Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon*

Yoshua Bengio[2,3], Andrea Lodi[1,3], and Antoine Prouvost[1,3]

`yoshua.bengio@mila.quebec`
`{andrea.lodi, antoine.prouvost}@polymtl.ca`

[1]Canada Excellence Research Chair in Data Science for Decision Making, École Polytechnique de Montréal
[2]Department of Computer Science and Operations Research, Université de Montréal
[3]Mila, Quebec Artificial Intelligence Institute

Abstract

This paper surveys the recent attempts, both from the machine learning and operations research communities, at leveraging machine learning to solve combinatorial optimization problems. Given the hard nature of these problems, state-of-the-art methodologies involve algorithmic decisions that either require too much computing time or are not mathematically well defined. Thus, machine learning looks like a promising candidate to effectively deal with those decisions. We advocate for pushing further the integration of machine learning and combinatorial optimization and detail methodology to do so. A main point of the paper is seeing generic optimization problems as data points and inquiring what is the relevant distribution of problems to use for learning on a given task.

1 Introduction

Operations research (OR), also referred to as prescriptive analytics, started in the first world war as an initiative to use mathematics and computer science to assist military planners in their decisions. Nowadays it is widely

*Submitted to European Journal of Operations Research.

used in the industry, including but not limited to transportation, supply chain, energy, finance, and scheduling. In this paper, we focus on OR problems formulated as integer constrained optimization, *i.e.* with integral or binary variables (called decision variables). While not all such problems are hard to solve (*e.g.* shortest path problems), we concentrate on combinatorial (NP-hard) problems. This is a worst case complexity for a general purpose problem, although in practice it is possible to solve instances with up to a million decision variables and constraints. This is in part due to a rich set of techniques which have been developed in the past.

Machine learning (ML), on the other hand, focuses on performing a task given some (finite and usually noisy) data. It is well suited for natural signals for which no clear mathematical formulation emerges because the true data distribution is not known analytically, such as when processing images, text, voice or molecules, or with recommender systems, social networks or financial predictions. Most of the times, the learning problem has a statistical formulation that is solved through mathematical optimization. Recently, dramatic progress has been achieved with deep learning, a ML sub-field building large parametric approximators by composing simpler functions. Deep learning excels when applied in high dimensional spaces with a large number of data points.

OR and ML are closely related, especially through optimization, *e.g.* minimizing the error between predictions and targets (see Section 2.2 for details). They can overlap, or complement in some areas. In this paper, we focus on the use of machine learning to build combinatorial OR optimization algorithms. To solve a combinatorial problem, an expert would have an intuition on which of the many approaches to use. Given that these problems are highly structured, we believe that expert intuition can be automated and augmented through machine learning – and especially deep learning to address the high dimensionality of such problems. In the following, we survey the attempts in the literature to achieve such automation and augmentation, and we present a methodological overview of those approaches.

1.1 Motivation

From the combinatorial optimization (CO) point of view, machine learning can serve two purposes. For the first one, the researcher assumes expert knowledge[1] about the optimization algorithm, but wants to replace some heavy computations by a fast approximation. Learning can be used to build

[1]Theoretical and/or empirical.

such approximations in a generic way, *i.e.* without the need to derive new explicit algorithms. Regarding the second motivation for using ML, expert knowledge may not be sufficient and some algorithmic decisions may be unsatisfactory. The goal is therefore to explore the space of these decisions, and learn out of this experience the best performing behavior (policy), hopefully improving on the state of the art. Even though ML is approximate, we will demonstrate through the examples surveyed in this paper that this does not systematically mean that incorporating learning will compromise overall theoretical guarantees.

From the point of view of using ML to tackle a combinatorial problem, CO can decompose the problem into smaller, hopefully simpler, learning tasks. The CO structure therefore acts as a relevant prior for the model. It is also an opportunity to leverage the CO literature, notably in terms of theoretical guarantees (*e.g.* feasibility and optimality).

Current ML algorithms can generalize[2] to examples from the same distribution but tend to have more difficulty generalizing out of distribution.[3] Mixed algorithms, *i.e.* algorithms that mix traditional CO components with ML models, do depend on the data used for learning. Therefore, they also are subject to generalization issues and can be expected to fail when the use cases are too far from what has been used for training the ML predictor. Although it is one of the main challenges reported in this paper, this behavior also provides an opportunity for the learning to specialize to sub-problems by fully exploiting their structure. Indeed, in many practical cases, the interest does not lie on solving any possible problem, for example all those representable by mixed-integer linear programmings (MILPs), but rather a smaller family of problems relevant to the application at hand, say the traveling salesman problem (TSP).[4] It is also worth noting that traditional CO algorithms might not even work consistently across all possible instances of a problem family, but rather tend to be more adapted to particular structures of problems, *e.g.* Euclidean TSPs.[5]

We end the section by noting that most of the literature we survey is exploratory, *i.e.* we aim at highlighting promising research directions in the

[2] Meaning the performance of the task (*e.g.* accuracy for classification) on unseen examples is similar to the performance on the examples used to learn the model.

[3] Defining what is in or out of the distribution when working with natural samples is often qualitative.

[4] A NP-hard problem defined on a graph: finding a cycle of minimum length visiting once and only once every node.

[5] Each node is assigned coordinates in a vector space or a cube and the cost of the edges is the Euclidian distance between the two nodes. Typically, the two dimensional unit cube in the plane is used.

use of ML within CO, instead of reporting on already mature algorithms.

1.2 Outline

We have introduced the context and motivations for building combinatorial optimization algorithms together with machine learning. The remainder of this paper is organized as follows. Section 2 provides minimal prerequisites in combinatorial optimization, machine learning, deep learning, and reinforcement learning necessary to fully grasp the content of the paper. Section 3 surveys the recent literature and derives two distinctive, orthogonal, views: Section 3.1 shows how machine learning policies can either be learned by imitating an expert or discovered through experience, while Section 3.2 discusses the interplay between the ML and CO components. Section 4 pushes further the reflection on the use of ML for optimization and brings to the fore some methodological points. In Section 5 we detail critical practical challenges of the field. Finally, some conclusions are drawn in Section 6.

2 Preliminaries

In this section, we give a basic (sometimes rough) overview of combinatorial optimization and machine learning, with the unique aim of introducing concepts that are strictly required to understand the remainder of the paper.

2.1 Combinatorial Optimization

Without loss of generality, a CO problem can be formulated as a constrained min-optimization program. Constraints model natural or imposed restrictions of the problem, variables define the decisions to be made, while the objective function, generally a cost to be minimized, defines the measure of the quality of every feasible assignment of values to variables. If the objective and constraints are linear, the problem is called a linear programming (LP) problem. If, in addition, some variables are also restricted to only assume integer values, then the problem is a mixed-integer linear programming (MILP) problem.

The set of points that satisfy the constraints is the feasible region. Every point in that set (often referred to as feasible solution) yields an upper bound on the objective value of the optimal solution. Exact solving is an important aspect of the field, hence a lot of attention is also given to find lower bounds to the optimal cost. The tighter the lower bounds, with respect to the optimal solution value, the higher the chances that the current algorithmic approaches to tackle MILPs described in the following could be successful, *i.e.* effective if not efficient.

