Injecting Domain Knowledge in Neural Networks: A Controlled Experiment on a Constrained Problem

Injecting Domain Knowledge in Neural Networks: a Controlled Experiment on a Constrained Problem

Mattia Silvestri[1] Michele Lombardi[1] and Michela Milano[1]

[1]University of Bologna
{mattia.silvestri4,michele.lombardi2,michela.milano}@unibo.it

Abstract. Recent research has shown how Deep Neural Networks trained on historical solution pools can tackle CSPs to some degree, with potential applications in problems with implicit soft and hard constraints. In this paper, we consider a setup where one has offline access to symbolic, incomplete, problem knowledge, which cannot however be employed at search time. We show how such knowledge can be generally treated as a propagator, we devise an approach to distill it in the weights of a network, and we define a simple procedure to extensively exploit even small solution pools. Rather than tackling a real-world application directly, we perform experiments in a controlled setting, i.e. the classical Partial Latin Square completion problem, aimed at identifying patterns, potential advantages, and challenges. Our analysis shows that injecting knowledge at training time can be very beneficial with small solution pools, but may have less reliable effects with large solution pools. Scalability appears as the greatest challenge, as it affects the reliability of the incomplete knowledge and necessitates larger solution pools.

1 Introduction

Given enough data, Deep Neural Networks (DNNs) are capable of learning complex input-output relations with high accuracy. Recent work has shown how this applies also to the solution process of Constraint Satisfaction Problems, at least to some degree: examples include the approach from [26], relying on a pool of solutions, or Reinforcement Learning approaches inspired by [2], relying on solution checkers/evaluators. This class of approaches, while still not close to the state of the art in combinatorial decision making, may have advantages in terms of robustness and when implicit soft or hard constraints are present. For example, course timetables often need to take into account both explicit constraints (e.g. preferences, capacities) and informal agreements or manually enforced rules. A second, less explored, area of application concerns problems with well-defined sources of symbolic knowledge, which cannot however be easily exploited at search time. Examples include simulators, complex nonlinear equations, or particularly expensive (e.g. NP-hard) propagators. In this context, a

Deep Learning approach may learn to satisfy such constraints without the need for a propagator at search time. In this paper, we focus on the latter use case and investigate methods for injecting offline information into DNNs designed to tackle combinatorial problems. Specifically, we will consider training a network for identifying variable-value assignments that are likely to be feasible. We will assume the availability of both implicit knowledge (from data), and explicit symbolic knowledge that can be accessed prior to the search process. Rather than tackling a real-world problem directly, *we perform experiments in a controlled setting*, with the aim to gauge the potential of the approach and identify the key challenges. The idea, in the spirit of [6], is to test the ground before starting the complex and time-consuming endeavor of applying such methods in a real-world use case.

In detail, we use as a benchmark the Partial Latin Square (PLS) completion problem, which requires to complete a partially filled $n \times n$ square with values in $\{1..n\}$, such that no value appears twice on any row or column. Despite its simplicity, the PLS is NP-hard, unless we start from an empty square, it has practical applications (e.g. in optical fiber routing), and serves as the basis for more complex problems (e.g. timetabling). We focus on the only PLS due to its clear structure, availability of multiple solutions that can be easily generated, and its single defining parameter (size). Using a classical constrained problem as a case study grants access to domain knowledge (the declarative formulation), and facilitates the generation of empirical data (problem solutions). This combination enables controlled experiments that are impossible to perform on real-world datasets.

As a baseline, we train on a pool of solutions *a problem-agnostic, data-driven, approach*. We devise *a simple method to extract multiple training examples* from a finite set of solutions, and we define a technique, building over Semantic Based Regularization [9] to *inject at training time domain knowledge coming from constraint propagators*. We then adjust the amount of initial data (empirical knowledge) and of injected constraints (domain knowledge) and assess the ability of the approach to identify feasible assignments. Our results show that even very small solution pools, provided they are coupled with offline knowledge injection, are enough for the DNN to identify feasible assignments with reliability comparable to a propagator at search time. When training solutions are plentiful, conversely, injecting offline knowledge has a less pronounced (or even deleterious) effect. Scalability appears as the greatest challenge, as it affects the reliability of the incomplete knowledge and necessitates larger solution pools.

The paper is organized as follows: Section 2 briefly surveys the related literature and motivates the choice of our baseline techniques; Section 3 discusses the details of the problem and methods we use; Section 4 presents the results of our analysis, while Section 5 provides concluding remarks.

2 Related Works and Baseline Choice

The analysis that we aim to perform requires 1) a data-driven technique that can solve a constrained problem, *with no access to its structure*; moreover, we need 2) methodologies for injecting domain knowledge in such a system. In this section, we briefly survey methods available in the literature for such tasks and we motivate our selection of techniques.

Neural Networks for Solving Constrained Problems. The integration of Machine Learning methods for the solution of constrained problems is an active research topic, recently surveyed in [3]. Many such approaches consider how ML can improve specific steps of the solution process: here, however, we are interested in methods that use learning to replace (entirely or in part) the modeling activity itself. These include Constraint Acquisition techniques (e.g. [4]), which attempt to learn a declarative problem description from feasible/infeasible variable assignments. These approaches may however have trouble dealing with implicit knowledge (e.g. preferences) that cannot be easily stated in a well-defined constraint language. Techniques for encoding Machine Learning models in constrained problems (e.g. [11, 16, 25, 20]) are capable of integrating empirical and domain knowledge, but not at training time; additionally, they require to know a-priori which variables are involved in the constraints to be learned.

