Model Agnostic Solution of CSPs via Deep Learning: A Preliminary Study

Conference Paper *in* Lecture Notes in Computer Science · June 2018

DOI: 10.1007/978-3-319-93031-2_18

Andrea Galassi
University of Bologna
10 PUBLICATIONS **18** CITATIONS
SEE PROFILE

Michele Lombardi
University of Bologna
73 PUBLICATIONS **541** CITATIONS
SEE PROFILE

Michela Milano
University of Bologna
236 PUBLICATIONS **2,413** CITATIONS
SEE PROFILE

Model agnostic solution of CSPs via Deep Learning: a preliminary study

Andrea Galassi(✉), Michele Lombardi, Paola Mello, and Michela Milano

Department of Computer Science and Engineering (DISI), University of Bologna
{a.galassi,michele.lombardi2,paola.mello,michela.milano}@unibo.it

Abstract. Deep Neural Networks (DNNs) have been shaking the AI scene, for their ability to excel at Machine Learning tasks without relying on complex, hand-crafted, features. Here, we probe whether a DNN can learn how to construct solutions of a CSP, without any explicit symbolic information about the problem constraints. We train a DNN to extend a feasible solution by making a single, globally consistent, variable assignment. The training is done over intermediate steps of the construction of feasible solutions. From a scientific standpoint, we are interested in whether a DNN can learn the structure of a combinatorial problem, even when trained on (arbitrarily chosen) construction sequences of feasible solutions. In practice, the network could also be used to guide a search process, e.g. to take into account (soft) constraints that are implicit in past solutions or hard to capture in a traditional declarative model. This research line is still at an early stage, and a number of complex issues remain open. Nevertheless, we already have intriguing results on the classical Partial Latin Square and N-Queen completion problems.

1 Introduction

Deep Neural Networks (DNNs) [12], are characterized by the ability to learn high-level concepts without the need of symbolic features. In this paper, we investigate the idea that DNNs could be capable of learning how to solve combinatorial problems, *with no explicit information about the problem constraints*. This is partially motivated by the results achieved in a previous work regarding the application of DNNs to a board game [3]. In particular, we train a DNN to extend a feasible partial solution by making a single, globally consistent, variable assignment.

In principle, *such a network could be used to guide a search process*: this may be used to take into account constraints that are either implicit in the training solutions, or too difficult to capture in a declarative model. In this sense, the approach is complementary to Empirical Model Learning (EML) [14], where the goal is instead to learn a constraint. The method presented here is applicable even when only positive examples (i.e. feasible solutions) are available. Moreover, using the DNN to guide search may also provide a speed-up when solving multiple instances of the same problem. *Practical applications are not our only driver*, however: there is a strong scientific interest in assessing to what extent a sub-symbolic technique, trained on arbitrarily chosen solution construction sequences, can learn something of the problem structure.

This line of research is at an early stage, and there are many complex issues to be solved before reaching practical viability. *So far, we have focused on two classical Constraint Satisfaction Problems* (CSPs), namely N-queen completion and Partial Latin Square. For these benchmarks we have intriguing results, the most striking being an impressive discrepancy between the (low) DNN accuracy and its (very high) ability to generate feasible assignments: this suggest that the network is indeed learning something about the problem structure, even if it has been trained to "mimic" specific solution construction sequences.

This is not the first time that Neural Networks have been employed to solve CSPs. For example, Guarded Discrete Stochastic networks [1] can solve generic CSPs in an unsupervised way. They rely on a Hopfield network to find consistent variable assignment, and on a "guard" network to force the assignment of all the variables. The GENET [16] method can construct neural networks capable of solving binary CSPs, and was later extended in EGENET [13] to support non-binary CSPs. In [2], a CSP is first reformulated as a quadratic optimization problem. Then, a heuristic is used to guide the evolution of a Hopfield network from an initial state representing an infeasible solution to a final feasible state. Crucially, *all these methods rely on full knowledge of the problem constraints to craft both the structure and the weights of the networks.* What we are trying to do is in fact radically different.

