ALMA MATER STUDIORUM UNIVERSITÁ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO INFORMATICA - SCIENZA E INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Intelligent Systems

Exact Combinatorial Optimization with Graph Convolutional Neural Networks

Candidato:

Nicola Ferroni

Relatore:

Chiar. ma Prof.ssa **Michela Milano**

Correlatori:

Prof. **Andrea Lodi**

Dott. **Maxime Gasse**

Dott. **Didier Chetelat**

Dott. **Michele Lombardi**

Anno Accademico 2017-2018

Sessione III

Acknowledgements

I would first like to thank my thesis advisor Prof. Michela Milano of the Engineering School at University of Bologna. She allowed me to have a beautiful experience in Canada at the cole Polytechnique de Montral, under the supervision of the Prof. Andrea Lodi and his research group, the Canada Excellence Research Chair in Data Science for Real-Time Decision Making.

I would also like to thank the experts who were involved in the validation survey for this research project, the postdoc Maxime Gasse, who taught me most of the skills I learned and is a true friend also out of the office, the researcher Didier Chtelat, who constantly supervised and supported me in the best possible way and all the other member of the team: Gabriele Dragotto, Gonzalo Muñoz, Jeff Sylvestre-Décary, Ashutosh Saboo, Aleksandr Kazachkov, Aurélien Serre, Antoine Prouvost, Mathieu Tanneau, Claudio Sole, Giulia Zarpellon, Mehdi Taobane, Laurent Charlin, Jaime E. Gonzalez, Koladé Nourou.

Finally, I must express my very profound gratitude to my parents and all my family for providing me with unfailing support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them. Thank you.

Author

Nicola Ferroni

Abstract

Combinatorial optimization problems are typically tackled by the branch-and-bound paradigm. We propose to learn a variable selection policy for branch-and-bound in mixed-integer linear programming, by imitation learning on a diversified variant of the strong branching expert rule. We encode states as bipartite graphs and parameterize the policy as a graph convolutional neural network. Experiments on a series of synthetic problems demonstrate that our approach produces policies that can improve upon expert-designed branching rules on large problems, and generalize to instances significantly larger than seen during training.

Abstract

I problemi di ottimizzazione combinatoria sono tipicamente affrontati tramite il paradigma branch-and-bound. L'obiettivo é imparare una policy di variable selection per il branch-and-bound nei problemi di mixed-integer linear programming, usando imitation learning su una variante della regola strong branching. Lo stato del solver viene codificato come grafo bipartito e la policy come graph convolutional neural network. Esperimenti dimostrano che l'approccio produce policy che possono velocizzare l'esecuzione dell'algoritmo rispetto all'uso di euristiche esperte su problemi di grandi dimensioni, generalizzando su problemi significativamente piú grandi rispetto a quelli visti durante la fase di training.

Contents

1 Introduction **1**

1.1 Context of the Study 1

1.2 Motivations 3

1.3 Overview of the Thesis 3

2 Operations Research Overview **5**

2.1 Combinatorial Optimization 6

2.1.1 Backtracking 7

2.2 Mixed-Integer Linear Programming 8

2.2.1 Cutting planes 9

2.2.2 Branch-and-Bound 10

2.2.3 Branch-and-Cut 10

2.2.4 Approximated Methods 10

2.3 Branch and Bound 11

2.3.1 Node Selection 13

2.3.1.1 Simple Heuristics 13

2.3.1.2 Relaxations 14

2.3.1.3 Is the Node Selection a Bottle-Neck? 15

2.3.2 Variable Selection 15

2.3.2.1 The Most Infeasible Branching 16

2.3.2.2 Hybrid Branching 16
2.3.2.3 Strong Branching 16
2.3.2.4 Pseudocost Branching 17
2.3.2.5 The State of the Art 17
2.3.3 The Branch-and-Bound formulation as a Markov Decision Process 18

3 Machine Learning **21**
3.1 What is Machine Learning? 22
3.2 Classification 23
3.2.1 Performance Measures 24
3.3 Information Theory 25
3.3.1 Entropy 25
3.3.2 Conditional Entropy 26
3.3.3 Cross Entropy 26
3.4 Overfitting 27
3.5 Decision trees 28
3.6 Support Vector Machines 29
3.7 Artificial Neural Networks and Deep Learning 31

4 Methodology **35**
4.1 Implementation Tools 35
4.1.1 Backend Solver 35
4.1.2 The Deep Learning Library 36
4.1.3 Programming Language 37
4.2 Imitation Learning 38
4.2.1 Imitation Target and Dataset Generation 39
4.3 State Encoding 40

4.4 Policy Parametrization 41
4.4.1 Graph Convolutional Neural Network 41
4.4.2 Implementation 44
4.4.3 Hyper-parameters tuning 46
4.4.4 Pre-norm Layer 48
4.5 Evaluation Procedure 50
4.5.1 Embedding the model in SCIP 51
4.5.2 Ablation Study 52

5 Related Works 55
5.1 Regression Model 55
5.2 Learn to Rank Model 56
5.3 Imitation learning and SVM 56
5.4 Portfolio-based Methods 57
5.5 Other Methods 57

6 Experimental results 59
6.1 Benchmarks 59
6.2 Experimental setup 60
6.3 Baseline competitors 60
6.4 Results 62
6.5 Target evaluation 66

7 Conclusions and Future Developments 67

References 69

List of Figures

2.1 Branch-and-bound variable selection as a Markov decision process. On the left, a state s_t comprised of the branch-and-bound tree, with a leaf chosen by the solver to be the next node from which to branch. On the right, a new state s_{t+1} resulting from branching on the variable $a_t = x_4$. 20

3.1 Overfitting detected after the 30th training iteration on the dataset, where the error after start to increase. 28

3.2 Comparison between a linearly separable dataset and a non linearly separable one. 30

3.3 A kernel trick transorm the original space in a new feature space where the dataset becomes linearly separable, i.e. with a hyperplane 31

3.4 A simple representation of a fully connected feed forward network with one hidden layer. 32

3.5 A comparation between the three most used activation functions, Sigmoid, Tanh and ReLU. 33

4.1 A bipartite state representation $\mathbf{s}_t = (\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$ with $n = 3$ variables and $m = 2$ constraints. Here $\mathbf{C} \in \mathbb{R}^{m \times c}$ represents the feature matrix of the constraint nodes on the left, $\mathbf{V} \in \mathbb{R}^{n \times d}$ the feature matrix of the variable nodes on the right, and $\mathbf{E} \in \mathbb{R}^{m \times n \times e}$ the (sparse) feature tensor of the edges between constraints and variables. 40

4.2 Our bipartite graph convolutional neural network architecture for parametrizing our policy π. 45

4.3 This figure shows the test error (cross-entropy) through the epochs during the training phase. We can observe that there is no overfitting and the curve is almost smooth and stable. 48

4.4 The architecture of the model is not the standard sequential one, but it begins with three parallels layers, that feed the first graph convolutional layer and then is a sequential model until the output. 49

4.5 The rectangles with rounded angles are matrices, the rectangles are fully connected layers, the dashed rounded rectangle are mathematical operation, and the sum convolution is represented by the parallelepiped. The Affinity layer is a parallel task with the feature layer, it is not represented because it is exactly the same and it would have made the graph unreadable. 54

List of Tables

4.1 Training results on maximum set cover instances. M is for the mean normalized convolution, L is for the usage of layernorm instead of our prenorm, A is for the usage of te affinity layer, N is for no normalization after the convolutions, 3 is for three convolutional layers. The measured time is the amount needed for a mini-batch forward step. 51

4.2 Test results on maximum set cover instances. 52

6.1 Training results on maximum set cover instances. M is for the mean normalized convolution, L is for the usage of layernorm instead of our prenorm, A is for the usage of te affinity layer, N is for no normalization after the convolutions, 3 is for three convolutional layers, E is for a model trained with decisions from strong branching only. 63

6.2 Test results on maximum set cover instances. 64

6.3 Test results maximum independent set instances. 65

6.4 Test results on multiple knapsack instances. 66

Chapter 1

Introduction

1.1 Context of the Study

Optimization is a discipline that studies different kind of problems where the result is a decision that maximizes (or minimizes) a quantity (such as a profit) by applying advanced mathematical techniques. The decision takes the name of optimal solution (or near optimal in case of approximated methods). It is used to improve decision-making, optimization and efficiency, and it is based on problem formulation, i.e. mathematical model, that should give the most precise representation of the problem. The nature of the problems could be very different, so the formulation can involve many different techniques between all the possible ones to find a solution considering the constraints, the computing power, and the amount of time.

Some of the most common scenario that the operation research improved are computing and information technologies, financial engineering, transports, simulations, stochastic models, game theory, facility location and scheduling.

The focus of this thesis is on combinatorial optimization problems. In these kind of problems it is almost impossible to perform an exhaustive search: the

space of possible solutions is typically too large to use brute force methods, and no algorithm is guaranteed to find the optimal solution nor to run in polynomial time in the worst case.

Combinatorial optimization problems can be solved using different methods, usually integer programming techniques. An integer programming problem is a formulation of a combinatorial optimization problem in which all the variables must be integers, and it is defined through a function that needs to be maximized (or minimized) and a list of constraints. We will focus on mixed-integer linear programming method (MILP) where there could be also non discrete variables. The linearity means that every mathematical expression that describes the model has a linear form. If we want to find an exact solution, these kind of problems are typically tackled with the branch-and-bound algorithm which is based on the decomposition of the main problem in many smaller and simpler problems [1]. It consists of a systematic enumeration of candidate solutions in a tree structure until the optimal one is found and proven to be optimal. A key step in the algorithm is the selection of a fractional variable to branch on because it can have a very significant impact on the size of the resulting search tree [2]. Branching rules are one of the most important choices of modern Mixed-Integer Linear Programming solvers [3; 4; 5; 6].

It has been shown [7] that to get significant improvements in number of explored nodes, we can perform an intensive dynamic use of strong branching rule [8]; but its main issue is that it takes a lot of computational power, and as a result, the total computing time is prohibitively high in practice. Many research efforts over the last decade have been spent on finding an alternative ways to make it lighter.

The most common technique, implemented by most modern solvers, is an hybrid method that involves strong branching to build the first nodes, and then

the pseudo-cost rule, which is very light and fast, but since it is based on past decisions, it can not be used at the beginning. The goal of the thesis it to use imitation learning to emulate the behavior of the strong branching rule, in order to find a better trade off between computational time and number of expanded nodes. It involves the usage of machine learning techniques, the generation of a dataset, the shift of computation on a GPU (Graphic Processing Unit, used by home personal computers as well, which are capable of high performances in algebra operations) and the definition of what is a state and how can it be encoded as a single instance of a dataset.

1.2 Motivations

There are two main reason why this research topic is interesting. The first one is that it may be possible to use machine learning and deep learning to provide a better trade-off for the variable selection process in terms of speed and precision. The second one is related to the branch-and-bound nature, which is inherently sequential.

It makes the usage of a multi-core CPU (Central Processing Unit, the basic computing core of computer machines) not well optimized since one of the main advantage of these CPU is the efficient management of parallel threads. Using machine learning on the GPU is a kind of optimization of the computational source and, given a money budget, it is easy to find out that a GPU is much cheaper and more powerful than a CPU's core.

1.3 Overview of the Thesis

In the first chapter there are details about combinatorial optimization and integer programming framework and which are the main techniques with their pros and

cons. There are details about the state of the art and an alternative formulation of the main algorithm as well.

In the second chapter there is an explanation about Machine Learning, which are the main techniques and tools and which are the biggest challenges of this project.

The third chapter contains all the details about the methodology, such as the production of the dataset, the architectural choices, the implementations, the hyper-parameter tunings and an ablation study to validate the global architecture. There is also a section about the used tools, such as the solver, and details about the choices.

In the fourth chapter there is an analysis about the related works: there has been many different attempts to improve the performances of the variable selection's state of the art, such as regression models, support vector machines, clustering based classifier and others.

In the fifth chapter there are all the collected results, such as benchmark on different kind of problems, the experimental comparisons and evaluations against the competitors and details about the experimental setup.

Finally, the last chapter contains the conclusions, where there is an over all evaluation of the results and how it can impact the future research and developments.

Chapter 2

Operations Research Overview

Summary

Operations research was born due to military needs, during the Second World War, to solve problems about strategy and tactics for defense. Most of the time, the field of research has been directly practical, so the solutions had to be found in a reasonable time and it became an application of the scientific method soon.

In the real field we often look for solutions that do not allow the presence of decimal values (such as variables that represent counters or enumerations). This gave rise to a new class of problems, now known as integer programming.

The theory of complexity, which studies the minimum necessary requirements (in terms of computational time and memory), considers these kind of problems hard to solve in the worst case. This theory divides the different problems in many classes of complexity based on the best known algorithm that can solve them. The most important separation is between easy problems and hard ones, the last ones are important because there are no efficient algorithms, it means that the needed time to compute the solution in the worst case is not polynomial, but at least exponential [9]. The importance of this separation is that if the worst

case computational time is exponential (or worse) it becomes impossible to solve big-size problems. An example could help to understand this concept better: the TSP (travelling salesman problem) is described by a set of city and the distances between all of them, the goal is to find out the shortest path that connects all the cities. If we set the number of cities to 10, the possible number of tours is 10^6 (which is solvable by a commercial computer), with 33 cities, we have 10^{37} tours (the number of cells in a human body is about 10^{14}), with 100 cities, we have 10^{158} tours (the number of particles in the visible universe is about 10^{89}).

A common field which needs the existence of these hard problems is modern cryptography, because it needs the secret to be "safe enough", i.e. there should not be algorithms that can break the system running in polynomial time.

2.1 Combinatorial Optimization

In the operations research, combinatorial optimization is a class of problems that consists of finding the optimal solution instance in a set of many possible instances. As we said in the introduction, in a combinatorial optimization problem an exhaustive search is not tractable, there are infinitely many solution instances (finite sets of points in the plane), so it is impossible to list an optimum permutation for each of them. What we can do is to design an algorithm that, for each instance, computes an optimum solution. The formal definition of a combinatorial optimization problem is given by a quadruple (I, f, m, g), where:

- I is the set of instances;
- given an instance $i \in I$, $S = f(i)$ is the set of feasible solutions;
- given an instance i and a feasible solution y of i, $m(i, y)$ denotes the measure of y which is usually a positive real;

- g is the goal function, either min or max.

The goal is then to find for some instance i an optimal solution, that is, a feasible solution y with $m(i, y) = g\{m(i, y') \mid y' \in S\}$. In the next section it will be shown how this class of problems can be easily converted into another representation as integer programming.

This kind of problems can be solved in exponential time, for instance: given a set of n points, we can enumerate all possible $n!$ orders, and for each compute the $L_\infty-$norm [10] (Chebyshev distance, it is defined in a vector space where the distance between two vectors is the greatest of their differences along any coordinate dimension) of the corresponding path.

2.1.1 Backtracking

There are many different algorithms in this context, and they could differ in the level of detail and the used formal language. A technique that allows us the enumeration [11] is backtracking, which consists in increasing the last component of a "*ones*" vector until we get n, then switch to the second-last, and so on. The order in which the vectors $\{1, ..., n\}^n$ are enumerated is known as lexicographical order. Once we enumerated all vectors of $\{1, ..., n\}^n$ we can simply check vector by vector which is shortest between that and the best so far. But in this case we enumerated n^n vectors, not $n!$ (which is much smaller), so we can do better: there is a path enumeration algorithm that can do it in about $n^2 \times n!$ steps[11]. It is possible considering the cost of each path and avoid apriori permutations that are for sure worse. The number of steps is close to what we refer as running time, since every step needs about the same amount of work, and the highest priority is to find an algorithm the guarantees us the best running time.

