**Université de Montréal**

# Learning to Compare Nodes in Branch and Bound with Graph Neural Networks

par

**Abdel Ghani Labassi**

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

Orientation intelligence artificielle

Le mercredi 10 août 2022

# Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

**Learning to Compare Nodes in Branch and Bound with Graph Neural Networks**

présenté par

## Abdel Ghani Labassi

a été évalué par un jury composé des personnes suivantes :

*Margarida Carvalho*
(président-rapporteur)

*Andrea Lodi*
(directeur de recherche)

*Didier Chételat*
(codirecteur)

*Laurent Charlin*
(membre du jury)

# Résumé

En informatique, la résolution de problèmes NP-difficiles en un temps raisonnable est d'une grande importance : optimisation de la chaîne d'approvisionnement, planification, routage, alignement de séquences biologiques multiples, inference dans les modèles graphiques probabilistes, et même certains problèmes de cryptographie sont tous des examples de la classe NP-complet. En pratique, nous modélisons beaucoup d'entre eux comme un problème d'optimisation en nombre entier, que nous résolvons à l'aide de la méthodologie *séparation et évaluation*. Un algorithme de ce style divise un espace de recherche pour l'explorer récursivement (séparation), et obtient des bornes d'optimalité en résolvant des relaxations linéaires sur les sous-espaces (évaluation). Pour spécifier un algorithme, il faut définir plusieurs paramètres, tel que la manière d'explorer les espaces de recherche, de diviser une recherche l'espace une fois exploré, ou de renforcer les relaxations linéaires. Ces politiques peuvent influencer considérablement la performance de résolution.

Ce travail se concentre sur une nouvelle manière de dériver politique de recherche, c'est à dire le choix du prochain sous-espace à séparer étant donné une partition en cours, en nous servant de l'apprentissage automatique profond. Premièrement, nous collectons des données résumant, sur une collection de problèmes donnés, quels sous-espaces contiennent l'optimum et quels ne le contiennent pas. En représentant ces sous-espaces sous forme de graphes bipartis qui capturent leurs caractéristiques, nous entraînons un réseau de neurones graphiques à déterminer la probabilité qu'un sous-espace contienne la solution optimale par apprentissage supervisé. Le choix d'un tel modèle est particulièrement utile car il peut s'adapter à des problèmes de différente taille sans modifications. Nous montrons que notre approche bat celle de nos concurrents, consistant à des modèles d'apprentissage automatique plus simples entraînés à partir des statistiques du solveur, ainsi que la politique par défaut de SCIP, un solveur open-source compétitif, sur trois familles NP-dures: des problèmes de recherche de stables de taille maximum, de flots de réseau multicommodité à charge fixe, et de satisfiabilité maximum.

**Mot-clés :** Optimisation combinatoire, Séparation et évaluation, Recherche de solutions, Plongement-à-l'optimum, Apprentissage par imitation, Réseaux de neurones graphiques

# Abstract

In computer science, solving NP-hard problems in a reasonable time is of great importance, such as in supply chain optimization, scheduling, routing, multiple biological sequence alignment, inference in probabilistic graphical models, and even some problems in cryptography. In practice, we model many of them as a mixed integer linear optimization problem, which we solve using the branch and bound framework. An algorithm of this style divides a search space to explore it recursively (branch) and obtains optimality bounds by solving linear relaxations in such sub-spaces (bound). To specify an algorithm, one must set several parameters, such as how to explore search spaces, how to divide a search space once it has been explored, or how to tighten these linear relaxations. These policies can significantly influence resolution performance.

This work focuses on a novel method for deriving a search policy, that is, a rule for selecting the next sub-space to explore given a current partitioning, using deep machine learning. First, we collect data summarizing which subspaces contain the optimum, and which do not. By representing these sub-spaces as bipartite graphs encoding their characteristics, we train a graph neural network to determine the probability that a subspace contains the optimal solution by supervised learning. The choice of such design is particularly useful as the machine learning model can automatically adapt to problems of different sizes without modifications. We show that our approach beats the one of our competitors, consisting of simpler machine learning models trained from solver statistics, as well as the default policy of SCIP, a state-of-the-art open-source solver, on three NP-hard benchmarks: generalized independent set, fixed-charge multicommodity network flow, and maximum satisfiability problems.

**Keywords :** Combinatorial Optimization, Branch and Bound, Solution Search, Diving-to-Optimum, Imitation Learning, Graph Neural Networks

# Contents

# List of tables

# List of figures

# List of acronyms and abbreviations

| | |
|---|---|
| OR | Operations Research |
| NP | Nondeterministic Polynomial (in time) |
| B&B | Branch-and-Bound |
| CO | Combinatorial Optimization |
| MILP | Mixed Integer Linear Program |
| TSP | Traveling Salesman Problem |
| GISP | Generalized Independant Set Problem |
| FCMCNF | Fixed Charge Multicommodity Network Flow Problem |
| MAXSAT | Maximum Satisfiability Problem |
| ML | Machine Learning |

| | |
|---|---|
| DP | Dynamic Programming |
| SA | Stochastic Approximation |
| SVM | Support Vector Machine |
| MLP | Multilayer Perceptron |
| GNN | Graph Neural Network |

# Acknowledgements

On the personal side, first, I am unceasingly grateful to my family in Canada and Algeria, especially my mom, for all the unconditional love, support, and sacrifice. Without her, I could not have become the man I am today. Rest in peace, mom. Also, special thanks to my friends and box/gym partners who helped me balance with academics to keep me mentally sane and productive. Couldn't make it without you guys!

On the professional side, I am grateful to my advisor, Dr. Andrea Lodi. His endless curiosity and optimism inspired me and kept me motivated despite the grief I faced along the way. His impressive mastery of combinatorial optimization and his advocacy in the community allowed me to learn a great deal about the field and to get on the right track to becoming an expert myself. I would also like to thank my co-advisor, Dr. Didier Chételat, who convinced me to pursue the project. His exceptional kindness, intelligence, patience, and constant advice have been invaluable to me. He was never too busy to let me probe his immense knowledge so that I could overcome numerous difficulties. He has forever instilled a never give up attitude.

Additionally, acknowledgments to all other *Data Science for Real-Time Decision-Making Canada Excellence Research Chair* members and technical staff, who were always friendly, welcoming, and never shy to offer advice. A special thanks to Khalid for his technical support and his shares of past research experiences. Not only did those make me understand academia better, but they also greatly inspired me! Other thanks to Mariaa and Mehdi: their daily assistance and availability provided me with more than just administrative problem-solving. To Ph.D. students, especially Frederico, Gabriele, and Defeng: although I was not there for an extended time like you guys were/are, and despite the COVID circumstances, I am grateful for the welcoming !

I am appreciative to the *Département d'informatique et de recherche opérationnelle de l'Université de Montréal* for providing incredible research and teaching opportunities and for promoting a culture of intellectual freedom and autonomy. Its remarkable concentration of computer science and mathematics talents allowed me to learn and grow so much these past five years. Finally, I acknowledge funding for this research project from the *Canadian Excellence Research Chair Program.*

# Chapter 1

# Introduction

In this section, we will review basic concepts of mixed integer programming. We will define a mixed integer programming as a special kind of mathematical optimization problem, and present the branch and bound family of algorithms, which is a framework designed to solve them. Several sequential decision-making problems occur within these algorithms that can drastically influence performance. We will contrast two approaches taken for the design of such decision-making scheme, namely handcrafted heuristics, and data-driven ones. Since we will use machine learning frameworks for the latter, we will present basic machine learning paradigms and models with a focus for the ones useful for our purposes.

## 1. Mixed integer linear programming

Mixed integer linear programming is a powerful and flexible tool for modeling large-scale discrete optimization problems of different application domains. According to NP-completeness theory (Stephen Cook, 1971), any NP-hard problem can be formulated as a MILP [1] since its solving is NP-hard. Many well-known theoretical problems have been heavily studied and modeled, such as traveling salesperson, satisfiability in Boolean circuits, subset sum, knapsack, and independent set problems. In a more applied context, NP-hard problems naturally arise in numerous sectors, such as multiple sequence alignment, a very active research area in bioinformatics with high applications in phylogenetics and genomics [2], or in industrial planning, such as optimal network flows, optimal lot sizing, backlogging [3], and symmetric-key primitives analysis in cryptology [4]. Firefighting [5], combinatorial drug therapy for cancer [6], optimal wind turbine planning [7], biodiversity conservation planning [8] are other examples of real-life problems often modeled as MILPs and then solved.

A mixed integer linear program (MILP) is a mathematical optimization problem of the form

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b, \\ & x_i \in \mathbb{Z} \quad \forall i \in I, \end{aligned} \tag{1.1}$$

where $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n$, and $I \subseteq \{1,2...n\}$ are the indices of the variables constrained to be integer. The 4-tuple $(A, b, c, I)$ characterizes the problem and is often referred to as the problem data. The function to optimize, $c^T x$, is called the objective function. The set $S(A, b, I) := \{x | Ax \leq b \wedge x_i \in \mathbb{Z}\ \forall i \in I\}$ will be referred to as the search space induced by $(A, b, I)$. An element in the search space is called a feasible solution of the problem. A feasible solution that minimizes the objective function is called an optimal solution or optimum, and its associated objective value is referred as the optimum's value.

Let $(A, b, c, I)$ be a MILP. A linear inequality $\alpha, \beta \in \mathbb{R}^n \times \mathbb{R}$, defined as a function $\mathbb{R}^n \to \{\textsc{True}, \textsc{False}\}$ of vectorial format $\alpha^T x \leq \beta$, is said to be *valid* on a MILP $(A, b, c, I)$ if

$$\alpha^T x \leq \beta \qquad \forall x \in S(A, b, I).$$

A convex hull on a search space $S(A, b, I)$ is, informally, the smallest polyhedron[1] containing the search space. More formally, we denote and define such set as follows:

$$CH(A, b, I) := \cap_{P \text{ a polyhedron } | S(A,b,I) \subseteq P} P.$$

Besides their elementary yet malleable structural properties, which enable painless and intuitive modeling, MILPs are interesting since their continuous relaxations (the problems resulting from the removal of integrality constraints $x_i \in \mathbb{Z}\ \ \forall i \in I$) are linear programs (LP), which can be solved in polynomial time [9], [10]. Thus, obtaining a good lower bound on the problem can be computationally cheap. One can argue that this property motivated the development of two families of algorithms for solving MILPs until optimality: Cutting Planes (CPs) [11] and Branch and Bound (B&B) [12]. Both frameworks are designed to solve MILPs iteratively to obtain increasingly satisfactory lower bounds until an optimum is found. Other frameworks, referred to as primal heuristics, are designed to find feasible solutions that are not required to be optimal. Examples of such approaches are simulated annealing [13], genetic algorithms [14], tabu search [15], local search [16], to name a few.

CPs can be used as an algorithm for solving certain MILPs: it can be proven that for certain classes of MILPs, and certain classes of valid inequalities, iteratively adding such inequalities will lead to the linear relaxation polytope to converge to the convex hull. In other words, for a MILP $(A, b, c, I)$, one can iteratively compute a collection of valid inequalities

[1] A polyhedron is a set of points in $\mathbb{R}^n$ satisfying a finite number of linear inequalities.

$Hx \leq h$ such that the optimization problem

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x_i \in \mathbb{Z} \quad \forall i \in I \end{aligned}$$

is equivalent, in the sense that the optimums coincide, to

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & Hx \leq h. \end{aligned}$$

The problem of solving the MILP (on the left) is thus reduced to the problem of solving an LP (on the right), which can be solved efficiently in theory and in practice. Unfortunately, however, all currently known approaches for computing such $Hx \leq h$ suffer from numerical instabilities, which prevent convergence in practice [17]. Thus this strategy is rather used as an aid to tighten the LP relaxation rather than to solve the MILPs *per se*.

B&B employs a divide-and-conquer approach, where the search space is recursively split by branching on unfeasible variables in the continuous relaxation optima. This recursive nature means that the solving process can be interpreted as a tree search process where nodes correspond to sub-MILPs resulting from such branchings. Algorithm 1 details the procedure, with $\mathbb{1}_i^n$ a $n$-vector with a one in the $i^{\text{th}}$ position, and zeros otherwise ("one-hot vector").