Linear and mixed-integer linear programming problems are the workhorse of CO because they can model a wide variety of problems and are the best understood, *i.e.* there are reliable algorithms and software tools to solve them. We give them special considerations in this paper but, of course, they do not represent the entire CO, mixed-integer nonlinear programming being a rapidly-expanding and very significant area both in theory and in practical applications. With respect to complexity and solution methods, LP is a polynomial problem, well solved, in theory and in practice, through the simplex algorithm or interior points methods. Mixed-integer linear programming, on the other hand, is an NP-hard problem, which does not make it hopeless. Indeed, it is easy to see that the complexity of MILP is associated with the integrality requirement on (some of) the variables, which makes the MILP feasible region nonconvex. However, dropping such integrality requirement (i) defines a proper relaxation of MILP (*i.e.* an optimization problem whose feasible region contains the MILP feasible region), which (ii) happens to be an LP, *i.e.* polynomially solvable. This immediately suggests the algorithmic line of attack that is used to solve MILP through a whole ecosystem of branch-and-bound (B&B) techniques to perform implicit enumeration. Branch and bound implemements a devide-and-conquer type of algorithm representable by a search tree in which, at every node, an LP relaxation of the problem (possibly augmented by branching decisions, see below) is efficiently computed. If the relaxation is infeasible, or if the solution of the relaxation is naturally (mixed-)integer, *i.e.* MILP feasible, the node does not need to be expanded. Otherwise, there exists at least one variable, among those supposed to be integer, taking a fractional value in the LP solution and that variable can be chosen for branching (enumeration), *i.e.* by restricting its value in such a way that two child nodes are created. The two child nodes have disjoint feasible regions, none of which containing the solution of the previous LP relaxation. We use Figure 1 to illustrate the B&B algorithm for a minimization MILP. At the root node in the figure, the variable x_2 has a fractional value in the LP solution (not represented), thus branching is done on the floor (here zero) and ceiling (here one) of this value. When an integer solution is found, we also get an upper bound (denoted as $\overline{z}$) on the optimal solution value of the problem. At every node, we can then compare the solution value of the relaxation (denoted as $\underline{z}$) with the minimum upper bound found so far, called incumbent solution value. If the latter is smaller than the former for a specific node, no better (mixed-)integer solution can be found in the sub-tree originated by the node itself, and it can be pruned.

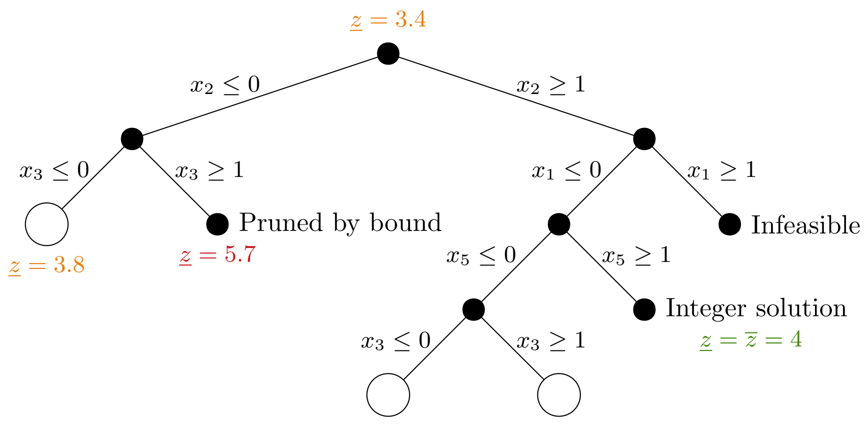

Figure 1: A branch-and-bound tree for MILPs. The LP relaxation is computed at every node (only partially shown in the figure). Nodes still open for exploration are represented as blank.

All MILP commercial and noncommercial solvers enhance the above enumeration framework with the extensive use of cutting planes, *i.e.* valid linear inequalities that are added to the original formulation (especially at the root of the B&B tree) in the attempt of strengthening its LP relaxation. The resulting framework, referred to as branch-and-cut algorithm, is then further enhanced by additional algorithmic components, preprocessing and primal heuristics being the most crucial ones.

The reader is referred to Wolsey (1998) and Conforti et al. (2014) for extensive textbooks on MILP and to Lodi (2009) for a detailed description of the algorithmic components of the MILP solvers.

2.2 Machine Learning

In supervised learning, a set of input (features) / target pairs is provided and the task is to find a function that for every input has a predicted output as close as possible to the provided target. Finding such a function is called learning and is solved through an optimization problem over a family of functions. The loss function, *i.e.* the measure of discrepancy between the output and the target, can be chosen depending on the task (regression, classification, *etc.*) and on the optimization methods. However, this approach is not enough because the problem has a statistical nature. It is usually easy enough to achieve a good score on the given examples but one wants to achieve a good score on unseen examples (test data). This is known as generalization. If a model (*i.e.* a family of functions) can represent many different functions, the model is said to have high capacity and is prone to over-fitting: doing well on the training data but not generalizing to the test data. Regularization is anything that can improve the test score at the expense of the training score and is used to restrict the practical capacity of a model. On the contrary, if the capacity is too low, the model under-fits and performs poorly on both sets.

In unsupervised learning, one does not have targets for the task one wants to solve, but rather tries to capture some characteristics of the joint distribution of the observed random variables. The variety of tasks include density estimation, dimensionality reduction, and clustering. Because unsupervised learning has received so far little attention in conjunction with CO and its immediate use seems difficult, we are not discussing it any further.

The reader is referred to Bishop (2006) and Murphy (2012) for textbooks on machine learning.

In reinforcement learning (RL), an agent interacts with an environment through a markov decision process (MDP), as illustrated in Figure 2. At every time step, the agent is in a given state of the environment and chooses an action according to its (possibly stochastic) policy. As a result, it receives from the environment a reward and enters a new state. The goal in RL is to train the agent to maximize the sum of future rewards, called the return. The problem can be formulated as dynamic programming, and solved approximately. The dynamics of the environment need not be known by the agent and are learned directly or indirectly, yielding an exploration *vs* exploitation dilemma: choosing between exploring new states for refining the knowledge of the environment for possible long-term improvements, or exploiting the best-known scenario learned so far (which tends to be in already visited or predictable states).

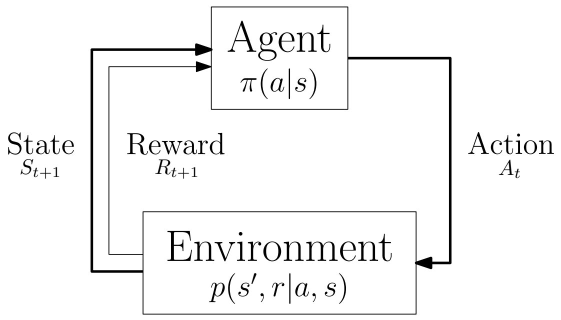

Figure 2: The Markov decision process associated with reinforcement learning, modified from Sutton and Barto (1998). The agent behavior is defined by its policy π, while the environment evolution is defined by the dynamics p. Note that the reward is not necessary to define the evolution and is provided only as a learning mechanism for the agent. Actions, states, and rewards are random variables in the general framework.

The state should fully characterize the environment at every step. When this is not the case, similar methods can be applied but we say that the agent receives an *observation* of the state. The Markov property no longer holds and the MDP is said to be partially observable.

Defining a reward function is not always easy. Sometimes one would like to define a very sparse reward, such as 1 when the agent solves the problem, and 0 the rest of the times. Because of its underlying dynamic programming process, RL is naturally able to credit states/actions that lead to future rewards. Nonetheless, the aforementioned setting is challenging as it provides no learning opportunity until the agent (randomly, or through advanced approaches) solves the problem. Furthermore, when the policy is approximated (for instance, by a linear function), the learning is not guaranteed to converge and may fall into local minima. For example, an autonomous car may decide not to drive anywhere for fear of hitting a pedestrian and receiving a dramatic negative reward. These challenges are strongly related to the aforementioned exploration dilemma.

The reader is referred to Sutton and Barto (1998) for an extensive textbook on reinforcement learning.

Deep learning is a successful method for building parametric composable functions in high dimensional spaces. In the simple case of a feedforward neural network (also called a multilayer perceptron (MLP)), the input data is successively passed through a number of layers. For every layer, an affine transformation is applied on the input vector, followed by a non-linear scalar function (named activation function) applied element-wise. The output of a layer, called intermediate activations, is passed on to the next layer. All affine transformations are independent and represented in practice as different matrices of coefficients. They are learned, *i.e.* optimized over, through stochastic gradient descent (SGD), the optimization algorithm used to minimize the selected loss function. The stochasticity comes from the limited number of data points used to compute the loss before applying a gradient update. In practice, gradients are computed using reverse mode automatic differentiation, a practical algorithm based on the chain rule, also known as back-propagation. Deep neural networks can be difficult to optimize, and a large variety of techniques have been developed to make the optimization behave better, often by changing architectural designs of the network. Because neural networks have dramatic capacities, *i.e.* they can essentially match any dataset, thus being prone to over-fitting, they are also heavily regularized. In addition, many hyper-parameters exist and different combinations are evaluated (known as hyper-parameters optimization). Deep learning also sets itself apart from more traditional ML techniques by taking as inputs all available raw features of the data, *e.g.* all pixels of an image, while traditional ML engineered a limited number of domain specific features.

