi-definite programming. The SDP relaxation used in their paper is as follows:

$$\max \sum_{(u,v) \in E} w_{uv} \left(\frac{1 - \boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_u}{2} \right) \text{ s.t. } \boldsymbol{\sigma}_u \in \mathbb{R}^n \text{ and } \boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_u = 1 \ \forall \, u \in V. \tag{2}$$

$$\tag{3}$$

Here $\boldsymbol{\sigma}_v$ is an n-dimensional unit vector associated with the vertex v. Note that the quantity we are maximizing depends only on the inner products of the vectors and is not affected by rotations in $\mathbb{R}^n$. In this paper, we only use the pairwise dot products of the vectors associated with the vertices and do not really need the vectors $\boldsymbol{\sigma}_v$.

Max-Clique: In the Max-Clique problem, we are given a graph $G = (V, E)$ and the goal is to find the largest subset $S \subseteq V$ which forms a clique i.e., each pair of vertices in S are connected by an edge in the graph. The following integer programming formulation can be used for solving this problem:

$$\max \sum_{v \in V} x_v \tag{4}$$

$$\text{s.t. } x_u \cdot x_v = 0 \quad \forall \, (u, v) \notin E \tag{5}$$

$$x_v \in \{0, 1\} \quad \forall v \in V \tag{6}$$

$$\tag{7}$$

Here is the SDP relaxation we use for this problem:

$$\max \sum_{v \in V} \boldsymbol{\sigma}_v \cdot \boldsymbol{\sigma}_v \tag{8}$$

$$\text{s.t. } \boldsymbol{\sigma}_0 \cdot \boldsymbol{\sigma}_0 = 1, \tag{9}$$

$$\boldsymbol{\sigma}_v \cdot \boldsymbol{\sigma}_v = \boldsymbol{\sigma}_v \cdot \boldsymbol{\sigma}_0 \tag{10}$$

$$\boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_v = 0 \, \forall \, (u, v) \notin E \tag{11}$$

As in the previous problem, $\boldsymbol{\sigma}_v$ is an n-dimensional vector (not necessarily of length 1) associated with the vertex v. In addition, we have a vector $\boldsymbol{\sigma}_0$.

Graph Coloring: In the graph colouring problem, we are given a graph $G = (V, E)$, and the goal is to colour the vertices with the minimum number of distinct colours so that any pair of vertices joined

by an edge $e \in E$ have different colours. The ILP formulation for the problem is as follows:

$$\begin{aligned}
\min \sum_{k=1}^{n} y_k & \\
\text{s.t. } \sum_{k=1}^{n} x_{uk} &= 1 \ \ \forall u \in V \\
x_{uk} &\leq y_k \ \ \forall\, k = 1, \ldots, n \ \ \text{and } \forall u \in V \\
x_{uk} + x_{vk} &\leq 1 \ \ \forall\, k = 1, \ldots, n \ \ \text{and } \forall\, (u, v) \in E \\
0 \leq x_{uk}, y_k &\leq 1 \ \ \forall\, k = 1, \ldots, n \ \ \text{and } \forall u \in V \\
x_{uk}, y_k &\in \mathbb{Z}
\end{aligned}$$

Here is the SDP relaxation we use:

$$\begin{aligned}
\min \ & t \\
\text{s.t. } & \boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_v \leq t \ \forall\, (u, v) \in E \\
& \boldsymbol{\sigma}_v \cdot \boldsymbol{\sigma}_v = 1 \ \forall\, v \in V \\
& \boldsymbol{\sigma}_v \in \mathbb{R}^n \ \forall\, v \in V
\end{aligned}$$

Here too, for every vertex $v \in V$, we have an associated vector $\boldsymbol{\sigma}_v$ of unit length.