2 General Method and Grounding

General Approach. We train a DNN to extend a partial solution of a combinatorial problem, *by making a single additional assignment that is globally consistent*, i.e. that can be extended to a full solution.

We use simple bit vectors for both the network input and output. We represent assignments using a one-hot encoding, i.e. for a variable with n values we reserve n bits; raising the i-th bit corresponds to assigning the i-th domain value. If no bit is raised, the variable is unassigned. Using such a simple format makes our input encoding *general* (any set of finite domain variables can be encoded), and truly *agnostic to the problem constraints.* As a major drawback, the method is currently restricted to problems of a pre-determined size.

Our training examples are obtained by *deconstructing a comparatively small set of solutions.* We considered two different strategies, referred to as *random* and *systematic deconstruction*, as described in Algorithm 1 and 2. Both methods operate by processing a partial solution s and populate a dataset T with pairs of partial solutions and assignments. In the pseudo code, s_i refers to the value of the i-th variable in s, and $s_i = \bot$ if the variable is unassigned. The random strategy generates in a backward fashion one arbitrary construction sequence for the solution. The systematic strategy generates all possible construction sequences. *When all the original solutions have been deconstructed, we prune the dataset* by considering all groups of examples sharing the same partial solution, and selecting a single representative at random.

Alg. 1 RandomDeconstruction(s)

```
Randomly choose a variable index $i$
$s' = s$ (copy the partial solution)
$s'_i = \bot$ (undo one assignment)
Insert $(s', s_i)$ in $T$
RandomDeconstruction($s'$)
```

Alg. 2 SystematicDeconstruction(s)

```
for all variable indices $i$ do
    $s' = s$ (copy the partial solution)
    $s'_i = \bot$ (undo one assignment)
    Insert $(s', s_i)$ in $T$
    SystematicDeconstruction($s'$)
```

The DNN is trained for a classification task: for each example, the target vector (i.e. the class label) contains a single raised bit, corresponding to the assignment s_i in the dataset. The network yields a normalized score for each bit in the output vector, which can be interpreted as a probability distribution. The bit with the highest score corresponds to the suggested next assignment. We take no special care to prevent the network from trying to re-assign an already assigned variable. These choices have three important consequences: 1) the network is *agnostic to the problem structure*; 2) the network is technically *trained to mimic specific construction sequences* of arbitrarily chosen solutions; 3) assuming that the DNN is used to guide a search process, it is *easy to take into account propagation* by disregarding the scores for variable-value pairs that have been pruned. As an adverse effect, we are forsaking possible performance advantages that could come by including information about the problem structure.

Grounding (Benchmark Problems). So far, we have grounded our approach on two classical CSPs, namely the N-queen completion and Partial Latin Square (PLS, see [4]) problems. Classical problems let us work in a controlled setting with well known properties [6,7,8], and simplifies drawing scientific conclusions.

The N-queen completion problem consist in placing n queens pieces on a $n \times n$ chessboard, so that no queen threats another. The PLS problem consist in filling an $n \times n$ square with numbers from 1 to n so that the same number appears only once per row and column. In both cases, some variables may be pre-assigned. We focus on N-queen problems of size 8 and PLSs of size 10. In both cases, we model assignments using a one-hot encoding, leading to vector of size $8 \times 8 = 64$ for the n-queens and $10 \times 10 \times 10 = 1,000$ for the PLS.

For the 8-queen problem, we have used 1/4 of the 12 non-symmetric solution to seed the training set, and the remaining ones for the test set. Both the training and the test set are then obtained by generating all the symmetric equivalents, and then by applying systematic deconstruction to the resulting solutions.

For the PLS, we have used an unbiased random generation method to obtain two "raw" datasets, respectively containing 10,000 and 20,000 solutions. The numbers are considerably large in this case, but they are very small compared to the number of size 10 PLS ($\sim 10^{31}$). As comparison, it is a bit like making sense of the layout of Manhattan from ~ 0.75 square nanometers of surface scattered all over the place. Each of the raw datasets is split into a training and test set, containing respectively 1/4 and 3/4 of the solutions. The actual examples have then been obtained by random deconstruction.