With advanced techniques and better analysis we can do even better and have a running time about $n \times n!$, but the main issue is that even with "little"

problems, the time grows too fast, exponentially with the number of points, and it is easy to reach the "fastest computer limit" with only moderate sizes.

The main focus of the combinatorial optimization is to find better algorithms in this kind of problems, to find the best element between a finite set of feasible solutions; the set is not explicitly listed, indeed it depends on the problem's structure.

Combinatorial optimization problems are often solved by using integer programming techniques and one of the most used is the branch-and-bound paradigm, that will be presented in its dedicated section 2.2.2.

2.2 Mixed-Integer Linear Programming

As presented so far, the topic of the thesis is not the general linear programming (LP), but the goal is to improve an algorithm that makes sense only in a scenario where there are variables forced to be integers and the mathematical expressions that describe the model are linear, so we must dive deeper in what mixed-integer linear programming (MILP) is.

The standard form of a MILP is the following:

$$\begin{aligned} \arg\min_{\mathbf{x}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b}, \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \\ & \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}, \end{aligned} \tag{2.1}$$

where $c \in \mathbb{R}^n$ is called the objective coefficient vector, $A \in \mathbb{R}^{m \times n}$ the constraint coefficient matrix, $b \in \mathbb{R}^m$ the constraint right-hand-side vector, $l, u \in \mathbb{R}^n$ respectively the lower and upper variable bound vectors, and $p \leq n$ the number of integer variables. Under this representation, the size of a MILP is typically

measured by the number of rows (m) and columns (n) of the constraint matrix.

Mixed-integer linear programming is a direct and easy extension [12] of a integer linear program, so these two classes share most of the solving techniques. Even if it could appear easier than the standard linear programming, integer linear programming is much harder and there are not algorithms that can guarantee to solve them in polynomial time.

Virtually all combinatorial optimization problems can be formulated as integer linear programming [13], the set of feasible solution can be written as $x : Ax \leq b, x \in \mathbb{Z}^n$ for some matrix A and some vector b. The set $P := \{x \in \mathbb{R}^n : Ax \leq b\}$ is a polyhedron, so we define $P_I = \{x : Ax \leq b\}_I$ the convex hull of the integral vectors in P, we call P_I the integer hull of P, where $P_I \subseteq P$.

In the following sections there are descriptions about the mainly used techniques to tackle integer linear programming problems, including hybrid and approximated techniques [14].

2.2.1 Cutting planes

The disequations that describe the model can be seen as a multi-dimensional space of solutions and the constraints limit this space in a polyhedron P that represents the set of all the feasible solutions. Trying to cut off parts of P is one of the most common techniques, and it takes the name of cutting planes; the idea is to cut off a certain part of P such that the resulting set is a polyhedron P' and $P^i \subset P' \subset P$, with P^i the integer program, if we are lucky the optimal solution is integer, otherwise we can repeat the cutting off procedure for P' until we get it. This idea was introduced for the TSP problem by Danzig, Fulkerson and Johnson [15] and Gomory proposed the first cutting planes algorithm [16].

2.2.2 Branch-and-Bound

Another important technique is the branch-and-bound algorithm, which is based on divide-and-conquer paradigm. The main polyhedron is separated by selecting one of the not integer variables in the solution of the linear problem relaxation and forcing it to be greater-or-equal than the rounded up solution value, or less-or-equal than the rounded down one, we will see details in its dedicated section 2.3.

2.2.3 Branch-and-Cut

An evolution of the branch-and-bound algorithm is the combination with the cutting planes and it takes the name of branch-and-cut. The main reason behind this algorithm is that branch-and-bound always split the solution space and we have to solve usually other two problems, while the cutting planes are able to completely remove part of the space of the solutions, allowing us a faster convergence.

2.2.4 Approximated Methods

Other techniques are the approximated methods, we don't always need the best (optimal) solution if it would take too much time (for example in a war context, but also in modern companies), but a fast computed and feasible approximation can often be found quickly. An important parameter is the approximation error ϵ between the optimal solution and its fast approximation, indeed a well designed approximated algorithm guarantees a percentage value of the biggest error from the optimal solution.

2.3 Branch and Bound

The branch-and-bound method is the most used tool for exact algorithm in combinatorial optimization field [17]. It reflects the definition of enumeration that we used to describe a generic combinatorial optimization problem, and it aims to find the best solution between a finite number due to some defined criteria (the case of an infinite number of feasible solutions is over the concept of combinatorial optimization, so we will not consider it). This algorithm aims at enumerating the solutions in a smart way: although the running time remains exponential in the worst case, many problems can often be solved in an acceptable time, even in case of big sizes.

Given S the set of the feasible solutions and $z : S \rightarrow R^1$ the objective function (that we want to maximize or minimize) that matches any element $y \in S$ to a value $z(y)$ [18]. The couple (z, S) defines an optimization problem, that aims to find the element $y^* \in S$ such that:

$$z(y^*) \geq z(y), \forall y \in S$$

(in case of a maximization problem).

We can call P^0 a problem defined by the couple $(z, S(P^0))$ and $Z(P^0)$ the value of its optimal solution, such that:

$$Z(P^0) = \max \{z(y) : y \in S(P^0)\},$$

the branch-and-bound algorithm is based on the separation of P^0 into a number of sub-problems $P^1, P^2, ...P^{n_0}$. It is obtained by separating $S(P^0)$ in subset

$S(P^1), S(P^2), ..., S(P^{n_0})$ such that:

$$\bigcup_{k=1}^{n0} S(P^k) = S(P^0)$$

and defining

$$Z(P^k) = max\{z(y) : y \in S(P^k)\}, \forall k \in \{1, ..., n_0\}.$$

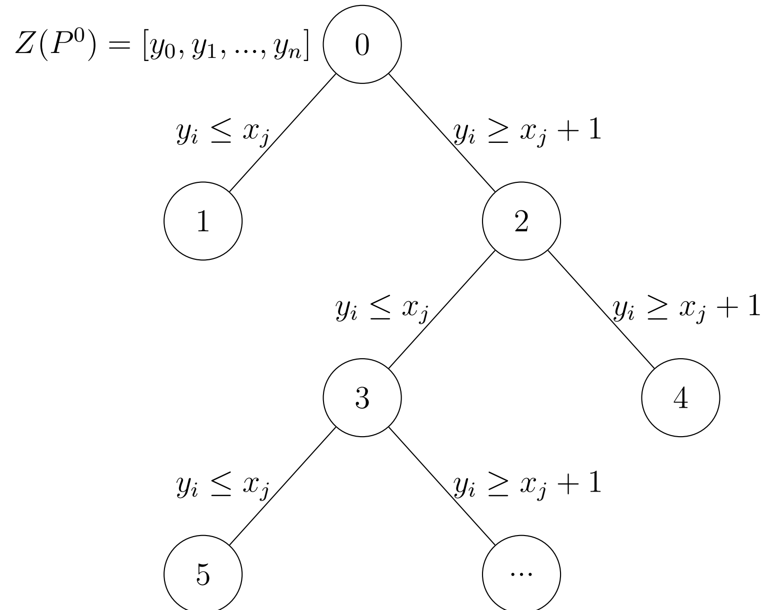
Since any feasible solution of P^0 is also a feasible solution of (at least) one between $P^1, P^1, ..., P^{n_0}$, it will be evident that

$$Z(P^0) = max\{Z(P^1), Z(P^2), ..., Z(P^{n_0})\}.$$

Instead of solving P^0 it will solve the sub-problems and, at each step, it will find an optimal solution for P^k, or prove that the solution for P^k is not feasible, or prove that the optimal solution for P^k can't be better than the best for other sub-problems. The advantage is that at each step we obtain smaller (and easier) problems (with less feasible solutions), but to make it effective we need also that

$$S(P^i) \cap S(P^j) = \emptyset \forall P^i, P^j : i \neq j.$$

Usually it is impossible to easily solve one or more of the sub-problems, so the separation will be repeated until it finds a solution generating a *decision tree* structure, then the next *branches* can be pruned if they provide a termination condition (for instance, the *lp* relaxation has a worse optimal solution value than the best integer one we found so far) or if it has not feasible solutions, or it can be explored until a better solution or a termination condition [19]. A graphical representation will be easier to understand:



P^0 is the original problem and in the tree representation it's the root, the nodes are the sub-problems. What happens at each node? This is the most important moment of the B&B for the thesis.

Considering a LP solution, suppose it is not acceptable for an integer program because it contains at least one continue variable, so how do we split the problem? The algorithm generates two new constraints (one for each sub-problem) that force one of the not-integer variables to be integer. Now we have to solve two decision problems, the node selection and the variable selection.

2.3.1 Node Selection

In this section there are the explanations about the main methods to speed-up the process by selecting the most promising node in order to produce small trees.

2.3.1.1 Simple Heuristics

We can compute for each node an upper and lower bound, but they can differ a lot from the best solution that the node can provide.

The first simple heuristic is the **Depth-first**, it starts producing the first child of the root (P^1) and computing the bounds, then it produces the first child of P^1 and so on, providing a sequence of forward steps in which it is always chosen the last generated node until we get an easy problem to solve or the upper-bound is not greater than Z, in which case a backtracking process is needed; the backtracking brings the algorithm back in the last node with an upper-bound greater than Z. The execution will terminate when the backtracking would try to find the root father. This algorithm is easy to explain and to implement, there are no complex structures to keep in memory and the number of active nodes is relatively low. The main advantage is that it can quickly produce feasible solutions, in this case a termination condition could be a timer.

Another simple heuristic is the **Highest-first**, that aims to expand the node with the highest upper-bound; it is done by keeping in memory a list of open nodes, with the associated upper-bound, and each time we select one of them, we have to remove and replace it with its children and the relative upper-bounds. The main advantage about this strategy is the choice of the most promising node at each step, it usually produces smaller trees in comparison with the depth-first one, but it also increases the complexity and the memory usage; the result is that a generic node expansion is more expensive than in the depth-first.

Another possible strategy is the hybrid one, that have the pros of a smart choice and a simple structure to keep in memory.

2.3.1.2 Relaxations

All the heuristics that we can take into consideration for choosing the node in a smart way are based on the bounds, which are computed on the continuous relaxation of the sub-problem, this particular case is obtained by removing the integrity constraint. Other relaxations could be obtained by deleting some con-

straints, or combining some of them in only one that allows more slack (surrogate relaxation). There is also the possibility of combining more relaxations, for example we can separate the set of constraints in two subsets so that the relaxation for both of them will be easy to solve, and we can consider our upper bound by selecting the worst between them. This concept is used by the decomposition relaxation, which aims to find a single (but better) bound by a weighted sum of the modified objective functions.

One of the most important is the Lagrangian relaxation, which is based on removing some of the constraints and tuning the objective function so that the removed constraints won't probably be violated. We need to put the constraints inside the objective function, if it is a max problem they will be probably ≥ 0. If the solution of the relaxation satisfies the constraints, then the optimal solution of the relaxed problem is equal to the original one, otherwise we have not that guaranteed. Generally there are dominance relations between this relaxations and they can be used to determine which one provides the best bound.

2.3.1.3 Is the Node Selection a Bottle-Neck?

How do these techniques perform in a real solver? Actually the node selection is not a crucial or computationally heavy process, it performs well and it is not a priority that needs research investments on. In the next section we face up to the bottle-neck of the branch and bound algorithm.

2.3.2 Variable Selection

The most important step of the branch and bound algorithm for this thesis is the variable selection: after choosing a node we need to decide which is the most convenient variable to branch on between all the non-integer variables of the LP solution of the node sub-problem. We could virtually generate and test all the

variables, but it would be too much expensive and useless for a practical use. The best strategies are based on heuristics that provide an estimation of the most promising variable; however, the selection of an optimal subset is no longer guaranteed.

2.3.2.1 The Most Infeasible Branching

The first easy heuristic that was used is now called **most infeasible branching**, it selects the most fractional variable (in a $0-1$ problem the closest to 0.5). Even if it is really fast to compute, it has been computationally shown [7] to be worse than a complete random choice (that's the reason of the modern name).

2.3.2.2 Hybrid Branching

To go over the random choice we need to consider a lot of data and to do much more work. An example of a more accurate branching rule is the **hybrid branching** [20], where we compute $five$ different measures for each candidate variable. These measure are then normalized to compute a single score by means of a weighted sum.

2.3.2.3 Strong Branching

The most reliable rule is usually the strong branching (**SB**) technique [8; 21]. There are few versions of the rule that differ for the level of precision (and, as a consequence, the computational weight); in its full version (full strong branching, **FSB**), at any node it branches on each candidate fractional variable and select the one in which the increase in the bound on the left branch times the one on the right branch is maximum. This way is of course too much heavy to compute, but it can be limited. For example we can take into consideration a smaller candidate set of variables, we can also limit to a fixed threshold the number of

simplex pivots to be performed in the variable evaluation. The consequences are a faster execution, paying the price of a lower accuracy. The main advantage of the strong branching is that it averagely produces smaller trees compared with any other known tool. Anyway, any accurate enough shape of the strong branching rule is too heavy and it will take a huge amount of time at every step of the tree.

2.3.2.4 Pseudocost Branching

Another common method used to tackle the variable selection is the pseudocost branching (**PC**) [22] which is based on the already performed branchings on each variable, it provides us a sort of quality score of the variable itself. The computation is fast and the results could be accurate, but the main problem is known as *cold start*, *i.e.* it doesn't know how to take decisions in the first nodes because there are not enough decisions to rely on.

2.3.2.5 The State of the Art

In practice, the most used technique for the variable selection task is the combination between the last two techniques: at the firsts nodes we can compute the **strong branching** (it will take a long time, but we will collect many very good predictions to rely on) and then we switch to **pseudocost**, this technique is known as reliability-pseudocost. The main parameter to fix is a **threshold** for where we are going to perform the switch, this threshold could be associated with the number of previous branching decisions that involved the variable or with the gap between the best bound and the best integer solution or other heuristics.

2.3.3 The Branch-and-Bound formulation as a Markov Decision Process

The sequential decisions made during branch-and-bound can be seen as a Markov decision process [23; 24].

A Markov decision process is a framework that models a decision process where some results could be considered as a random choices (by the environment), and sometime as supervised ones (by the agent). It is a very commonly used concept in fields like economy, robotics, automation and industrial production as well as optimization and reinforcement learning.

At each discrete time step, the process is in the state s_t, and an agent can perform an action a between any allowed action in the state s_t. The process responds at the next time step by "randomly" moving into a new state s_{t+1}, and giving the decision maker a corresponding reward $R_a(s_t, s_{t+1})$, but the probability of moving to s' is conditioned by choosing the action a in the next step. This probability is given by the state transition function $P_a(s_{t+1}|s_t, a)$.