**Algorithm 1** The branch and bound framework for mixed integer linear programming

**function** SOLVE_MILP_BB($A, b, c, I$)
  $P, ub \leftarrow \{(A, b, c, I)\}, +\infty$
  **while** $P \neq \varnothing$ **do**
    $P$.`remove(` $A, b, c, I \leftarrow$ `choose_node(` $P$ `)` `)` ▷ Searching in the tree
    $A, b, x \leftarrow$ `solve_LP(`$A, b, c$`)` ▷ Bounding
    **if** $\forall i \in I, x_i \in \mathbb{Z}$ **then**
      $ub \leftarrow$ `min(`$ub, c^T x$ `)`
      **if** $ub = c^T x$ **then** $x^* \leftarrow x$
      **end if**
    **else if** $c^T x < ub$ **then** ▷ Branching
      $i \leftarrow$ `choose_branch(`$i \in I \,|\, x_i \notin \mathbb{Z}$`)`
      $P$.`add` $\left( \begin{pmatrix} A \\ +\mathbb{1}_i^n \end{pmatrix}, \begin{pmatrix} b \\ +\lfloor x_i \rfloor \end{pmatrix}, c, I \right)$
      $P$.`add` $\left( \begin{pmatrix} A \\ -\mathbb{1}_i^n \end{pmatrix}, \begin{pmatrix} b \\ -\lceil x_i \rceil \end{pmatrix}, c, I \right)$
    **end if**
  **end while**
  **return** $x^*$

In practice, modern commercial and open-source MILP solvers are B&B-based, but often use CPs to refine search space, and primal heuristics at opportune moments to speed up the search [18]–[20]. Adding well-chosen valid inequalities can lead the linear relaxation polyhedron closer to the convex hull, and thus increase the lower bound this relaxation, thus leading to less branching (more nodes not satisfying the `else if` condition) and faster solving. In parallel, well-chosen primal heuristics help lower upper bounds early in the tree, which also helps reduce the amount of branching necessary.

A major property of the B&B framework is the presence of several arbitrary decisions that must be repeatedly taken, such as the variable over which to branch (branching), or the selection of the next node to process (searching in the tree). These are resolved by decision rules, whose design is tightly linked to solving performance [21]. General decision rules are often used, although highly problem-specific branching, searching, and refining rules can lead to state-of-the-art solving for many heavily studied NP-hard problems, such as knapsack with conflict graphs [22], maximum independent set [23], multicommodity network flow [24], and capacitated facility location [25]. Unfortunately, the design of such rules requires in practice considerable research efforts from combinatorial optimization experts, which might not be available to practitioners. Thus, in practice, most practitioners simply fall back on general-purpose rules.

A novel idea, however, has arisen recently which proposes to automatically derive good problem-specific rules without expert knowledge. In many applications, similar MILPs are repeatedly solved, leading to potential statistical similarities which could be exploited for improved solving. In particular, one could potentially use machine-learning methods to derive data-driven decision rules from already solved instances, which could hopefully then lead to faster solving in future instances [26]. This approach, which will be referred to as *data-driven heuristics in integer programming*, is, therefore, both problem-independent and problem-adaptive. Before reviewing the field's literature, an introduction of useful machine learning concepts will follow.

## 2. Machine learning paradigms

Machine learning (ML) has as objective to give the computer the ability to learn, that is, to automatically improve itself with experience for accomplishing a task [27]. It can be seen as a field of applied statistics where the focus is to develop computational methods for approximating unknown functions rather than providing confidence intervals for such functions [28].

ML paradigms are meta-frameworks for these methods. They traditionally consist of supervised learning, unsupervised learning, and reinforcement learning. Other paradigms exist, e.g., semi-supervised learning, self-supervised learning, and imitation learning [29],

[30]. We will focus on this thesis on supervised learning and imitation learning; because imitation learning is tightly linked to reinforcement learning, an overview of the latter will also be provided.

## 2.1. Supervised learning

We will introduce the general problem in supervised learning using a probabilistic approach. We start by defining some important probability concepts.

Given a distribution with probability density function $p$ with support $\text{SUPP}(p) = \text{closure}(\{x \in \mathbb{R}^d \,|\, p(x) > 0\})$, the log-likelihood of a sample $x \in \text{SUPP}(p)$ is defined as $\log p(x)$. The cross-entropy between two distributions with probability density functions $p$ and $q$ acting on the same support is the quantity

$$H(p, q) = -\mathbb{E}_{X\sim p}\Big[\log q(X)\Big].$$

This quantity can be interpreted as a measure of distance between two distributions, since it is minimized when $p = q$. Note that it is not symmetrical in general, that is $H(p, q) \neq H(q, p)$.

In supervised learning, one observes a dataset $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ coming from a joint probability distribution function $p(x, y)$ over some space $\mathcal{X} \times \mathcal{Y}$, where $\mathcal{X} \subset \mathbb{R}^{d_{\text{in}}}$ is called the feature space and $\mathcal{Y} \subset \mathbb{R}^{d_{\text{out}}}$ is called the response space, and one assumes that there is a function $f$ such that $f(x) \approx y$ for $(x, y) \in \mathcal{X} \times \mathcal{Y}$. The goal of supervised learning is to recover this mapping from the sampled data.

More formally, the objective is to estimate, or model, the conditional distribution function $p(y|x)$. Methods that try to estimate this function directly are called discriminative supervised learning methods. Alternatively, some methods aim to estimate the joint distribution $p(x, y)$, from which the conditional distribution follows by $p(y|x) = p(x, y)/\int p(x, y)dy$. Methods taking this indirect route are called generative supervised learning methods. The mapping $f$, from inputs to outputs, is reinterpreted as the expectation $f(x) = \text{E}_{Y\sim p(.|X=x)}[Y]$.

For the rest of this thesis, we will assume a direct, discriminative approach to supervised learning. The main approach runs as follows. To approximate or "learn" the conditional distribution $p(y|x)$, one decides on a class of probability density functions $p_{\text{model}}(y|x; \theta)$ parameterized by some vector $\theta \in \mathbb{R}^p$, called the model. Then, ones tries to find the parameters $\theta$ that minimize the cross-entropy between the model and empirical distribution $\hat{p}$ of the observed samples [28]:

$$\min_\theta \mathbb{E}_{X,Y\sim\hat{p}}\Big[-\log p_{\text{model}}(Y|X; \theta)\Big] = -\min_\theta \sum_{i=1}^{n} \log p_{\text{model}}(y_i|x_i; \theta).$$

This quantity is called the empirical risk, and this approach is known as empirical risk minimization. Since one is looking for the parameters $\theta$ that maximize the likelihood of the observed samples, this is also called the maximum likelihood approach [28]. The optimization

process is referred to as training: the model is being fit to the data. It is worth mentioning that the general goal of supervised learning is not to maximise this log-likelihood: instead, it is to choose a model and an optimization scheme such that after fitting the model to the data, the resulting is able to perform well on unseen data. This is captured through a performance function to be selected according to the learning task to accomplish, and may have no relation to $p_{model}$.

We will now describe a typical train/evaluate scheme ML practitioners perform to learn such discriminative function $f$ that performs well on unseen data. First, given a dataset $\mathcal{D}$, the idea would be to partition it in three: $\mathcal{D}_{train}, \mathcal{D}_{valid}$ and $\mathcal{D}_{test}$. The dataset $\mathcal{D}_{train}$ is used to fit the model to the data, i.e., to learn parameters for the model. This step is called the training phase and corresponds to empirical risk minimization. Some models are endowed with so-called hyperparameters. A hyperparameter may be defined as an unlearnable model parameter, i.e., parameters that cannot be affected by the training phase (e.g., kernel functions in SVM and kernel size for CNNs, neural network size and activation functions, and so on, see section 3 for more details about models). To fit hyperparameters to the data, machine learning practitioners tune them manually and retrain on $\mathcal{D}_{train}$ until a satisfactory performance on $\mathcal{D}_{valid}$. This step is called hyperparameter tuning. Finally, evaluation, i.e., performance of the model obtained after training and hyperparameter tuning, is done using $\mathcal{D}_{test}$. A satisfactory model on $\mathcal{D}_{test}$ is said to generalize well, and the goal of ML practitioners is solely to obtain such model.

Since hyperparameter tuning is out-of-the-scope of this thesis, we will denote $\mathcal{D}_{train}$ as $\mathcal{D}$. For evaluation, we will either specify that we using the function $f$ on $\mathcal{D}_{test}$, or simply on new/unseen samples.

## 2.2. Markov decision processes

Many mathematical optimization problems can be formulated as the task of finding how to behave in an environment that gives feedback, known as a control problem. This is usually formulated as the task of finding an optimal policy in a Markov decision process (MDP), where an agent observes the environment to be in a state $s_t \in \mathcal{S}$, takes an action $a_t \in \mathcal{A}$ according to a policy $\pi : \mathcal{S} \mapsto \mathcal{A}$, receives a reward $r_t$ and the environment transitions to a new state $s_{t+1} \in \mathcal{S}$. This continues until the environment reaches a terminal state $s_T$: the run $s_0, \ldots, s_T$ is called an episode. The objective is to find the policy that maximizes the expected return

$$J(\pi) = \mathbb{E}_\pi \left[ \sum_{t=1}^{T} R_t \right] \tag{2.1}$$

over episodes.

The standard approach to solving a control problem is through reinforcement learning, which optimizes a policy through repeated trial-and-error interactions with the environment. Many reinforcement learning frameworks exists, of which the simplest are the REINFORCE and Q-learning algorithms. In REINFORCE, the idea is to represent states and actions vectorially, thus making it possible to model the policy as a learnable vector of parameters. Then, one looks for the optimal parameters using first-order methods (see Section 4) [31]. Q-learning, in contrast, is more indirect. The idea is to learn offline a score function of state-action pairs, called the action-value function, that captures the expected total reward [32]. Once the action-value is known, one can derive an optimal policy from it. The approaches form the basis of the two main families of algorithms, known as policy methods and value methods: modern methods usually combine the two.

Because of its generality and its treatment of the environment as a black-box, reinforcement learning is often applicable where other methods cannot be applied. Unfortunately, this generality comes at a cost, and reinforcement learning methods suffer from numerous downsides, such as high computational requirements and difficulties learning in environments with a high number of actions or long episodes.

## 2.3. Imitation learning

An alternative to reinforcement learning for solving control problems is imitation learning, where one aims to mimic an expert policy. This is only possible in contexts where an expert that achieves a high return exists, but which for some reason cannot be used directly. The most important advantage of this approach is its simplicity, and efficiency. An additional reason to prefer imitation learning over reinforcement learning is that, in reinforcement learning, directly optimizing the reward function is often unfeasible, leading one to use proxy reward functions that are easier to optimize, but only roughly correlate with the desired reward function. In contrast, designing an expert agent might be much more straightforward.

More precisely, in imitation learning we are given a dataset of state-action pairs $(s_i, a_i)$ from the expert policy $\pi_{\text{expert}}$, and we aim to train a machine learning model $\pi_{\text{model}}$ such that $\pi_{\text{model}}(s_i) \approx a_i$ over these samples. Two approaches exist [33]: directly learn to mimic $\hat{p}$ (direct imitation learning) [34], or instead learn a reward function that suits $\hat{p}$ (inverse reinforcement learning) [35], after which one can use standard reinforcement learning techniques.

In this thesis, we will focus on direct imitation learning. Since the objective is to find a function such that $\pi_{\text{model}}(s_i) \approx a_i$, it is appealing to simply treat the task as a supervised learning problem. This approach is known as behavioral cloning, and often works surprisingly well [36]–[38]. Nevertheless, it suffers from two major problems: $\text{SUPP}(\pi_{\text{expert}})$ may not be independent and identically distributed, and it may not cover states encountered by the

learned policy. To address those issues, one can use the stochastic mixing iterative learning (SMILE) algorithm (Ross & Bagnell, 2010) [39] or the dataset aggregation (DAGGER) algorithm (Ross, Gordon & Bagnell, 2011) [40]. Alternatively, a simpler approach consists of intentionally diversifying state-action pairs by occasionally not following the expert (making mistakes), that is registering expert's decisions but not taking its recommended actions. Performing standard supervised learning methods on such a diversified dataset can lead to learning a policy that better captures the expert behavior.

# 3. Machine learning models

The previous section described the machine learning paradigms, where it was explained that one must select a parameterized family of probability distributions, known as a model, to approximate the true distribution. In this section, we will discuss popular choices, with particular details for supervised and, by extension, imitation learning. Selection of the model is a critical choice, and can make in important difference between excellent and poor generalization ability [41], [42].

## 3.1. Linear and logistic regression

Besides the discriminative/generative distinction between supervised learning methods outlined in the previous section, another important dichotomy is between regression and classification problems. A regression problem is one where the output space is continuous, while a classification problem is one where the output space is discrete.

Let us first look at regression. A simple example could be a college bookstore that wants to predict how many instances of a given book are in stock for a given class, knowing how many students are registered in such class. The history may be plotted as depicted in Figure 1.1. To solve this problem, the simplest approach would be to use a linear regression model. Formally, linear regression assumes that $p_{\text{model}}(y|x;\theta,\theta_0) = p_{\text{Normal}}(y \mid \theta^T x + \theta_0, 1)$[2], parameterized by parameters $(\theta,\theta_0) \in \mathbb{R}^d \times \mathbb{R}$. Under this model, the log-likelihood simplifies as the mean-square error with linear predictions $\mathbb{E}_{X,Y\sim\hat{p}}\left[\left(\theta^T X + \theta_0 - Y\right)^2\right]$. Geometrically, solving this optimization problem can be interpreted as attempting to make the hyperplane $\theta^T x + \theta_0 = 0$ fit the data points as well as possible, in the sense that the mean distance of the points to the hyperplane is as small as possible. After solving the supervised learning optimization problem using gradient descent (see Section 4), Figure 1.2 shows the learned line (one-dimensional hyperplane parameterized by $(\theta,\theta_0)$) that would be used for upcoming inferences. Given a test dataset, a performance measure could be the square root of the mean square error.