Some approaches (e.g. [1, 5]) rely on carefully structured Hopfield Networks to solve constrained problems, but designing these networks (or their training algorithms) requires full problem knowledge. Recently, Reinforcement Learning and Pointer Networks [2] or Attention [14] have been used for solving specific classes of constrained problems, with some measure of success. These approaches also require a high degree of problem knowledge to generate the reward signal, and to some degree for the network design. The method from [26] applies Neural Networks to predict the feasibility of a binary CSP, with a very high degree of accuracy; the prediction is however based on a representation of the allowed variable-value pairs, and hence requires explicit information about the problem.

In the approach from [12], from some of the authors of this paper, a Neural Network is used to learn how to extend a partial variable assignment so as to retain feasibility. Despite its limited practical effectiveness, this method shares the best properties of Constraint Acquisition (no explicit problem information), without being restricted to constraints expressed in a classical declarative language. *This last approach was chosen as our baseline*, since it represents (to the best of our knowledge) the data driven method for constraint problems that requires the least amount of problem knowledge. In particular, it requires neither information about the problem constraints (like e.g. [26]), nor a fully known (or at least evaluable) problem model like all Reinforcement Learning approaches.

Domain Knowledge in Neural Networks. There are several approaches for incorporating external knowledge in Neural Networks, none of which has been applied so far on constrained decision problems. One method to take into account domain knowledge *at training time* is Semantic Based Regularization (SBR) [8],

which is based on the idea of converting (logical) constraints into regularizing terms in the loss function used by a gradient-descent algorithm. Differentiability is achieved by means of fuzzy logic. In a similar way, [27] describes a semantic loss function that quantifies how much the network is satisfying constraints defined as sentences of propositional logic.

The approach can be pushed to an extreme by entirely replacing the loss function with a logical formula (again in fuzzy form), such as in Logic Tensor Networks (LTNs) [23]. LTNs are connected to Differentiable Reasoning [24], which uses relational background knowledge to benefit from unlabeled data.

Domain knowledge has also been introduced in differentiable Machine Learning (mainly Deep Networks) by adjusting their structure, rather than the loss function: examples include Deep Structured Models, e.g. [15] and [17], the latter integrating deep learning with Conditional Random Fields. The authors of [7] have developed a method to inject the domain knowledge encoded as First Order Logic formulas in Neural Networks generating an additional final layer that modifies the predictions according to the knowledge. Integration of external knowledge in Neural Networks *after training* is considered for example in DeepProbLog [18], where DNNs with probabilistic output (classifiers in particular) are treated as predicates. Markov Logic Networks achieve similar results via the use of Markov Fields defined over First Order Logic formulas [21], which may be defined via probabilistic ML models. [22] presents a Neural Theorem Prover using differentiable predicates and the Prolog backward chaining algorithm.

Some works attempt to both learn symbolic knowledge and enable reasoning with predicates represented by ML models. The method in [19] are similar in spirit to SBR or LTN, but they enable learning the weights of constraint terms (based on compatibility with the data), rather than having them fixed by an expert. This connects the approach to Differentiable Inductive Logic Programming, which attempts to learn (soft) logic problem from noisy data [10], by building over Inductive Logic Programming ideas.

We use a method loosely based on SBR for injecting knowledge at training time, as it offers a good compromise between flexibility and simplicity. In addition, since we regularize the propagator output rather than the constraint itself, our predicates are unary and hence we have no relational terms, making approaches like [23], [7] and [27] extremely similar to SBR in our setup.

3 Basic Methods

We reimplemented the approach from [12] and extended it via a number of techniques, described in this section together with our evaluation procedure.

Neural Network for the Data Driven Approach. The baseline approach is based on training a Neural Network to extend a partial assignment (also called a *partial solution*) by making one additional assignment, so as to preserve feasibility. Formally, the network is a function:

$$f : \{0,1\}^m \to [0,1]^m \tag{1}$$

Algorithm 1 DECONSTRUCT(x)

```
D = ∅
while ‖x‖₁ > 0 do
    Let y = 0    # zero vector
    Select a random index i s.t. x_i = 1
    Set x_i = 0, set y_i = 1
    Add the pair ⟨x, y⟩ to D
return D
```

whose input and output are m dimensional vectors. Each element in the vectors is associated to a variable-value pair $\langle z_j, v_j \rangle$, where z_j is the associated variable and v_j is the associated value. We refer to the network's input as x, assuming that $x_j = 1$ iff $z_j = v_j$. Each component $f_j(x)$ of the output is proportional to the probability that pair $\langle z_j, v_j \rangle$ is chosen for the next assignment. This is achieved in practice by using an output layer with m neurons with a sigmoid activation function. The setup makes no assumptions on the constraint structure but requires a fixed problem size and variables with finite domains.

Dataset Generation Process. The input of each training example corresponds to a partial solution x, and the output to a single variable value assignment (represented as a vector y using a one-hot encoding). The training set is constructed by repeatedly calling the randomized deconstruction procedure of Algorithm 1 on an initial set of full solutions (referred to as *solution pool*). Each call generates a number of examples that are used to populate a dataset. At the end of the process, we discard multiple copies of identical examples. Two examples may have the same input, but different output, since a single partial assignment may have multiple viable completions.

Unlike [12], here we sometimes perform *multiple calls to Algorithm 1 for the same starting solution.* This simple approach enables to investigate independently the effect of the training set size and of the actual amount of empirical knowledge (the size of the solution pool).