B Hopfield Network for Max-Clique

For this problem, we treat a neuron's output as the probability that it belongs to the Max-Clique. As the SDP formulation of the Max-Clique problem indicates, for any vertex v, $\boldsymbol{\sigma}_v \cdot \boldsymbol{\sigma}_v$ can be thought of as the "probability that v belongs to the clique". We therefore set $\boldsymbol{\sigma}_v \cdot \boldsymbol{\sigma}_v$ as the bias for the neuron corresponding to v. In addition, since we want to avoid picking non-adjacent vertices in the clique, we set the interaction between the neurons corresponding to the vertices u and v as -1 if u and v are non-adjacent and 1 otherwise. One difference with the Max-Cut problem is that whereas any cut is a feasible solution to the Max-Cut problem, arbitrary subsets of the vertices do not generally form a clique and, therefore, are not feasible solutions. Unlike for Max-Cut, we initialize the outputs for each of the neurons to 0, so that after the first round of updates, the neuron corresponding to vertex v has a pre-activation output of $\boldsymbol{\sigma}_v \cdot \boldsymbol{\sigma}_v$.

Experimental results. For this architecture, we tested its performance on the IMDB-Binary (1000 instances with 19 nodes and 96 edges on average), Google Colab [45] (5000 instances with 74 nodes and 2457 edges on average) and Twitter [40] graph datasets along with a Custom Dataset consisting of 1000 Erdős-Rényi graphs with an edge probability of 0.5 and a planted hidden clique that is twice the size of the Max-Clique within the graph. We carefully ensure that we have a similar degree distribution between nodes within the planted clique and nodes outside it. We compare the performance with the Erdős Goes Neural [41] fast GNN architecture as well as a greedy heuristic. The greedy heuristic builds a clique by starting with the highest-degree node in a clique and iteratively adding the next highest-degree node that is connected to all current members of the clique until no further nodes can be added. Table 4 shows that our Hopfield network-based approach returns optimal or near-optimal maximum cliques on these graph collections. The mean optimality ratio of our approach is considerably better than the Erdős Goes Neural GNN technique and the greedy heuristic.

The "Erdoes Goes Neural" approach always produces feasible solutions. Our approach based on Hopfield networks can sometimes result in infeasible solutions. However, as shown in Table 4, even for our approach, the frequency of producing infeasible solutions is very small (0.4% on IMDB-Binary, 0.3% on Collab, 0% on Twitter instances, and 0. 3% on Custom clique instances).

Note that for this problem, one can always return a subset of the vertices in the returned solution that form a maximal clique to ensure feasibility. Similarly, in the coloring problem, one can modify the algorithm so that it colors vertices one by one and we only allow a set of colors to a vertex which are distinct from already colored neighbors. For many problems, such postprocessing of the solution is possible.

	IMDB-Binary	Collab	Twitter	Custom Clique
Hopfield Network	0.993(±0.068)	0.996(±0.058)	0.978(±0.115)	0.994 (±0.062)
Erdős Goes Neural [41]	1.0(±0.0)	0.982(±0.063)	0.924(±0.133)	0.810(±0.226)
Greedy Heuristic	0.954(±0.133)	0.886(±0.195)	0.848(±0.154)	0.740(±0.238)
% of Invalid Cliques	0.4%	0.3%	0%	0.3%
SDP Runtime	0.194(±0.258)	2.589(±6.046)	21.219(±38.190)	4.385(±0.536)
Hopfield Runtime	0.0006(±0.0007)	0.006(±0.0130)	0.0132(±0.0094)	0.0117(±0.0007)

Table 4: Mean optimality ratio and the standard deviation for Max-Clique over graphs in different collections, runtimes in seconds and percentage of infeasible cliques found.

C Hopfield Network for Graph Coloring

Graph Name	$\chi(G)$	Hopfield $\chi(G)$	SDP Runtime	Hopfield Runtime
1-Insertions-4	5	5	3.968	0.043
2-Insertions-4	5	5	63.319	0.156
Anna	11	11	3.878	0.576
David	11	11	1.624	0.354
Games120	9	9	4.563	0.462
Huck	11	11	1.121	0.279
Mugg88-1	4	4	28.658	0.067
Myciel5	6	6	1.059	0.057
Myciel6	7	7	5.53	0.319
Myciel7	8	8	46.526	0.817
Queen5-5	5	5	0.303	0.071
Queen6-6	7	8	1.197	0.016
Queen7-7	7	9	1.191	0.032
Queen8-8	9	10	1.975	0.063
Queen8-12	12	12	3.590	0.113
Queen9-9	10	11	2.628	0.076
Queen11-11	11	14	6.931	0.307
Queen13-13	13	15	20.522	0.351

Table 5: Optimal chromatic number and chromatic number returned by Hopfield network for several graphs and runtimes in seconds.