[2]That is, the normal distribution with mean $\theta^T X + \theta_0$ and identity variance.

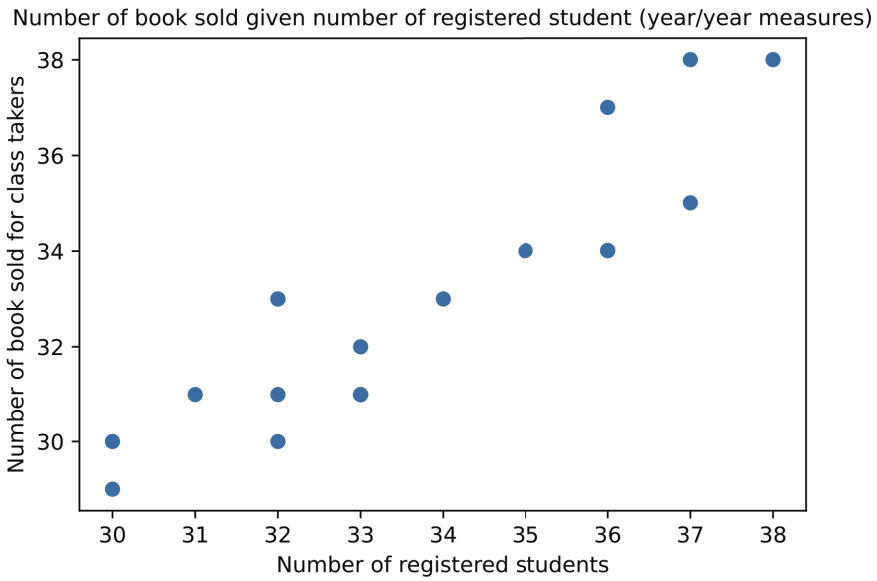


**Fig. 1.1.** An example of a supervised learning dataset.

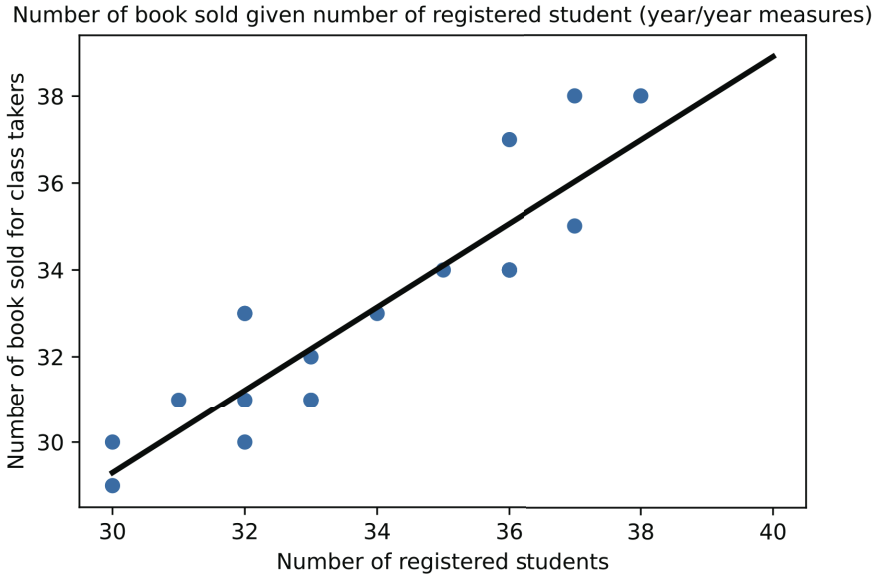


**Fig. 1.2.** An example of a data-fitting hyperplane for the linear regression problem.

Let us now turn to classification. As a simple example, the college bookstore might know information about its clients (age, major, or other characteristics) for each year, and might want to predict whether the client is likely to make a purchase at the bookstore in the upcoming semester. The features are the yearly information about the client $x$, and the response is a binary variable $y = 1$ if the client makes a purchase, $y = 0$ if not. To tackle this problem, the simplest approach is logistic regression, which is the analogue of linear

regression for classification. Let the $\sigma(x) = 1/(1 + e^{-x})$ denote the sigmoid function. This function is useful for two reasons: its output may be interpreted as a probability measure, and the derivative of $\log \sigma(x)$ has the particularly simple form $\sigma(-x) = 1 - \sigma(x)$.

Logistic regression assumes that $p_{\text{model}}(y|x; \theta, \theta_0) = p_{\text{Bernoulli}}(y \,|\, \sigma(\theta^T X + \theta_0))$[3], the resulting model is logistic regression. The resulting empirical risk is known as the binary cross-entropy loss function (to not be confused with cross-entropy operator between two distributions, which is a more general concept) and is of the form $\mathbb{E}_{X,Y\sim\hat{p}}\left[Y \log \sigma(\theta^T X + \theta_0) + (1 - Y)(1 - \log \sigma(\theta^T X + \theta_0))\right]$. Geometrically, solving this optimization problem can be interpreted as attempting to make the hyperplane $\theta^T x + \theta_0 = 0$ separate data points into two classes as well as possible, in the sense that the average signed distance of the points to the hyperplane is as high as possible.

As an example, Figure 1.3 shows a training set consisting of 2-D featured inputs with their associated class in the $R^2$ cartesian plan, while Figure 1.4 shows the linear decision boundary obtained by fitting a logistic regression model. As can be seen, the line is quite successful on the data used for training. Figure 1.5 shows this decision boundary plotted on top of new testing samples. Two points were misclassified over twenty points, yielding an accuracy of 90%.

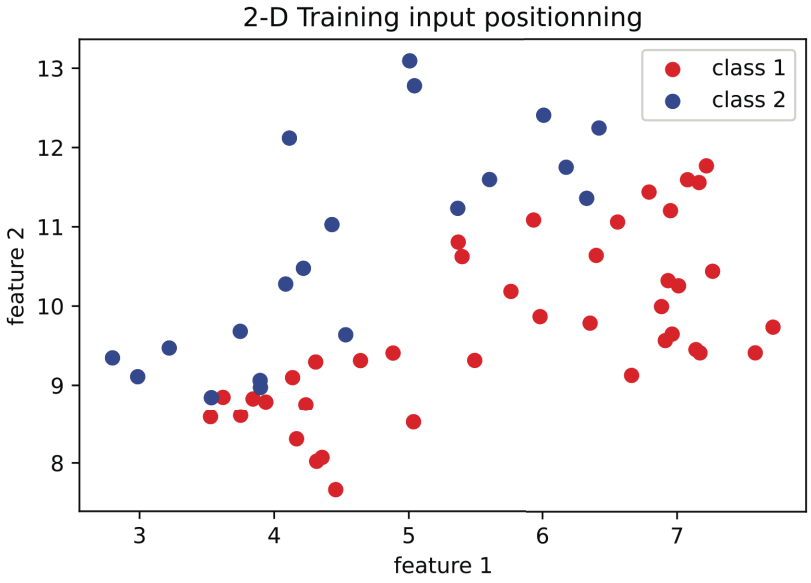


**Fig. 1.3.** The 2D binary classification dataset ABDl-DS40 sampled from the ABDl distribution.

[3]That is, the Bernoulli distribution with probability $\sigma(\theta^T X + \theta_0)$.

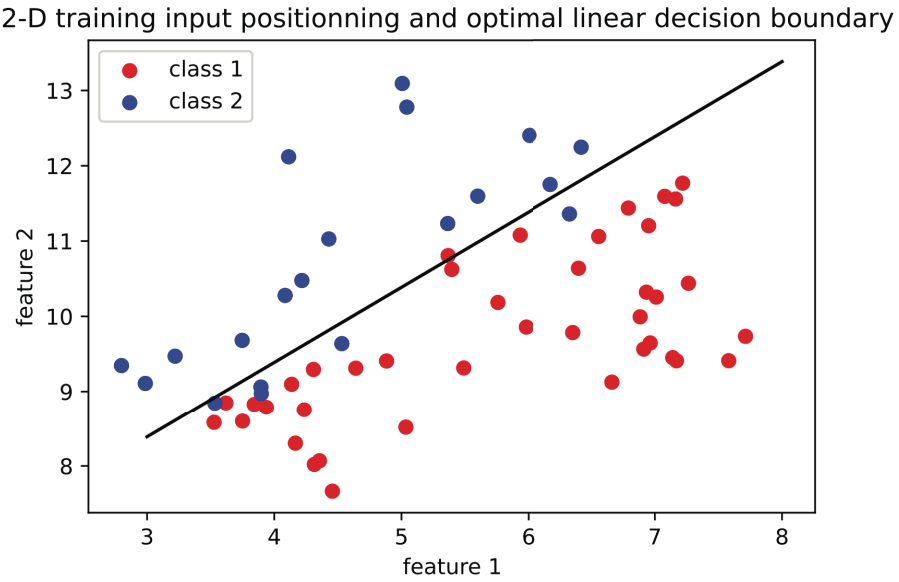


**Fig. 1.4.** Logistic regression's decision boundary on ABDl-DS40

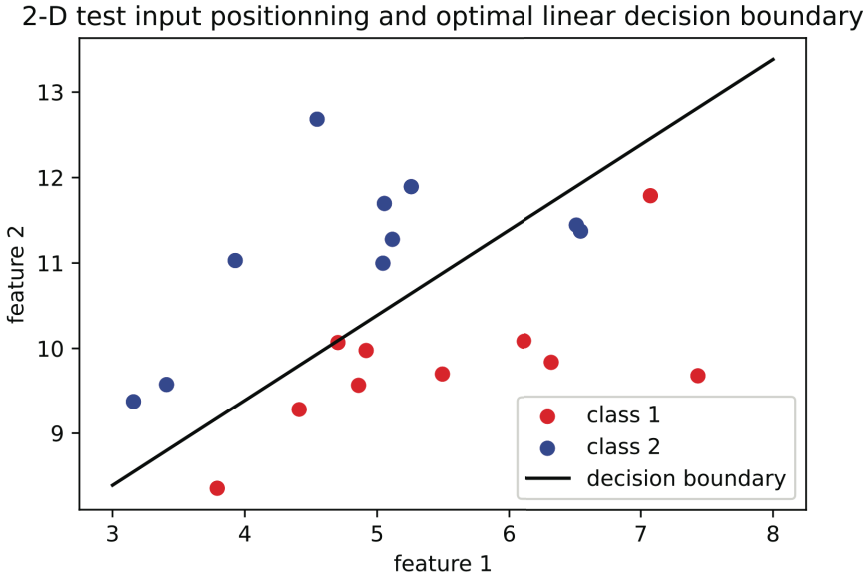


**Fig. 1.5.** Separating 20 new samples of ABDl

## 3.2. Support vector machines

The models introduced in the previous section have in common that they try to fit linear objects (hyperplanes) onto the data, either as the closest fit (for regression) or as a separation boundary (for classification). Accordingly, these are referred to as linear methods. Unfortunately, data is rarely well approximated or separated by linear planes: for such data,

it would be desirable to have models that can use more flexible objects. For example, the manual transformation depicted in Figure 1.6, which enables linear classifiers to operate effectively, can be learned by an MLP.

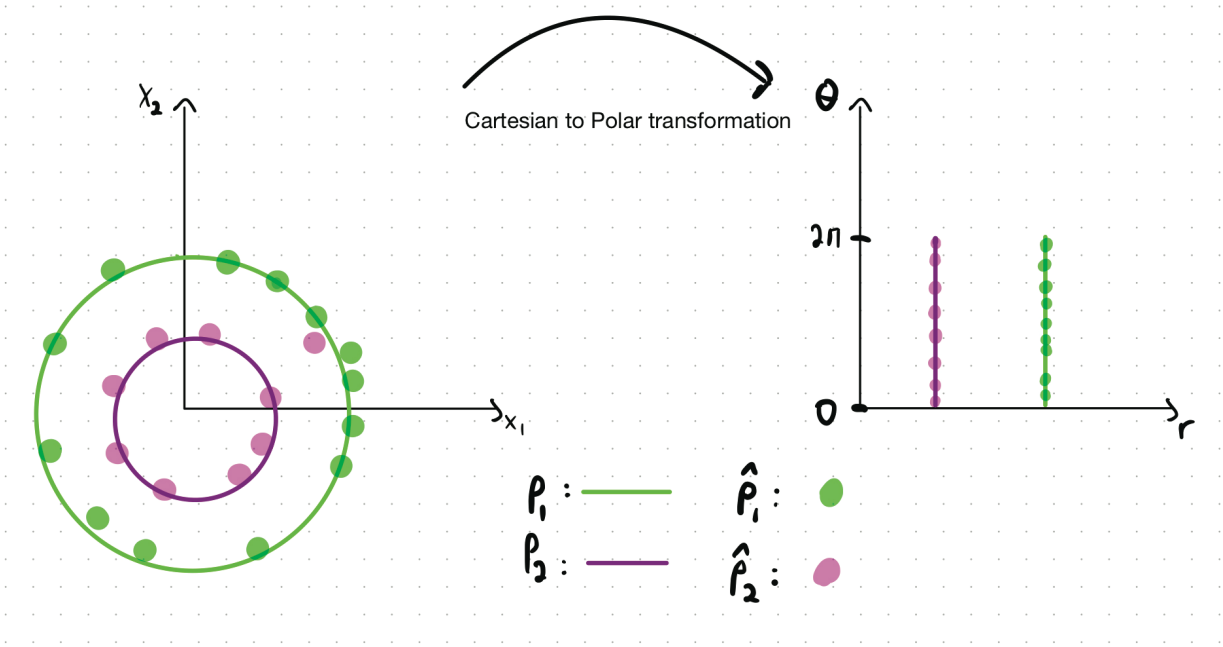


**Fig. 1.6.** An example of a transformation that enables linear models to effectively discriminate data of the ABDL-Circles distribution.