Training and Knowledge Injection. The basic training for the NN is the same as for neural classifiers. Since the network output can be assimilated to a class, we process the network output through a softmax operator, and then we use as a loss function the categorical cross-entropy H. Additionally, we inject domain knowledge at training time via an approach that combines ideas of Semantic Based Regularization (SBR) and Constraint Programming.

Without loss of generality, we assimilate *domain knowledge to a constraint propagator*, in the sense that it can be used to flag specific variable-value pairs as either feasible or infeasible. In our experimentation, we indeed use a classical propagator (Forward Checking) as the source of domain knowledge.

Formally, given a constraint (or a collection of constraints) C, here we will treat its associated propagator as a multivariate function such that $C_j(x) = 1$ iff assignment $z_j = v_j$ has not been marked as infeasible by the propagator,

while $C_j(x) = 0$ otherwise. Given that, we formulate three different approaches to augment the loss function with an SBR inspired term.

The first one relies on the usual assumption that pruned values are supposed to be provably infeasible. Given an example $\langle x, y \rangle$, we have:

$$L_{sbr}^{negative}(x) = \sum_{j=0}^{m-1}((1 - C_j(x)) \cdot f_j(x)) \tag{2}$$

i.e. increasing the output of a neuron corresponding to a pair flagged as infeasible incurs in a penalty that grows with $f_j(x)$.

For the other two methods, we just acknowledge that the domain knowledge may be incomplete, discouraging provably infeasible pairs, and encouraging the remaining ones. The only difference is in the cost function. In one instance the cost function is the binary cross-entropy, since for each partial solution there may exist many global viable completions, and the SBR inspired term is:

$$L_{sbr}^{bce}(x) = \sum_{j=0}^{m-1}(C_j(x) \cdot log(f_j(x)) + (1 - C_j(x)) \cdot log(1 - f_j(x)) \tag{3}$$

In the other case instead we employ the mean squared error as cost function for the SBR inspired regularization:

$$L_{sbr}^{mse}(x) = \sum_{j=0}^{m-1}(C_j(x) - f_j(x))^2 \tag{4}$$

Our full loss is hence given by:

$$L(x, y) = H\left(\frac{1}{Z}f(x), y\right) + \lambda L_{sbr}(x) \tag{5}$$

where Z is the partition function and the scalar λ controls the balance between the cross-entropy term H and the SBR term, i.e. the amount of trust we put in the incomplete domain knowledge. Since we assume the domain knowledge/propagator to be incomplete, there is a risk of injecting incorrect information into the model. In practice, this is balanced by the presence of the categorical cross-entropy term in the loss: only the single pair that comes from the deconstruction of a full solution will be associated with a non-null component, and this pair is guaranteed to be *globally feasible*.

The method can be applied for all known propagators with discrete, finite domain, variables. By adapting the structure of the SBR term, it can be made to work for important classes of numerical propagators (e.g. those that enforce Bound Consistency).

Evaluation and Knowledge Injection. We evaluate the approach via a constraint solver, a classical PLS model, and a randomized search strategy. Formally, we assume access to a function SOLVE(x, C, h), where x is the starting partial

Algorithm 2 FEASTEST(X, C, h)

```
J* = arg max{h_j(x) | C_j(x) = 1}        # Most likely assignments
Pick j* uniformly at random from J*
Set x_{j*} = 1
if SOLVE(x, C_pls, h_rnd) ≠ ⊥ then
  return 1                                # Globally feasible
else
  return 0                                # Globally infeasible
```

assignment, C is the considered (sub)set of problem constraints, and h is a probability estimator for variable-value pairs (e.g. our trained NN). The function runs a Depth First Search using the Google or-tools constraint solver: the variable-value pair for the left branch is chosen at random with probabilities proportional to $h(x')$, where x' is the current state of assignments. The SOLVE function returns either a solution, or $\perp$ in case of infeasibility.

Our main evaluation method tests the ability of the NN to identify individual assignments that are globally feasible, i.e. that can be extended into full solutions. This is done via Algorithm 2, which 1) starts from a given partial solution; 2) relies on a constraint propagator C (if supplied) to discard some of the provably infeasible assignments; 3) uses the NN to make a (deterministic) single assignment; 4) attempts to complete it into a full solution (taking into account all problem constraints, i.e. C_{pls}). Replacing the NN with a uniform probability estimator provides an uninformed search strategy. We repeat the process on all partial solutions from a test set and collect statistics. This approach is identical to one of those in [12], with one major difference, i.e. the ability to use a constraint propagator for "correcting" the output of the probability estimator. This enables us to assess the impact of using the offline knowledge directly during the search, something that is allowed in our controlled setting, but that would be impossible (e.g.) with an actual simulator.

Unlike in typical Machine Learning evaluations, accuracy is not a meaningful metric in our case, as it is tied to the (practically irrelevant) ability to replicate the same sequence of assignments observed at training time. Incidentally, accuracy is very low when measured in the traditional way in all our experiments.

4 Empirical Analysis

In this section we discuss our experimental analysis, which is designed around three key questions:

Q1: Does injecting knowledge at training time improve the network's ability to identify feasible assignments?
Q2: What is the effect of adjusting the amount of available empirical knowledge?
Q3: Can knowledge injection improve the ability to satisfy constraints in a soft fashion, i.e. in terms of the number of violations?