In the branch and bound context, we can consider the environment to be the solver, and the brancher the agent. At the tth decision the solver is in a state s_t, which comprises the B&B tree with all past branching decisions, the best integer solution found so far, the LP solution of each node, the currently focused leaf node, as well as any solver statistics (such as, *e.g.*, the number of times every primal heuristic has been called). The brancher then selects a variable a_t among all fractional variables $A(s_t) \subseteq \{1, ..., p\}$ at the currently focused node, according to a policy $\pi(a_t|s_t)$. The solver in turn extends the B&B tree, solves the two child LP relaxations, runs any internal heuristic, prunes the tree if warranted, and finally selects the next leaf node to split. We are then in a new state s_{t+1}, and the brancher is called again to take the next branching decision. This process continues until the instance is solved, i.e. when there are no leaf node left for

branching. As a Markov decision process, B&B is episodic (it has a defined end state), where each episode amounts to solving a MILP instance. Initial states correspond to an instance being selected randomly among a group of interest, while final states mark the end of the optimization procedure. With this setup, the probability of a trajectory (specific sequence of states) $\tau = (s_0, ..., s_T) \in \tau$ depends on both the branching policy π and the remaining components of the solver, namely

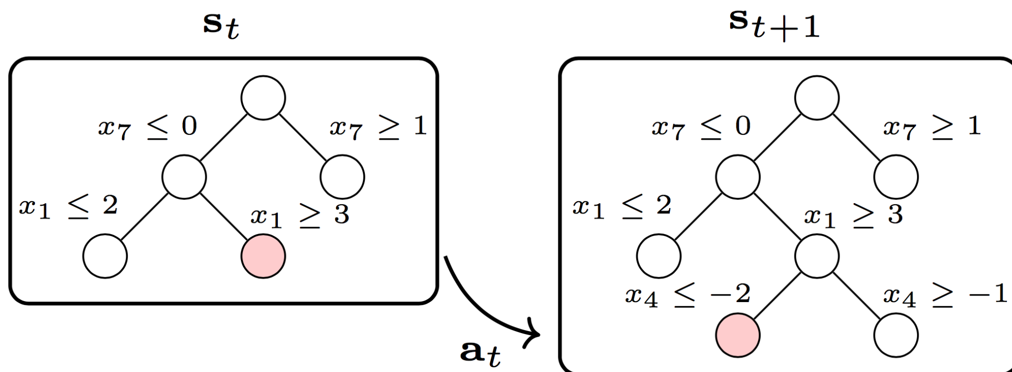
$$p_\pi(\tau) = p(s_0) \prod_{t=0}^{T-1} \sum_{a \in A(s_t)} \pi(a|s_t) p(s_{t+1}|s_t, a).$$

In practice, one might look for policies which minimize running time, but other measures that are less hardware-dependent could also be of interest, such as the negative upper bound integral for finding integer solutions quickly, the negative lower bound integral for tightening the MILP relaxation quickly, or the negative duality gap integral, which is frequently used in the combinatorial optimization community to measure the quality of a solver. Many of these measures can be formulated as expected long-term returns (feedback) for a suitable reward function, in which case solving the Markov decision is equivalent to the control problem

$$\arg\max_\pi E_{\tau \sim p_\pi}[r(\tau)],$$

where $r : \tau \to \mathbb{R}$ is the reward function to be maximized. A natural approach to solve this problem is reinforcement learning. However, this raises several key issues, which we circumvent by imitation learning as discussed in the next chapter.

Figure 2.1: Branch-and-bound variable selection as a Markov decision process. On the left, a state s_t comprised of the branch-and-bound tree, with a leaf chosen by the solver to be the next node from which to branch. On the right, a new state s_{t+1} resulting from branching on the variable $a_t = x_4$.



Chapter 3

Machine Learning

Summary

The desire of building a machine that thinks dates back at least to the time of ancient Greece [25]. The mythical figures Pygmalion, Daedalus, and Hephaestus may all be interpreted as legendary inventors, and Galatea, Talos, and Pandora may all be regarded as articial life [26; 27; 28].

When programmable computers were first conceived, people wondered whether such machines might become intelligent, over a hundred years before one was built [29]. Today the Artificial Intelligence is used in many different practical applications and is an active research topic. It can be used in automated works, in voice recognition, image analysis and also for support in medical diagnosis.

At the beginning, there was a lot of excitement for the first results, indeed there are some problems that are hard for the human intelligence, but relatively straightforward for computers. This is the case of problems that can be described by a list of formal mathematical rules. However the real challenge turned out to be the design of machines able to solve problems that are easy to understand for people, but hard to describe formally. Examples include the recognition of a face

in an image, a voice in a call, and many other processes that we do automatically, especially those that involve our perceptive senses.

3.1 What is Machine Learning?

Machine learning is a branch of the artificial intelligence that provides systems the ability to automatically learn and improve from experience without being explicitly programmed [30]. Machine learning aims to develop computer programs that, accessing data, can learn patterns by themselves.

There are two main approaches [31]:

- **unsupervised learning** builds a mathematical model to learn patterns over datasets without labels. Some applications that involve unsupervised learning methods are **clustering**, which is a grouping technique of a set of points in the space or the **anomaly** (or **outlier**) **detection**, which aims to find rare patterns that could threaten the integrity of a complex system;

- **supervised learning**, on the other hand, builds the model by knowing the labels of datasets instances. It is theoretically more problem specific and more efficient, but it also needs the presence of an expert and the availability of a dataset.

- There is also a hybrid category, the semi-supervised learning, which is computed when there are missing labels in the dataset.

The supervised learning tasks could be split in two main types: **classification** and **regression**.

- In regression tasks the goal is the estimation of a continuous value for a variable (for instance in forecast and marketing fields). It usually helps to

understand how the typical value of a dependent variable changes if the value of an independent variable is changed, keeping the others fixed.

- Classification models are those that, through the experience from already classified instances (by the supervisor), should be able to output the label (the class) of new unlabeled instances. The output of the model should be a discrete value, where each possible value represents the class of the instance. In this thesis the focus will be about classification tasks.

3.2 Classification

A classification architecture could give as output either the predicted class, or a probabilistic distribution that covers each possible class, in the first case the algorithm is named **crisp** (for instance support vector machines), otherwise it is named **probabilistic** (an example are artificial neural networks).

A common workflow is the following:

1. Design and implement the architecture of the parameterized model;

2. Split the dataset in two parts, that represent the training set and the test set; the split ratio usually varies from $2:1$ to $5:1$, it depends on the context and on the available dataset instances.

3. Perform the training phase of the model using **only** the instances of the training set, in this phase the parameters change iteratively improving (hopefully) the accuracy performances.

4. Test the quality of the model using **only** the instances of the test set, comparing the inferred class with the known one.

5. Use the trained and tested model to predict the classes of the new instances.

It is really important to use the training set and the test set on their own phase, because we need to perform the test on instances that the model has never seen before, i.e. we don't want the classification model to be specialized on the training instances, as a matter of fact we want the training phase to be as general as possible. It requires the dataset to be representative of all the possible instances, it should have enough instances and it should be well balanced.

Intuitively, the classifier has to retreive a pattern such that, if some attributes (features) of the instances have some specific values, then the predicted class should be inferred correctly. Among the most commonly used classifier we find decision trees, support vector machines and artificial neural networks.

3.2.1 Performance Measures

To evaluate the abilities of a machine learning model there are several measures. For classification, **accuracy** is one of the easiest and most natural to compute after the test phase, it is just the ratio between the correct predictions and the total ones. A mathematically equivalent measure is the **error rate**, which is the ratio between the incorrect predictions and the total ones. For a binary classifier there are also other measures, such as **precision**, **recall**, **specificity** or **f-measure**; there is not a best one between these measures, as it depends on the balance and distribution of the dataset.

In case of probabilistic classifiers, these kind of measures have some issues: they take into consideration only the result of the model and not the probability distribution over all the classes.

For instance, if the output between three classes is the distribution $[0.3, 0.3, 0.4]$ or $[0.2, 0.1, 0.8]$ then both yield correct classifications, but the second one is clearly better because of the higher probability assigned to the right class.

An error measure that takes care about the probability distribution is the

sum of squared errors (or the mean of squared errors, which is based on the same concept), defined as follows:

$$SSE = \sum_{i=0}^{N} (y_i - \hat{y}_i)^2,$$

where y_i is the $i-th$ component of the output probability distribution and $\hat{y}_i$ is always zero but when i corresponds to the class. The main issue about this error function is that it gives too much emphasis on the incorrect outputs, so it is used for regression tasks instead of classification. We will see some more details in the section about information theory.

3.3 Information Theory

Information theory is a branch of applied mathematics that aims to quantify how much information there is in a signal [32]. It was born to analyze events about measurement and transmission of information through a physical communication channel. The key measure for this theory is the **entropy**.

3.3.1 Entropy

Entropy is a measure that quantifies the amount of uncertainty involved in the value of a random variable or the result of a random process [32]. The mathematical formulation is the following:

$$H(X) = -\sum_{x \in X} p(x) \log(p(x));$$

where X is a feature that can take different values and $p(x)$ is the probability that the feature X is equal to x. In this context it represents the probability to *confuse*

the possible values: a high entropy value means that the probabilities about the possible values are similar, or that there are many different possible values; a low Entropy value means that some values are more probable than others.

3.3.2 Conditional Entropy

Information theory provides us also the definition of conditional entropy $H(Y|X)$ which means the entropy of Y where we know that X is equal to a certain value, i.e. the probability of confusing the value of Y knowing the value of X. Intuitively it should be a weighted sum of specific entropy values for each possible value of X:

$$H(Y|X) = \sum_{x \in X, y \in Y} P(x,y) log \frac{p(x,y)}{p(x)}.$$

This measure is useful in order to compute the **information gain**, which gives us the amount of information the value of X provides about y. It is defined as follows:

$$IG(Y|X) = H(X) - H(Y|X).$$

It represents the number of bits that we can save if both the ends of the transmission channel know the value of X, and in a machine learning context, it evaluates a sort of correlation between two features. This measure is widely used with decision trees methods.

3.3.3 Cross Entropy

Another important measure from the information theory about machine learning is the **cross entropy** between two probability distributions p and q defined over the same domain. It measures the average number of necessary bit in order to identify a specific instance drawn from a distribution p, if a coding scheme is

used that is optimized for an "artificial" probability distribution q, rather than the "true" distribution p. It is given by:

$$H(p, q) = -\sum_{x} p(x) \log q(x);$$

where $p(x)$ is the wanted probability and $q(x)$ is the encoding one.

Why is the cross entropy measure important? In the section about the performance measures we described why the accuracy and the error rate don't fit our needs as we would a performance measure that takes care about the probability distribution. The cross entropy represents exactly what we need and it is commonly used as performance measure.

3.4 Overfitting

As we said before, the challenge of a machine learning model is to perform well on new instances, i.e. instances that it never saw during the training phase. This ability is known as **generalization**.

Usually, when we are performing the training phase, we can compute some error measure on the training set which we want to minimize. A way to measure the generalization of a model is to compute, every step of the training phase, the **test error** (i.e. the error on the test set, also known as **generalization error**) which we want to minimize as well.

It is possible (and frequent) that, after a long training phase, minimizing the training error induces a raise of the generalization error. It means that the model is specializing on the training instances and it can't generalize and perform well on new ones. The best choice to make when this event is detected is to stop the training phase or, depending on the machine learning model, there could be some available regularization techniques to reduce it.

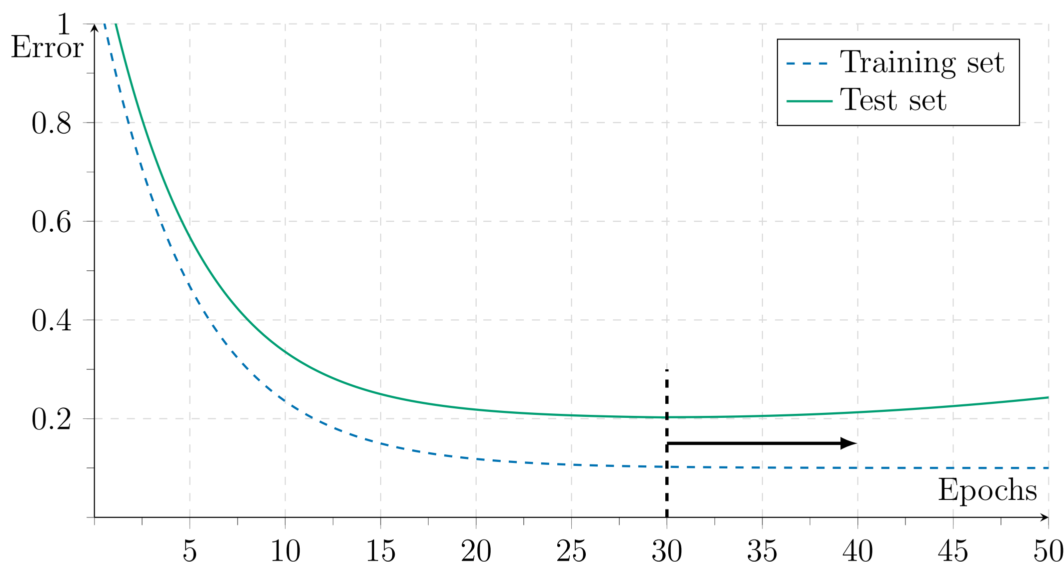


Figure 3.1: Overfitting detected after the 30th training iteration on the dataset, where the error after start to increase.

3.5 Decision trees

Decision trees (DT) are efficient and nonparametric methods [33], mainly used for classification tasks (but they can also be used in regression tasks with a slightly different formulation). A DT is represented as a tree structure graph $G = (V, E)$, where V is a set of nodes and E is a set of edges, and the leaves correspond to the possible classes.

The prediction about an instance is performed by navigating through the tree structure. At each node a test is computed on a specific attribute (feature) and depending on its result the task continues on the corresponding edge until a leaf is reached. There are different ways to build the tree and usually the simpler the structure, the more efficient is the classification task. It is proven [34] that building the minimal decision tree consistent with a training set is an NP-hard problem.

3.6 Support Vector Machines

Support vector machines (SVM) are hyper plane classifiers [35]: every instance of the dataset is represented as a point in an input space and the algorithm searches a hyper plane (in case of binary classification) which optimizes the separation between the points of the two classes. After the training phase, the hyper plane is fixed and the new instances are classified according to the portion of the semi space where they belong.

Choosing the hyper plane that optimizes the separation between points belonging to different classes is an optimization problem. Ideally, we should consider the closest points to the hyper plane and try to maximize their distance to it. If we achieve this we obtain the maximum margin hyper plane, and its closest points are called *support vectors*. This definition makes support vector machines crisp classifiers. If we take a look at the mathematical expression used to classify the new instances we can extract some more information.

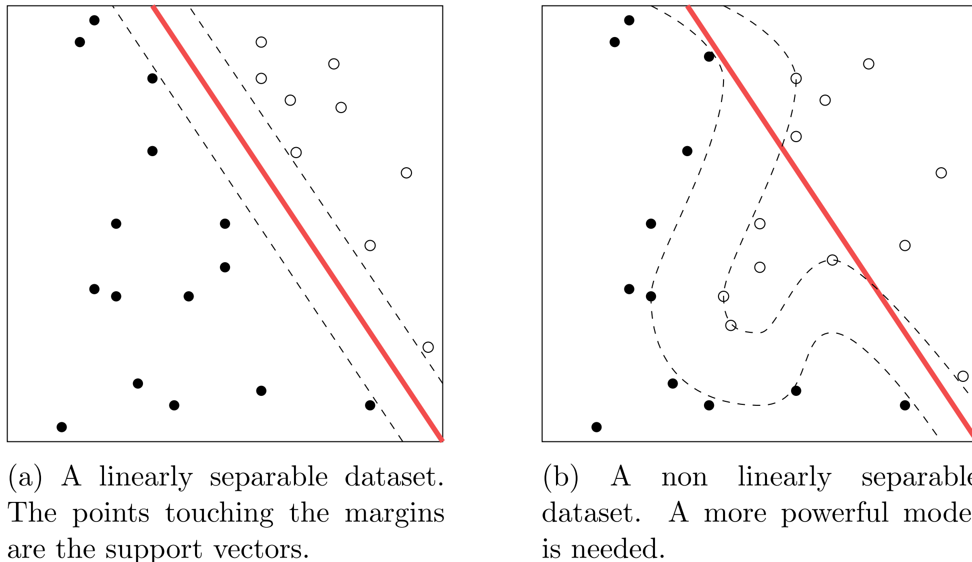
The hyper plane equation is:

$$\sum_{i=0}^{n}(\omega_i \cdot x_i) + I = 0,$$

where n is the number of feature, i.e. the dimension of the hyper space, the ω_i are the weights associated to each feature, the x_i are the inputs of the SVM, and I is the intercept.