We now introduce a more complex model which can efficiently find nonlinear boundaries, known as support vector machines (SVMs). This model is usually introduced for binary classification only: multiclass SVMs exist, and an analog for regression (known as support vector regression) exists, although it is less commonly used. For simplicity, in this section, we will only detail the classical binary classification case.

We start by describing SVMs with linear decision boundaries. The goal of logistic regression was to maximize the points' average signed distance to the hyperplane. SVMs, in contrast, operate according to a different principle: it attempts to fit *two* hyperplanes, parallel to each other, in between the points of the two classes, so that the distance between the hyperplanes (the "margin") is as large as possible. The hyperplane $\theta^T x + \theta_0 = 0$ midway between the parallel planes can then be taken as the linear decision boundary.

Let $n := |\mathcal{D}_{train}|$. Formally, finding the highest-margin hyperplane $\theta^T x + \theta_0 = 0$ can be cast as the optimization problem

$$\min_{(\theta, \theta_0) \in \mathbb{R}^d \times \mathbb{R}} \quad \frac{1}{2}||\theta||^2 \tag{3.1a}$$

$$\text{s.t.} \quad y_i(\theta^T x_i + \theta_0) \geq 1 \quad \forall i = 1, \ldots, n, \tag{3.1b}$$

where constraints (3.1b) force the hyperplane to correctly classify all training samples (of output space $\mathcal{Y} = \{-1,1\}$, for convenience). This is called the hard-margin primal formulation.

Two comments are in order. First, technically, this approach does not quite fit with the empirical risk minimization/maximum likelihood approach described in Section 2.1: the SVM was designed before probabilistic approaches became prevalent in machine learning. Accordingly, attributing probabilities SVMs predictions is impossible, among other limitations. However, it is possible to regard SVMs as an approximation of another model called the relevance vector machine (RVM), which does have a probabilistic grounding and fits within the framework of Section 2.1 [43].

Second, the quadratic program above does not usually have a solution. Finding the best hyperplane that separates the data is only possible if at least one such hyperplane exists, and this is not usually the case. To address this, one can relax the constraints, turning them into a cost that encourages separation but allows for misclassification errors. This is the so-called soft-margin primal formulation and reads as follows:

$$\min_{(\theta,\theta_0)\in\mathbb{R}^d\times\mathbb{R}} \quad \frac{1}{2}||\theta||^2 + C\sum_{i=1}^{n}\xi_i \tag{3.2a}$$

$$\text{s.t.} \quad y_i(\theta^T x_i + \theta_0) \geq 1 - \xi_i \quad \forall i = 1, ..., n, \tag{3.2b}$$

$$\xi_i \geq 0 \quad \forall i = 1, \ldots, n. \tag{3.2c}$$

This problem includes a hyperparameter $C > 0$, which controls how sensitive one is to misclassification errors.

So far, the approach we have described only fits a linear hyperplane. We will now describe an extension of this paradigm which effectively allows SVMs to fit nonlinear boundaries, known as the *kernel trick*. This trick, in essence, maps the data to a higher dimensional space where a linear boundary can be fit, without directly computing the higher-dimensional projections. Doing so allows for decision boundaries with unorthodox shapes, allowing SVMs to classify well nonlinear data such as depicted in Figure 1.6.

A kernel, in the support vector machine literature, is a mapping $K : \mathcal{X}^2 \to \mathbb{R}$ with the property that for any collection of points $\{x_1, \ldots, x_n\}$, the matrix $K_{ij} = K(x_i, x_j)$ is positive semi-definite [43]. Popular choices include the Gaussian kernel $K(x, y) = \exp(-\|x - y\|^2/\sigma^2)$ with hyperparameter $\sigma^2 > 0$, and the polynomial kernel $K(x, y) = (x^T y + 1)^d$ with hyperparameter $d \in \mathbb{N}$. Using a linear kernel $K(x, y) = x^T y$ recovers the classical case. The choice of the kernel is considered hyperparameter tuning. Fitting a SVM with kernel $K$ can then be performed by solving the optimization problem

$$\min_{\alpha\in\mathbb{R}^n} \quad \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i \cdot \alpha_j y_i \cdot y_j \cdot K(x_i, x_j) - \sum_{i=1}^{n}\alpha_i \tag{3.3a}$$

$$\text{s.t.} \qquad \sum_{i=1}^{n} \alpha_i \cdot y_i = 0, \tag{3.3b}$$

$$C \geq \alpha_i \geq 0 \quad \forall i = 1, ..., n. \tag{3.3c}$$

with parameters $(\alpha_1, \ldots, \alpha_n)$. Once these parameters have been found, one can then compute the "bias"

$$b = \frac{1}{|J|} \sum_{i \in J} \Big[ y_i - \sum_{j \in J} \alpha_i y_i K(x_i, x_j) \Big]$$

where $J = \{i \,|\, \alpha_i > 0\}$, and make a prediction at a point $z$ using the formula

$$\hat{y} = \frac{1}{2} + \frac{1}{2} \text{sign}\Big( \sum_{i \in J} \alpha_i y_i K(x_i, z) - b \Big).$$

## 3.3. Neural networks and deep learning

In recent years there has been increased interest in another class of nonlinear models, called neural networks. These models became noteworthy for their performance on large data volume, as well as for their ability to perform well with minimal feature engineering [28], [44]. It is thus unsurprising that the approach is state of the art in heavily studied computer science problems, such as speech recognition [45], automatic translation [46], sentiment analysis [47], language generation [48], object detection [49], face recognition [50], image restoration [51] and video tracking [52].

The simplest neural network is called the perceptron [53]. This model takes the form $p_{\text{model}}(y|x; w, b) = \exp(\ell(a(w^T X + b), y))$ for parameters $(w, b)$, where $\ell$ is the negative log-likelihood of a probability distribution of interest, also called the loss function, and $a$ is a function called the activation function. Popular choices for $a$ include the relu activation function $a(x) = \max(0, x)$, and the hyperbolic tangent $a(x) = \tanh(x)$. The perceptron generalizes the simpler models of Section 3.1: taking the activation function to be the identity function and the loss function to be the mean square error function yields linear regression, while taking the activation function to be the sigmoid function $\sigma$ and the loss function to be the binary cross entropy loss yields logistic regression.

In general, regardless of the choice of loss and activation function, the perceptron is a linear model. To increase the ability to model problems that are not easily approximated by hyperplanes, one can use the multilayer perceptron (MLP) model, also called a fully connected neural network [41], [54]. Intuitively, a MLP is a stack of perceptrons, called layers, that feed into each other to make a prediction. Conceptually, using nonlinear activation functions, the model should be able to linearize the data for appropriate prediction by the final layer. In fact, the universal approximation theorem in MLPs claims that the simplest irreducible MLP can learn any discriminative/regressive function [55], as long as enough parameters ("neurons") are used.

Formally, an $L$-layer MLP is defined by

$$p_{\text{model}}(y|x; W_1, b_1, \ldots, W_L, b_L) = \exp(\ell(h_L(x), y)),$$

where $h_L(x)$ is defined recursively by

$$h_l(x) = a_l(W_l^T h_{l-1}(x) + b_l) \quad \text{for } l = 1, \ldots, n$$

and $h_0(x) = x$. The $W_l \in \mathbb{R}^{d_{l-1} \times d_l}, b_l \in \mathbb{R}^{d_l}$ are the parameters, and the architecture of the MLP is the layer sizes $d_0, d_1, ..., d_L$, which specify the number of parameters.

As an example, let us look back at the ABDL-Circles classification problem depicted in Figure 1.6. A MLP for this classification problem can be of the architecture presented in Figure 1.7, with the binary cross-entropy loss function. After training the model using gradient-descent (see Section 4), Figure 1.8 shows the original Cartesian coordinate features, and Figure 1.9 shows their projection to the hidden layer's linearly separable learned feature space.

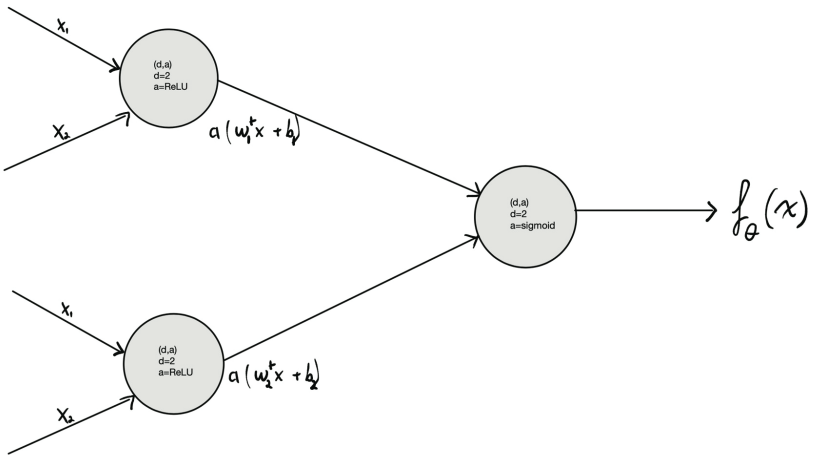


**Fig. 1.7.** Architecture of a simple MLP for binary classification

So far, all models we have covered have in common that they expect inputs (features) and outputs (responses) to have a certain fixed dimension $d_{\text{in}}, d_{\text{out}}$. A recent evolution in machine learning has been the development of models, based on feedforward neural networks, that generalize these models to other types of data with varying input size. One such class of models are recurrent neural networks (RNNs) [56], which can take as input varying-length sequences of feature vectors, outputting a final dimensional vector. An extension is sequence-to-sequence models [57] which can map sequences of inputs to sequences of outputs. In another direction, convolutional neural networks (CNNs) [58] have been developed for grid-like inputs, such as images. These models rely on applying a convolution operation (a kind of local transformation) with a learned kernel which is particularly natural for certain tasks, such as computer vision. These are in turn generalized by graph neural networks (GNNs)

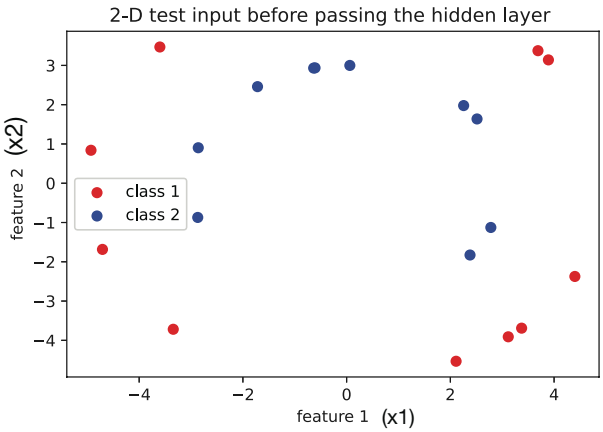


**Fig. 1.8.** ABDL-Circle original feature space samples

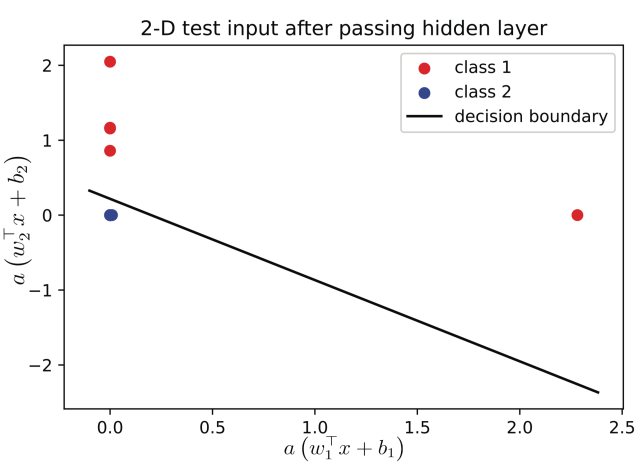


**Fig. 1.9.** ABDL-Circles samples after passing the first layer of MLP in Figure 1.7, after training the model. This learned transformation tends to make the datapoints linearly separable.

[59], which can take as input arbitrary graphs as features. These latest class of models have proven particularly successful for the field of machine learning in combinatorial optimization. Consequently, we next provide a detailed overview of CNNs and GNNs.

## 3.4. Convolutional and graph neural networks

As discussed in the preceding section, one often wants to make predictions for datasets composed of related content, but of varying sizes. For example, a dataset of images might contain pictures of different sizes, or a dataset of natural language might contain sentences of different sizes. Processing data of varying size has motivated the development of CNNs [58], which have been the angular stone for state-of-the-art neural network architectures

for computer vision [60]–[62] and have been associated with competitive ones for natural language processing [63], [64].