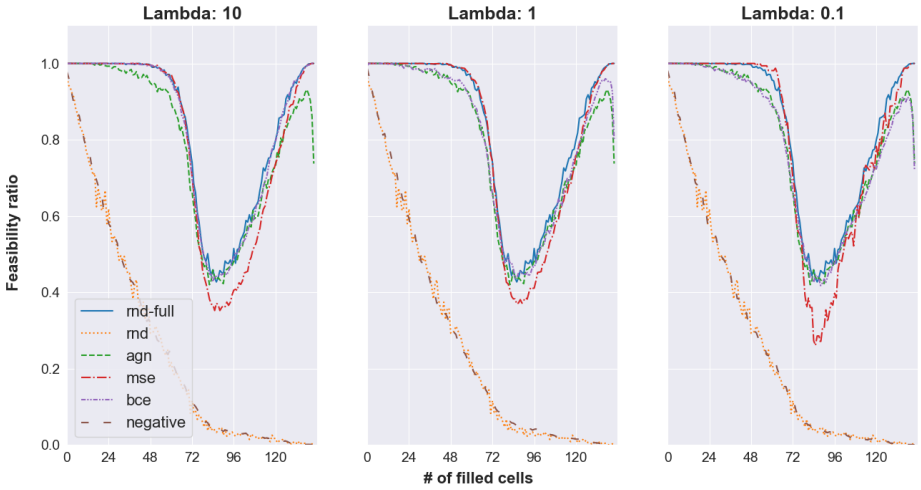

Fig. 1. Effect of the injection of the all constraints at training time comparing the regularization methods for different λ values, on the PLS-12. The dataset is generated from a 10,000 solutions pool.

While Q1 and Q2 focus on the feasibility of individual assignments, Q3 assumes that some degree of infeasibility can be tolerated. We present a series of experiments in our controlled use case that investigate such research directions. Details about the rationale and the setup of each experiment are reported in dedicated sections, but some common configurations can be immediately described.

We perform different experiments on 7×7, 10×10 and 12×12 PLS instances, resulting respectively in input and output vectors with 343, 1000 and 1728 elements. For all the experiments, we use a feed-forward, fully-connected Neural Network with three hidden layers, each with 512 units having ReLU activation function. This setup is considerably simpler than the one we used in [12], but manages to reach very similar results. We employ the Adam optimizer from Keras-TensorFlow 2.0, with default parameters. We use a batch size of 2048 for experiments on the PLS-7, whereas we adopt a batch size of 50,000 for the ones on PLS-10 and PLS-12.

4.1 Regularization methods comparison and λ-tuning

As a first step to evaluate the impact of knowledge injection at training time, we compare the regularization methods and evaluate how the λ value affects the performance of each of them. We focus on the PLS-12, which is the greatest dimension among the ones examined in this work so that advantages and limitations for each method can easily emerge. We refer as NEGATIVE, BCE and MSE to the methods which respectively employ the SBR inspired loss functions described in eq. (2), eq. (3) and eq. (4).

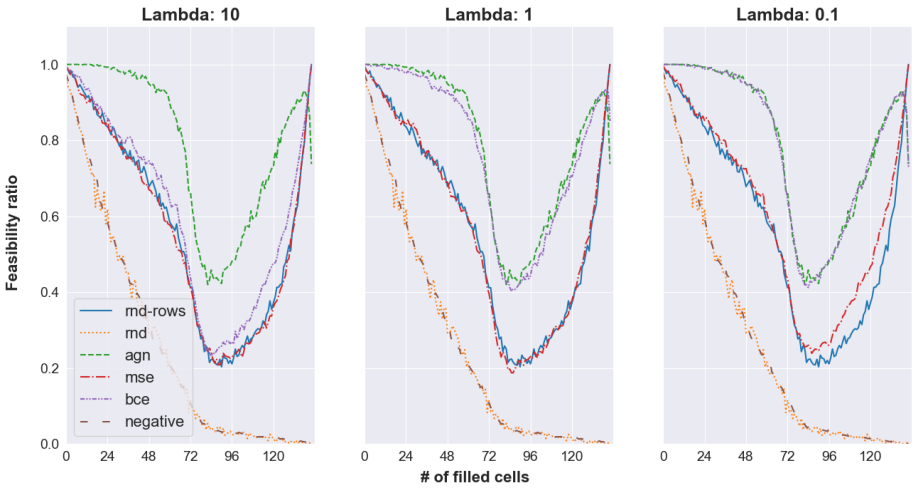

Fig. 2. Effect of the injection of the only rows constraints at training time comparing the regularization methods for different λ values, on the PLS-12. The dataset is generated from 10,000 solutions pool.

The evaluation concerns whether injecting domain knowledge *at training time* may help the NN in the identification of feasible assignments, *assuming the same knowledge is not available at search time*. We also assume in this instance that a large number of historical solutions is available.

This experimentation is motivated by practical situations in which: 1) a domain expert has only partial information about the problem structure, but a pool of historical solutions is available; 2) some constraints (e.g. from differential equations or discrete event simulation) cannot be enforced at search time. In detail, the training set is generated using the deconstruction approach from Section 3, starting from a set of 10,000 PLS solutions, 75% of which are used for training and the remaining ones for testing. Each solution is then deconstructed exactly once, yielding a training set of 1,000,000 examples. An additional validation set of 5,000 partial solutions is adopted to assess the improvements during training via the FEASTEST procedure, using the network as the heuristic h and an empty set of constraints as C (no propagation when choosing the assignment to be checked). Since this computation is really expensive, we perform the assessment every 10 epochs. If for 10 successive checks the best global feasibility ratio found so far is not improved then we stop the training.