Grounding (Networks and Training). Due to the impressive results obtained in computer vision tasks, we have chosen to use pre-activated Residual Networks [9,5,10]. We have adapted the architecture to use fully-connected layers rather than convolutional ones, as the latter are not well suited to deal with generic combinatorial problems.

We have trained the networks in a supervised fashion, using 10% of the examples (chosen at random) as a validation set. The loss function is the negative log-likelihood of the target class, with a 10^{-4} L1 regularization coefficient. The choice of the network and training hyper-parameters has been made after an informal tuning. We have eventually settled for using the Adam [11] optimizer, with parameters $\beta_1 = 0.9$ and $\beta_2 = 0.99$. The initial learning rate α_0, was progressively annealed through epochs with decay proportional to training epoch t, resulting in a learning rate $\alpha = \frac{\alpha_0}{1+k\times t}$ with $k = 10^{-3}$. Training was stopped after there was no improvement on the validation accuracy for e epochs.

For the 8-queens problem we have used an initial layer of 200 neurons, than 100 residual blocks, each one composed by two layers of 500 and 200 neurons, and finally an output layer of 64 neurons, for a total of more than 200 layers. Batch optimization has been employed, using a initial learning rate of $\alpha_0 = 0.1$ and a patience of $e = 200$ epochs. Dropout [15] has been applied to each input and hidden neuron with probability $p = 0.1$.

For the PLS problem, we have used a smaller network because of the bigger input/output vectors and the larger datasets would have required too much training time. Therefore we have used an initial layer of 200 neurons, then 10 residual blocks, each one composed by two layers of 300 and 200 neurons, and a final output layer of 1000 neurons, for a total of 22 layers. Mini-batch optimization has been employed, using shuffling in each epoch, using a initial learning rate of $\alpha_0 = 0.03$ and a patience of $e = 50$ epochs. Dropout has been applied to hidden neuron with probability $p = 0.1$. The size of the mini batch has been setted to 50,000 for the training on the 10k dataset and to 100,000 for the 20k dataset.

3 Experimentation

We designed our experiments to address four main questions. First, we want to assess how well the DDNs are actually learning their designated task, i.e. to guess the "correct" assignment according to the employed deconstruction method. Second, we are interested in whether the DNNs learn to generate feasible assignments, no matter whether those are "correct" according to datasets. Third, assuming that the networks are actually learning something about the problem constraints, it makes sense to check whether some constraint types are learned better than others. Finally, we want to investigate whether using the DNNs to guide an actual tree search process leads to a reduction in the number of fails.

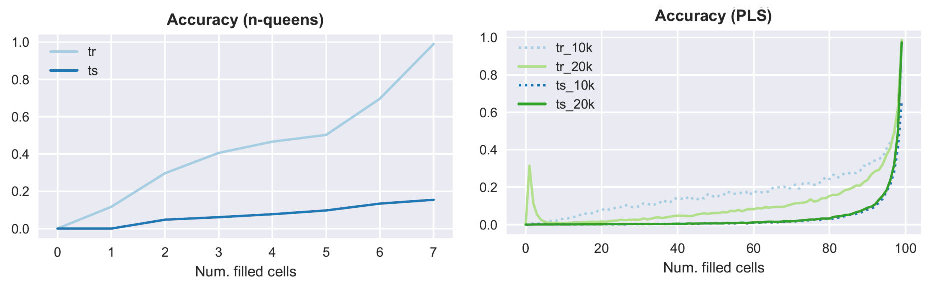


Fig. 1. Accuracy on the training and test sets

Network Accuracy. Here we are interested in assessing the performance of our DNNs in their natural task, i.e. learning "correct" variable-value assignment, as defined by our deconstruction procedure. Figure 1 shows the accuracy reached by our DNNs on both the training and the test sets, grouped by the number of pre-assigned variables in the example input. For comparison, random guessing would reach an accuracy of $1/64 \simeq 0.015$ for the 8-queens and $1/1,000$ for the PLS. There are three notable facts:

1. The accuracy is at least one order or magnitude larger than random guessing, but still rather low, in particular for the PLS; this suggest that *the networks are not doing particularly well at the task they are being trained for.*
2. Second, *the accuracy on the test set if considerably lower than on the training set*; normally this is symptomatic of overfitting, but in this case there is also *a structural reason.* The pruning in the last phase of dataset generation introduces a degree of ambiguity in our training: as an extreme case, for the same partial assignment, the training and the test set may report different "correct" assignments that cannot be both predicted correctly.
3. Third, the accuracy tends to increase with the number of filled cells. Having many filled cells means having very few feasible completions, and therefore it is more unlikely for the same instance to appear both in the test and train set with a different target. In this situation it is intuitively easier for the network to label a specific assignment as the "correct" one.

The third observation leaves an open question: while the small number of feasible completions can explain why the accuracy raises, it fails to explain the magnitude of the increase. *The result would be much easier to explain by assuming that the DNN has somehow learned something about the problem constraints.*

Feasibility Ratio. It makes sense to evaluate the ability of the DNNs to yield globally consistent assignments, even if those are not chosen as "correct", since this is our primary goal. Figure 2 show the ratio of predictions of the DNNs (both on the training and test set) that could be expanded to full solutions. For

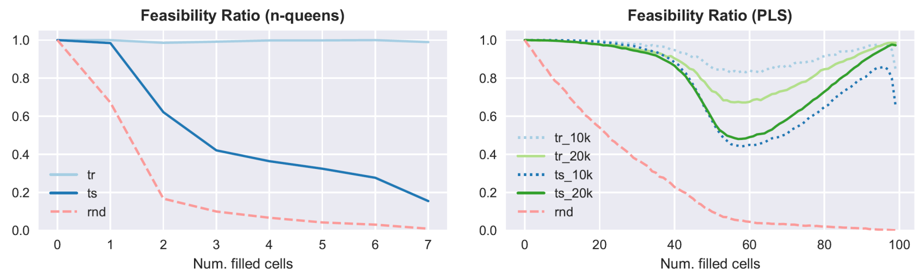


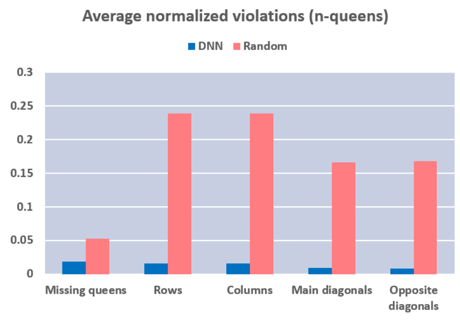
Fig. 2. Feasibility ratios on the training and test sets

comparison, the figures report also the results that can be obtained by guessing at random on the test sets. There are three very relevant observations to make:

1. There is *a striking difference between the accuracy values from Figure 1 and the feasibility ratios.*
2. Such discrepancy may be due to the fact that the more a partial solution is empty, the more are the feasible assignments that can be found even by guessing. However, *the reported feasibility ratio are also significantly higher than the random baseline.* This is hard to explain, unless we assume that the DNNs have somehow learned the semantic of the problem constraints.
3. The feasibility ratios for the PLS networks have a dip between 50 and 60 pre-assigned variables, and then tend to raise again. *This is exactly the behavior that one would expect thank to constraint propagation*: when many variables are bound many values are pruned and the number of available assignments is reduced. However, *the DNNs at this stage do not rely on propagation at all.* Even the higher accuracy from Figure 1 is not enough to justify how much the feasibility rations tend to increase for almost full solutions. Assuming that the DNN has learned the problem constraints can explain the increase, but not so easily the dip.