A CNN is similar to a feedforward neural network, instead of being composed of perceptrons, it is composed of convolutional layers. Intuitively, convolutions aggregate local information by performing an averaging with a matrix of weights in a neighborhood of each point [28]. Formally, given an input $X \in \mathbb{R}^{n \times n \times d_{in}}$, of which $X_{i,j}$ denotes an element in $\mathbb{R}^{d_{in}}$ for every pair of $(i,j) \in \{1,...,n\}^2$, a convolution layer is defined to return $\widetilde{X} \in \mathbb{R}^{n \times n \times d_{out}}$ as follows:

$$\widetilde{X}_{i,j} = \sum_{k'_1, k'_2 \in \{1..k\}^2} W^T_{k'_1, k'_2} X_{i+k'_1-1, j+k'_2-1}$$

where $W \in \mathbb{R}^{k \times k \times d_{in} \times d_{out}}$ contains the so-called filter weights and $k \in \mathbb{N}$ is called the kernel size. A filter or kernel is simply a $k \times k$ matrix. Thus, in this model, there are $d_{in} \times d_{out}$ number of filters. Figure 1.10 provides an example of the computation of a convolution operation over an image, where $X$ is an $200 \times 300$ dimensional image, $(d_{in}, d_{out}) = (1, 1)$, and $k = 3$.

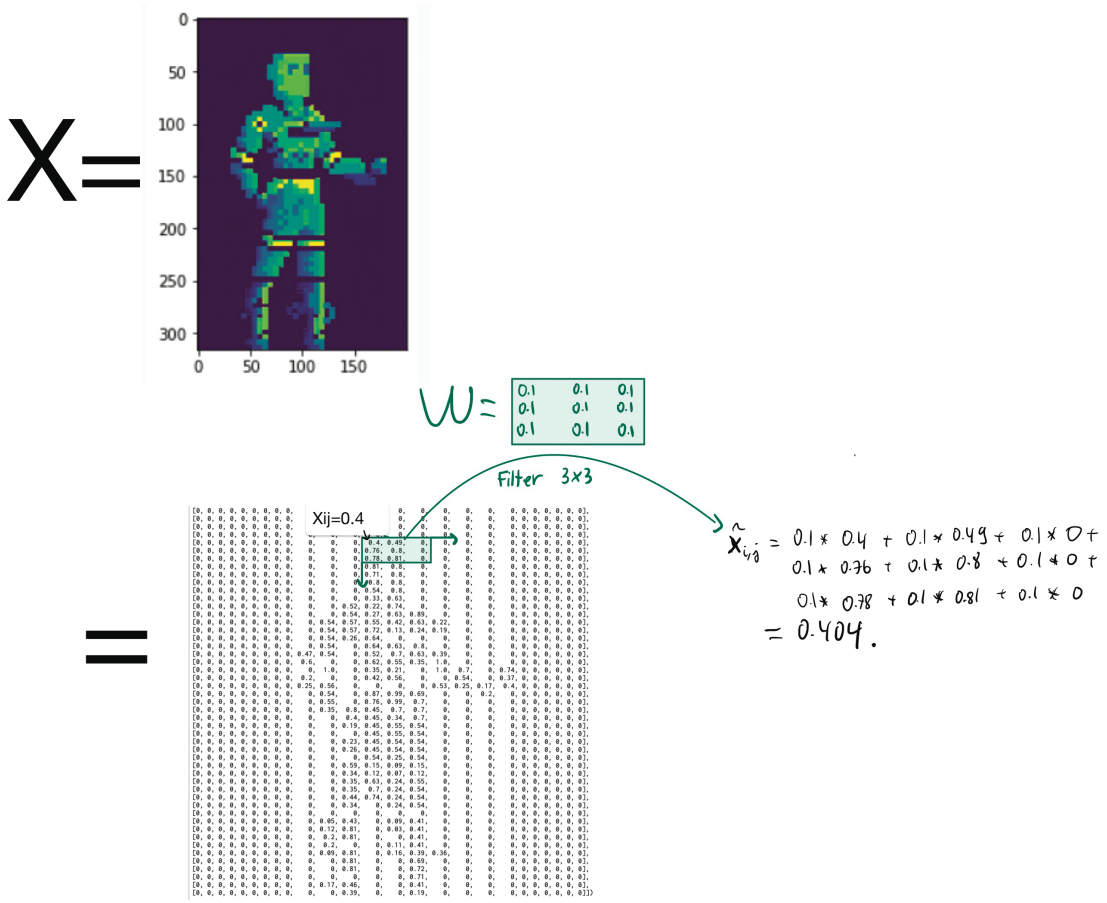


**Fig. 1.10.** Convolutional layer operating on an image

CNNs are applicable when data is structured in a grid, such as in an image. This grid can be interpreted as a graph with a lattice topology. A powerful generalization of CNNs, called graph neural networks (GNNs), apply a convolution operation, but on an arbitrary graph with features. Just as a CNN is composed of convolutional layers, a GNN is composed of graph convolutional layers, which aggregate information for each node from adjacent nodes. Consider a graph $G = (X,A)$, where $X \in \mathbb{R}^{n \times d_{in}}$ corresponds to individual node features, and $A \in \mathbb{R}^{n \times n}$ the adjacency matrix (with $d_{in}$ the feature dimensionality of individual nodes, and $n$ the number of nodes in the graph). Figure 1.11 shows a concrete example of how a graph is modeled according to this representation scheme.

Formally, given a graph $(X,A)$, a graph convolution computes an output $\widetilde{X}$ of dimensionality $\mathbb{R}^{n \times d_{out}}$ as follows [59]:

$$\widetilde{X} = (\widetilde{D}^{\frac{1}{2}} \widetilde{A} \widetilde{D}^{-\frac{1}{2}}) XW,$$

where $\widetilde{A} = A + I$, $D_{ii} = \sum_j \widetilde{A_{ij}}$ a diagonal matrix, $W \in \mathbb{R}^{d_{in} \times d_{out}}$ are the model parameters, and $X \in \mathbb{R}^{N \times d_{in}}$ are the graph's input nodes representations, with $d_{in}$ and $d_{out}$ being respectively node representation input size (also called in-channel) and node representation output size (also called out-channel). Thus a graph convolution outputs the same input topology but with the nodes' features altered with neighboring information. GNNs have been reported to state-of-the-art performances in tasks where data are naturally representable as graphs, e.g., disease–gene association, recommendation systems, adversarial attack prevention [65] and combinatorial optimization [26], [66].

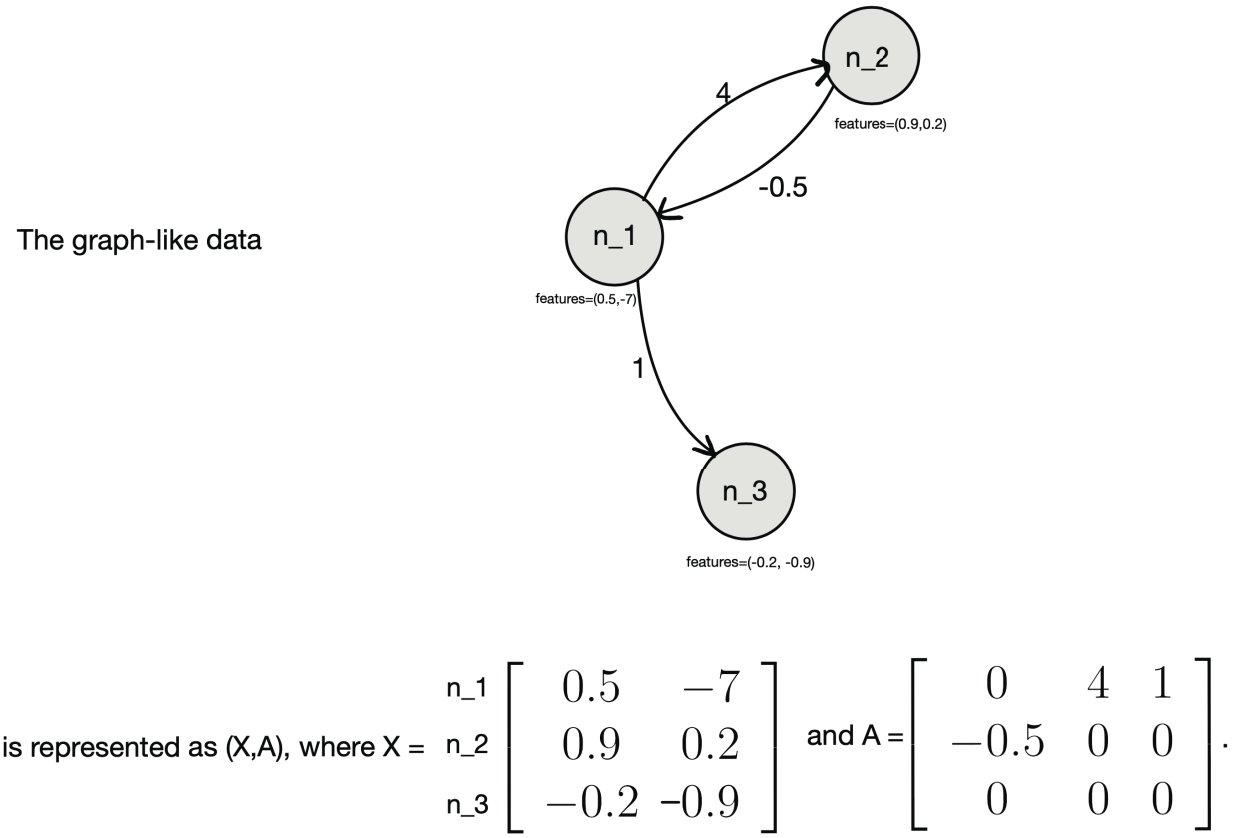


**Fig. 1.11.** Representing a graph-like data

# 4. Optimization for neural networks

We have discussed so far in Section 2 the paradigm of supervised machine learning, which can be framed as a continuous optimization problem, and in Section 3 various popular choices of models that completely characterize these problems. The final aspect we are left to discuss is how these optimization problems are solved in practice. We will define some important concepts in continuous optimization before introducing the algorithms, with a particular emphasis on those that are relevant for neural networks.

Consider a generic continuous optimization problem

$$\begin{aligned} \min_{\theta \in \mathbb{R}^n} \quad & f(\theta) \\ \text{s.t.} \quad & \theta \in \Theta \end{aligned} \tag{4.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable of gradient $\nabla f$. A problem is considered unconstrained if $\Theta = \mathbb{R}$; otherwise, it is said to be constrained. Let $||.|| := \sqrt{.^T.}$ (the Euclidean norm). A point $\theta \in \Theta$ such that $\nabla_\theta f(\theta) = 0$ is said to be

- A local minimum if $\exists \epsilon \in \mathbb{R}_{>0}$ s.t. $\forall \theta^{'} \in \Theta$ satisfying $||\theta - \theta^{'}|| \leq \epsilon$, we have $f(\theta) \leq f(\theta^{'})$;

- A local maximum if $\exists \epsilon \in \mathbb{R}_{>0}$ s.t. $\forall \theta^{'} \in \Theta$ satisfying $||\theta - \theta^{'}|| \leq \epsilon$, we have $f(\theta) \geq f(\theta^{'})$;
- A saddle point, otherwise.

A point $\theta \in \Theta$ is said to be a global minimum if $f(\theta) \leq f(\theta') \ \ \forall \theta^{'} \in \Theta$. It is easy to prove that all global minima are local minima, and so one can look global minima by inspecting all local minima as candidates if there are only finitely many. Figure 1.12 provides an illustration of this situation. In turn, finding local minima can be done by finding zero-valued gradient points, and this can be done analytically or iteratively. Except for simple cases, such as the optimization problem arising in linear regression, training supervised learning models usually requires an iterative method.