For each new point after the training phase, computing the left side of the described expression, we will have a positive value in case of one class, or a negative one in case of the other class. If it is exactly zero, then there are no reasons to chose between the available classes. The value provides a useful information. Intuitively, if it is far from zero, then the prediction should be more accurate than a value close to zero.

The main problem of this kind of support vector machines is the hypothesis that the problem is linearly separable, the figure 3.2 shows the differences in the case of a 2-dimensional space.



(a) A linearly separable dataset. The points touching the margins are the support vectors.

(b) A non linearly separable dataset. A more powerful model is needed.

Figure 3.2: Comparison between a linearly separable dataset and a non linearly separable one.

There is a solution to deal with non-linearly separable dataset, which is called kernel trick [36]. The main idea to apply this trick to SVMs [37] is to replace the dot product with a nonlinear kernel function, which measures the similarity between two points. This allows the algorithm to fit the maximum-margin hyper plane in a transformed feature space, which is usually higher-dimensional than the original space. The figure 3.3 shows how a kernel function can transform the space.

On one hand, SVMs are simple models and are very fast both for training and predicting new instances. On the other hand, there are many dataset on which SVMs don't perform well; they are too sensitive to noise and to class imbalance; and in presence of outliers the performances can dramatically decrease.

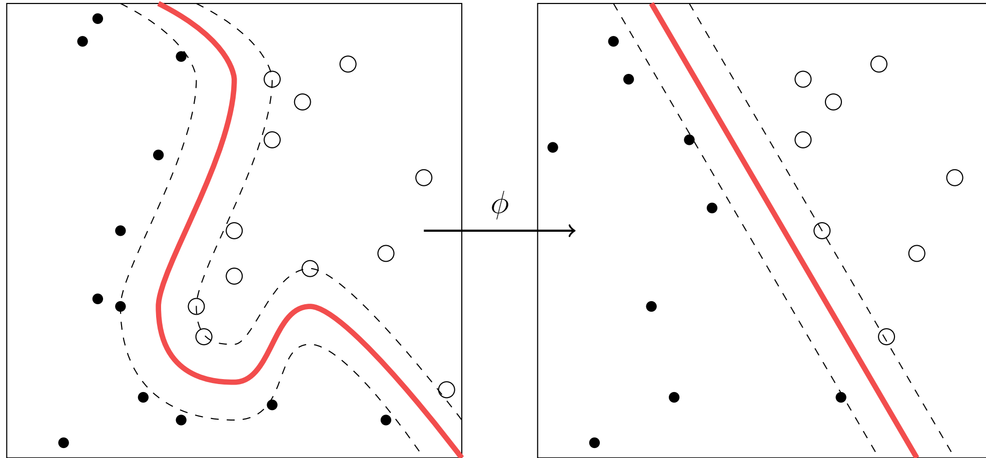


Figure 3.3: A kernel trick transorm the original space in a new feature space where the dataset becomes linearly separable, i.e. with a hyperplane

3.7 Artificial Neural Networks and Deep Learning

Artificial neural networks are one of the most powerful classification methods, they are inspired by biological neural networks and are basically a set of nodes (neurons) linked in a hierarchical structure.

The definition of a single neuron is based on a famous proposal from 1943 [38], which represents the neuron as a thresholded linear combinator, with several inputs and one output.

Until 1986, this model didn't find a real application because it was not able to solve useful problems. But in that year the idea of the **backpropagation error** [39] was introduced, which let the network learn through examples modifying some internal values related to the links between the neurons. This method is based on **gradient descent**, which is an optimization method that aims to find the local minimum of a function in an n-dimensional space.

To complete the structure of an artificial network, we have to define what is

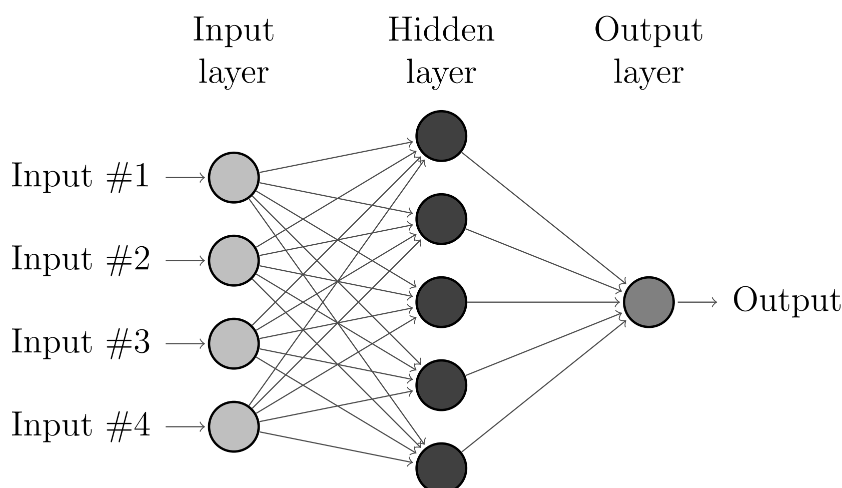


Figure 3.4: A simple representation of a fully connected feed forward network with one hidden layer.

a link, how to threshold the outputs and how to design the hierarchical structure (figure 3.4 shows a classical representation of an artificial neural network):

- A link is a relation between two nodes and it has a real value, which represents the **weight** of the link.
- The basic hierarchical model is a fully connected feed-forward network, where each neuron is linked with all the next layer's neurons and **only** with them. The number of layers is often fixed to three (input, hidden and output layer). However there are other architectures, for example a fully-connected layer doesn't perform well under some circumstances, like object recognition in images. Other possible architectures include **convolutional**, **recurrent**, **graph convolutional** or even other layers. If we put more hidden layers, the model is usually referred as a **deep neural network**. The result is powerful but also heavier than the standard three-layers architecture. These became useful models only in the last decade, thanks to hardware advances such as programmable GPUs.

- The threshold is defined as a mathematical function that takes in input the output of a neuron (a real value) and output another number. The first used threshold function was the sigmoid, defined as:

  $$\Phi(x) = \frac{1}{1+e^{-x}}.$$

  This function is differentiable and non linear; this last feature is useful to face up to the noise. However it is not widely used now. Other common threshold (or activation) functions are the tanh (hyperbolic tangent, which is similar to the sigmoid) and the **ReLU** (Rectified Linear Unit). The latter is currently the most frequently used activation function and it is defined as follows:

  $$R(x) = max(0, x).$$

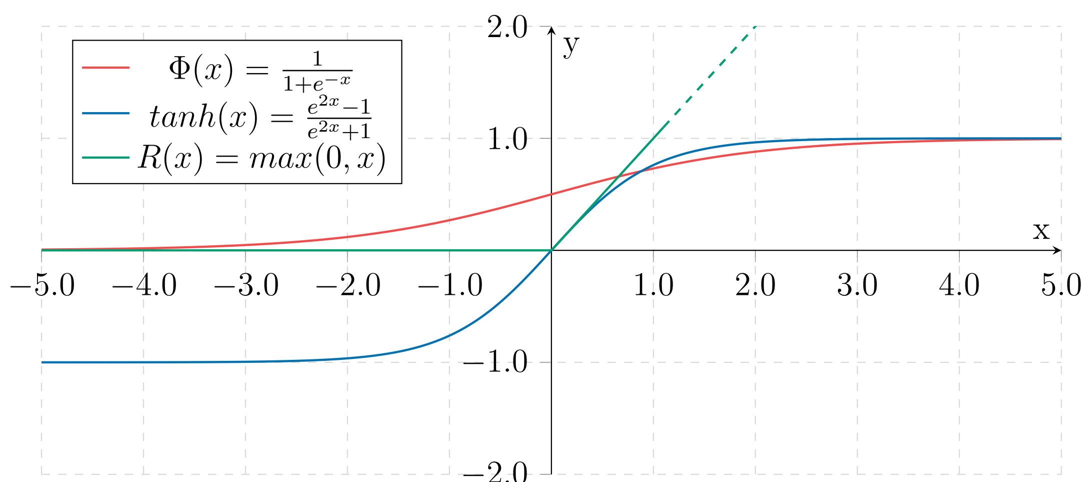


Figure 3.5: A comparation between the three most used activation functions, Sigmoid, Tanh and ReLU.

Other design parameters include the **number of epochs**, one epoch is a whole training phase iteration over the dataset; **how many nodes** for each hidden layer, indeed the first and the last layers have a fixed size since the first must have the

size of the input, and the last one must encode the output, so the number of the available classes; the **learning rate**, which is a hyper parameter that controls how much the network adjusts the weights with respect to optimizer, it should be big enough to avoid local minima loops and to make the network fast, but small enough to guarantee more precision, it could be changed over the epochs, usually decreased; a **stop criterion**, such as a time-out, a threshold on the performance measure or a detected overfitting situation.

There are two kinds of learning: in **stochastic learning** the backpropagation is computed after each forward step (i.e. the computation from the input to the output, through the hidden layers), it is slower and more noise sensitive than the **batch learning**, which computes several forward step before the backpropagation. It is faster and more robust to the noise.

Chapter 4

Methodology

Summary

This chapter describes our machine learning architecture, including implementation details, motivations about the tools used and an overview of the alternatives.

4.1 Implementation Tools

4.1.1 Backend Solver

A combinatorial optimization solver is a software that takes as input a formal description of an operation research problem, computes the solution using any available tool it knows about the class of the provided problem, and outputs the result. There are simple solvers embedded in Microsoft Excel and in Matlab, these solvers are basic and are not useful in case of big problems, both for the efficiency and some internal limit, such as the number of decision variables. Usually these kinds of software are very complex and the licenses for commercial solvers are really expensive, they should be able to solve many kind of optimization problem and they have to guarantee good computational performances also

with big instances.

CPLEX is one of the most powerful available solvers, it is developed by **IBM** and it is averagely the fastest available one. It can be seen as a black-box solver due to the impossibility to see any details about the used algorithms and settings. Other famous and modern solvers are Gurobi, MOSEK, SCIP, XPRESS, GLPK and others. Between the multitude of available solvers, we discarded the not famous and less used ones just for a competitive purpose, then we discarded also IBM Cplex, since it is not open-source and it would have been impossible to retreive all the informations about the state of the solver.

The final choice has been **SCIP** because it is one of the fastest non-commercial solvers for mixed integer programming (linear and not), it allows for total control of the solution process and the access of detailed information down to the guts of the solver [40], which is exactly what we need in order to implement a custom branching rule.

4.1.2 The Deep Learning Library

Another important architectural choice for the implementation is the software library that implements the basic tools for deep learning. Between the most famous machine learning libraries there are Tensorflow, Theano, Keras, Caffe, Pytorch and others.

The main separation between these library is the graph definition, which can be static or dynamic.

- In a **static graph** defined environment, the definition of the model must be done before running it, defining all the tensors that are involved in the computation and all the details must be given in advance. Then the running code can be executed and the tensors will be replaced with external data.

- In a **dynamic** execution the flow is more imperative, changes are allowed on the go as most of the imperative programming language, it is very flexible and more intuitive to use. This flexibility comes with a price, the performances are usually worse than a compiled graph environment.

PyTorch is one of the most important dynamic libraries, it is easy to use, the commands are very well integrated with the programming language, it is easy to debug and the performances are close to the static graph libraries.

Tensorflow is its main competitor, it was born as a static library and it is one of the most supported by the scientific community, it is very efficient and it allows the deployment of computation across a variety of platform (CPU, GPU, TPU) and from desktops to clusters of servers to mobile and edge devices.

In the last years the team worked on a variety of it, **Tensorflow Eager**, which implements the dynamic graph declaration, increasing the expressiveness but paying the performance price. It is not as complete as the standard version, some methods are not implemented yet and some objects are not compatible. The main advantage is that it is easy to convert a dynamic model into a static one, so the suggestion from the developing team is to use the eager mode for experiments and the modeling phase, then switch to the static one to maximize the performances.

Since the main issue of the strong branching is the efficiency, the choice about the library has been Tensorflow, using it in eager mode during the experimental phase and switching to the compiled graph execution for the deployment.

4.1.3 Programming Language

There are many languages supported by the main machine learning libraries (Java, Javascript, Python, R and also C/C++), but the most used and supported is Python. It is directly supported by Tensorflow and it is well considered

by the scientific communities, the speed is not its killer feature, however it can also interface with other languages, indeed many libraries where performances are important (such as Numpy, the most used standard for Python's collections) are written in C/C++ (a language known for its efficiency). A very good feature of Python is that it is very easy to read and write, the syntax is light, but it allows designing complex software architectures. It is object oriented (organized about the concept of objects instead of actions, and data instead of logic), functional (functions can be assigned to variables) and interpreted (the code runs line by line without a compiling phase that could take time and make the debug hard to understand, it comes at the price of the performances as said before).

4.2 Imitation Learning

Since there is no "perfect" branching rule, in this project the choice is to learn directly from an expert branching rule, an approach usually referred to as imitation learning [41]. More precisely, we train by behavioral cloning [42].

We first run the expert on a collection of training instances, recording its decision with the state at the time the decision was made, resulting in a dataset of expert state-action pairs $\mathcal{D} = (\mathbf{s}_i, \mathbf{a}_i^*)_{i=1}^N$. To learn our policy, we minimize the cross-entropy loss

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(\mathbf{s},\mathbf{a}^*)\in\mathcal{D}} \log \pi_\theta(\mathbf{a}^* \mid \mathbf{s}). \tag{4.1}$$

On one hand the imitation learning performs well because it is a supervised method and there are many techniques to tackle it, on the other hand, imitation learning will never perform better than the expert rule. So it is known in advance that the performances of our approach are bounded, but if the model is fast enough to output good enough branching decisions in a lower time it would be a success.

4.2.1 Imitation Target and Dataset Generation

Training by imitation requires defining a target policy to learn from. The choice has been **Strong Branching**, introduced before. Its main downside, the high computational cost, is less important here since we can in principle take as much time as needed to produce a dataset of decisions on the training set of instances.

However, during evaluation the learning brancher will inevitably make mistakes, after which it will have to take decisions in states unlike those visited by the expert brancher. This is a well-recognized problem with behavioral cloning [43; 44]. Solutions that address this issue include inverse reinforcement learning methods [41; 45], which are usually very computational demanding.

On the other hand, in our context we are free to design the expert in a way that could improve learning. A simple and adopted solution is to flip a biased coin at every branching decision from the initial state. If the coin flips heads, say with 95% probability, we follow the decision of a mediocre brancher, such as pseudocost branching [22]. If we hit tails, we use strong branching and record its decision with the current state.

The resulting dataset of expert decisions thus records strong branching in a wider range of states, which helps our policy recover from mistakes during evaluation. Note that we never record and learn from the mediocre brancher, only strong branching decisions. As an additional advantage, this procedure makes state-action pairs closer to having been sampled independently, as assumption upon which supervised learning relies for good performance. We call the overall procedure explore-then-strong-branch and demonstrate that it improves generalization.

4.3 State Encoding

We saw in Section 2.3.3 that branch-and-bound variable selection can be interpreted as a Markov decision process. However, as described, solver states are so large and complex that it seems very challenging to learn a policy directly from the complete states. Therefore, we approximate the states through manual feature engineering. This technically turns the process into a partially-observable Markov decision process [46], which is a generalization of the MDP where the agent can not observe the underlying state, with the state approximation as observation vector. We claim that, for the problem at hand, it is reasonable to use such an approximation with a well-chosen set of features, and support our choice by noting that an excellent variable selection policy such as strong branching seems to do well despite relying only on a subset of the solver state.