For each regularization approach, we train two neural networks: one trained with knowledge about row constraints and another trained with knowledge about row and column constraints. For the first network, we use the SBR-inspired methods (and a Forward Checking propagator) to inject knowledge that both assigning a variable twice and assigning a value twice on the same row is forbidden. For the second one, we do the same, applying the Forward Checking propagator also to column constraints (i.e. no value can appear twice on the

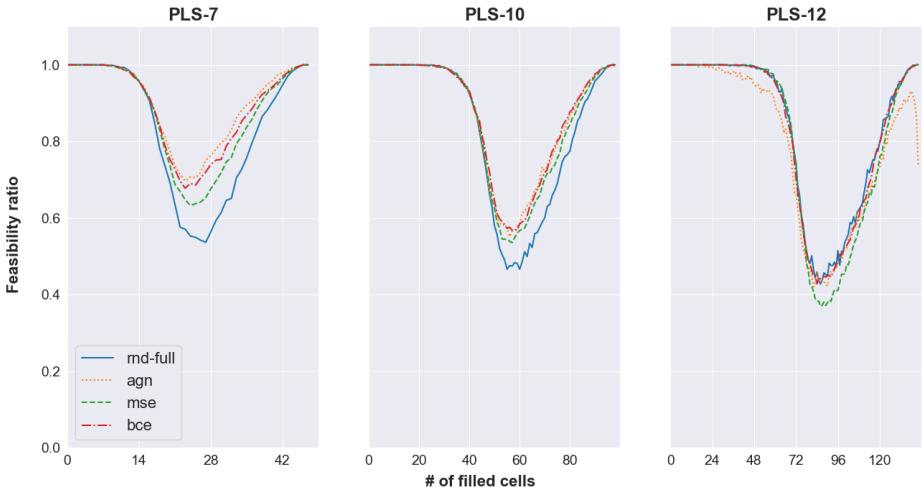

Fig. 3. Full constraints injection at training time on different problem dimensions.

same column). Due to the use of an incomplete propagator, both the networks make use of incomplete knowledge.

In addition, we train a model-agnostic neural network that lacks even the basic knowledge that a variable cannot be assigned twice, since this is not enforced by our input/output encoding, and must infer that from data.

We evaluate the resulting approaches via the FEASTEST procedure, using the separated test set as X, the trained networks as h, and an empty set of constraints (i.e. no propagation at test time). We compare them with methods that randomly choose an assignment with an uniform probability distribution *but that can rely on a set of constraints C during the evaluation*. We consider the two scenarios in which C is the set of the row constraints (RND-ROWS) and the one in which C is the set of column and row constraints (RND-FULL). These methods are representative of the behavior (at each search node) of a Constraint Programming solver having access to either only row constraints or the full problem definition. It allows us to gauge the ideal effect of the offline symbolic knowledge. Finally, we consider a very pessimistic baseline, referred to as RND, which again randomly chooses an assignment with an uniform probability distribution but does not rely on the propagation of any constraints (i.e. C is the empty set). We then produce "feasibility plots" that report on the x-axis the number of assigned variables (filled cells) in the considered partial solutions and on the y-axis the ratio of suggested assignments that are globally feasible. Since RND-ROWS and RND-FULL methods are the only ones that can rely on *online* constraints propagation, *we have highlighted them using solid lines*. In fig. 1, we show results when all the constraints are employed by the Forward Checking constraints propagator, whereas in fig. 2 we do not propagate the columns constraints. The balance between learning the constraints from empirical data and the Forward Checking

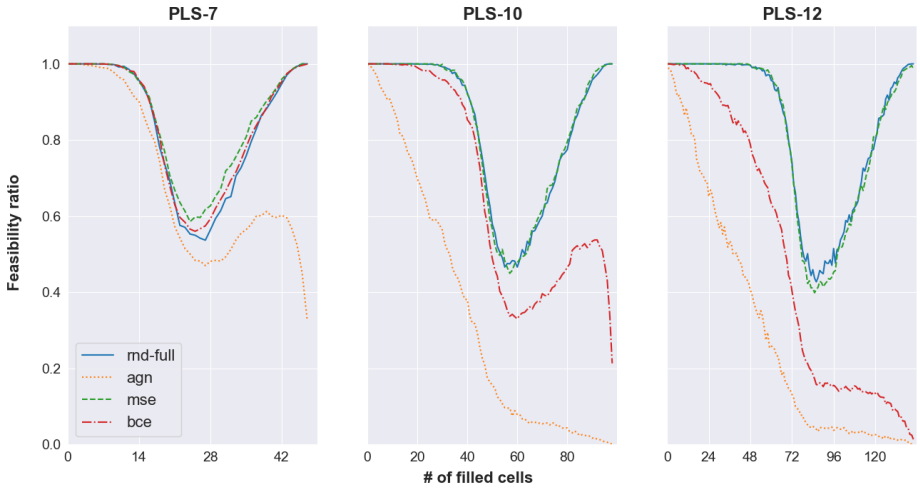

Fig. 4. Full constraints injection at training time when the dataset is reduced to the 10% of its initial size.

propagator is tuned by λ: reducing its value means giving more emphasis on the global feasible assignments obtained by deconstruction of the complete solutions rather than on the incomplete knowledge. We report results for λ equal to 10, 1 and 0.1

For all the λ values, the NEGATIVE approach's behavior is hardly distinguishable from RND. A reasonable explanation is that it encourages the network to keep the output the lowest as possible instead of discouraging the network to make provably infeasible assignments. Since this approach is not effective at all, we do not consider it for further analysis.