Constraints Preference. Next, we have designed an experiment to investigate whether some constraints are handled better than others. We start by generating a pool of (partial) solutions by using the DNN to guide a randomized constructive heuristic. Given a partial solution, we use the DNN to obtain a probability distribution over all possible assignment, one of which is chosen randomly and performed. Starting from an empty solution, the process is repeated as many times as there are variables, and relies on our low-level, bit vector, representation of the partial solution. As a consequence, at the end of the process there may be variables that have been “assigned multiple times”, and therefore also unassigned variables. We have used this approach to generate 10,000 solutions for each DNN, and for comparison we have done the same using a uniform distribution.

Once we have such a pool of partial solution, we count the average degree of violation of each abstract problem constraints, e.g. the number of rows with mul-



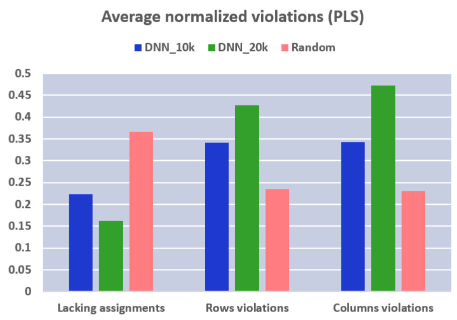


Fig. 3. Average violations on the two problems

tiple queens. Each quantity is then normalized over the corresponding maximum (i.e. the number of row/columns, or the number of variables). Looking at the average violations for the random baseline intuitively tells the natural difficulty of satisfying a constraint type. Comparing such values with those of the DNN allows to evaluate how well the DNN is faring.

As reported in Figure 3, for the N-queens problem the network gets much closer to feasibility than the random baseline and all problem constraints are handled equally well. For the PLS problem, the DNNs violates the row and column constraints significantly more than the random baseline, but they also tend to leave fewer variable unassigned. There is a logic correlation between these two values, since assigning more variables increases the probability to violate a row or a column constraint.

Guiding Tree Search. Finally, we have tried using our DNNs to guide a Depth First Search process for the Partial Latin Square[1]. In particular, we always make the assignment with the largest score, excluding bounded variables and values pruned by propagation.

We employ a classic CP model for the PLS (one finite domain variable per cell), and use the GAC AllDifferent propagator for the row and column constraint. We compare the results of the two DNNs with those of heuristic that pick uniformly at random both the variable and the value to be assigned. Given that our research is at a early stage, we have opted for a simple (but inefficient) implementation relying on the Google or-tools python API: for this reason we focus our evaluation on the number of fails. As a benchmarks, we have sampled 4,000 partial solutions from the 20k training and test set, at the complexity peak. All instances have been solved with a cap at 10,000 fails.

The results of this experimentation are reported in Figure 4, using box plots. Apparently, the DNN trained on the 10k dataset is more efficient than the random baseline, but the opposite holds for the one trained in the 20k dataset. This

[1] The 8-queens problem is too easy to provide meaningful measurements.

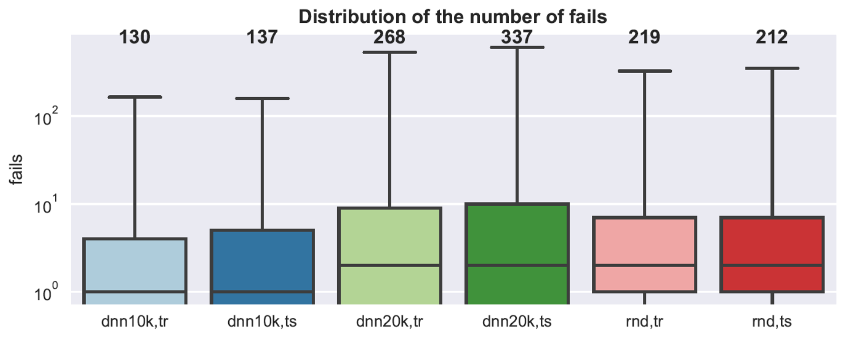


Fig. 4. Distribution of the number of fails on the train and test sets. At the top of each box we report the number of times the fail cap was reached.

matches the results obtained in our analysis of violated constraints, but not those obtained for the feasibility ratio. We suspect however that explaining the performance (and obtaining practical speed-up) will require to take into account the complex trade-off making choice that are likely feasible, and recovering quickly from the inevitable mistakes. This is a well know open problem in Constraint Programming, that we plan to tackle as part of future work.