Deep learning researchers have developed different techniques to tackle this variety of structured data in a manner that is invariant to size. For instance, recurrent neural networks (RNNs) can operate on sequence data by *sharing parameters* across different sequence steps. More precisely, a same neural network is successively applied at every step of the sequence. The specificity of such a network is the presence of recurrent layers: layers that take as input both the activation vector of the previous layer and its own activation vector on the preceding sequence step (called hidden state vector), as illustrated in Figure 3.

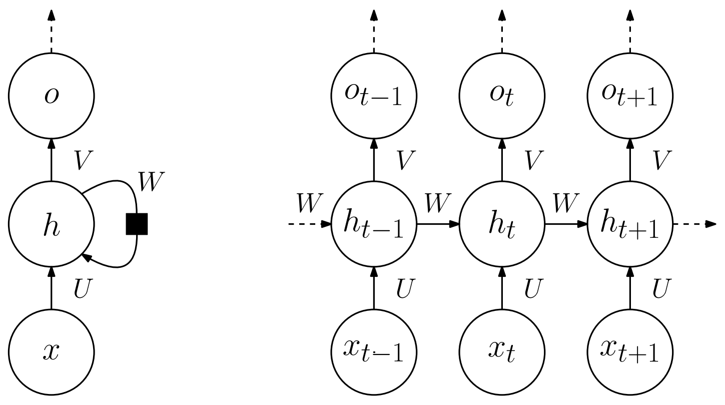

Figure 3: A simple RNN modified from Goodfellow et al. (2016). On the left, the black square indicates a one step delay. On the right, the same RNN is shown unfolded. Three sets U, V, and W of parameters are represented and re-used at every time step.

Attention mechanism is another size-invariant technique used to process data where each data point corresponds to a set. In that context, parameter sharing is used to address the fact that different sets need not to be of the same size. Attention is used to query information about all elements in the set, and merge it in a neural network, as depicted in Figure 4. An affinity function takes as input the query and a representation of an element of the set (both are activation vectors) and outputs a scalar. This is repeated over all elements in the set for the same query. Those scalars are normalized (for instance with a softmax function) and used to define a weighted sum of the representations of elements in the set that can, in turn, be used in the neural network making the query. A more general explanation of attention mechanisms is given by Vaswani et al. (2017). Attention can be used to build graph neural networks (GNNs), *i.e.* neural networks able to process graph structured input data, as done by Veličković et al. (2018). In this architecture, every node attends over the set of its neighbors. The process is repeated multiple times to gather information about nodes further away. GNNs can also be understood as a form of message passing (Gilmer et al., 2017).

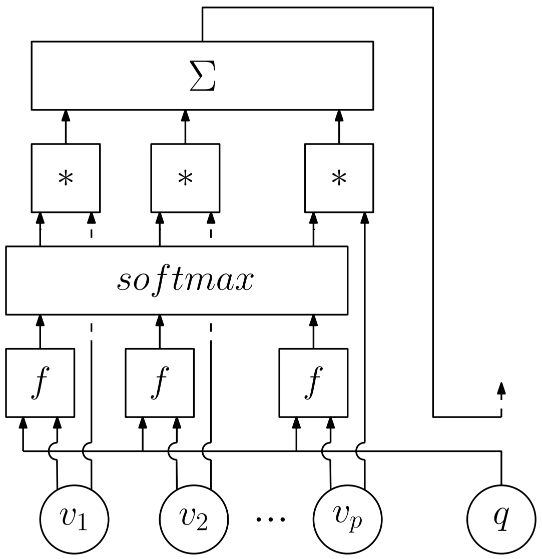

Figure 4: A simple attention mechanism where a query q is computed against a set of values $(v_i)_i$. An affinity function f, such as a dot product is used. If it includes some parameters, the mechanism can be learned.

Deep learning can be used in supervised, unsupervised, or reinforcement learning. The reader is referred to Goodfellow et al. (2016) for a textbook on deep learning.

3 Recent approaches

We survey different uses of ML to help solve combinatorial optimization problems and organize them along two orthogonal axes. First, in Section 3.1 we illustrate the two main motivations for using learning: approximation and discovery of new policies. Then, in Section 3.2, we show examples of different ways to combine learned and traditional algorithmic elements.

3.1 Learning methods

This section relates to the two motivations reported in Section 1.1 for using ML in CO. In some works, the researcher assumes theoretical and/or empirical knowledge about the decisions to be made for a CO algorithm, but wants to alleviate the computational burden by approximating some of those decisions with machine learning. On the contrary, we are also motivated by the fact that, sometimes, expert knowledge is not satisfactory and the researcher wishes to find better ways of making decisions. Thus, ML can come into play to train a model through trial and error.

We frame both these motivations in the state/action MDP framework introduced in section 2.2, where the environment is the internal state of the algorithm. We care about learning algorithmic decisions utilized by a CO algorithm and we call the function making the decision a *policy*, that, given all available information,[6] returns (possibly stochastically) the action to be taken. The policy is the function that we want to learn using ML and we show in the following how the two motivations naturally yield two learning settings.

In the case of using ML to approximate decisions, the policy is often learned by *imitation learning*, thanks to *demonstrations*, because the expected behavior is shown (demonstrated) to the ML model by an expert (also called oracle, even though it is not necessarily optimal), as shown in Figure 5.

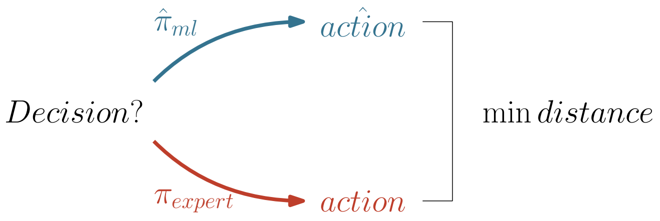

Figure 5: In the demonstration setting, the policy is trained to reproduce the action of an expert policy by minimizing some discrepancy in the action space.

In the case where one cares about discovering new policies, *i.e.* optimizing an algorithmic decision function from the ground up, the policy is learned by reinforcement learning through *experience*, as shown in Figure 6.

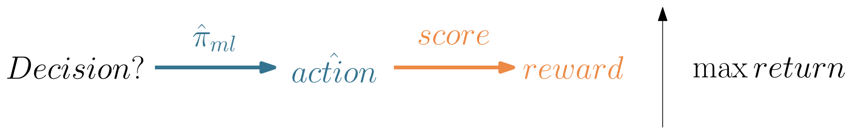

Figure 6: When learning through a reward signal, no expert is involved; only maximizing the expected sum of future rewards (the return) matters.

[6] A *state* if the information is sufficient to fully characterize the environment at that time in a Markov decision process setting.

It is critical to understand that in the imitation setting, the policy is learned through supervised targets provided by an expert for every action (and without a reward), whereas in the experience setting, the policy is learned from a reward (possibly delayed) signal using RL (and without an expert). In imitation, the agent is taught *what* to do, whereas in RL, the agent is encouraged to quickly *accumulate* rewards. The distinction between these two settings is far more complex than the distinction made here. We explore some of this complexity, including their strengths and weaknesses, in Section 4.1.

In the following, few papers demonstrating the different settings are surveyed.