We choose the best λ parameters for the BCE and MSE regularization methods with the aim of distilling the constraints propagator in the neural network's weights, finding a tradeoff between learning from correct knowledge and the incomplete one. Considering the overall performance, the MSE regularization method provides better results with $\lambda = 1$, so this value is chosen for the successive analysis. The BCE approach provides the best performance with $\lambda = 10$. Despite in fig. 2 lower values of λ provide better feasibility ratios, these results are not preferable since they make the regularization not effective, i.e. the methods collapse to AGN. The BCE method provides a little improvement over the MSE one but, as we will see when answering question 2, it is not robust when only a limited amount of empirical knowledge is available.

4.2 Domain Knowledge at Training Time for different problem dimensions

Unlike the previous section, here we extend the analysis to the PLS of dimensions 7 and 10, considering the only MSE and BCE regularization methods together

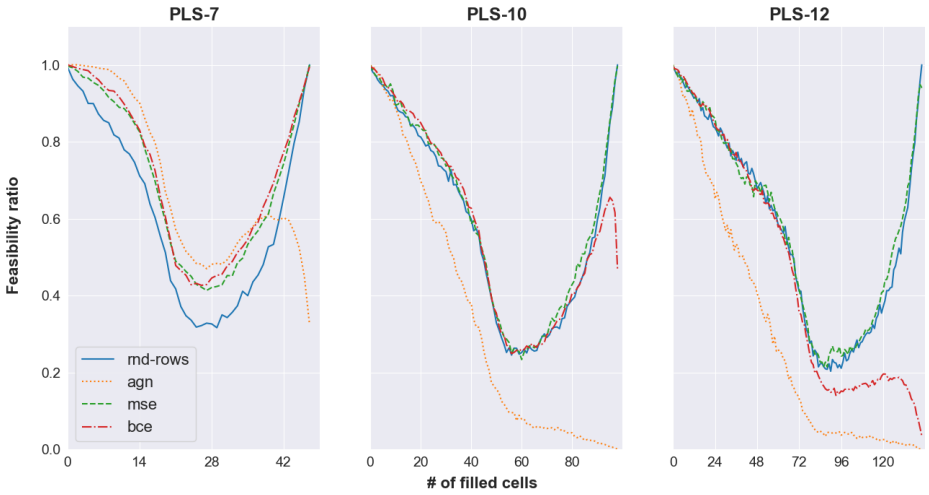

Fig. 5. Rows constraints injection at training time when the dataset is reduced to the 10% of its initial size.

with their best λ values. The datasets are generated as described in the previous section, yielding training sets of size $350,000$ and $700,000$ for respectively the PLS-7 and PLS-10.

In fig. 3, we show results when all the constraints are employed by the forward checking constraints propagator. As long as the problem size is small enough, AGN performs considerably better than RND-FULL, even if no propagation is employed at evaluation time: this is symptomatic of the network actually managing to learn the problem constraints from the available data, which (unlike the propagator output) is guaranteed feasible. As the problem size grows, the gap decreases, until it almost disappears for PLS-12.

For PLS-7, injecting incomplete symbolic knowledge appears to have an *adverse* effect, as it biases the network toward trusting too much the incomplete propagator. With a large problem dimension (i.e. PLS-12) the benefits introduced by knowledge injection become more visible, especially when using the BCE regularization method. The decreasing performance of the data driven methods is likely a consequence of the training set size staying constant, in the face of a search space that becomes increasingly large. In all cases, the feasibility ratio is high for almost empty and almost full squares, with a noticeable drop when $\sim$60% of the square is filled. The trend may be connected to a known phase transition in the complexity of this problem [13].

4.3 Training Set Size and Empirical Information

Next, we proceed to tackle Question 2, by acting on the training set generation process. In classical Machine Learning approaches, the amount of available information is usually measured via the training set size: this is a reasonable approach

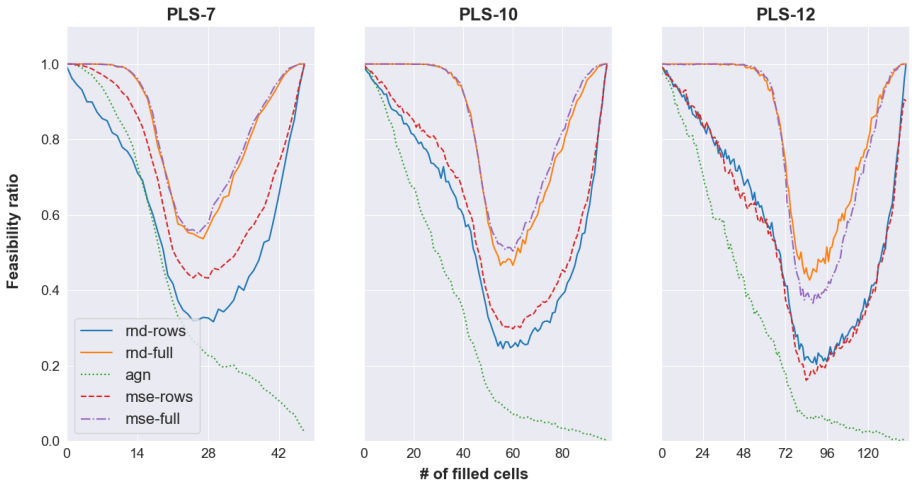

Fig. 6. Effect of reducing the solution pool size from 10,000 to 100.

since the number of training examples has a strong impact on the ability of an ML method to learn and generalize. We performed experiments to probe the effect of the training set size on the performance of the data-driven approaches: the training sets are reduced to the 10% of the initial size, i.e. $35,000$, $70,000$ and $100,000$ for respectively PLS of size 7, 10 and 12. In fig. 4 and fig. 5, we show results when respectively all the constraints and the only rows constraints are injected via the regularization methods. *In this case, knowledge injection at training time has a dramatic effect*: the AGN approach is very sensitive to the available number of examples and it has a great drop in performance. Despite being less pronounced, the BCE method has a major drop in performance too. Instead, the MSE approach provides much more robust results.