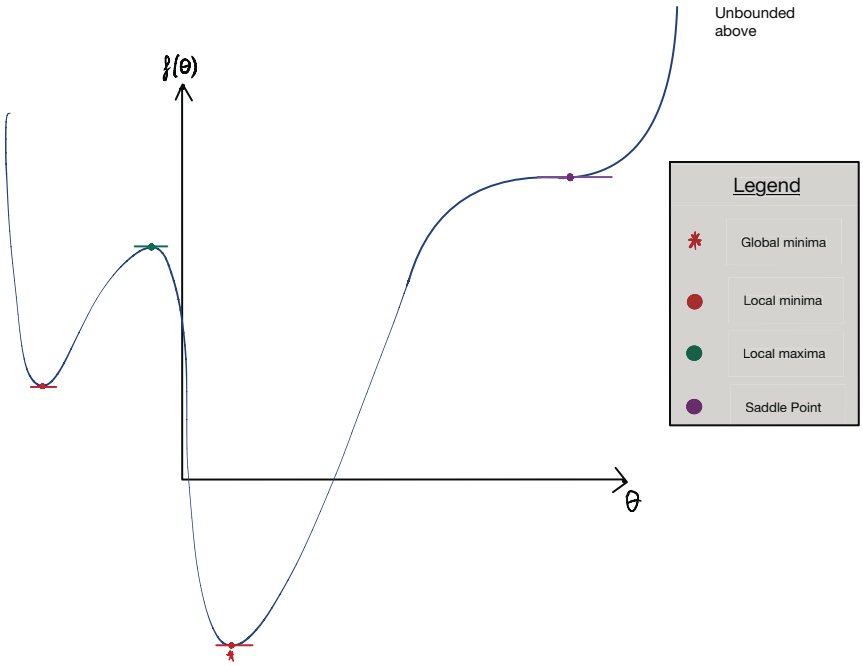


**Fig. 1.12.** 1D function with its local/global minima/maxima and saddle points

The simplest kind of iterative method for solving unconstrained optimization problems where the objective function $f$ is differentiable are first-order methods, which rely solely on the gradient $\nabla_\theta f = (\partial f/\partial \theta_1, \ldots, \partial f/\partial \theta_n)$ of the function with respect to $\theta$. These methods exploit the fact that the gradient gives, locally to a point $\theta$, the direction and magnitude of the steepest ascent when evaluated at this point.

The angular stone of first-order methods is the gradient descent algorithm. The algorithm converges to the closest point with zero gradient starting at some point $\theta^0 \in \mathbb{R}^n$ by iteratively taking a step controlled by a parameter $\eta \in \mathbb{R}^{>0}$, called learning rate, towards the opposite gradient direction (as we are minimizing). Algorithm 2 shows its functioning.

**Algorithm 2** Gradient descent algorithm

**function** GRADIENT_DESCENT($f$,$\theta^0$,$\eta$, $\epsilon$)
    $t \leftarrow 0$
    **repeat**
        $t \leftarrow t + 1$
        $\theta^t \leftarrow \theta^{t-1} - \eta \nabla_\theta f(\theta^{t-1})$
    **until** $||\theta^t - \theta^{t-1}|| \leq \epsilon$
    **return** $\theta^t$

**Algorithm 3** Stochastic gradient descent for supervised learning using mini-batch sampling scheme

**function** STOCHASTIC_GRADIENT_DESCENT_MINIBATCH($p_{model}$,$\mathcal{D}_{train}$, $m$, $\theta^0$, $\eta$, $\epsilon$)
    $t \leftarrow 0$
    **repeat**
        $t \leftarrow t + 1$
        Sample a subset $J \subset \mathcal{D}_{train}$ of size $m$
        $\theta^t \leftarrow \theta^{t-1} - \frac{\eta}{m} \sum_{j \in J} \nabla_\theta \log p_{\text{model}}(y_j | x_j; \theta^{t-1})$
    **until** $||\theta^t - \theta^{t-1}|| \leq \epsilon$
    **return** $\theta^t$

In recent years, a concern of great importance has been the training of machine learning models when the dataset is very large. This is particularly the case for the training of neural network models detailed in Sections 3.3 and 3.4, which often require large amount of data to be competitive. Consequently, a variant of gradient descent has become standard for training those models, called stochastic gradient descent (SGD). This algorithm aims to solve unconstrained continuous problems where the function to minimize has a particular form

$$f(\theta) = \frac{1}{n} \sum_{i=1}^{n} g(x_i; \theta).$$

When $n$ is very large, the gradient $\nabla_\theta f(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta g(x_i; \theta)$ might be expensive to compute. The idea of stochastic gradient descent is to instead sample a small minibatch $J$ of the indices $\{1, \ldots, n\}$, where $|J| \ll n$, and use the approximate gradient update

$$\theta' = \theta - \eta \frac{1}{|J|} \sum_{j \in J} g(x_i; \theta).$$

Algorithm 3 details the algorithm as applied to a supervised learning training problem. In practice, besides the memory savings of the approach, it is generally believed that the additional noise brought by the approximative gradient is helpful for highly nonlinear models, such as neural networks.

**Algorithm 4** The Adam algorithm [67]

**function** ADAM($f$,$\theta^0$, $m$,$\eta$, $\epsilon$, $\beta_1$, $\beta_2$)
  $m^0 \leftarrow 0$ ▷ first moment vector
  $v^0 \leftarrow 0$ ▷ second moment vector
  $t \leftarrow 0$
  **repeat**
    $t \leftarrow t + 1$
    Sample a subset $J \subset \{1, \ldots, n\}$ of size $m$
    $g^t \leftarrow \frac{1}{m} \sum_{j \in J} \nabla_\theta \log p_{\text{model}}(y_j | x_j; \theta^{t-1})$
    $m^t \leftarrow \beta_1 m^{t-1} + (1 - \beta_1) g^t$ ▷ Update biased moments
    $v^t \leftarrow \beta_2 v^{t-1} + (1 - \beta_2) g^t \odot g^t$
    $\hat{m}^t \leftarrow \frac{m^t}{1-\beta_1^t}$ ▷ Correct the bias to obtain moments estimates
    $\hat{v}^t \leftarrow \frac{v^t}{1-\beta_2^t}$
    $\theta^t \leftarrow \theta^{t-1} - \eta \frac{\hat{m}^t}{\sqrt{\hat{v}^t} + \epsilon}$ ▷ Update rule
  **until** $||\theta^t - \theta^{t-1}|| \leq \epsilon$
  **return** $\theta^t$

Many variants of this approach have been proposed. Instead of pursuing the gradient direction at each iteration, modern SGD-based optimizers usually take into account the existing movement captured by previous gradient evaluations, a mechanism called momentum SGD. This approach can make the optimization process less oscillating [68]. Moreover, using adaptive learning rates for each parameter eliminates the need to manually tune the learning rate and lead to faster and more reliable convergence [69]. One of the current state-of-the-art optimizers is Adam (Kingma & Ba, 2015) [67], which computes individual learning rates by approximating the first and second moments of the current gradient, a random variable, using previously computed and realized gradients. Algorithm 4 details this popular approach, where $\odot$ denotes elementwise multiplication.

A final remark is in order. In every gradient descent algorithm, whether Algorithm 2, 3 or 4, one needs to provide the gradient of the negative log-likelihood (loss function). To do so, one could in principle derive the gradient analytically, and code it. In practice, however, it is simpler to simply analytically derive the Jacobian of each layer, and compute the gradient using the chain rule. This approach is usually called backpropagation in the neural network literature [68].

# Chapter 2

# Data-driven heuristics for integer programming

As described in Chapter 1, it is common in applications to repeatedly solve similar mixed-integer linear programs. This has motivated much work at the intersection of machine learning and combinatorial optimization. Typically, the idea is to take an existing solver and improve one of its components using a problem-adaptive policy instead of a handcrafted generic one using data. Some past work has focused on improving separately four significant modern B&B solvers' components: primal heuristics design and their scheduling, cutting plane selection, branching, and searching. We provide in this chapter an overview of the work in this area.

## 1. Learning primal heuristics

It is possible to train neural networks to output good feasible solutions to CO problems. The first architecture of this kind was introduced in *Pointer Networks*, Vinyals et al., 2015 [70]. These consist of long short-term memory (LSTM) networks [56] specifically designed to output variable-size discrete tokens sequences. In their work, Vinyals et al. [70] trained this network to output good Traveling Salesman Problem (TSP) solutions using behavioral cloning. They reported being competitive with some handcrafted Christofides-based [71] open-source primal heuristics.[1] In 2016, Bello et al. [72] took the same architecture but used it inside an RL framework, namely the gradient-policy method REINFORCE [31], and scaled it on larger TSP instances and more NP-hard problems. Subsequently, in 2017, Khalil et al. [73] used the `structure2vec` architecture [74] to work on data that directly captures problems' structure and trained using fitted Q-learning [75] (S2V-DQN) instead of REINFORCE. The authors reported better performance than past approaches on three

[1]For the specific algorithms, see last three references in Section 3.3 of *Pointer Networks*, Vinyals et al., 2015 [70].

classes of NP-hard problems representable on graphs. Then, in 2018, Li and al. [76] reported surpassing S2V-DQN on four classes of NP-hard problems by training a Graph Convolution Network with behavioral cloning. Overall, these approaches suffer from two major issues: scalability is limited, and real-life CO problems with hard constraints are problematically represented [77].

## 2. Learning to select cutting planes

The first work concerning using machine learning to select cutting planes was initiated by Radu et al. [78], who score cutting planes by predicting the objective variation brought by adding the cut. In 2020, Tang and al. [79] proposed to formulate the cut selection problem using reinforcement learning, and used a recurrent neural network with the attention mechanism [80] and LSTM cells [56] trained with evolutionary techniques [81]. They reported beating some well-known handcrafted cutting plane selectors[2] using Gurobi solver [19]. Subsequently, in 2021, Huang et al. [83] proposed to use multiple instance learning [84] for cut ranking formulation and trained an MLP at the task. They reported beating some manually-designed cutting planes heuristics in a Huawei-owned B&B-based solver. In 2022, Paulus et al. [85] were able to design a look-ahead expert for the task, inspired by strong-branching [86]. This new expert design leveraged the use of imitation learning. As a learning model, they designed *NeuralCut*, a GNN inspired by Gasse et al. [87], with extra attention blocks [80]. They reported beating 2020's Tang et al. [79] approach on four classes of NP-hard problems.

## 3. Learning to branch

Previous operations research work on branching includes: branching to the most infeasible variable, random branching, strong branching, pseudocost branching, reliability branching, and hybrid branching [88]–[90]. Because strong branching generally produces the smallest tree size and because it requires heavy computation [90], it is an ideal expert to imitate. Thus, in 2016, Alvarez et al. [89] proposed an offline, regression-based model using Extremely Randomized Trees (ERT) [91], whereas Khalil et al. [92] used pairwise ranking of variables with a SVM rank model (SVMRANK) [93]. In 2018, Hansknecht et al. [94] proposed to use LambdaMart [95] as a ranking model instead of an SVM Rank (LMART) and specialize their work on the TSP problem. Subsequently, in 2019, Gasse et al. [87] proposed a classification formulation using a new data representation: variable-to-constraint graphs. They reported beating ERT, SVMRANK, and LMART on four classes of NP-hard problems. Then, in 2021, Zarpellon et al. [96] used a variable and tree search parameterization on custom

[2]For a description of these handcrafted cutting planes heuristics, we refer the interested reader to Section 2.2 of *Implementing cutting plane management and selection techniques*, Wesselmann and Shulh, 2012 [82]

deep learning architecture to show that using structured information from the tree enabled learning problem-independent branching policies, whereas Gasse et al. [87] approach could not.

# 4. Learning to search

Finally, a fourth topic that has received attention is the usage of machine learning methods to improve search heuristics. This is the central problem of this thesis, and so we provide a detailed explanation of the search mechanism and its associated handcrafted heuristics, with particular emphasis on the implementation in the SCIP open-source MILP solver.

## 4.1. Searching mechanism in SCIP

SCIP [97], standing for Solving Constrained Integer Programs, is a free, open-source `C++`, academics-targeted software first by Achterberg for his Ph.D. thesis in 2004 and currently maintained, i.e., updated, improved, and documented, by the Zuse Institute Berlin Group (ZIB)[18]. The software provides a framework to create branch-and-bound-based search algorithms. Figure 2.1 shows how the different highly customizable SCIP components interact with each other to solve a problem.

This thesis aims at improving node selection, aka SCIP's B&B searching mechanism, without touching any other components. In SCIP, each node selection policy is associated with a `Nodesel` object, referred to as node selector, and with an execution priority. The `Nodesel` class should implement the following methods [18]:

- `nodecomp`: given two nodes, returns which one is preferred or equality
- `nodeselect`: given a list of open nodes, returns which one to explore next

Even if implementing only one of `nodeselect` or the `nodecomp` methods is sufficient for specifying a search policy, SCIP makes use of both of them in order to maximize solving performance. The `nodecomp` is used to maintain a priority queue of nodes to explore (by inserting new nodes in the appropriate spot), whereas the `nodeselect` decides to either simply remove and return the queue's top node, aka the dequeue operation implemented by `getBestNode` function, or to select another open-node. On extreme cases, only using the priority queue in `nodeselect` may overcharge the `nodecomp` operator, whereas not using at all the priority queue may overcharge the `nodeselect` operator. Though both are about queue usage, more specifically on insertions, we believe that the latter extreme case is a worse case since having no ordering information on open nodes means that reevaluating each node each selection is unavoidable.