3.1.1 Demonstration

In Baltean-Lugojan et al. (2018), the authors use a neural network to approximate the lower bound improvement generated by tightening the current relaxation via cutting planes (cuts, for short). More precisely, Baltean-Lugojan et al. (2018) consider non-convex quadratic programming problems and aim at approximating by a linear program the associated semidefinite programming (SDP) relaxation, known to be strong but time-consuming. A straightforward way of doing that is to iteratively add (linear) cutting planes associated with negative eigenvalues, especially considering small-size (square) submatrices of the original quadratic objective function. That approach has the advantage of generating sparse cuts[7] but it is computationally challenging because of the exponential number of those submatrices and because of the difficulty of finding the right metrics to select among the violated cuts. The authors propose to solve SDPs to compute the bound improvement associated with considering specific submatrices, which is also a proxy on the quality of the cuts that could be separated from the same submatrices. In this context, supervised (imitation) learning is applied offline to approximate the objective value of the SDP problem associated with a submatrix selection and, afterward, the model can be rapidly applied to select the most promising submatrices without the very significant computational burden of solving SDPs. Of course, the rational is that the most promising submatrices correspond to the most promising cutting planes and Baltean-Lugojan et al. (2018) train a model to estimate the objective of an SDP problem only in order to decide to add the most promising cutting planes. Hence, cutting plane selection is the ultimate policy learned.

[7] The reader is refereed to Dey and Molinaro (2018) for a detailed discussion on the interest of sparse cutting planes in MILP.

Another example of demonstration is found in the context of branching policies in B&B trees of MILPs. The choice of variables to branch on can dramatically change the size of the B&B tree, hence the solving time. Among many heuristics, a well-performing approach is *strong branching* (Applegate et al., 2007). Namely, for every branching decision to be made, strong branching performs a one step look-ahead by tentatively branching on many candidate variables, computes the LP relaxations to get the potential lower bound improvements, and eventually branches on the variable providing the best improvement. Even if not all variables are explored, and the LP value can be approximated, this is still a computationally expensive strategy. For these reasons, Marcos Alvarez et al. (2014, 2017) use a special type of decision tree (a classical model in supervised learning) to approximate strong branching decisions using supervised learning. Khalil et al. (2016) propose a similar approach, where a linear model is learned on the fly for every instance by using strong branching at the top of the tree, and eventually replacing it by its ML approximation. The linear approximator of strong branching introduced in Marcos Alvarez et al. (2016) is learned in an active fashion: when the estimator is deemed unreliable, the algorithm falls back to true strong branching and the results are then used for both branching and learning. In all the branching algorithms presented here, inputs to the ML model are engineered as a vector of fixed length with static features descriptive of the instance, and dynamic features providing information about the state of the B&B process. Node selection, *i.e.* deciding on the next branching node to explore in a B&B tree, is also a critical decision in MILP. He et al. (2014) learn a policy to select among the open branching nodes the one that contains the optimal solution in its sub-tree. The training algorithm is an online learning method collecting expert behaviors throughout the entire learning phase. The reader is referred to Lodi and Zarpellon (2017) for an extended survey on learning and branching in MILPs.

As already mentioned at the beginning of Section 3.1, learning a policy by demonstration is identical to supervised learning, where training pairs of input state and target actions are provided by the expert. In the simplest case, expert decisions are collected beforehand, but more advanced methods can collect them online to increase stability as previously shown in Marcos Alvarez et al. (2016) and He et al. (2014).

3.1.2 Experience

Considering the TSP on a graph, it is easy to devise a greedy heuristic that builds a tour by sequentially picking the nodes among those that have not been visited yet, hence defining a permutation. If the criterion for selecting the next node is to take the closest one, then the heuristic is known as the nearest neighbour. This simple heuristic has poor practical performance and many other heuristics perform better empirically, *i.e.* build cheaper tours. Selecting the nearest node is a fair intuition but turns out to be far from satisfactory. Khalil et al. (2017a) suggest learning the criterion for this selection. They build a greedy heuristic framework, where the node selection policy is learned using a GNN (Dai et al., 2016), a type of neural network able to process input graphs of any size by a mechanism of message passing (Gilmer et al., 2017). For every node to select, the authors feed to the network the graph representation of the problem – augmented with features indicating which of the nodes have already been visited – and receive back an action value for every node. Action values are used to train the network through RL (Q-learning in particular) and the partial tour length is used as a reward.

This example does not do justice to the rich TSP literature that has developed far more advanced algorithms performing orders of magnitude better than ML ones. Nevertheless, the point we are trying to highlight here is that given a fixed context, and a decision to be made, ML can be used to discover new, potentially better performing policies. Even on state-of-the-art TSP algorithms (*i.e.* when exact solving is taken to its limits), many decisions are made in heuristic ways, *e.g.* cutting plane selection, thus leaving room for ML to assist in making these decisions.

Once again, we stress that learning a policy by experience is the framework of reinforcement learning, where an agent maximizes the return (defined in Section 2.2). By matching the reward signal with the optimization objective, the goal of the learning agent becomes to solve the problem, without assuming any expert knowledge.

We close this section by noting that demonstration and experience are not mutually exclusive and some learning tasks can be tackled in both ways. In the case of selecting the branching variables in an MILP branch-and-bound tree, one could adopt anyone of the two prior strategies. On the one hand, Marcos Alvarez et al. (2014, 2016, 2017); Khalil et al. (2016) estimate that strong branching is an effective branching strategy but computationally too expensive and build a machine learning model to approximate it. On the other hand, one could believe that no branching strategy is good enough and try to learn one from the ground up, for instance through reinforcement learning as suggested (but not implemented) in Khalil et al. (2016).

3.2 Algorithmic structure

In this section, we survey how the learned policies (whether from demonstration or experience) are combined with traditional CO algorithms, *i.e.* considering ML and explicit algorithms as building blocks, we survey how they can be laid out in different templates. The three following sections are not necessarily disjoint nor exhaustive but are a natural way to look at the literature.

3.2.1 End to end learning

A first idea to leverage machine learning to solve discrete optimization problems is to train the ML model to output solutions directly from the input instance, as shown in Figure 7.

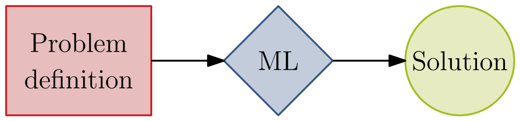

Figure 7: Machine learning acts alone to provide a solution to the problem.

This approach has been explored recently, especially on Euclidean TSPs. To tackle the problem with deep learning, Vinyals et al. (2015) introduce the pointer network wherein an encoder, namely an RNN, is used to parse all the TSP nodes in the input graph and produces an encoding (a vector of activations) for each of them. Afterward, a decoder, also an RNN, uses a mechanism similar to attention (Section 2.2) over the previously encoded nodes in the graph to produce a probability distribution over these nodes (through the softmax layer previously illustrated in Figure 4). Repeating this decoding step, it is possible for the network to output a permutation over its inputs (the TSP nodes). This method makes it possible to use the network over different input graph sizes. The authors train the model through supervised learning with precomputed TSP solutions as targets. Bello et al. (2017) use a similar model and train it with reinforcement learning using the tour length as a reward signal. They address some limitations of supervised (imitation) learning, such as the need to compute optimal (or at least high quality) TSP solutions (the targets), that in turn, may be ill-defined when those solutions are not computed exactly, or when multiple solutions exist. Kool and Welling (2018) introduce more prior in the model using a GNN instead of an RNN to process the input. Emami and Ranka (2018) and Nowak et al. (2017) explore a different approach by directly approximating a double stochastic matrix in the output of the neural network to characterize the permutation. The work of Khalil et al. (2017a), introduced in Section 3.1.2, can also be understood as an end to end method to tackle the TSP, but we prefer to see it under the eye of Section 3.2.3. It is worth noting that tackling the TSP through ML is not new. Earlier work from the nineties focused on Hopfield neural networks and self organizing neural networks, the interested reader is referred to the survey of Smith (1999).