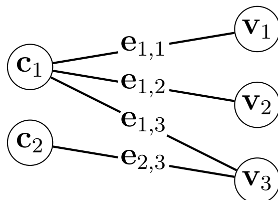


Figure 4.1: A bipartite state representation $\mathbf{s}_t = (\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$ with $n = 3$ variables and $m = 2$ constraints. Here $\mathbf{C} \in \mathbb{R}^{m \times c}$ represents the feature matrix of the constraint nodes on the left, $\mathbf{V} \in \mathbb{R}^{n \times d}$ the feature matrix of the variable nodes on the right, and $\mathbf{E} \in \mathbb{R}^{m \times n \times e}$ the (sparse) feature tensor of the edges between constraints and variables.

We encode the state $\mathbf{s}_t$ of the B&B process at time t as a bipartite graph with node and edge features $(\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$, described in Figure 4.1. On one side of the graph are the constraint nodes, one per row in the current node's LP relaxation. On the other side are the variable nodes, one per LP column, and an edge $(i, j) \in \mathcal{E}$ connects a constraint node i and a variable node j if the latter is involved in the former, that is if $\mathbf{A}_{ij} \neq 0$ ($\mathbf{A}$ is the adjacency matrix). Note that under proper restrictions in the branch-and-bound solver (namely, by disabling

node cuts), the graph structure is the same for all LPs in the B&B tree as it is in the original MILP. For each edge (i, j), we attach the constraint coefficient $\mathbf{A}_{ij}$ as the only feature.

4.4 Policy Parametrization

In this section the focus is on the architectural design of the network model that parametrizes the policy. Since our state encoding is a graph structure and its size could vary between the instances, the most promising architecture that can deal with this kind of data is the **Graph Convolutional Neural Network** (GCNN) [47].

4.4.1 Graph Convolutional Neural Network

There are many problems of the real world that involve data with a specific graph based structure, for example social networks, molecolar and proteins structures, hypertext in the web, document classification in citation networks and others. These structures can't be represented by the standard neural network layers and in the last five years it has been an important research topic.

One graph can be seen as an extension of a grid where there are no limits on the number of neighbours, indeed in a 2-dimensional grid there are four or eight neighbours, in a 3-dimensional grid there are six, eighteen or twenty-six neighbours, it depends on the definition of distance; but once the distance is chosen, the number is fixed and equal for all the nodes.

A GCNN should be able to take in input a graph structure usually summarized by $G = (V, E)$, we can encode it as follows:

- for each node there are D features that describe the node, the total number of node is N. We can represent this data in an $N \times D$ matrix;

- the structure of the graph is defined by the edges, which can be represented by the adjacency matrix A, this is a sparse matrix with a value in correspondence of a match between two nodes, and zero anywhere else. The value is usually one, but there may be different values.

So the output of a layer will be a $N \times F$ matrix, where F is the number of the output features per node (specific of the layer), then there is a possibility to add some pooling operation like in the common convolutional layer [48].

Each GCNN layer can be written as non linear function that represents the input-output relation:

$$H^{(l+1)} = f(H^{(l)}, A),$$

with $H^{(0)}$ the input data of the network, L the number of layers, $H^{(L)}$ the output of the network. So we can try to define the function f to realize a graph convolution propagation rule.

Every neural network layer has an activation function σ and a weight matrix W, usually the activation is applied on the output and the weight matrix multiplies the output of the previous layer, so an intuitive propagation rule could be the following:

$$f(H^{(l)}, A) = \sigma(AH^{(l)}W^{(l)}).$$

This formulation has two main issues:

- the multiplication A means that we will sum all the features of the neighbours of each node excluding the node itself unless there is a loop on it;
- A is usually not normalized so the multiplication with it could potentially modify the scale of the features vectors.

The first issue is easy to fix by adding the matrix I to A, by forcing a sort of cyclic loop on each node of the graph. For the second one we need to normalize

it and we can consider a matrix D as a diagonal node degree matrix, then we can compute the reverse of D and multiply it with A. The result of the multiplication $D^{-1}A$ represents the average of the neighbours' features, but in practice a symmetric normalization produces better results, so it could be replaced by $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$. If we combine these two tricks we can obtain the propagation rule presented by Kipf and Welling:

$$f(H^{(l)}, A) = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}),$$

where $\hat{A} = A + I$ and $\hat{D}$ is its diagonal node degree matrix.

These GCNNs exhibit many properties which make them a natural choice for graph-structured data in general, and MILP problems in particular:

1. they are well-defined no matter the input graph size;
2. their computational complexity is directly related to the density of the graph, which makes it an ideal choice for processing typically sparse MILP problems;
3. they are permutation-invariant, that is they will always produce the same output no matter the order in which the nodes are presented.

Our model takes as input our bipartite state representation $\mathbf{s}_t = (\mathcal{G}, \mathbf{C}, \mathbf{V}, \mathbf{E})$ and performs a single graph convolution, in the form of two interleaved half-convolutions. In detail, because of the bipartite structure of the input graph, our graph convolution can be broken down into two successive passes, one from variable to constraints and one from constraints to variables. These passes take

the form:

$$\mathbf{c}_i \leftarrow \mathbf{f}_{\mathcal{C}}\left(\mathbf{c}_i, \sum_{j}^{(i,j)\in\mathcal{E}} \mathbf{g}_{\mathcal{C}}\left(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j}\right)\right), \quad \forall i \in \mathcal{C},$$

$$\mathbf{v}_j \leftarrow \mathbf{f}_{\mathcal{V}}\left(\mathbf{v}_j, \sum_{i}^{(i,j)\in\mathcal{E}} \mathbf{g}_{\mathcal{V}}\left(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j}\right)\right), \quad \forall j \in \mathcal{V},$$

with $\mathbf{f}_{\mathcal{C}}$, $\mathbf{g}_{\mathcal{C}}$, $\mathbf{f}_{\mathcal{V}}$, $\mathbf{g}_{\mathcal{V}}$ multilayer perceptions with prenorm layers (described in Section 4.4.4). From a mathematical point of view this property is given by the fact that the adjacency matrix of a bipartite graph can be split in two different adjacency sub-matrix, indeed there are no edges between variables or between constraints, but only from variables to constraints or viceversa.

Following this graph convolution layer, we obtain a bipartite graph with the same topology as the input, but with potentially different node features. We obtain our policy by discarding the constraint nodes, applying a fully connected layer to the variable nodes and using a masked softmax activation to produce a probability distribution over the admissible variables for branching (i.e., the fractional variables). Figure 4.2 provides an overview of our architecture.

4.4.2 Implementation

Another important advantage about using Tensorflow is the support for sparse tensors, the adjacency matrix could be really big and without this support every attempt could be vain. Thanks to it there are many structures and methods that we can use to have more control about the graph convolutions and reduce the computational time.

Another important used tool is Keras, it is a high level API written in python that runs on the top of Tensorflow. It allows a higher expressiveness, so a faster prototyping, it implements many neural network architectures and layers as well

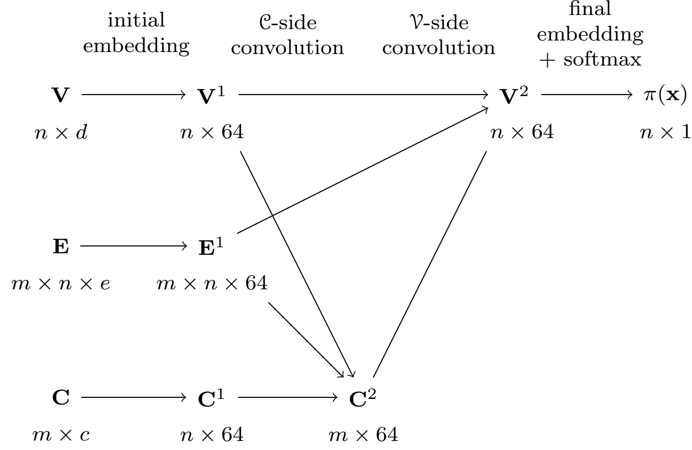


Figure 4.2: Our bipartite graph convolutional neural network architecture for parametrizing our policy π.

as any other useful functions (such as activations, losses, optimizers, etc).

The model architecture is coded as an implementation of the Keras Model interface, inside it there is the definition of some fixed values, such as the embedding size, the number of variables, constraint and edge features, the activation function, the weights initializer and all the layers; it takes a bipartite graph as input. The firsts layers are embedding layers, so they are fully-connected layers that expand the feature vectors to the fixed embedding size. Then they feed the input of the first graph convolutional layer and after the convolutions, finally there are other two fully connected output layers. The Figure 4.4 shows the architecture of the model.

Also the bipartite graph convolutional layer implements the interface Keras Model, indeed there are definitions of other layers inside, mixed to build the graph convolution described before in a more efficient way. First there are three parallel embedding layers, this time also the edges are expanded to the same embedding size as constraint and variable features. Constraints and variables are then indexed (using the efficient tensorflow.gather function), according to the

edge indices, to reach the same size as the embedded edges, they are summed in the so called VCE matrix that feeds a feature layer. This layer normalizes the input, applies the activation function and then feeds a fully connected layer. The core of the convolution is obtained by the sum over the neighbours using the tensorflow.scatter_nd method. In the end there is another normalization layer and an output layer; the Figure 4.5 shows the non linear architecture of the graph convolution.

Every sequential part of the model is usually wrapped in a Keras.Sequential model, to guarantee a clean, readable and optimized code.

4.4.3 Hyper-parameters tuning

As said in section 3.7 there are many parameters that define a deep learning model. In this section it is described how all the hyper-parameter of the model are set.

- The maximum number of epochs is fixed to 1000, but one epoch is considered the iteration over 312 batch. This is done to pick up random samples to compose the batch without discarding those samples until the next epoch. This practice is useful to randomize the choice.

- The batch size is limited to 32, the main issue is the physical available memory on the GPU, with 32 it requires more than 11gb, and the used GPU has 12gb, making it impossible to go over 32.

- The Learning Rate is initially set to 0.001 and it is then decreased according to reduce on plateau technique, which is an adaptive method that reduces the value by a factor, in this case 5, after a certain number of epoch without improvement; this parameter is called patience and it is set to 10.

- The termination condition is the early stopping, it means that after a number of epochs without improvement, 20 in this case, it stops the training phase.
- For the optimizer one of the most common choices is the AdamOptimizer [49], which is provided by Tensorflow and guarantees a fast and accurate convergence.

Other more specific hyper-parameters are related to architecture modifications that can improve performance or speed, such as normalization layer, implementation of an affinity layer or the graph convolution through the mean instead of the sum.

To implement the mean over the neighbours instead of the sum, we should count, for each node, the number of neighbours and divide the already computed sum by its count.

The **Affinity Layer** is a method [50] that aims to compute a kind of *affinity* between a node and its each neighbours: the idea is that some edges could be more important than other. This layer act before the convolution and in parallel with the feature layer, so there are other embedding fully-connected layers and indexing operations that feed the affinity fully-connected layer. Then the feature and affinity layers are multiplied by an element-wise multiplication.

The choice of the **normalization layer** has initially been the layer normalization [51], a modern variant of the batch normalization [52], which is computed for each batch. It is very accurate and it let the model avoid the exploding gradient situations, but it is also heavy to compute. So we decided to implement a new kind of normalization layer that is based on the standard formula of the normalization, we call it **prenorm layer**. This layer introduce a pre-train phase to compute the normalization parameters for each prenorm layer, but once they are computed, their values are fixed and nothing else is computed, so it is faster

and also let us keep the dataset with the raw un-normalized data. There are details about the implementation in the section 4.4.4

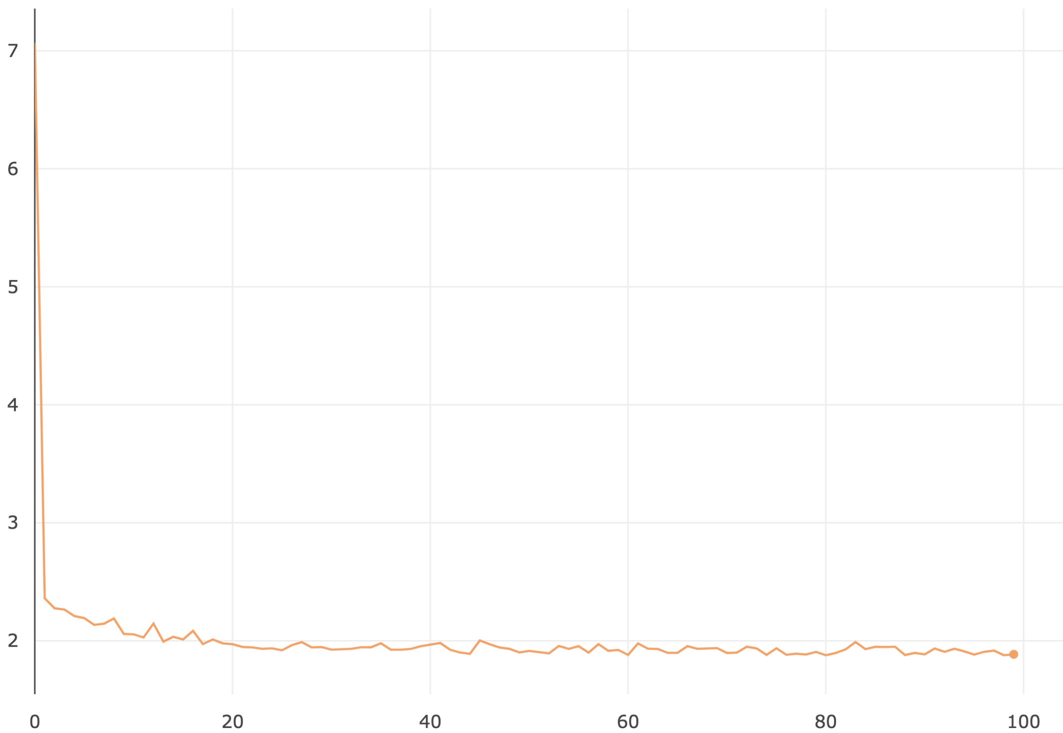


Figure 4.3: This figure shows the test error (cross-entropy) through the epochs during the training phase. We can observe that there is no overfitting and the curve is almost smooth and stable.