For the graph colouring problem, instead of directly minimizing the number of colours required via a Hopfield network, we fix a parameter k and use a Hopfield network to try to find a k-coloring. If the network is not able to find such a colouring after several attempts, we increase k and retry with a larger k. For the remainder of this subsection, we assume that k is fixed, and our goal is to find a k-coloring. In this case, we want the output of each layer to be a probability vector whose i^{th} entry corresponds to the probability that the vertex has colour i. This is done by having k neurons corresponding to a vertex v - one for each of the colours and using the softmax function as the activation function. Let t^* be the objective value of the SDP solution. By definition, if vertices u and v are neighbours in the graph, then $\boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_v \leq t^*$. In this case, we do not want u and v to have the same colour and to ensure this; we set the interaction between u_i and v_i for every $i \in \{1, \cdots, k\}$ to $-2n$, where n is the number of vertices in the graph. Otherwise, if $\boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_v > t^*$, we set the interaction between u_i and v_i to $\boldsymbol{\sigma}_u \cdot \boldsymbol{\sigma}_u - t^*$ so that the larger the quantity, the more we "encourage" u and v to have the same colour. We choose the initial output of each neuron to be the uniform distribution over the k colours. Once the output probability distribution (on the k colours) for each vertex is fixed, we choose a colour for the vertex from this probability distribution.

Experimental results. To test the schema, we tested the performance on several graphs from the COLOR02/03/04 dataset[2]. We selected a subset of graphs similar to that used in a recent paper on GNNs for graph colouring [46]. Table 5 shows that our approach finds the optimal or near-optimal chromatic number on these graphs.

[2] https://mat.tepper.cmu.edu/COLOR02/

D Generalization of learnt Hopfield network to different graph classes

	SF-295 Test	Twitter	Custom Cut
Hopfield Network	0.979($\pm$0.105)	0.986($\pm$0.013)	0.998($\pm$0.002)
Runtime	0.443($\pm$0.478)	9.512($\pm$7.546)	7.364($\pm$0.306)

Table 6: Mean optimality ratio and runtimes in seconds for Max-Cut with Learnt Models

	IMDB-Binary Test	Collab	Twitter	Custom Clique
% of Invalid Cliques	2%	2.3%	35.1%	17%
Hopfield Network	1.0($\pm$0.000)	0.996($\pm$0.004)	0.991($\pm$0.024)	1.0($\pm$0.000)
Runtime	0.222($\pm$0.298)	4.940($\pm$11.486)	10.291($\pm$8.122)	8.101($\pm$0.083)

Table 7: Percentage of valid cliques found, mean optimality ratio of valid cliques and runtimes in seconds for Max-Clique with learnt models.

To test the generalization ability of our learning models to learn the interaction weights on Hopfield network, we tested our neural network performance of our learnt model for Max-Cut on the Twitter dataset and the custom cut dataset consisting of 128 nodes Erdős-Rényi graphs. Even though our training dataset consisted of 10-50 node SF-295 graphs (average number of nodes 31 with density 0.08) which is very different from the Twitter (average number of nodes is 130 with density 0.18) and Custom Cut of 128 node instances (average density of 0.48), Table 6 shows that the model can still produce edge weights that the Hopfield network can decode to produce cuts of high quality. While the optimality ratio obtained on the test part of the SF-295 dataset is $0.979(\pm 0.105)$, the optimality ratio for Twitter and Custom dataset is $0.986(\pm 0.013)$ and $0.998(\pm 0.002)$, respectively which is comparable to the instances from the same distribution. Likewise, for Max-Clique, we tested our neural network performance on the Google Collab, Twitter and Custom Clique datasets to show the generalization performance of our learnt model (Table 7). The distribution of our training dataset (average number of nodes is 20 with density 0.52) again differs from the Google Collab (average number of nodes is 74 with density 0.51), Custom Clique of 128 node instances (average density 0.49) and Twitter datasets (average number of nodes is 130 with density 0.18), but still produces edge weights that result in high-quality cliques. This shows that the model has learnt a function which can be used on graphs outside of its training data/graph size and can generalize well across distributions and larger size instances.