In our setup, we have also the possibility to apply the deconstruction process multiple times, so that the number of different examples that can be obtained from a single solution grows with the number of possible permutations of the variable indices (i.e. $O(n^2!)$ for the PLS). The approach opens up the possibility to *generate large training sets from very few starting solutions*. This is scientifically interesting since the "actual" empirical information depends on how many solutions are available; it is also very useful in practice since in many practical applications only a relatively small number of historical solutions exists.

The results of this evaluation are shown in Figure 6 for a solution pool of 100 elements, rather than the original 10,000. Due to the bad results provided with the reduced datasets, we do not further investigate the BCE regularization approach but we examine the only MSE method. For this analysis, we collapse the feasibility results of the neural network trained with full knowledge injection (referred to as MSE-FULL) and of the network trained without the columns constraints knowledge injection (MSE-ROWS) in a single plot. The size of the

generated training set is comparable to the original. Despite the dramatically reduced number of training solutions, the MSE-ROWS and MSE-FULL methods perform really close to respectively RND-ROWS and RND-FULL, i.e. they behave similarly to what the propagator would if employed at search time. Instead, the performance of the AGN drops dramatically, stressing again its sensitivity to the available empirical information.

From a practical point of view, it seems that injecting constraints during training can be a very effective strategy when only a small number of training solutions is available. Constraint injection tends to be redundant if the same type of propagation can be performed at search time, but can be very useful in cases when this is not possible.

4.4 Constraint Violation Assessment

In the last set of our experiments, we investigate the effectiveness of the trained NNs at guiding a search process toward solutions that are close to being feasible, but not necessarily so. This is equivalent to treating constraints as soft and may be of practical relevance on overconstrained problems (e.g. many real-world timetabling applications). This setup tends to be more challenging for the ML models, since chains of variable-value assignments may lead to partial solutions that are remarkably different from those observed at training time.

In detail, we used each trained neural network as a value selection heuristic in Depth First Search, once again for PLS of sizes 7, 10 and 12; we used for this experiment a fixed variable ordering. As a baseline for the comparison, we consider (uniformly) random value selection referred to as RND, while for the NNs we select a random value with probability proportional to the network output. We generate a fixed number of solutions (500) from an empty square, rather than starting from partially filled ones. When generating the solutions, we never propagate the entirety of the PLS constraints: this setup serves as a controlled experiment for use cases where some constraints are either unknown or cannot be enforced at search time. We measure the degree of feasibility of the generated solutions by quantifying the violations for the constraints that were not propagated at search time. For this purpose, we measure violations by counting how many times a value is not appearing exactly once in the same row or column, depending on which constraint is being considered.

We train two model-agnostic neural networks: one on the dataset obtained by random deconstruction of 10,000 solutions (referred to as AGN-10K) and the other one on the dataset obtained by multiple random deconstructions of 100 solutions (referred to as AGN-100). Similarly, we train two neural networks with knowledge injection at *training time* of all the constraints by means of the mean squared error version of the SBR-inspired method and the Forward Checking propagator (referred to as SBR-10K and SBR-100). Neither row nor column constraints are propagated during the search, and therefore we count the violations of both in the final solutions. Results are shown in table 1: *the SBR-inspired approach allows to significantly reduce the number of violations, and it achieves very similar results even when only a small amount of empirical knowledge is*

Table 1. Number of soft constraints violations per generated solution.

	rnd		**agn-10k**		**sbr-10k**		**agn-100**		**sbr-100**	
	rows	cols	rows	cols	rows	cols	rows	cols	rows	cols
PLS-7	29	29	11	9	4	3	20	20	4	4
PLS-10	61	61	28	25	8	7	52	53	7	7
PLS-12	88	88	56	53	22	30	70	76	17	20

available. The AGN approach performs considerably better than RND, as long as a large pool of solutions is available, but the gap narrows when trained on examples generated from 100 solutions. It is interesting to see how, when constraints are interpreted in a soft fashion, injecting full problem knowledge at training time has a much more robust effect compared to the analysis in section 4.2.

5 Conclusion

We considered injecting domain knowledge in Deep Neural Networks to account for domain knowledge that cannot be easily enforced at search time. We chose the PLS as a case study and extended an existing NN approach to enable knowledge injection. We performed controlled experiments to investigate three main questions, drawing the following conclusions:

Q1: As long as enough empirical data is available w.r.t. the problem size, an agnostic data-driven approach can be better at identifying feasible assignments than random choice supported by propagation at search time. However, the performance gap narrows quickly as the problem size grows. Injecting incomplete domain problem knowledge at training time does not appear to provide reliable advantages.

Q2: A pure data-driven approach is very sensitive to the available empirical information. Injecting knowledge at training time significantly improves robustness: if both row and column constraints are considered, only a limited performance drop is observed with as few as 100 historical solutions.

Q3: If constraints are relaxed and treated as soft, injecting domain knowledge can be very effective.