4.4.4 Pre-norm Layer

In the literature of GCNN, it is common to normalize each convolution operation by the number of neighbours [47]. We argue that this might result in a loss of expressiveness for our model, as it then becomes unable to perform a simple counting operation (e.g., in how many constraints does a variable appears). Therefore we opt for un-normalized convolutions, which improves our generalization performance on larger problems as shown in the table 6.2. However, this introduces a

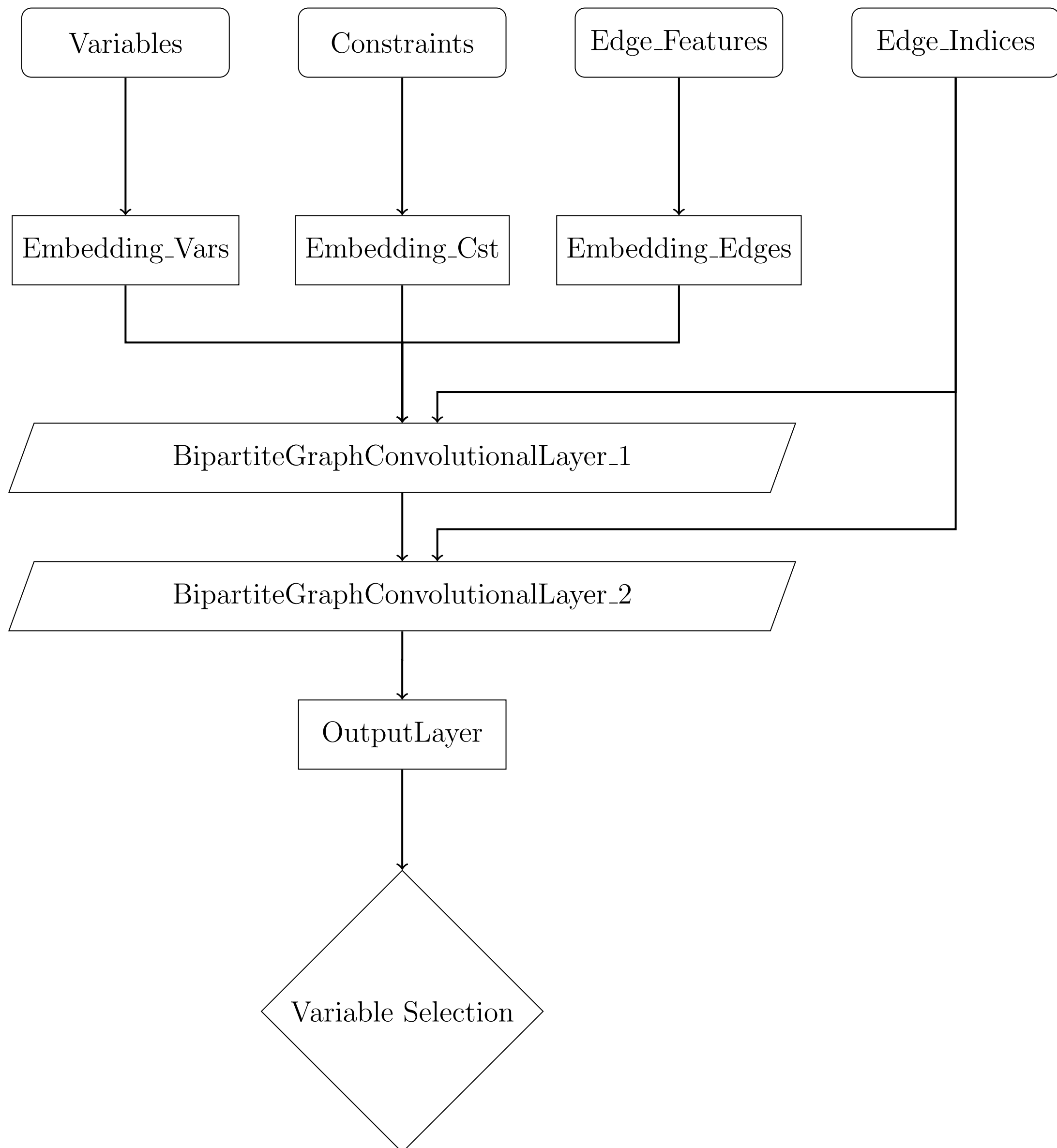


Figure 4.4: The architecture of the model is not the standard sequential one, but it begins with three parallels layers, that feed the first graph convolutional layer and then is a sequential model until the output.

weight initialization issue. Indeed, weight initialization in standard CNNs relies on the number of input units to normalize the initial weights [53], which in a GCNN is unknown beforehand and depends on the dataset. To overcome this is-

sue and stabilize the learning procedure, we adopt a simple affine transformation $\mathbf{x} \leftarrow \frac{\mathbf{x}-\beta}{\sigma}$, which we call a *prenorm* layer, applied right after the summation in the convolution operation. The β and σ parameters are pre-computed with respectively the empirical mean and standard deviation of $\mathbf{x}$ on the training dataset, and fixed once and for all before the actual training. Namely, we run through each neural network layer one at a time, starting from the inputs, and when we encounter a prenorm layer we fix its parameters. After this pretraining procedure, we proceed with the neural network training. The implementation of this class is based on the interface Keras.Layer, that allows us set for each variable of the layer the flag *trainable* to False.

4.5 Evaluation Procedure

After implementing the model, it has not been easy to validate which was the best one, we tried many different settings of the model and the best loss function score has been obtained by the model with three convolutional layers, the convolution with the sum (not with the mean), the layer normalization instead of the prenorm, and the usage of the affinity layer.

But we observed that this configuration was much slower than a configuration with two graph convolutional layers, and the loss score was only slightly worse. A similar situation has been detected about the layer normalization and the affinity layer, which improve the performances, but beeing slower.

After observing these results it was almost impossible to decide which model works better on new instances considering both loss score and speed, so we need to embed the trained models in SCIP and evaluate them.

Set Cover Training Results

model	Cross-Entropy	acc@1	acc@5	acc@10	time
gcnn	1.887 ± 0.004	39.3 ± 0.1	83.8 ± 0.1	95.5 ± 0.0	5.26 ± 0.21
gcnn+a	1.879 ± 0.003	39.5 ± 0.1	84.1 ± 0.1	95.7 ± 0.0	6.88 ± 0.24
gcnn+m	1.891 ± 0.004	39.3 ± 0.1	83.6 ± 0.2	95.4 ± 0.1	5.61 ± 0.23
gcnn+l	1.883 ± 0.002	39.3 ± 0.1	83.9 ± 0.1	95.5 ± 0.0	5.69 ± 0.28
gcnn+n	1.895 ± 0.010	39.2 ± 0.2	83.6 ± 0.2	95.4 ± 0.1	$\mathbf{5.19} \pm 0.24$
gcnn+3	$\mathbf{1.834} \pm 0.012$	$\mathbf{39.8} \pm 0.3$	$\mathbf{84.5} \pm 0.3$	$\mathbf{96.2} \pm 0.2$	8.76 ± 0.27

Table 4.1: Training results on maximum set cover instances. M is for the mean normalized convolution, L is for the usage of layernorm instead of our prenorm, A is for the usage of te affinity layer, N is for no normalization after the convolutions, 3 is for three convolutional layers. The measured time is the amount needed for a mini-batch forward step.

4.5.1 Embedding the model in SCIP

This task is done by implementing the SCIP interface Branchrule, which basically allows the embedding of new custom rules and load a trained GCNN model as a policy. The loading is done using also the tensorflow.defun() method that compiles the computational graph for a faster evaluation, indeed it was written using Tensorflow eager.

Then we need to overwrite the method branchexeclp(), which is the code that will be executed at each new node for the variable selection, so we need to extract the state from the solver, convert it to a tensor and give it as input to the policy.

The output of the policy is a distribution with the highest value in correspondence to the variable to select, but before picking up the variable with the highest value we have to apply a mask that deletes the non selectable variables. When the best variable after applying the mask is selected, it is given as input to the model.branchVar() method of Scip to continue the algorithm with the selected variable.

4.5.2 Ablation Study

After evaluating different instances we noticed that the best model in term of the adopted performance measure was not the best in term of solving time. The most evident result is model with three graph layers, that showed a great behaviour in the little instances both in term of time and explored nodes, but on medium and big instances the time was not competitive with other architectures, and also the number of explored nodes was higher.

Set Cover Test Results

	Easy			Medium			Hard		
model	Nodes	Time	Wins	Nodes	Time	Wins	Nodes	Time	Wins
gcnn	190	**3 s.**	**5/10**	3296	**56 s.**	**6/10**	**116704**	**2549 s.**	**6/6**
gcnn+m	190	**3 s.**	0/10	3391	60 s.	0/10	146111	3097 s.	0/6
gcnn+a	183	**3 s.**	1/10	3301	59 s.	1/10	141254	2862 s.	0/6
gccn+l	190	**3 s.**	1/10	3298	58 s.	2/10	145211	2979 s.	0/6
gcnn+n	188	**3 s.**	1/10	3304	58 s.	1/10	123605	2734 s.	0/6
gcnn+3	176	**3 s.**	2/10	3861	105 s.	0/10	455005	9675 s.	0/6

Table 4.2: Test results on maximum set cover instances.

This event is explained by a high level of specialization, indeed the training phase has been done on little instances. Using a mean normalization after the sum convolution does not improve the result and make the computation slower, it was noticed also at training time and probably it is caused by the loss of informations about the number of neighbours. Also the usage of the affinity layer did not improve the solving time, indeed it involves heavy computations, and it is not worth it. On the other hand the prenorm layer has outperformed the layer normalization and the un-normalized convolution, confirming that a simple and fast approach could be better than a complex and heavy one.

So with this knowledge we performed an ablation study on the baseline model

by modifying each hyper-parameter to validate the architectural choices. At the end of it every change on it provided worse performances, even if the gap was sometime very tight. Empirical results of this ablation study can be seen in the table 4.2.

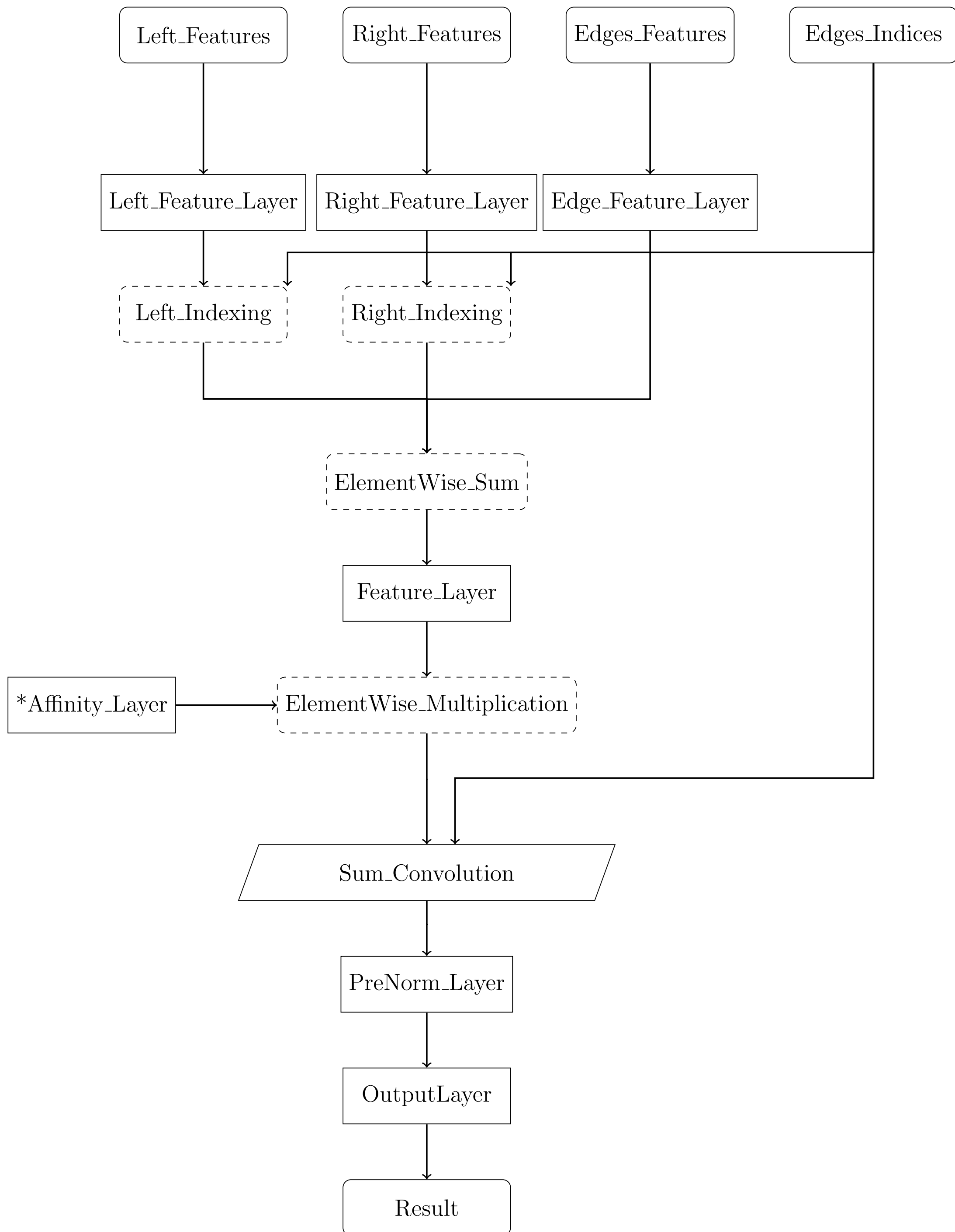


Figure 4.5: The rectangles with rounded angles are matrices, the rectangles are fully connected layers, the dashed rounded rectangle are mathematical operation, and the sum convolution is represented by the parallelepiped. The Affinity layer is a parallel task with the feature layer, it is not represented because it is exactly the same and it would have made the graph unreadable.

Chapter 5

Related Works

Summary

In this chapter there is a study that goes beyond the state of the art of the branch and bound variable selection; these alternative procedures have also been implemented in order to perform a comparison over different kind of problems with our model and default Scip brancher.

5.1 Regression Model

In [54], the authors perform imitation learning of strong branching across instances by fitting a regression model to predict strong branching scores from feature vectors that describe a variable within the current B&B state. They train on synthetic bin packing and multiple knapsack, and evaluate on MIPLIB [55; 56], a standard benchmark of heterogeneous instances.

5.2 Learn to Rank Model

In [57], the authors also perform imitation learning of strong branching across instances, but by fitting a learning-to-rank model. A learn to rank model is a machine learning model that aims to order a list of items, by giving a numerical or ordinal score for each of them. In this context it is used to predict strong branching orderings of variables from feature vectors, training and evaluating on time-dependent traveling salesman problems. Since these approaches solve the same problem as ours, we compare against them in Section 6.3, and demonstrate that our contributions offer substantial improvement over these baselines. Implementation details can be found in Section 6.3.

5.3 Imitation learning and SVM

A different use case is explored in [58]. They also learn a branching policy by imitation learning from strong branching – however, within the instances themselves. Namely, when solving an instance, they first run strong branching for a few hundred nodes, recording states and strong branching scores. Then, a support vector machine is trained to rank variable pairs [59] to imitate strong branching preferences. Finally, for the rest of the B&B process, branching occurs on the variable ranked highest by the machine learning model. This yields a branching strategy that adapts to combinatorial problems it might have never seen before. A downside is that the machine learning model must be kept simple, as the training data is limited and too much training time would erase potential gains from better decisions. Thus, whereas our approach is adapted to the recurrent solving of instances from the same distribution, their approach is designed for the on-the-fly solving of heterogeneous instances.

5.4 Portfolio-based Methods

Less end-to-end approaches include portfolio-based methods [60], which are strategies to run several algorithms in parallel, sometime different each other, assigning them different amount of CPU if needed.

In [61] the model learns a clustering-based classifier to pick a variable selection rule at every branching decision up to a certain depth. In contrast, [62] use the fact that many variable selection rules in B&B explicitly score the candidates, selecting the one with the highest score. They propose to learn a weighting of different score-based variable selection rules to combine their strengths. Other works learn variable selection policies, but for less general algorithms than B&B. In [63] learn a variable selection policy for SAT (boolean satisfiability problems) solvers using a bandit approach; it is a method where there are competing choices among a limited set of resources, these choices have to maximize a gain, but their properties are only partially known. [64] extend their work by taking a full-fledged reinforcement learning approach with graph convolutional neural networks. Unlike our approach, these works are restricted to conflict-driven clause learning methods in SAT solvers, and cannot be readily extended to B&B methods for arbitrary combinatorial optimization problems. In the same vein, [65] learn by imitation learning a variable selection procedure for SMT (Satisfiablity Modulo Theories) solvers that exploits specific aspects of this type of solver.

5.5 Other Methods

Some work in the literature has focused on learning other aspects of B&B algorithms than variable selection. [24] learned a node selection heuristic by imitation learning of the oracle procedure (i.e. a function that always chooses the right decision) that expands the node whose feasible set contains the optimal solution, while

[66] learned primal heuristics for B&B algorithms. More generally, many authors have proposed machine learning approaches to fine-tune exact optimization algorithms, not necessarily for MILPs in general: a recent survey is provided by [67]. Beyond those already cited, recent works include [68; 69; 70; 71; 72; 73]. Finally, this work fits within the wider literature that uses machine learning methods for combinatorial optimization, which have focused mostly on building heuristics. Recent works on this topic using deep supervised learning methods include the work of [74; 75; 76; 77; 78; 79; 80; 81; 82; 83; 84], while deep reinforcement learning approaches include [85; 86; 87; 88; 89; 90; 91; 92; 93; 94; 95].