E Sensitivity of Hopfield network to the noise and precision of SDP Gram matrix

A potential criticism of our approach is that it relies on a computationally expensive step of calculating SDP vectors. In this section, we show that we do not actually need the exact computation of SDP vectors. In fact, a coarse approximation of SDP vectors can already yield near-optimal solutions. Such low-accuracy SDP vectors can be computed efficiently and scalably. For instance, a recent pre-print by Yau et al. [28] shows that GNNs can be used to learn a low-rank SDP. Yurtsever et al. [47] solved the Max-Cut SDP (to moderate accuracy) for a sparse graph with over 20 million vertices, where the matrix variable has over 10^{14} entries on a laptop equivalent machine. In this context, we also refer the reader to a review article by Majumdar et al. [48], a recent paper by Durante et al. [49] and the book chapter on "Approximately Solving Semidefinite Programs" by Gärtner and Matoušek [1].

Figure 4 shows that our approach based on the Hopfield network is significantly more robust to the addition of noise than the Goemans-Williamson approximation algorithm [16]. In this experiment, we modified each value in our Gram matrix by adding some random value from a uniform distribution. A magnitude of M in Figure 4 refers to adding a uniform random noise in the range $[-M, M]$ to each element in the Gram matrix. Note that both – our approach and the Goemans-Williamson algorithm – rely on SDP vectors. However, the optimality ratio of the Goemans-Williamson algorithm goes down considerably as we insert noise in the Gram matrix, whereas our approach returns optimal or near-optimal solutions even when a large amount of noise is added. As expected, too much noise kills

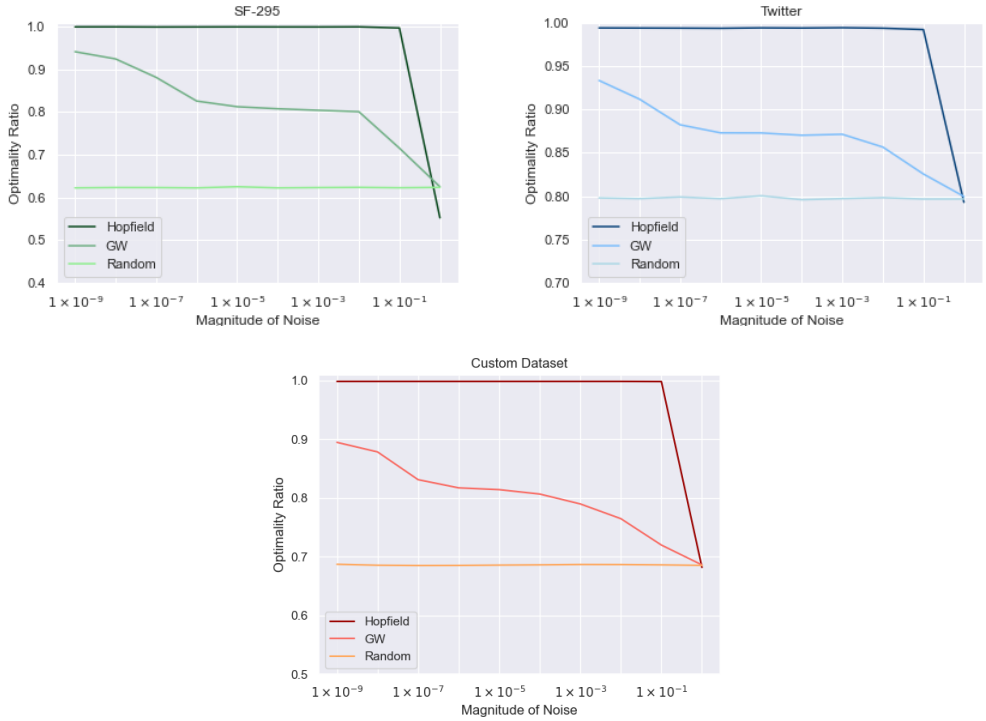

Figure 4: The plot shows how the mean optimality ratio decreases with increasing noise for our approach based on Hopfield network/SDP vectors and for the Goemans-Williamson approximation algorithm. The results are shown for the SF-295 graphs (top-left), the collection of Twitter graphs (top-right) and our collection of Custom Cut Graphs with 128 nodes (bottom).

the performance of our approach as well, indicating that SDP solutions do contain useful information for our approach.