As a side product of our analysis, we have formulated and tested different regularization approaches to develop an SBR-inspired method to constraint propagators into a source of training-time information, plus a technique to extract multiple training examples from a few historical solutions. An open question and future research direction is the experimentation with different problem types to make sure that our results hold in general.

References

1. Adorf, H.M., Johnston, M.D.: A discrete stochastic neural network algorithm for constraint satisfaction problems. In: Proc. of IJCNN. pp. 917–924 vol.3 (June 1990). https://doi.org/10.1109/IJCNN.1990.137951
2. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. arXiv preprint arXiv:1611.09940 (2016)
3. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. arXiv preprint arXiv:1811.06128 (2018)
4. Bessiere, C., Koriche, F., Lazaar, N., O'Sullivan, B.: Constraint acquisition. Artificial Intelligence **244**, 315–342 (2017). https://doi.org/10.1016/j.artint.2015.08.001, https://doi.org/10.1016/j.artint.2015.08.001
5. Bouhouch, A., Chakir, L., Qadi, A.E.: Scheduling meeting solved by neural network and min-conflict heuristic. In: Proc. of IEEE CIST. pp. 773–778 (Oct 2016). https://doi.org/10.1109/CIST.2016.7804991
6. Cauwelaert, S.V., Lombardi, M., Schaus, P.: Understanding the potential of propagators. In: Michel, L. (ed.) Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9075, pp. 427–436. Springer (2015). https://doi.org/10.1007/978-3-319-18008-3_29, https://doi.org/10.1007/978-3-319-18008-3_29
7. Daniele, A., Serafini, L.: Neural networks enhancement through prior logical knowledge. arXiv preprint arXiv:2009.06087 (2020)
8. Diligenti, M., Gori, M., Sacca, C.: Semantic-based regularization for learning and inference. Artificial Intelligence **244**, 143–165 (2017)
9. Diligenti, M., Gori, M., Saccà, C.: Semantic-based regularization for learning and inference. Artificial Intelligence **244**, 143 – 165 (2017). https://doi.org/https://doi.org/10.1016/j.artint.2015.08.011, http://www.sciencedirect.com/science/article/pii/S0004370215001344, combining Constraint Solving with Mining and Learning
10. Evans, R., Grefenstette, E.: Learning explanatory rules from noisy data. Journal of Artificial Intelligence Research **61**, 1–64 (2018)
11. Fischetti, M., Jo, J.: Deep neural networks as 0-1 mixed integer linear programs: A feasibility study. In: Proc. of CPAIOR (2018)
12. Galassi, A., Lombardi, M., Mello, P., Milano, M.: Model agnostic solution of csps via deep learning: A preliminary study. In: van Hoeve, W.J. (ed.) Proc. of CPAIOR. pp. 254–262. Springer International Publishing, Cham (2018)
13. Gomes, C.P., Selman, B., et al.: Problem structure in the presence of perturbations. AAAI/IAAI **97**, 221–226 (1997)
14. Kool, W., Hoof, H., Welling, M.: Attention solves your tsp, approximately. Statistics **1050**, 22 (2018)
15. Lin, G., Shen, C., Van Den Hengel, A., Reid, I.: Efficient piecewise training of deep structured models for semantic segmentation. In: Proc. of the IEEE CVPR. pp. 3194–3203 (2016)
16. 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, https://doi.org/10.1016/j.artint.2016.01.005
17. Ma, X., Hovy, E.: End-to-end sequence labeling via bi-directional lstm-cnns-crf. In: Proc. of ACL. pp. 1064–1074. Association for Computational Linguistics (2016). https://doi.org/10.18653/v1/P16-1101, http://aclweb.org/anthology/P16-1101

18. Manhaeve, R., Dumančić, S., Kimmig, A., Demeester, T., De Raedt, L.: Deepproblog: Neural probabilistic logic programming. arXiv preprint arXiv:1805.10872 (2018)
19. Marra, G., Giannini, F., Diligenti, M., Gori, M.: Integrating learning and reasoning with deep logic models. In: Proc. of ECML (2019)
20. Mišić, V.V.: Optimization of tree ensembles. arXiv preprint arXiv:1705.10883 (2017)
21. Richardson, M., Domingos, P.: Markov logic networks. Machine learning **62**(1-2), 107–136 (2006)
22. Rocktäschel, T., Riedel, S.: End-to-end differentiable proving. In: Advances in Neural Information Processing Systems. pp. 3788–3800 (2017)
23. Serafini, L., Garcez, A.d.: Logic tensor networks: Deep learning and logical reasoning from data and knowledge. arXiv preprint arXiv:1606.04422 (2016)
24. Van Krieken, E., Acar, E., Van Harmelen, F.: Semi-supervised learning using differentiable reasoning. Journal of Applied Logic (2019), to Appear
25. Verwer, S., Zhang, Y., Ye, Q.C.: Auction optimization using regression trees and linear models as integer programs. Artificial Intelligence **244**(Supplement C), 368 – 395 (2017). https://doi.org/https://doi.org/10.1016/j.artint.2015.05.004, http://www.sciencedirect.com/science/article/pii/S0004370215000788, combining Constraint Solving with Mining and Learning
26. Xu, H., Koenig, S., Kumar, T.S.: Towards effective deep learning for constraint satisfaction problems. In: Proc. of CPAIOR. pp. 588–597. Springer (2018)
27. Xu, J., Zhang, Z., Friedman, T., Liang, Y., Broeck, G.: A semantic loss function for deep learning with symbolic knowledge. In: International Conference on Machine Learning. pp. 5502–5511. PMLR (2018)