4 Conclusions

We have performed a prelimiary investigation to understand whether DNNs can learn how to solve combinatorial problems. We have adopted a general setup, totally agnostic to the problem structure, and we have trained the networks on arbitrarily chosen solution construction sequences.

Our experimentation has provided evidence that, despite having low accuracy at training time, a DNN can become capable of generating globally consistent variable-value assignments. This cannot be explained assuming that the networks only mimic the assignment sequences in the training set, but it compatible with the hypothesis that the DNNs have learned something about the problem structure. The networks do not seem to favor any abstract constraint in particular, suggesting that what they are learning does not match our usual understanding of CSPs. When used for guiding a search process, our DNNs have provided mixed results, highlighting that achieving performance improvements may require to deal more explicitly with the peculiarities of a specific solution technique (e.g. constraint propagation).

This research line is still at an early stage: there are considerable overheads that make practical applications still far, and the method is currently limited to problems of fixed size, a problem that maybe could be solved using only convolutional layers. However, we believe the approach to have enough potential to deserve further investigation.

References

1. Adorf, H.M., Johnston, M.D.: A discrete stochastic neural network algorithm for constraint satisfaction problems. In: 1990 IJCNN International Joint Conference on Neural Networks. pp. 917–924 vol.3 (June 1990)
2. Bouhouch, A., Chakir, L., Qadi, A.E.: Scheduling meeting solved by neural network and min-conflict heuristic. In: 2016 4th IEEE International Colloquium on Information Science and Technology (CiSt). pp. 773–778 (Oct 2016)
3. Chesani, F., Galassi, A., Lippi, M., Mello, P.: Can deep networks learn to play by the rules? a case study on nine men's morris. IEEE Transactions on Games PP(99), 1–1 (2018), https://doi.org/10.1109/TG.2018.2804039
4. Colbourn, C.J.: The complexity of completing partial latin squares. Discrete Applied Mathematics 8(1), 25–30 (1984)
5. Ebrahimi, M.S., Abadi, H.K.: Study of residual networks for image recognition
6. Gent, I.P., Jefferson, C., Nightingale, P.: Complexity of n-queens completion. Journal of Artificial Intelligence Research 59, 815–848 (2017)
7. Gomes, C.P., Selman, B., Crato, N.: Heavy-tailed distributions in combinatorial search. In: Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings. pp. 121–135 (1997), https://doi.org/10.1007/BFb0017434
8. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA. pp. 431–437 (1998), http://www.aaai.org/Library/AAAI/1998/aaai98-061.php
9. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 770–778 (2016)
10. He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks. In: European Conference on Computer Vision. pp. 630–645. Springer (2016)
11. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. CoRR abs/1412.6980 (2014), http://arxiv.org/abs/1412.6980
12. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature 521(7553), 436–444 (2015)
13. Lee, J.H.M., Leung, H.F., Won, H.W.: Extending genet for non-binary csp's. In: Proceedings of 7th IEEE International Conference on Tools with Artificial Intelligence. pp. 338–343 (Nov 1995)
14. Lombardi, M., Milano, M., Bartolini, A.: Empirical decision model learning. Artif. Intell. 244, 343–367 (2017), https://doi.org/10.1016/j.artint.2016.01.005
15. Srivastava, N., Hinton, G.E., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. Journal of machine learning research 15(1), 1929–1958 (2014)
16. Wang, C.J., Tsang, E.P.K.: Solving constraint satisfaction problems using neural networks. In: 1991 Second International Conference on Artificial Neural Networks. pp. 295–299 (Nov 1991)