Chapter 6

Experimental results

We now present a comparative experiment against three baseline competitors to validate the value of our approach and choices.

6.1 Benchmarks

We evaluate on three problem benchmarks. All are NP-hard and challenging for state-of-the-art solvers. In all cases the training sets consist of 100,000 samples from 10,000 randomly generated instances, and 20,000 samples from 2,000 instances for our validation set.

Our first benchmark is composed of set cover instances, generated by the software of [96] following [97], with 1000 variables. We train and validate on problems with 500 constraints and evaluate generalization on 10 test problems with 500 (EASY), 1,000 (MEDIUM) and 6 test problems with 2,000 (HARD) constraints. The second benchmark is composed of maximum independent set instances, generated from Erdös-Rényi random graphs with edge probability $p = 0.9$ following the procedure of [98]. We train and validate on problems with graphs of 200 nodes, but we test on 10 problems with 200 (EASY), 300 (MEDIUM) and 400

(HARD) nodes. Finally, the third benchmark is composed of multiple knapsack instances. We follow the "weakly correlated" generation approach of [99], which basically is generation of instances with a bounded correlation value, with two knapsacks. We train and validate on problems with 18 items, while we test on 20 problems with 18 (EASY), 26 (MEDIUM) and 34 (HARD) items.

6.2 Experimental setup

The training procedure for our GCNN model remains the same throughout all experiments. We disable presolving, separation, and conflict analysis in the SCIP solver in order to keep the number of rows and columns constant throughout the solving process. All other parameters are kept at the default. All experiments are performed on a machine with Intel Xeon E5-2650 2.20GHz CPUs and an Nvidia Tesla P100 GPU.

6.3 Baseline competitors

We compare our model against three baselines. First, we compare against SCIP's default branching procedure, *reliability pseudocost branching*, an implementation of hybrid branching [20]. This rule incorporates many sophisticated heuristics and is usually very competitive.

The second baseline follows the approach of [54]. Strong branching makes its decisions by assigning to every candidate variable i a score σ_i, then branching on the variable with the highest score. The authors imitate strong branching by recording, when in a state $\mathbf{s}_t$, a feature vector $\phi_i(\mathbf{s}_t)$ as well as the strong branching score $\sigma_{i,t}$ for every candidate variable i. This creates a dataset of states-scores pairs $\{(\phi_i(\mathbf{s}_t), \sigma_{i,t})\}$. They then train by supervised learning an ExtraTrees [100] regression model $\hat{\sigma}$ to predict scores for a given variable from its feature vector

by minimizing mean-squared error. In evaluation, at every branching decision the variable-wise features $\phi_i(\mathbf{s}_t)$ are computed, and they branch on the variable with the highest predicted score $i^* = \arg\max_i \hat{\sigma}(\phi_i(\mathbf{s}_t))$. Since our model uses a slightly different target and different features, we adapted this approach to allow comparisons.

First, we record strong branching scores from our explore-then-strong-branch procedure, rather than strong branching proper. Second, we compute the variable-wise features from the bipartite state as follows. For every variable node, we take all edge and constraint features of its neighbors: the variable-wise feature vector is the concatenation of the node feature with the component-wise minimum, mean and maximum of this set.

Finally, the third baseline follows the approach of [57], which is similar to the one of [54] except in that they predict strong branching ranks by fitting a LambdaMART [101] learning-to-rank model.

Again, we vary from the original authors by recording candidate ranks from our explore-then-strong-branch procedure, rather than strong branching proper. Moreover, in the original article incorporated features that were specific to time-dependent traveling salesman problems – we generalize their approach by reusing the variable-wise features described earlier. In detail, the training set is then composed of rankings over tuples of variable-wise features

$$\{(\phi_1(\mathbf{s}_t), \ldots, \phi_{|\mathbf{V}|}(\mathbf{s}_t)), (\rho_1, \ldots, \rho_{|\mathbf{V}|})\},$$

and we train a LambdaMART model to maximize normalized discounted cumulative gain over this training set. At test time, we branch on the variable to which the model $\hat{\rho}$ gives the highest rank, $i^* = \arg\max_i \hat{\rho}(\phi_1(\mathbf{s}_t), \phi_2(\mathbf{s}_t) \ldots, \phi_{|\mathbf{V}|}(\mathbf{s}_t))_i$.

For both the ExtraTrees and the LambdaMART models, as [54] mentions, inference time increases with the training set size, and thus it is valuable to limit

the training set. To address this, we sampled 4,000 branching decisions as training set and 1,000 branching decisions as validation set. This yielded an average of 121,199 training observations over five seeds,[1] a figure roughly in line with the 100,000 training set size used in [54].

6.4 Results

Table 6.1 reports the validation cross-entropy achieved by each model (Cross-Entropy), as well as the percentage of the time that the decision made by strong branching was the highest ranked decision of the model (acc@1), within the five highest (acc@5) and the ten highest (acc@10). Results are the mean ± standard deviation over five seeds.

Tables 6.2–6.4 summarize our test results. Since the ultimate objective is to solve optimization problems as fast as possible, we report the mean time needed to solve the instances (time). As this is hardware-specific, it is common in the literature to report the number of nodes obtained by the final B&B tree (e.g. as in [54; 57]). Consequently, we report the mean number of nodes as well (nodes). Finally, we report the number of instances for which each model was the fastest (win). Results were averaged over five seeds.

The models included are our graph convolutional neural network (gcnn), as well as variants described later in the target evaluation and ablation study sections. We also report results of SCIP's reliability pseudocost branching (rpb), the ExtraTrees regression model (trees) and the LambdaMART ranking model (lmart).

The results in Tables 6.2–6.4 show that gcnn clearly outperforms the other machine learning approaches and is very competitive with the default branching

[1] All major MILP solvers have a parameter, seed, that randomizes some tie-breaking rules, so as to be able to run the same instance multiple times and avoid overfitting.

Set Cover Training Results

model	Cross-Entropy	acc@1	acc@5	acc@10
gcnn	1.887 ± 0.004	39.3 ± 0.1	83.8 ± 0.1	95.5 ± 0.0
gcnn+a	1.879 ± 0.003	39.5 ± 0.1	84.1 ± 0.1	95.7 ± 0.0
gcnn+m	1.891 ± 0.004	39.3 ± 0.1	83.6 ± 0.2	95.4 ± 0.1
gcnn+l	1.883 ± 0.002	39.3 ± 0.1	83.9 ± 0.1	95.5 ± 0.0
gcnn+n	1.895 ± 0.010	39.2 ± 0.2	83.6 ± 0.2	95.4 ± 0.1
gcnn+3	$\mathbf{1.834} \pm 0.012$	$\mathbf{39.8} \pm 0.3$	$\mathbf{84.5} \pm 0.3$	$\mathbf{96.2} \pm 0.2$
gcnn+e	2.163 ± 0.035	32.7 ± 0.8	76.9 ± 0.9	92.2 ± 0.5
trees	-	10.3 ± 0.0	35.2 ± 0.0	52.5 ± 0.0
lmart	-	4.6 ± 0.0	13.7 ± 0.0	19.8 ± 0.0

Table 6.1: Training results on maximum set cover instances. M is for the mean normalized convolution, L is for the usage of layernorm instead of our prenorm, A is for the usage of te affinity layer, N is for no normalization after the convolutions, 3 is for three convolutional layers, E is for a model trained with decisions from strong branching only.

strategy of a state-of-the-art MILP solver. Recall that the two metrics used in the tables, i.e., the mean number of branch-and-bound nodes and computing times are the standard ones for evaluating MILP search performances.

For this reason, the results are particularly impressive: for the first time in the literature a machine-learning-based approach is compared with a MILP solver at its best and not in a controlled mode for the only purpose of assessing viability. What the results of Tables 6.2–6.4 are telling is that gcnn is a very serious candidate to be implemented within a MILP solver to provide an additional tool to speed up search for mixed-integer linear programming problems, which, in turn, means that more could be gained by a tight integration within a complex

software like any MILP solver is.

More in details, the results in Table 6.2 for set covering show that GCNN improves over RPB both in terms of number of explored nodes and in terms of computing time. Those results are also consistent in terms of the number of wins on which GCNN dominates RPB. Further, the model generalizes very well larger instance sizes as shown by the increasing relative improvement with respect to RPB.

The results in Table 6.3 for maximum independent set are also very positive and interesting. On the one hand, GCNN continues to dominate the other machine learning approaches and to be significantly faster than RPB (also considering the number of wins). On the other hand, the number of nodes explored by RPB is on average significantly smaller than for GCNN. This can be explained by the fact that the behavior of SCIP on this class of instances between RPB and

SET COVER TEST RESULTS

	EASY			MEDIUM			HARD		
MODEL	NODES	TIME	WINS	NODES	TIME	WINS	NODES	TIME	WINS
GCNN	190	**3 s.**	**4/10**	3296	**56 s.**	**6/10**	**116704**	**2549 s.**	**5/6**
GCNN+M	190	**3 s.**	0/10	3391	60 s.	0/10	146111	3097 s.	0/6
GCNN+A	183	**3 s.**	1/10	3301	59 s.	1/10	141254	2862 s.	0/6
GCCN+L	190	**3 s.**	1/10	3298	s.	2/10	145211	2979 s.	0/6
GCNN+N	188	**3 s.**	1/10	3304	58 s.	1/10	123605	2734 s.	0/6
GCNN+3	176	**3 s.**	2/10	3861	105 s.	0/10	455005	9675 s.	0/6
GCNN+E	187	**3 s.**	1/10	**3265**	57 s.	0/10	126581	2829 s.	1/6
RPB	**106**	6 s.	0/10	4662	80 s.	0/10	145351	2639 s.	0/6
TREES	247	7 s.	0/10	5170	158 s.	0/10	>1M	>1 h.	0/6
LMART	2708	43 s.	0/10	>1M	>1 h.	0/10	>1M	>1 h.	0/6

Table 6.2: Test results on maximum set cover instances.

MAXIMUM INDEPENDENT SET TEST RESULTS

	EASY			MEDIUM			HARD		
MODEL	NODES	TIME	WIN	NODES	TIME	WIN	NODES	TIME	WIN
GCNN	1253	**11 s.**	**7/10**	3864	**61 s.**	**10/10**	8478	**214 s.**	**7/10**
RPB	**236**	18 s.	0/10	**422**	83 s.	0/10	**801**	221 s.	3/10
TREES	1151	12 s.	1/10	4534	92 s.	0/10	9990	349 s.	0/10
LMART	1423	12 s.	2/10	4808	89 s.	0/10	10167	335 s.	0/10

Table 6.3: Test results maximum independent set instances.

pure (i.e., aggressive) strong branching is very similar. In other words, for these problems SCIP automatically decides (according to some internal analysis) to be aggressive at each node for variable selection, and obtains a good performance in terms of explored nodes by paying the price of a larger computing time because of expensive variable selection.

This SCIP behavior is somehow reversed for multiple knapsack instances, Table 6.4. There, although GCNN significantly improves with respect to RPB in terms of number of nodes, it turns out to be slower. Looking at the behavior of single instances, it is clear that GCNN is faster when it reduces the number of nodes by at least one order of magnitude and slower otherwise.

We believe that the observations for independent set and multiple knapsack instances are very interesting because they highlight, on the one hand, the fact that the time spent on variable selection for SCIP is tuned and problem dependent (with respect to our approach for which it is constant), and, on the other hand, provides plenty of room for hydrid approaches combining traditional methods and machine learning.

	Multiple Knapsack Test Results								
	Easy			Medium			Hard		
model	Nodes	Time	Win	Nodes	Time	Win	Nodes	Time	Win
gcnn	**540**	0.8 s.	1/20	**3789**	4.5 s.	0/20	**14981**	16.2 s.	2/20
rpb	770	**0.2 s.**	**19/20**	6840	**1.2 s.**	**20/20**	34772	**5.8 s.**	**18/20**
trees	569	1.3 s.	0/20	11453	25.9 s.	0/20	249200	597 s.	0/20
lmart	588	0.5 s.	0/20	8105	6.8 s.	0/20	133174	126 s.	0/20

Table 6.4: Test results on multiple knapsack instances.

6.5 Target evaluation

We also evaluated the impact of our explore-then-strong-branch procedure detailed in Section 4.2.1. Namely, we compared our results on the same set cover instances as in Section 6.3 with a graph convolutional neural network trained with decisions from strong branching only (gcnn+e). As can be seen in Table 6.2, our imitation target allows for substantially better generalization of the resulting policy.

Chapter 7

Conclusions and Future Developments

In this thesis it has been shown that a machine learning approach could be a significative tool to take in consideration for solving combinatorial optimization problems. Our formulation of branch-and-bound as a Markov decision process involves the usage of graph convolutional neural networks, trained by imitation learning on a dataset produced by the new approach explore-then-strong-branch, and results in a competitive variable selection rule for exact methods. The representation of the solver states as bipartite graphs matches perfectly with the graph convolutional neural networks architecture, and learns well the behaviour of an expert rule such as strong branching, a gold standard brancher. The ablation study helped to select the best performing architecture after embedding a new custom brancher inside the solver, emphasizing that the absolutely best model could be worse than a trade-off one for several reasons. After validating our architectural choices, the experiments on new problem instances showed promising results, outperforming the older machine learning approaches and often the reliability pseudocost branching, the highly competitive default brancher of SCIP, a state-of-the-art open-source solver. The results were not surprising on small instances since the reliability pseudocost uses mainly strong branching at the

beginning, which is known for not having a competitive running time trade-off. However, on large instances reliability pseudocost uses mostly a hybrid strategy which is known for being very competitive, and outperforming that brancher in this case has been surprising, especially considering that the training phase has been done on small instances to test the generalization capabilities of our learned policy.

The main future development is to improve the policy performances embedding the GCNN model in a reinforcement learning actor critic approach, where the GCNN is the actor and the critic could be a machine learning model, for example predicting the number of missing nodes or time for the branch-and-bound algorithm. Another possible future development could be the improvement of the policy running time trade-off with simple heuristics, for example the probability distribution provided by the policy could be used for more than one node by selecting the next best variables, thereby saving some GPU computations and reducing the overall running time. The key feature of this heuristic must be the speed, indeed the resulting policy will probably be less accurate than using the GCNN model, resulting in a higher number of explored nodes, but hopefully also a reduced computing time.