In another example, Larsen et al. (2018) train a neural network to predict the solution of a stochastic load planning problem for which a deterministic MILP formulation exists. Their main motivation is that the application needs to make decisions at a tactical level, *i.e.* under incomplete information, and machine learning is used to address the stochasticity of the problem. The authors use operational solutions, *i.e.* solutions to the deterministic version of the problem, and aggregate them to provide (tactical) solution targets to the ML model. As explained in their paper, the highest level of description of the solution is its cost, whereas the lowest (operational) is the knowledge of values for all its variables. Then, the authors place themselves in the middle and predict an aggregation of variables (tactical) that corresponds to the stochastic version of their specific problem. Furthermore, the nature of the application requires to output solutions in real time, which is not possible either for the stochastic version of the load planning problem or its deterministic variant and by using state-of-the-art MILP solvers. Then, ML turns out to be suitable for obtaining accurate solutions in short computing times because some of the complexity is addressed offline, *i.e.* in the learning phase, and the run-time (inference) phase is by construction extremely quick. Finally, note that in Larsen et al. (2018) an MLP, *i.e.* a simple feedforward neural network, is used to process the input instance as a vector, hence integrating little prior knowledge about the problem structure.

3.2.2 Learning meaningful properties of optimization problems

In many cases, using only machine learning to tackle the problem may not be the most suitable approach. Instead, ML can be applied to provide additional pieces of information to a CO algorithm as illustrated in Figure 8. For example, ML can provide a parametrization of the algorithm (in a very broad sense).

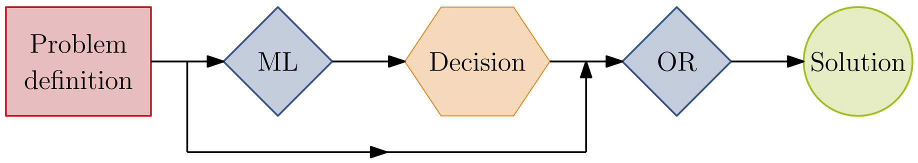

Figure 8: The machine learning model is used to augment an operation research algorithm with valuable pieces of information.

In this context, Kruber et al. (2017) use machine learning on MILP instances to estimate beforehand whether or not applying a Dantzig-Wolf decomposition will be effective, *i.e.* will make the solving time faster. Decomposition methods can be powerful but deciding if and how to apply them depends on many ingredients of the instance and of its formulation and there is no clear cut way of optimally making such a decision. In their work, a data point is represented as a fixed length vector with features representing instance and tentative decomposition statistics. In another example, in the context of mixed-integer quadratic programming, Bonami et al. (2018) use machine learning to decide if linearizing the problem will solve faster. When the quadratic programming (QP) problem given by the relaxation is convex, *i.e.* the quadratic objective matrix is semidefinite positive, one could address the problem by a B&B algorithm that solves QP relaxations[8] to provide lower bounds. Even in this convex case, it is not clear if QP B&B would solve faster than linearizing the problem (by using McCormick (1976) inequalities) and solving an equivalent MILP. This is why ML is a great candidate here to fill the knowledge gap. In both papers (Kruber et al., 2017; Bonami et al., 2018), the authors experiment with different ML models, such as support vector machines and random forests, as is good practice when no prior knowledge is embedded in the model.

As previously stated, the parametrization of the CO algorithm provided by ML is to be understood in a very broad sense. For instance, in the case of radiation therapy for cancer treatment, Mahmood et al. (2018) use ML to produce candidate therapies that are afterward refined by a CO algorithm into a deliverable plan. Namely, a generative adversarial networks (GAN) is used to color CT scan images into a potential radiation plan, then, inverse optimization (Ahuja and Orlin, 2001) is applied on the result to make the plan feasible (Chan et al., 2014). In general, GANs are made of two distinct networks: one (the generator) generates images, and another one (the discriminator) discriminates between the generated images and a dataset

[8] Note that convex QPs can be solved in polynomial time.

of real images. Both are trained alternatively: the discriminator through a usual supervised objective, while the generator is trained to fool the discriminator. In Mahmood et al. (2018), a particular type of GAN (conditional GAN) is used to provide coloring instead of random images. The interested reader is referred to Creswell et al. (2018) for an overview on GANs.

We end this section by noting that a ML model used for learning some representation may in turn use as features pieces of information given by another CO algorithm, such as the decomposition statistics used in Kruber et al. (2017), or the LP information in Bonami et al. (2018).

3.2.3 Machine learning alongside optimization algorithms

To generalize the context of the previous section to its full potential, one can build CO algorithms that repeatedly call a ML model throughout their execution, as illustrated in Figure 9. A master algorithm controls the high-level structure while frequently calling a ML model to assist in lower level decisions. The key difference between this approach and the examples discussed in the previous section is that the *same ML model* is used by the CO algorithm to make the same type of decisions a number of times in the order of the number of iterations of the algorithm. As in the previous section, nothing prevents one from applying additional steps before or after such an algorithm.

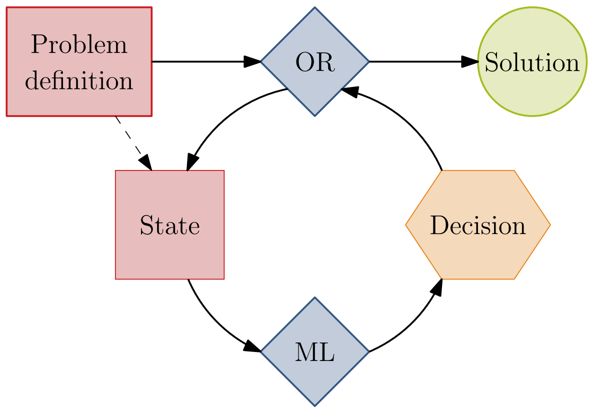

Figure 9: The combinatorial optimization algorithm repeatedly queries the same ML model to make decisions. The ML model takes as input the current state of the algorithm, which may include the problem definition.

This is clearly the context of the branch-and-bound tree for MILP, where we already mentioned how the task of selecting the branching variable is either too heuristic or too slow, and is therefore a good candidate for learning (Lodi and Zarpellon, 2017). In this case, the general algorithm remains a branch-and-bound framework, with the same software architecture and the same guarantees on lower and upper bounds, but the branching decisions made at every node are left to be learned. Another important aspect in solving MILPs is the use of primal heuristics, *i.e.* algorithms that are applied in the B&B nodes to find feasible solutions, without guarantee of success. On top of their obvious advantages, good solutions also give tighter upper bounds (for minimization problems) on the solution value and make more pruning of the tree possible. Heuristics depend on the branching node (as branching fix some variables to specific values), so they need to be run frequently. However, running them too often can slow down the exploration of the tree, especially if their outcome is negative, *i.e.* no better upper bound is detected. Khalil et al. (2017b) build a ML model to predict whether or not running a given heuristic will yield a better solution than the best one found so far and then greedily run that heuristic whenever the outcome of the model is positive.

The approximation used by Baltean-Lugojan et al. (2018), already discussed in Section 3.2.1, is an example of predicting a high-level description of the solution to an optimization problem, namely the objective value. Nonetheless, the ultimate goal is to solve the original QP. Thus, the learned model is queried repeatedly to select promising cutting planes. The ML model is used only to select promising cuts, but once selected, cuts are added to the LP relaxation, thus embedding the ML outcome into an exact algorithm. This approach highlights promising directions for this type of algorithms. The decision learned is critical because adding the best cutting planes is necessary for solving the problem fast (or fast enough, because in the presence of NP-hard problems, optimization may time out before any meaningful solving). At the same time, the approximate decision (often in the form of a probability) does not compromise the exactness of the algorithm: any cut added is guaranteed to be valid. This setting leaves room for ML to thrive, while reducing the need for guarantees from the ML algorithms (an active and difficult area of research). In addition, note that, the approach in Larsen et al. (2018) is part of a master algorithm in which the ML is iteratively invoked to make booking decisions in real time.