The SCIP software is delivered with six handcrafted node selectors implementing both `nodecomp` and `nodeselect` [18]:

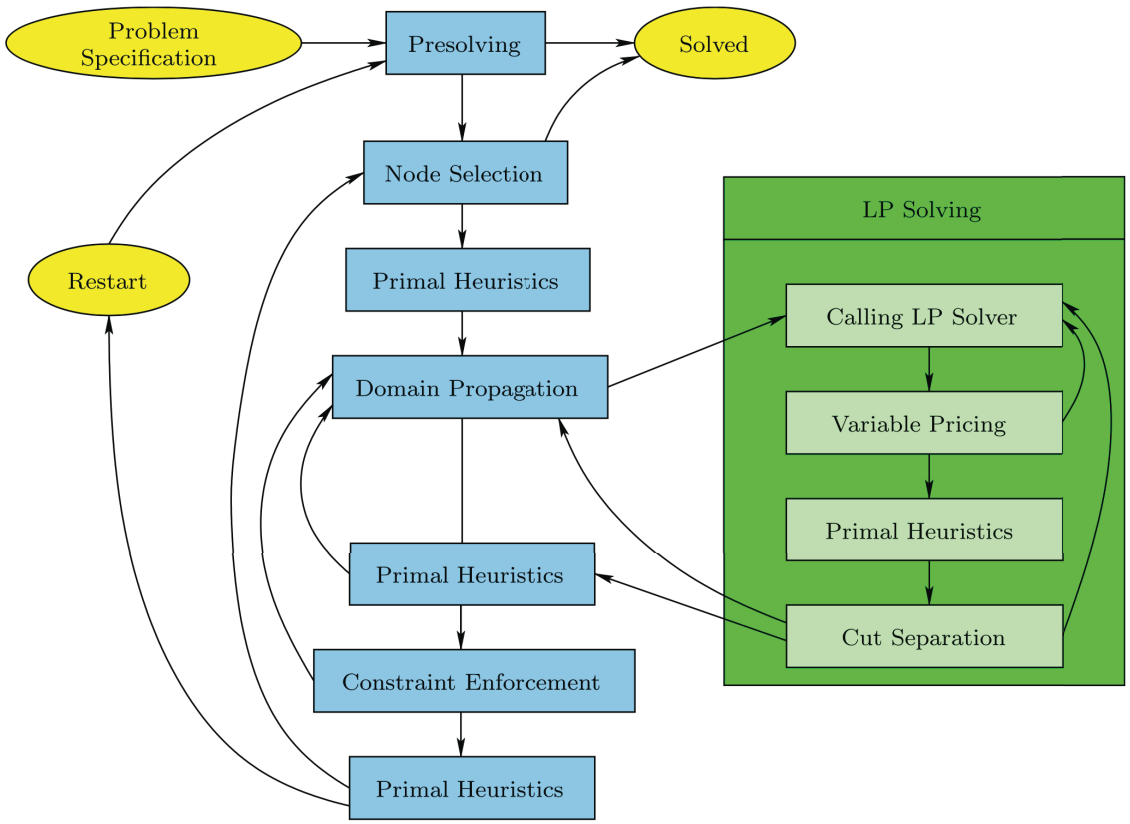


**Fig. 2.1.** SCIP solver general workflow [97]. Node selection refers to the process of searching in the feasible region partition, where each equivalence class is referred to as a node. Branching procedures are not explicitly displayed but occur just after calling the LP solver and before pricing variables.

- best-estimate: from solver's pseudostatistics, computes (as node score) an estimate of the objective decrease resulting in node potential selection [90], [97]. Prefers the highest score;
- best-first search: prefers nodes with higher dual lower bounds [98];
- depth-first search (DFS): prefers deeper and newer nodes;
- breadth-first search: prefers shallower and older nodes;
- hybrid estimate (also called best-bound search): computes (as node score) a weighted sum of node's dual lower bound and its primal upper bound [18]. Prefers the lowest score;
- restart DFS: a recursive depth-first search.

The solver operates in two computational modes: NORMAL MODE and MEMORY SAVING MODE. Depending on the node selectors' priorities and the active mode, the appropriate node selector will be used at that moment. Table 2.1 shows the default, initial priority table of SCIP built-in node selectors. In MEMOMRY SAVING MODE, the active policy is depth-first as it tends to produce fewer unprocessed nodes while reducing the number of open nodes,

freeing up some memory. In contrast, in NORMAL MODE, the active node selection procedure is BEST ESTIMATE, which corresponds to node selection's state of the art [99], [100].

**Table 2.1.** SCIP built-in node selectors and their default execution priorities (higher = more important)

| | estimate | bfs | dfs | breadth-first | nodesel_hybridestim | ,restartdfs |
|---|---|---|---|---|---|---|
| normal mode | 200,000 | 100 000 | 0 | -10,000 | 50,000 | 10,000 |
| memory saving mode | 100 | 0 | 100,000 | -1,000,000 | 50 | 50,000 |

One can use a plain `nodeselect`, i.e., a method that just dequeues the priority list and focus entirely on specializing the `nodecomp` operator. An active, added customized node selection policy will be entirely specified by its `nodecomp` method and a maximal execution priority. This is the Node Comparison Problem in SCIP (in a plain setting). Since SCIP's searching mechanism is similar to other B&B-based solvers', we will refer to this as the Node Comparison Problem in Branch and Bound. Comparing nodes enables intuitive machine learning formulation. One of interests is to guide search heuristics on where the optimum is. It is possible to approximate such behavior by learning to discriminate between optimum-containing and non-optimum-containing nodes, aka optimal and non-optimal nodes. We hypothesize that the solving performance gain made possible by such specialization can provide enough room to beat SCIP both `nodecomp`/`nodeselect`-implementing handcrafted heuristics.

## 4.2. Data-driven node comparison

The first steps towards data-driven node comparison were taken by He et al. [101]. They proposed to train an SVM model using the DAGGER algorithm [40] to imitate the node comparison operator of a diving oracle, i.e., an expert that plunges to optimum. They used it in combination with a learned pruning model that cuts off unpromising branches of the branch-and-bound tree, yielding something more analogous to a primal heuristic. They reported improvements in the optimality gap against SCIP under a node limit and against Gurobi [19] under a time limit on four benchmarks.

Subsequently, Song et al. [102] trained an MLP RankNet model to perform node comparison using a novel approach they call retrospective imitation learning. In this approach, as applied to the branch-and-bound algorithm, a solver is executed until a specific node limit (or potentially until optimality). The node selection trajectory is then corrected into the shortest path to the best solution found during the process. When the solver is executed until optimality, this is identical to trajectories generated by the diving oracle. In practice, they generated trajectories using Gurobi and trained using DAGGER and SMILE [39] algorithms. Unlike He et al., they provided results that only use the learned node comparator

without an additional pruning operator. On a collection of path planning integer programs, they reported impressive improvements in the optimality gap under a node limit against Gurobi and SCIP. However, their appendix also reported less thorough results on a more challenging combinatorial auctions benchmark used by He et al.

Finally, and more recently, Yilmaz and Yorke-Smith [103] proposed to learn a limited form of feedforward neural network node comparison operator that decides whether the branch-and-bound algorithm should expand the left child, right child, or both children of a node. This operator can then be combined with a backtracking algorithm to provide a full node selection policy. This approach can be interpreted as a combination of the neural network node comparator of Song et al. [102] with a node selection rule that only calls it on children of the current node and reverts to depth-first search otherwise. The authors used the state encoding from Gasse et al. [87] and trained their model using behavioral cloning [34] to imitate an oracle that prioritizes nodes on a path towards one of the $k \geq 1$ best solutions – in practice, a generalization of the He et al. [101] oracle. On three NP-hard benchmarks, they reported improvements in time and number of nodes against He et al. [101], and sometimes in nodes against SCIP; in a fourth benchmark, they were slightly worse than He et al. [101].

## 4.3. Node comparison with GNNs

Our approach aims at training a GNN to output a node "score" given a graph bipartite representation, which we take as the one of Gasse et al. [87]. This score serves to discriminate between optimal and non-optimal nodes. Intuitively, deeper optimal nodes should have the highest scores and shallow non-optimal ones the lowest. To learn such a scoring function, we generate data using a flawed branch-and-bound algorithm to obtain as many optimal/non-optimal discriminations and depth differences as possible. We then use a standard classification learning framework with exponential emphasis on higher depth differences between pairs of nodes. Such formulation leverages inherent deep learning optimization schemes to implicitly learn the scoring function of interest. The article of the next chapter summarizes this work and experimental results.

**First Article.**

# Learning to Compare Nodes in Branch and Bound with Graph Neural Networks

by

Abdel Ghani Labassi[1], Didier Chételat [2], and Andrea Lodi[3]

([1]) Université de Montréal
2920 Chemin de la Tour, Montréal, QC, Canada

([2]) Polytechnique Montréal
2900 Boul. Edouard-Montpetit, Montréal, QC, Canada

([3]) Cornell University
2 West Loop Road, New York, NY, USA

This article was submitted in Neural Information Processing Systems (NeurIPS) 2022.

Contributions of the author to the paper: Most of the project was done by the main author, including model and data collection design, coding, running the experiments and writing the paper. The co-authors 2 and 3 provided scientific and technical guidance, and helped with paper structure.

Résumé. Les approches branch-and-bound dans la programmation en nombres entiers nécessitent d'ordonner des portions de l'espace à explorer ensuite, un problème connu sous le nom de comparaison de nœuds. Nous proposons un nouveau modèle de réseau de neurones à graphes siamois pour résoudre ce problème, où les nœuds sont représentés sous forme de graphes bipartis avec des attributs. Semblable aux travaux antérieurs, nous entraînons notre modèle pour imiter un oracle qui plonge vers la solution optimale. Nous évaluons notre méthode en résolvant les instances dans un cadre simple où les nœuds sont explorés selon leur rang. Sur trois problèmes NP-hard choisis pour être particulièrement difficiles au niveau primal, notre approche conduit à une résolution plus rapide et à des arbres de recherche plus petits qu'en utilisant la fonction de classement par défaut du solveur SCIP, ou bien aux autres modèles d'apprentissage automatique que constituent nos compétiteurs. De plus, ces résultats se généralisent à des instances plus grandes que celles utilisées pour l'entraînement.
**Mots clés :** Optimisation combinatoire, Branch-and-Bound, Recherche de solutions, Plonger-à-l'optimum, Apprentissage par imitation, Réseaux de neurones à graphes

Abstract. Branch-and-bound approaches in integer programming require ordering portions of the space to explore next, a problem known as node comparison. We propose a new siamese graph neural network model to tackle this problem, where the nodes are represented as bipartite graphs with attributes. Similar to prior work, we train our model to imitate a diving oracle that plunges towards the optimal solution. We evaluate our method by solving the instances in a plain framework where the nodes are explored according to their rank. On three NP-hard benchmarks chosen to be particularly primal-difficult, our approach leads to faster solving and smaller branch-and-bound trees than the default ranking function of the open-source solver SCIP, as well as competing machine learning methods. Moreover, these results generalize to instances larger than used for training.
**Keywords:** Combinatorial Optimization, Branch-and-Bound, Solutions Search, Dive-to-Optimum, Imitation Learning, Graph Neural Networks

# 1. Introduction

Mixed-integer linear programming is an optimization paradigm with applications as varied as airline scheduling [104], CPU management [105], auction design [106] and industrial process scheduling [107]. Modern solvers rely on the branch-and-bound (B&B) algorithm, which recursively divides the search space into a tree, solving relaxations of the problem until an integral solution is found and proven optimal [108]. Throughout this procedure, numerous decisions must be repeatedly made, such as the choice of the variable on which to branch or the choice of primal heuristics to run at every node. These decisions often dramatically impact final performance yet are still poorly understood [109]. Traditionally, these would be made according to hard-coded expert heuristics implemented in solvers. Recently, however, there has been a surge of interest in using machine learning methods to learn such heuristics [26], in particular for variable selection [87], [96], [110]–[112].

Despite this success, other critical branch-and-bound decision tasks remain poorly studied. One of the most important is the node comparison problem. Throughout solving, the algorithm must repeatedly select the next node to subdivide, a task known as node selection. It maintains a priority list of the open nodes, ordered according to a node comparison function. This list is then used to select the next node to subdivide, either by simply choosing the highest-ranked node or through some more complex paradigm. Interestingly, a few works have proposed to use machine learning methods to derive node comparison functions [101]–[103]. This is particularly promising since the problem is naturally amenable to statistical learning methods. However, despite promising results, challenges hinder progress in this area. Most prominently, it is unclear how to represent nodes, which can vary in the number of variables and constraints. Existing approaches have so far relied on fixed-dimensional representations that necessarily lose information.

In this paper, inspired by work on the related problem of variable selection in branch and bound [87], [96], [110], [111], we propose to tackle this problem by an approach based on graph neural networks (GNNs) [113]. We represent nodes by bipartite graphs with attributes and use a siamese architecture to model the node comparison function. This node representation allows complete information regarding the nodes to be provided to the model, reducing the amount of manual feature engineering. In line with previous work on this node comparison problem [101]–[103], we train the network using imitation learning to approximate a diving oracle that plunges towards the optimal solution.

We compare our GNN approach against the support vector machine approach of He et al. [101], the feedforward neural network approach of Song et al. [102] and Yilmaz and Yorke-Smith [103], and the default node selection rule of the open-source solver SCIP [18]. In addition, we compare against the node comparator of this same branching rule but with a highest-rank node selection rule. Results show that our approach leads to improved node selection compared to competing machine learning approaches and, in fact, often improves on the default rule in SCIP itself. In addition, these results generalize to instances larger than those used for training.