References

[1] A. H. Land and A. G. Doig, “An automatic method of solving discrete programming problems,” *The Econometric Society*, 1960. 2

[2] T. Achterberg and R. Wunderling, “Mixed integer programming: Analyzing 12 years of progress,” *Facets of Combinatorial Optimization*, pp. 449–481, 2013. 2

[3] J. Linderoth and M. Savelsbergh, “A computational study of search strategies for mixed integer programming,” *Informs Journal on Computing*, pp. 173–187, 1999. 2

[4] J. Patel and J. W. Chinneck, *Active-constraint variable ordering for faster feasibility of mixed integer linear programs.* Springer, 9 2007. 2

[5] T. Achterberg and T. Berthold, “Hybrid branching,” *Lecture Notes in Computer Science*, pp. 309–311, 2009. 2

[6] M. Fischetti and M. Monaci, *Cutting plane versus compact formulations for uncertain (integer) linear programs.* Springer, 2012. 2

[7] T. Achterberg, T. Koch, and A. Martin, “Branching rules revisited,” *Operations Research Letters*, vol. 33, pp. 42–54, 2005. 2, 16

[8] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, “Finding cuts in the tsp,” tech. rep., DIMACS, 1995. 2, 16

[9] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Engelwood Cliffs, 1982. 5

[10] C. D. Cantrell, *Modern Mathematical Methods for Physicists and Engineer*. Cambridge University Press, 12 2000. 7

[11] T. Ibaraki, "Enumerative approaches to combinatorial optimization," *Annals of Operations Research*, pp. 10, 11, 1987. 7

[12] A. Schrijver, *Theory of Linear and Integer Programming*. Wiley, Chichester, 1986. 9

[13] B. Korte and J. Vygen, *Combinatorial Optimization, Theory and Algorithms*. Springer, 2012. 9

[14] L. A. Wolsey, *Integer Programming*. Wiley, 1998. 9

[15] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem," *Journal of the Operations Research Society of America*, pp. 393–410, 11 1954. 9

[16] R. E. Gomory, "Outline of an algorithm for integer solutions to linear programs," *Bulletin of the American Mathematical Society*, pp. 275–278, 5 1958. 9

[17] S. Martello, "Algoritmi branch-and-bound: Strategie di esplorazione e rilassamenti," pp. 1,2, 02 2006. 11

[18] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. Wiley, Chichester, 1988. 11

[19] J. Shapiro, *Mathematical Programming: Structures and Algorithms*. Wiley, Chichester, 1979. 12

[20] T. Achterberg and T. Berthold, "Hybrid branching," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2009. 16, 60

[21] J. T. Linderoth and M. W. P. Savelsbergh, "A computational study of search strategies for mixed integer programming," *INFORMS Journal on Computing*, vol. 11, pp. 173–187, 1999. 16

[22] M. Bénichou, J.-M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent, "Experiments in mixed-integer linear programming," *Mathematical Programming*, vol. 1, pp. 76–94, 1971. 17, 39

[23] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press, 1960. 18

[24] H. He, H. I. Daum, and J. Eisner, "Learning to search in branch and bound algorithms," in *Advances in Neural Information Processing Systems 27*, pp. 3293–3301, 2014. 18, 57

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. 21

[26] Ovid and Martin, *Metamorphoses*. W. W. Norton Company, 2004. 21

[27] Sparkes, *The Red and the Black: Studies in Greek Pottery*. Routledge, 1996. 21

[28] Tandy, *Works and Days: A Translation and Commentary for the SocialSciences*. University of California Press, 1997. 21

[29] A. Lovelace, *Notes upon L. F. Menabreas Sketch of the Analytical Engineinvented by Charles Babbage*. 1842. 21

[30] E. S. Team, "What is machine learning? a definition," 2017. 22

[31] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, 1959. 22

[32] Wikipedia, "Information theory." 25

[33] R. C. Barros, A. C. de Carvalho, and A. A. Freitas, *Automatic Design of Decision-Tree Induction Algorithms.* Springer, 2015. 28

[34] T. H. et al., "Lower bounds on learning decision lists and trees," *Information and Computation*, 1996. 28

[35] B. Scholkpof and A. J. Smola, *Support Vector Machines, Regularization, Optimization, and Beyond.* The MIT Press Cambridge, Massachusetts, London, England, 2002. 29

[36] M. A. Aizerman, E. A. Braverman, and L. Rozonoer, "Theoretical foundations of the potential function method in pattern recognition learning.," in *Automation and Remote Control,*, no. 25 in Automation and Remote Control,, pp. 821–837, 1964. 30

[37] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," *Proceedings of the fifth annual workshop on Computational learning theory*, p. 144, 1992. 30

[38] W. McCulloch and W. Bulletin, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, 1943. 31

[39] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, pp. 533–536, 1986. 31

[40] Scip, "Solving constraint integer programs." 36

[41] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods," *ACM Computing Surveys*, vol. 50, p. 21, 2017. 38, 39

[42] D. A. Pomerleau, "Efficient training of artificial neural networks for autonomous navigation," *Neural Computation*, vol. 3, pp. 88–97, 1991. 38

[43] S. Ross and D. Bagnell, "Efficient reductions for imitation learning," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 661–668, 2010. 39

[44] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 627–635, 2011. 39

[45] S. Russell, "Learning agents for uncertain environments," in *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, pp. 101–103, 1998. 39

[46] K. J. Åström, "Optimal control of Markov processes with incomplete state information," *Journal of Mathematical Analysis and Applications*, vol. 10, pp. 174–205, 1965. 40

[47] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proceedings of the International Conference on Learning Representations*, 2017. 41, 48

[48] D. K. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints.," in *Advances in Neural Information Processing Systems 28*, pp. 2224–2232, 2015. 42

[49] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014. 47

[50] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017. 47

[51] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," in *https://arxiv.org*, 2016. 47

[52] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML*, 2015. 47

[53] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, pp. 249–256, 2010. 49

[54] A. M. Alvarez, Q. Louveaux, and L. Wehenkel, "A machine learning-based approximation of strong branching," *INFORMS Journal on Computing*, vol. 29, pp. 185–195, 2017. 55, 60, 61, 62

[55] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. Savelsbergh, "An updated mixed integer programming library: MIPLIB 3.0," tech. rep., Rice University, 1998. 55

[56] T. Achterberg, T. Koch, and A. Martin, "Miplib 2003," *Operations Research Letters*, vol. 34, pp. 361–372, 2006. 55

[57] C. Hansknecht, I. Joormann, and S. Stiller, "Cuts, primal heuristics, and

learning to branch for the time-dependent traveling salesman problem." arXiv:1805.01415, 2018. 56, 61, 62

[58] E. B. Khalil, P. L. Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pp. 724–731, 2016. 56

[59] T. Joachims, "Optimizing search engines using clickthrough data," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 133–142, ACM, 2002. 56

[60] B. A. Huberman, R. M. Lukose, and T. Hogg, "An economics approach to hard computational problems," *Science*, 1 1997. 57

[61] G. Di Liberto, S. Kadioglu, K. Leo, and Y. Malitsky, "Dash: Dynamic approach for switching heuristics," *European Journal of Operational Research*, vol. 248, pp. 943–953, 2016. 57

[62] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik, "Learning to branch," in *Proceedings of the International Conference on Machine Learning*, 2018. 57

[63] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 123–140, 2016. 57

[64] G. Lederman, M. N. Rabe, and S. A. Seshia, "Learning heuristics for automated reasoning through deep reinforcement learning." arXiv:1807.08058, 2018. 57

[65] M. Balunovic, P. Bielik, and M. Vechev, "Learning to solve SMT formulas," in *Advances in Neural Information Processing Systems 31*, pp. 10338–10349, 2018. 57

[66] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao, "Learning to run heuristics in tree search," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pp. 659–666, 2017. 58

[67] A. Lodi and G. Zarpellon, "On learning and branching: a survey," *TOP*, vol. 25, pp. 207–236, 2017. 58

[68] G. Nannicini, P. Belotti, J. Lee, J. Linderoth, F. Margot, and A. Wächter, "A probing algorithm for MINLP with failure prediction by SVM," in *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 154–169, 2011. 58

[69] A. Sabharwal, H. Samulowitz, and C. Reddy, "Guiding combinatorial optimization with UCT," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pp. 356–361, Springer, 2012. 58

[70] A. Balafrej, C. Bessiere, and A. Paparrizou, "Multi-Armed bandits for adaptive constraint propagation," in *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pp. 290–296, 2015. 58

[71] A. Hottung, S. Tanaka, and K. Tierney, "Deep learning assisted heuristic tree search for the container pre-marshalling problem." arXiv:1709.09972, 2017. 58

[72] W. Xia and R. H. C. Yap, "Learning robust search strategies using a bandit-based approach." arXiv:1805.03876, 2018. 58

[73] Q. Cappart, E. Goutierre, D. Bergman, and L.-M. Rousseau, "Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning." arXiv:1809.03359, 2018. 58

[74] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015. 58

[75] D. Levy and L. Wolf, "Learning to align the source code to the compiled object code," in *Proceedings of the Thirty-Fourth International Conference on Machine Learning*, pp. 2043–2051, 2017. 58

[76] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna, "Revised note on learning algorithms for quadratic assignment with graph neural networks." arXiv:1706.07450, 2017. 58

[77] R. B. Palm, U. Paquet, and O. Winther, "Recurrent relational networks." arXiv:1711.08028, 2017. 58

[78] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from Single-Bit supervision." arXiv:1802.03685, 2018. 58

[79] H. A. A. Nomer, A. Aboutahoun, and A. Elsayed, "Long term memory network for combinatorial optimization problems." OpenReview, 2018. 58

[80] S. Gu and T. Hao, "A pointer network based deep learning algorithm for 0-1 knapsack problem," in *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, pp. 473–477, Mar. 2018. 58

[81] Y. Kaempfer and L. Wolf, “Learning the multiple traveling salesmen problem with permutation invariant pooling networks.” arXiv:1803.09621, 2018. 58

[82] J. Song, R. Lanka, A. Zhao, Y. Yue, and M. Ono, “Learning to search via retrospective imitation.” arXiv:1804.00846, 2018. 58

[83] M. O. R. Prates, P. H. C. Avelar, H. Lemos, L. Lamb, and M. Vardi, “Learning to solve NP-Complete problems - a graph neural network for decision TSP.” arXiv:1809.02721, 2018. 58

[84] Z. Li, Q. Chen, and V. Koltun, “Combinatorial optimization with graph convolutional networks and guided tree search,” in *Advances in Neural Information Processing Systems 31*, pp. 536–545, 2018. 58

[85] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu, and D. Hassabis, “Hybrid computing using a neural network with dynamic external memory,” *Nature*, vol. 538, pp. 471–476, 2016. 58

[86] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, “Neural combinatorial optimization with reinforcement learning.” arXiv:1611.09940, 2016. 58

[87] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, “Learning combinatorial optimization algorithms over graphs.” arXiv:1704.01665, 2017. 58

[88] H. Hu, X. Zhang, X. Yan, L. Wang, and Y. Xu, "Solving a new 3D bin packing problem with deep reinforcement learning method." arXiv:1708.05930, 2017. 58

[89] V. Nair, K. Dvijotham, I. Dunning, and O. Vinyals, "Learning fast optimizers for contextual stochastic integer programs," in *Conference on Uncertainty in Artificial Intelligence*, 2018. 58

[90] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau, "Learning heuristics for the TSP by policy gradient," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 170–181, 2018. 58

[91] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takac, "Reinforcement learning for solving the vehicle routing problem," in *Advances in Neural Information Processing Systems 31*, pp. 9861–9871, 2018. 58

[92] W. Kool, H. van Hoof, and M. Welling, "Attention solves your TSP, approximately." arXiv:1803.08475, 2018. 58

[93] P. Emami and S. Ranka, "Learning permutations with sinkhorn policy gradient." arXiv:1805.07010, 2018. 58

[94] Y. Ye, X. Ren, J. Wang, L. Xu, W. Guo, W. Huang, and W. Tian, "A new approach for resource scheduling with deep reinforcement learning." arXiv:1806.08122, 2018. 58

[95] A. Laterre, Y. Fu, M. K. Jabri, A.-S. Cohen, D. Kas, K. Hajjar, T. S. Dahl, A. Kerkeni, and K. Beguir, "Ranked reward: Enabling Self-Play reinforcement learning for combinatorial optimization." arXiv:1807.01672, 2018. 58

[96] S. Umetani, "Exploiting variable associations to configure efficient local search algorithms in large-scale binary integer programs," *European Journal of Operational Research*, vol. 263, pp. 72–81, 2017. 59

[97] E. Balas and A. Ho, "Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study," in *Combinatorial Optimization*, pp. 37–60, Springer, 1980. 59

[98] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker, *Decision diagrams for optimization*. Springer, 2016. 59

[99] A. S. Fukunaga, "A branch-and-bound algorithm for hard multiple knapsack problems," *Annals of Operations Research*, vol. 184, pp. 97–119, 2011. 60

[100] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, pp. 3–42, 2006. 60

[101] C. J. Burges, "From RankNet to LambdaRank to LambdaMART: An Overview," tech. rep., Microsoft Research, 2010. 61

[102] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig, "The SCIP Optimization Suite 6.0," technical report, Optimization Online, July 2018.

[103] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization." arXiv:1607.06450, 2016.

[104] A. Y. Ng and S. J. Russell, "Algorithms for inverse reinforcement learning.," in *Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 663–670, 2000.

[105] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT Press, second ed., 2018.

[106] J. P. M. Silva and K. A. Sakallah, "Grasp - a new search algorithm for satisfiability," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 220–227, IEEE, 1997.

[107] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *arXiv preprint arXiv:1811.06128*, 2018.

[108] J. J. Hopfield and D. W. Tank, "neural computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52, pp. 141–152, 1985.

[109] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, pp. 497–520, 1960.

[110] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the Thirty-Second International Conference on Machine Learning*, pp. 448–456, 2015.

[111] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," in *Proceedings of the International Conference on Learning Representations*, 2017.

[112] M. Kruber, M. E. Lübbecke, and A. Parmentier, "Learning when to use a decomposition," in *Integration of AI and OR Techniques in Constraint*

Programming (D. Salvagnin and M. Lombardi, eds.), pp. 202–210, Springer International Publishing, 2017.

[113] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 5998–6008, Curran Associates, Inc., 2017.

[114] W. Kool, H. van Hoof, and M. Welling, "Attention solves your tsp, approximately.," 2018. arXiv:1803.08475.

[115] L. A. Wolsey, *Integer Programming*. Wiley-Blackwell, 1988.

[116] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig, "The SCIP Optimization Suite 6.0," zib-report, Zuse Institute Berlin, July 2018.

[117] J. Caldwell, R. Watson, C. Thies, and J. Knowles, "Deep optimisation: Solving combinatorial optimisation problems using deep neural networks," *arXiv preprint arXiv:1811.00784*, 2018.

[118] J. Patel and J. W. Chinneck, "Active-constraint variable ordering for faster feasibility of mixed integer linear programs," *Mathematical Programming*, vol. 110, pp. 445–474, 2007.

[119] M. Fischetti and M. Monaci, "Branching on nonchimerical fractionalities," *Operations Research Letters*, vol. 40, pp. 159 – 164, 2012.

[120] T. Powers, R. Fakoor, S. Shakeri, A. Sethy, A. Kainth, A.-r. Mohamed, and R. Sarikaya, "Differentiable greedy networks," *arXiv preprint arXiv:1810.12464*, 2018.

[121] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," in *Proceedings of the International Conference on Learning Representations*, 2014.

[122] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, pp. 61–80, Jan 2009.

[123] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, vol. 2, pp. 729–734, 2005.

[124] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *Proceedings of the International Conference on Learning Representations*, 2016.

[125] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the Thirty-Fourth International Conference on Machine Learning*, pp. 1263–1272, 2017.

[126] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," 2018. arXiv:1806.08804.

[127] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau, "Learning heuristics for the tsp by policy gradient," in *Integration of Con-*

straint Programming, Artificial Intelligence, and Operations Research (W.-J. van Hoeve, ed.), (Cham), pp. 170–181, Springer International Publishing, 2018.

[128] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 6348–6358, 2017.

[129] C. Grozea and M. Popescu, "Can machine learning learn a decision oracle for np problems? a test on sat.," *Fundamenta Informaticae*, vol. 131, pp. 441–450, 2014.

[130] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization* (C. A. C. Coello, ed.), (Berlin, Heidelberg), pp. 507–523, Springer Berlin Heidelberg, 2011.

[131] V. Nair, K. Dvijotham, I. Dunning, and O. Vinyals, "Learning fast optimizers for contextual stochastic integer programs.," in *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018*, 2018.

[132] J. Song, R. Lanka, A. Zhao, Y. Yue, and M. Ono, "Learning to search via retrospective imitation," in *ArXiv preprint*, 2018.