The work of Khalil et al. (2017a), presented in Section 3.1.2, also belongs to this setting, even if the resulting algorithm is heuristic. Indeed, a ML model is asked to select the most relevant node, while a (simple) master algorithm maintains the partial tour, computes its length, *etc.* Because the master algorithm is very simple, it is possible to see the contribution as an end-to-end method, as stated in Section 3.2.1, but it can also be interpreted more generally as done here.

Before ending this section, it is worth mentioning that learning recurrent algorithmic decisions is also used in the deep learning community, for instance in the field of meta-learning to decide how to apply gradient updates in stochastic gradient descent (Andrychowicz et al., 2016; Li and Malik, 2017; Wichrowska et al., 2017).

4 Methodology

In the previous section, we have surveyed the existing literature by orthogonally grouping the main contributions of ML for CO into families of approaches, sometimes with overlapping. In this section, we present a methodological view based on learning theory concepts.

4.1 Demonstration and experience

In order to learn a policy, we have highlighted two methodologies: demonstration, where the expected behavior is shown by an expert or oracle (sometimes at a considerable computational cost), and experience, where the policy is learned through trial and error with a reward signal.

In the demonstration setting, the performance of the learned policy is bounded by the expert, which is a limitation when the expert is not optimal. More precisely, without a reward signal, the imitation policy can only hope to marginally outperform the expert (for example because the learner can reduce the variance of the answers of the expert). The better the learning, the closer the performance of the learner to the expert's. This means that imitation alone should be used only if it is significantly faster than the expert to compute the policy. Furthermore, the performance of the learned policy may not generalize well to unseen examples and small variations of the task and may be unstable due to errors in the dataset and model approximations. Some downsides of supervised (imitation) learning can be overcome with more advanced algorithms, including active methods to query the expert as an oracle to improve behavior in uncertain states.

On the contrary, with a reward, the algorithm learns to optimize for that signal and can potentially outperform any expert, at the cost of a much longer training time. Learning from a reward signal (experience) is also more flexible when multiple decisions are (almost) equally good in comparison with an expert that would favor one (arbitrary) decision. Experience is not without flaws. In the case where policies are approximated (*e.g.* with a neural network), the learning process may get stuck around poor solutions if exploration is not sufficient or solutions which do not generalize well are found. Furthermore, it may not always be straightforward to define a reward signal. For instance, sparse rewards may be augmented to value intermediate accomplishments (see Section 2.2).

Often, it is a good idea to start learning from demonstrations by an expert, then refine the policy using experience and a reward signal. This is what was done in the original AlphaGo paper (Silver et al., 2016), where human knowledge is combined with reinforcement learning.

The reader is referred to Hussein et al. (2017) for a survey on imitation learning covering most of the discussion in this section.

4.2 Partial observability

We mentioned in section 2.2 that sometimes, the states of an MDP are not fully observed and the Markov property does not hold, *i.e.* the probability of the next observation, conditioned on the current observation and action, is not equal to the probability of the next observation, conditioned on all past observations and actions. An immediate example of this can be found in any environment simulating physics: a single frame/image of such an environment is not sufficient to grasp notions such as velocity and is therefore not sufficient to properly estimate the future trajectory of objects. It turns out that, on real applications, partial observability is closer to the norm than to the exception, either because one does not have access to a true state of the environment, or because it is not tractable to represent and needs to be approximated. A straightforward way to tackle the problem is to compress all previous observations using an RNN. This can be applied in the imitation learning setting, as well as in RL, for instance by learning a recurrent policy (Wierstra et al., 2010).

How does this apply in the case where we want to learn a policy function making decisions for a CO algorithm? On the one hand, one has full access to the state of the algorithm because it is represented in exact mathematical concepts, such as constraints, cuts, solutions, B&B tree, *etc.* On the other hand, these states can be exponentially large. This is an issue in terms of computations and generalization. Indeed, if one does want to solve problems fast, one needs to have a policy that is also fast to compute, especially if it is called frequently as is the case for say branching decisions. Furthermore, considering too high-dimensional states is also a statistical problem for learning, as it may dramatically increase the required number of samples, decrease the learning speed, or simply fail altogether. Hence, it is necessary to keep these aspects in mind while experimenting with different representations of the data.

4.3 Evaluation

The question of evaluating the quality of the algorithm is not easy. A policy that fails to learn anything and turns out being random could be a legitimate, even sometimes well-performing, policy for some problems. As in this context we care about solving an optimization problem, the usual metrics (time, lower bound, upper bound *i.e.* solution values, *etc.*) should matter. Nonetheless, because a learning component was introduced, we need to ask ourselves on which problems (instances) the algorithms are evaluated on.

4.3.1 Effect of machine learning performance

Can the error metrics of a ML model, *e.g.* accuracy for classification and mean square error for regression, give us any information about the performance of the overall optimization algorithm? Intuitively, in the supervised (imitation) context, learning should drive the policy in the right direction as the model learns to replicate the expert policy, and so we expect the performance of the optimization algorithm to improve. Under some conditions, it is possible to relate the performance of the learned policy to the performance of the expert policy, but covering this aspect is out of the scope of this paper. The opposite is not true, if learning fails, the policy may still turn out to perform well (by encountering an alternative good decision). Indeed, when making a decision in contradiction with a target, the learning will be fully penalized when, in fact, the decision could be almost as good (as explained in Section 4.1).

In the case of reinforcement learning, if the reward signal is shaped to reflect the optimization objective, it is straightforward that both performances are correlated, even proportional. In practice, however, one might design a surrogate reward signal to encourage intermediate accomplishments so there may be some discrepancies, especially if the learned policy falls near a bad minimum.

Overall, the learning process encourages the policy to improve the performance of the original optimization algorithm. However, to report final results, only the true observed performance of the optimization algorithm is significant.

4.3.2 Generalization

Another genuine question is to know whether or not a learned policy will perform decently in a different environment. Or does it matter? To make it easier, let us take the example of a branching policy for MILPs. At one end of the spectrum, we consider one fixed instance that we need to solve. We can make as many runs (episodes) and as many calls to the expert as we want, but ultimately we only care about solving this one instance. At the other end of the spectrum, we want our policy learned on a finite set of instances to perform well (generalize) to any given MILP instance. Learning a policy for a single instance should require a smaller model, which could thus be achieved with less training examples. Nonetheless, in the single instance case, one learns the policy from scratch at every new instance. This means starting the timer at the beginning of learning and competing with other solvers to get the solution the fastest (or get the best results within a time limit). This is not the case when the policy generalizes, because in this case, learning can be decoupled from solving as it can be done offline.

One way to think about the training data from multiple instances is like a multi-task learning setting. The different tasks have underlying aspects in common, and they may also have their own peculiar quirks. One way to take advantage of these commonalities is to learn a single policy that generalizes within a distribution of instances. This compromise sounds promising as it could give a policy to use out of the box for similar instances, while keeping the learning problem offline and hopefully reasonable. This is also in line with many business applications: one company does not care for every possible instance, but, generally, it is interested in solving similar (yet distinct) problems repeatedly.

A compromise between instance-specific learning and learning a generic policy is what we typically have in multi-task learning: some parameters are shared across tasks and some are specific to each task. A common way to do that (in the transfer learning scenario) is to start from a generic policy and then adapt it to the particular instance by a form of fine-tuning procedure: training proceeds in two stages, first training the generic policy across many instances from the same distribution, and then continuing training on the examples associated with a given instance on which we are hoping to get more specialized and accurate predictions.

The question of distribution contains many aspects. Two intuitive axes are "*structure*" and "*size*". A TSP and a scheduling problem seems to have fairly different structure, and one can think of two planar euclidean TSPs to be way more similar. Still, two of these TSPs can have dramatically different sizes (as in number of nodes).

Choosing how ambitious one should be in defining the characteristics of the distribution is a hard question. It is also too early to provide insights about it, given the limited literature in the field.