Next, we explore the effect of reducing the precision by rounding the values in the Gram matrix to the nearest decimal place. Figure 5 shows that when we reduced the precision of Gram Matrix instead of adding random noise, we could still obtain optimal or near-optimal solutions on most problem instances. Again, this is in contrast with the Goemans-Williamson algorithm, for which the optimality ratio degrades considerably. We obtained similar noise/precision results for the graphs from IMDB and the Google Collab collection as well.

We obtained similar results for the Max Clique problem instances when we reduced the precision by rounding the values in the Gram matrix to the nearest decimal place.From the above discussion, we conclude that our rounding approach is tolerant to noise and can potentially leverage a low-precision SDP Gram matrix (e.g., using an approximation) to extract good-quality solutions.

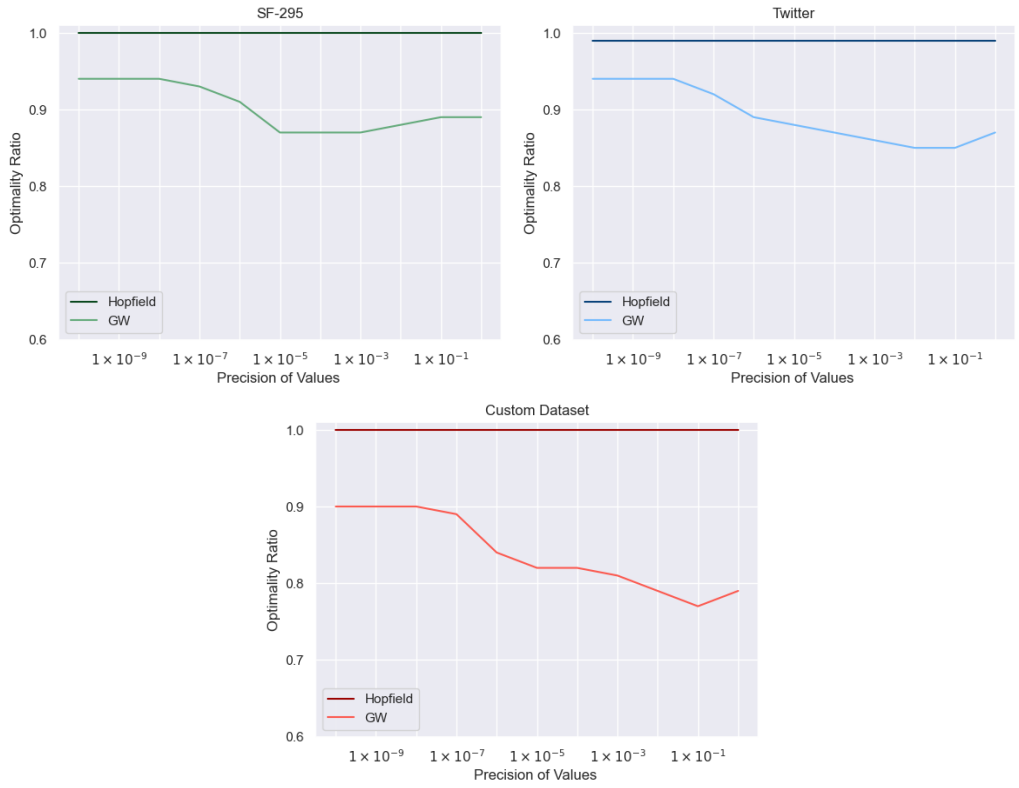

Figure 5: The plot shows how the mean optimality ratio decreases with reduced precision for our approach based on Hopfield network/SDP vectors and for the Goemans-Williamson approximation algorithm. The results are shown for the SF-295 graphs (top-left), the collection of Twitter graphs (top-right) and our collection of Custom Cut Graphs with 128 nodes (bottom).