Machine learning advances in the areas of meta-learning and transfer learning are particularly interesting. Meta-learning considers two levels of optimization: the inner loop trains the parameters of a model on the training set in a way that depends on meta-parameters, which are themselves optimized in an outer loop (*i.e.*, obtaining a gradient for each completed inner-loop training or update). When the outer loop's objective function is performance on a validation set, we end up training a system so that it will generalize well. This can be a successful strategy for generalizing from very few examples if we have access to many such training tasks. It is related to transfer learning, where we want that what has been learned in one or many tasks helps improve generalization on another. These approaches can help rapidly adapt to a new problem, which would be useful in the context of solving many MILPs instances, seen as many related tasks.

To stay with the branching example on MILPs, one may not want the policy to perform well out of the box on new instances (from the given distribution). Instead, one may want to learn a policy offline that can be adapted to a new instance in a few training steps, every time it is given one.

This setting, which is more general than not performing any adaptation of the policy, has potential for better generalization. Once again, the scale on which this is applied can vary depending on ambition. One can transfer on very similar instances, or learn a policy that transfers to a vast range of instances.

Meta-learning algorithms were first introduced in the 1990s (Bengio et al., 1991; Schmidhuber, 1992; Thrun and Pratt, 1998) and have since become particularly popular in ML, including, but not limited to, learning a gradient update rule (Hochreiter et al., 2001; Andrychowicz et al., 2016), few shot learning (Ravi and Larochelle, 2017), and multi-task RL (Finn et al., 2017).

4.3.3 Other learning metrics

Other metrics from the process of learning itself are also relevant, such as how fast the learning process is, the sample complexity (number of examples required to properly fit the model), *etc.* As opposed to the metrics suggested earlier in this section, these metrics provide us with information not about final performance, but about offline computation or the number of training examples required to obtain the desired policy. This information is, of course, useful to calibrate the effort in integrating ML into CO algorithms.

4.4 Exactness and approximation

In the different examples we have surveyed, ML is used in both exact and heuristic frameworks, for example Baltean-Lugojan et al. (2018) and Larsen et al. (2018), respectively. Getting the output of a ML model to respect advanced types of constraints is a hard task. In order to build exact algorithms with ML components, it is necessary to apply the ML where all possible decisions are valid. Using only ML as surveyed in Section 3.2.1 cannot give any optimality guarantee, and only weak feasibility guarantees (see Section 5.1). However, applying ML to select or parametrize a CO algorithm as in Section 3.2.2 will keep exactness if all possible choices that ML discriminate lead to complete algorithms. Finally, in the case of repeated interactions between ML and CO surveyed in Section 3.2.3, all possible decisions must be valid. For instance, in the case of MILPs, this includes branching *among fractional variables* of the LP relaxation, selecting the node to explore *among open branching nodes* (He et al., 2014), deciding on the frequency to run heuristics on the B&B nodes (Khalil et al., 2017b), selecting cutting planes *among valid inequalities* (Baltean-Lugojan et al., 2018), removing previous cutting planes *if they are not original constraints or branching decision*, *etc.*

5 Challenges

In this section, we are reviewing some of the algorithmic concepts previously introduced by taking the viewpoint of their associated challenges.

5.1 Feasibility

In Section 3.2.1, we pointed out how ML can be used to directly output solutions to optimization problems. Rather than learning the solution, it would be more precise to say that the algorithm is learning a *heuristic*. As already repeatedly noted, the learned algorithm does not give any guarantee in terms of optimality, but it is even more critical that feasibility is not guaranteed either. Indeed, we do not know how far the output of the heuristic is from the optimal solution, or if it even respects the constraints of the problem. This can be the case for every heuristic and the issue can be mitigated by using the heuristic within an exact optimization algorithm (such as branch and bound).

Finding feasible solutions is not an easy problem (theoretically NP-hard for MILPs), but it is even more challenging in ML, especially by using neural networks. Indeed, trained with gradient descent, neural architectures must be designed carefully in order not to break differentiability. For instance, both pointer networks (Vinyals et al., 2015) and the Sinkhorn layer (Emami and Ranka, 2018) are complex architectures used to make a network output a permutation, a constraint easy to satisfy when writing a classical CO heuristic.

5.2 Modelling

In ML, in general, and in deep learning, in particular, we know some good prior for some given problems. For instance, we know that a convolutional neural network (CNN) is an architecture that will learn and generalize more easily than others on image data. The problems studied in CO are different from the ones currently being addressed in ML, where most successful applications target natural signals. The architectures used to learn good policies in combinatorial optimization might be very different from what is currently used with deep learning. This might also be true in more subtle or unexpected ways: it is conceivable that, in turn, the optimization components of deep learning algorithms (say, modifications to SGD) could be different when deep learning is applied to the CO context.

Current deep learning already provides many techniques and architectures for tackling problems of interest in CO. As pointed out in section 2.2, techniques such as parameter sharing made it possible for neural networks to process sequences of variable size with RNNs or, more recently, to process graph structured data through GNNs. Processing graph data is of uttermost importance in CO because many problems are formulated (represented) on graphs. For a very general example, Selsam et al. (2018) represent a satisfiability problem using a bipartite graph on variables and clauses. This can generalize to MILPs, where the constraint matrix can be represented as the adjacency matrix of a bipartite graph on variables and constraints.

5.3 Scaling

Scaling to larger problems can be a challenge. If a model trained on instances up to some size, say TSPs up to size fifty nodes, is evaluated on larger instances, say TSPs of size a hundred, five hundred nodes, *etc*, the challenge exists in terms of generalization, as mentioned in Section 4.3.2. Indeed, all of the papers tackling TSP through ML and attempting to solve larger instances see degrading performance as size increases (Vinyals et al., 2015; Bello et al., 2017; Khalil et al., 2017a; Kool and Welling, 2018). To tackle this issue, one may try to learn on larger instances, but this may turn out to be a computational and generalization issue. Except for very simple ML models and strong assumptions about the data distribution, it is impossible to know the computational complexity and the sample complexity, i.e. the number of observations that learning requires, because one is unaware of the exact problem one is trying to solve (the true data generating distribution).

5.4 Data generation

Collecting data (for example instances of optimization problems) is a subtle task. Larsen et al. (2018) claim that "*sampling from historical data is appropriate when attempting to mimic a behavior reflected in such data*". In other words, given an external process on which we observe instances of an optimization problem, we can collect data to train some policy needed for optimization, and expect the policy to generalize on future instances of this application. A practical example would be a business that frequently encounters optimization problems related to their activities, as discussed in Section 4.3.2.

In other cases, *i.e.* when we are not targeting a specific application for which we have historical data, how can we proactively train a policy for problems that we do not yet know of? As partially discussed in Section 4.3.2, we first need to define to which family of instances we want to generalize. For instance, we might decide to learn a cutting plane selection policy for Euclidian TSP problems. Even so, it remains a complex effort to generate problems that capture the essence of real applications. Moreover, CO problems are high dimensional, highly structured, and troublesome to visualize. The sole exercise of generating graphs is already a complicated one.

Deciding how to represent the data is also not an easy task, but can have a dramatic impact on learning. For instance, how does one properly represent a B&B node, or even the whole B&B tree? These representations need to be expressive enough for learning, but at the same time, concise enough to be used frequently without excessive computations.

6 Conclusions

We have surveyed and highlighted how machine learning can be used to build combinatorial optimization algorithms that are partially learned. We have suggested that imitation learning alone can be valuable if the policy learned is significantly faster to compute than the original one provided by an expert, in this case a combinatorial optimization algorithm. On the contrary, models trained with a reward signal have the potential to outperform current policies, given enough training and a supervised initialization. Learning a policy that generalizes to unseen problems is a challenge, this is why we believe learning should occur on a distribution small enough that the policy could fully exploit the structure of the problem and give better results. 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.

Other than performance incentives, there is also interest in using machine learning as a modelling tool for operations research, as done by Lombardi and Milano (2018), or to extract intuition and knowledge about algorithms as mentioned in Bonami et al. (2018); Khalil et al. (2017a).

Although most of the approaches we discussed in this paper are still at an exploratory level of deployment, at least in terms of their use in general-purpose (commercial) solvers, we strongly believe that this is just the beginning of a new era for combinatorial optimization algorithms.

Acknowledgments

The authors are grateful to Emma Frejinger, Simon Lacoste-Julien, Jason Jo, Laurent Charlin, Matteo Fischetti, Rémi Leblond, Michela Milano, Sébastien Lachapelle, Eric Larsen, Pierre Bonami, Martina Fischetti, Elias Khalil, Bistra Dilkina, Sebastian Pokutta, Marco Lübbecke, Andrea Tramontani, Dimitris Bertsimas and the entire CERC team for endless discussions on the subject and for reading and commenting a preliminary version of the paper.

References

Ahuja, R. K. and Orlin, J. B. (2001). Inverse Optimization. *Operations Research*, 49(5):771–783.

Andrychowicz, M., Denil, M., Gómez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, pages 3981–3989. Curran Associates, Inc.

Applegate, D., Bixby, R., Chvátal, V., and Cook, W. (2007). *The traveling salesman problem. A computational study.* Princeton University Press.

Baltean-Lugojan, R., Misener, R., Bonami, P., and Tramontani, A. (2018). Strong sparse cut selection via trained neural nets for quadratic semidefinite outer-approximations. Technical report, Imperial College, London.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2017). Neural Combinatorial Optimization with Reinforcement Learning. In *International Conference on Learning Representations.*

Bengio, Y., Bengio, S., Cloutier, J., and Gecsei, J. (1991). Learning a synaptic learning rule. In *IJCNN*, pages II–A969.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning.* springer.

Bonami, P., Lodi, A., and Zarpellon, G. (2018). Learning a Classification of Mixed-Integer Quadratic Programming Problems. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Lecture Notes in Computer Science, pages 595–604. Springer, Cham.

Chan, T. C. Y., Craig, T., Lee, T., and Sharpe, M. B. (2014). Generalized Inverse Multiobjective Optimization with Application to Cancer Therapy. *Operations Research*, 62(3):680–695.

Conforti, M., Conrnuéjols, G., and Zambelli, G. (2014). *Integer Programming.* Springer.

Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., and Bharath, A. A. (2018). Generative Adversarial Networks: An Overview. *IEEE Signal Processing Magazine*, 35(1):53–65.

Dai, H., Dai, B., and Song, L. (2016). Discriminative Embeddings of Latent Variable Models for Structured Data. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2702–2711, New York, New York, USA. PMLR.

Dey, S. and Molinaro, M. (2018). Theoretical challenges towards cutting-plane selection. *Mathematical Programming*, 170:237–266.

Emami, P. and Ranka, S. (2018). Learning Permutations with Sinkhorn Policy Gradient. *arXiv:1805.07010 [cs, stat]*.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135, International Convention Centre, Sydney, Australia. PMLR.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural Message Passing for Quantum Chemistry. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272, International Convention Centre, Sydney, Australia. PMLR.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT press.

He, H., Daume III, H., and Eisner, J. M. (2014). Learning to Search in Branch and Bound Algorithms. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 3293–3301. Curran Associates, Inc.

Hochreiter, S., Younger, A. S., and Conwell, P. R. (2001). Learning to learn using gradient descent. In Dorffner, G., Bischof, H., and Hornik, K., editors, *Artificial Neural Networks — ICANN 2001*, pages 87–94, Berlin, Heidelberg. Springer Berlin Heidelberg.

Hussein, A., Gaber, M. M., Elyan, E., and Jayne, C. (2017). Imitation Learning: A Survey of Learning Methods. *ACM Computing Surveys*, 50(2):21:1–21:35.

Khalil, E., Dai, H., Zhang, Y., Dilkina, B., and Song, L. (2017a). Learning Combinatorial Optimization Algorithms over Graphs. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 6348–6358. Curran Associates, Inc.

Khalil, E. B., Bodic, P. L., Song, L., Nemhauser, G., and Dilkina, B. (2016). Learning to Branch in Mixed Integer Programming. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 724–731, Phoenix, Arizona. AAAI Press.

Khalil, E. B., Dilkina, B., Nemhauser, G. L., Ahmed, S., and Shao, Y. (2017b). Learning to Run Heuristics in Tree Search. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 659–666.

Kool, W. W. M. and Welling, M. (2018). Attention Solves Your TSP, Approximately. *arXiv:1803.08475 [cs, stat]*.

Kruber, M., Lübbecke, M. E., and Parmentier, A. (2017). Learning When to Use a Decomposition. In *Integration of AI and OR Techniques in Constraint Programming*, Lecture Notes in Computer Science, pages 202–210. Springer, Cham.

Larsen, E., Lachapelle, S., Bengio, Y., Frejinger, E., Lacoste-Julien, S., and Lodi, A. (2018). Predicting Solution Summaries to Integer Linear Programs under Imperfect Information with Machine Learning. *arXiv:1807.11876 [cs, stat]*.

Li, K. and Malik, J. (2017). Learning to Optimize Neural Nets. *arXiv:1703.00441 [cs, math, stat]*.

Lodi, A. (2009). MIP computation. In Jünger, M., Liebling, T., Naddef, D., Nemhauser, G., Pulleyblank, W., Reinelt, G., Rinaldi, G., and Wolsey, L., editors, *50 Years of Integer Programming 1958-2008*, pages 619–645. Springer-Verlag.

Lodi, A. and Zarpellon, G. (2017). On learning and branching: A survey. *TOP*, 25(2):207–236.

Lombardi, M. and Milano, M. (2018). Boosting Combinatorial Problem Modeling with Machine Learning. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 5472–5478. International Joint Conferences on Artificial Intelligence Organization.

Mahmood, R., Babier, A., McNiven, A., Diamant, A., and Chan, T. C. Y. (2018). Automated Treatment Planning in Radiation Therapy using Generative Adversarial Networks. In *Proceedings of Machine Learning for Health Care*, volume 85 of *Proceedings of Machine Learning Research*.

Marcos Alvarez, A., Louveaux, Q., and Wehenkel, L. (2014). A supervised machine learning approach to variable branching in branch-and-bound. Technical report, Université de Liège.

Marcos Alvarez, A., Louveaux, Q., and Wehenkel, L. (2017). A Machine Learning-Based Approximation of Strong Branching. *INFORMS Journal on Computing*, 29(1):185–195.

Marcos Alvarez, A., Wehenkel, L., and Louveaux, Q. (2016). Online Learning for Strong Branching Approximation in Branch-and-Bound. Technical report, Université de Liège.

McCormick, G. P. (1976). Computability of global solutions to factorable nonconvex programs: Part I — Convex underestimating problems. *Mathematical Programming*, 10(1):147–175.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT press.

Nowak, A., Villar, S., Bandeira, A. S., and Bruna, J. (2017). A Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks. *arXiv:1706.07450 [cs, stat]*.

Ravi, S. and Larochelle, H. (2017). Optimization as a model for few-shot learning. In *International Conference on Learning Representations*.

Schmidhuber, J. (1992). Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. (2018). Learning a SAT Solver from Single-Bit Supervision. *arXiv:1802.03685 [cs]*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Smith, K. A. (1999). Neural Networks for Combinatorial Optimization: A Review of More Than a Decade of Research. *INFORMS Journal on Computing*, 11(1):15–34.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* MIT press Cambridge.

Thrun, S. and Pratt, L. Y., editors (1998). *Learning to Learn.* Kluwer Academic.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is All you Need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. (2018). Graph attention networks. In *International Conference on Learning Representations*.

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer Networks. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc.

Wichrowska, O., Maheswaranathan, N., Hoffman, M. W., Colmenarejo, S. G., Denil, M., de Freitas, N., and Sohl-Dickstein, J. (2017). Learned Optimizers that Scale and Generalize. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3751–3760, International Convention Centre, Sydney, Australia. PMLR.

Wierstra, D., Förster, A., Peters, J., and Schmidhuber, J. (2010). Recurrent policy gradients. *Logic Journal of the IGPL*, 18(5):620–634.

Wolsey, L. A. (1998). *Integer Programming.* Wiley.