The paper is divided as follows. In Section 2, we review the related literature, while in Section 3, we describe the branch-and-bound algorithm and the node comparison problem. In Section 4, we describe our state representation, neural network architecture, as well as training procedure. Finally, we detail experimental results in Section 5.

## 2. Related works

The first steps towards learning node comparison heuristics in branch and bound were taken by He et al. [101]. In this work, they propose to train a support vector machine (SVM) model using the DAGGER algorithm [40] to imitate the node comparison operator

of a diving oracle. However, they only use it in combination with a learned pruning model, which cuts off unpromising branches of the branch-and-bound tree, yielding something more analogous to a primal heuristic. They report improvements in the optimality gap against SCIP under a node limit and Gurobi [19] under a time limit on four benchmarks.

Subsequently, Song et al. [102] trained a multilayer perceptron (MLP) RankNet model to perform node comparison using a novel approach they call retrospective imitation learning. In this approach, as applied to the branch-and-bound algorithm, a solver is run until a certain node limit (or potentially until optimality). The node selection trajectory is then corrected into a shortest path to the best solution found during the process. When the solver is run until optimality, this is in effect identical to trajectories generated by the diving oracle. In practice, they generated trajectories using Gurobi and trained using the DAGGER and SMILe [39] imitation learning algorithms. Unlike He et al., they provided results that only use the learned node comparator without an additional pruning operator. On a collection of path planning integer programs, they report impressive improvements in the optimality gap under a node limit against Gurobi and SCIP. However, their appendix also reports more mitigated results on a more challenging combinatorial auctions benchmark used by He et al.

Finally, and more recently, Yilmaz and Yorke-Smith [103] proposed to learn a limited form of feedforward neural network node comparison operator that decides whether the branch-and-bound algorithm should expand the left child, right child or both children of a node. This operator can then be combined with a backtracking algorithm to provide a full node selection policy: in effect, this can be interpreted by combining the neural network node comparator of Song et al. with a node selection rule that only calls it on children of the current node, and reverts to depth-first search otherwise. They use the state encoding from Gasse et al. [87] and train their model using behavioral cloning [34] to imitate an oracle that prioritizes nodes on a path towards one of the $k \geq 1$ best solutions - in effect a generalization of the He et al. oracle. On three benchmarks, they report improvements in time and number of nodes against He et al., and sometimes in nodes against SCIP; in a fourth, they are slightly worse than He et al.

# 3. Background

A mixed-integer linear program (MILP) is an optimization problem of the form

$$\arg\min_{x \in \mathbb{Z}^k \times \mathbb{R}^{n-k}} \{c^t x : Ax \geq b\},$$

for a matrix $A \in \mathbb{R}^{m \times n}$ and vectors $b \in \mathbb{R}^m, c \in \mathbb{R}^n$. The branch-and-bound algorithm solves this problem recursively as follows. First, the linear program (LP) relaxation $\arg\min_{x \in \mathbb{R}^n} \{c^t x : Ax \geq b\}$ is solved, which can be done efficiently in practice. This relaxation yields a solution $x^*$, with a lower bound $c^t x^*$ to the MILP. If the LP solution satisfies

the integrality constraints, $x^* \in \mathbb{Z}^k \times \mathbb{R}^{n-k}$, the problem is solved. Otherwise, we can take any non-integer $x_i^*$ for some $1 \leq i \leq k$, and divide the problem into two subproblems

$$\arg\min_{x \in \mathbb{Z}^k \times \mathbb{R}^{n-k}} \{c^t x : Ax \geq b, x_i \leq \lfloor x_i^* \rfloor\}, \qquad \arg\min_{x \in \mathbb{Z}^k \times \mathbb{R}^{n-k}} \{c^t x : Ax \geq b, x_i \geq \lfloor x_i^* \rfloor + 1\}$$

The process then starts again, recursively constructing a tree of subproblems with their associated linear relaxation solutions. The branching stops when subproblems are found unfeasible or when their linear relaxations are integral, in which case they furnish feasible solutions. These solutions can be used to prune parts of the branching tree, whose dual bounds are worse than the best-found solution so far.

Throughout this algorithm, nodes in the branch-and-bound tree, corresponding to subproblems, must be selected for further branching: this is known as the node selection problem. In SCIP, this is implemented through a NODESELECT method that takes as argument the list of open nodes and must choose one for subdivision. In practice, it is expensive to rank open nodes at every node selection step, and solvers maintain a priority list of open nodes throughout solving. Whenever new nodes are created, they are inserted in the priority list according to a node comparison function NODECOMP, which takes two nodes as argument and returns whether the first node, the second node or none are to be preferred. The NODESELECT function can then make use of the ranking; in the simplest strategy, it simply selects the node with the highest rank. More complex node selection strategies are also possible, such as prioritizing the highest-ranked children or sibling of the currently opened node over arbitrary leaves. Although this description uses SCIP terminology, other solvers work similarly.

The current state-of-the-art NODECOMP rule, used by default in most solvers, is best estimate search [99], [100]. In this scheme, every node is associated with an estimate of the increase in objective value resulting from selecting the node, computed from pseudocost statistics. The heuristic then selects the node with the highest estimate. Other popular rules include best-first search [114], which prioritizes nodes with the best dual bound, and depth-first search [115], which prioritizes the deepest node.

As detailed by He et al. [101], the task of designing a good NODECOMP function can be assimilated to finding a good policy in a Markov decision process. In this process, the solver is interpreted as the environment, which calls the NODECOMP($\texttt{node}_1, \texttt{node}_2$) policy whenever it needs two open nodes compared. This policy is provided information about nodes, which can be interpreted as a state $\mathbf{s} = (\text{node}_1, \text{node}_2)$, and then takes an action as to whether to prefer the first node, the second node, or none, $\mathbf{a} \in \{\text{node}_1\text{-better}, \text{node}_2\text{-better}, \text{equal}\}$. This repeated decision making continues until no new nodes need insertion, that is, until the solving is complete. The process is illustrated in Figure 1.

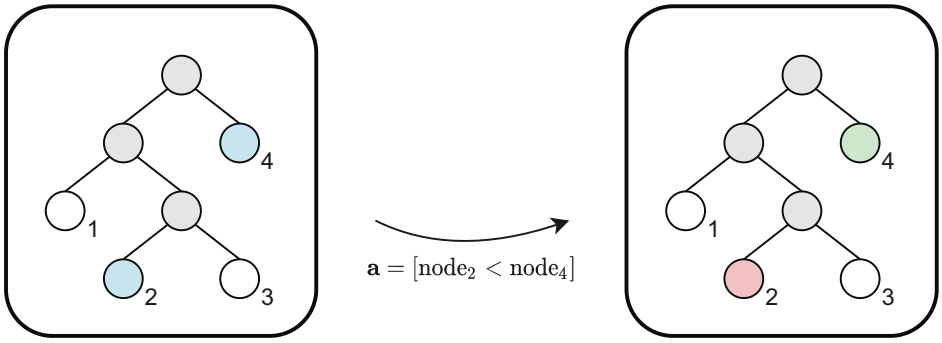


**Fig. 1.** The node comparison problem. Here the solver is asking the NODECOMP function to rank the open nodes 2 and 4, which chose to prioritize the latter over the former.

# 4. Methodology

We now describe our approach to learning good NODECOMP functions. Since the problem can be assimilated to a Markov decision process, we follow previous work [101]–[103] and train by imitation learning to mimic an expert policy.

## 4.1. State representation

Our learned NODECOMP$(\text{node}_1, \text{node}_2)$ takes as input a state $\mathbf{s} = (\text{node}_1, \text{node}_2)$, which represents a pair of nodes. In the related problem of variable selection, important advances were achieved by the usage of bipartite graph representations of the state of the solver, allowing for the first time machine learning to improve over human-designed heuristics in full-fledged solvers [87], [96], [110], [111]. These representations form the current state-of-the-art approach on this problem, and offer many advantages, such as being permutation-invariant in the labeling of the variables and constraints. Inspired by this innovation, we similarly propose to represent each node as a bipartite graph, where on one side there are as many vertices as constraints, and on the other side as many vertices as variables, in the sub-MILP encoded by the node. We draw an edge between a constraint and a variable vertex if the coefficient associated with the variable in the constraint is nonzero. To each constraint vertex $i$ we associate a vector of features, namely its right-hand side $b_i$ and its type ($>$, $<$ or $=$). Similarly, to each variable vertex $j$ we associate a vector of features, namely its objective coefficient $c_j$, upper and lower bounds $u_j$ and $l_j$, and type (binary, integer, or continuous). In addition, we associate the nonzero coefficient of the variable in the constraint to each edge. Finally, an additional global vertex of attributes associated with the whole node is added, unconnected with the rest. To this vertex, we associate two features, namely an estimate of the objective value of the best feasible solution in the subtree of the node and an estimate of the dual bound achieved at the node, through the `SCIPnodeGetEstimate` and

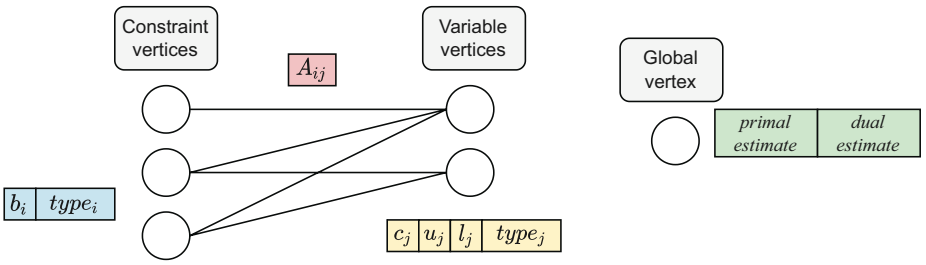


**Fig. 2.** Bipartite graph representation of a node.

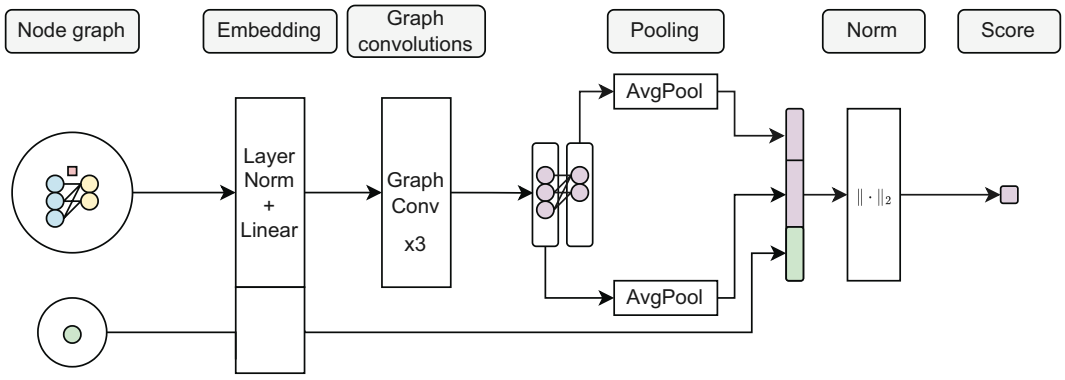


**Fig. 3.** Architecture of the GNN scoring function $g$.

`SCIPnodeGetLowerbound` SCIP functions, respectively. The representation is illustrated in Figure 2.

## 4.2. Model

Our NODECOMP function has the form

$$\text{NODECOMP}(\text{node}_1, \text{node}_2) = \begin{cases} \text{node}_1\text{-better} & \text{if } f(\text{node}_1, \text{node}_2) \leq 0.5, \\ \text{node}_2\text{-better} & \text{if } f(\text{node}_1, \text{node}_2) > 0.5, \end{cases} \tag{4.1}$$

where $f \in [0,1]$ is a classification machine learning model. Our model always prefers one node over the other, and never returns $\mathbf{a}$=equal as an action. The classification model takes the form $f(\text{node}_1, \text{node}_2) = \sigma\big(g(\text{node}_1) - g(\text{node}_2)\big)$ where $\sigma$ stands for the sigmoid function, and $g \in$ is a scoring function with a single dimensional, real-valued output. This siamese architecture [116] is naturally symmetric, in the sense that our model satisfies $f(\text{node}_2, \text{node}_1) = 1 - f(\text{node}_1, \text{node}_2)$.

Just as for the design of the state representation, we take inspiration from the latest line of work on variable selection [87], [96], [110], [111] and use a graph neural network [113] model for our scoring function $g$, with suitable modifications for this node comparison problem. A diagram of the architecture is provided as Figure 3. An important advantage of this model is that, just as in the variable selection setting, the same model can be trained

on problems of varying number of variables and constraints. In detail, the constraint and variable features of the node are first transformed by an 32-dimensional embedding layer and then pass through three graph convolutional layers, with 8, 4 and 4 dimensions each. Each layer uses a ReLU activation function. The representations of the constraint and variable vectors are then pooled by average separately and then concatenated with the global features of the node. Finally, the resulting vector's $\ell_2$ norm is taken, which is outputted as the score.

## 4.3. Training procedure

Our training procedure is similar to the one of He et al. [101]. Just like them, we aim to imitate a "diving oracle" NODECOMP policy, which prioritizes a node if it contains the optimal solution $x^*$, and falls back on another heuristic (we use best estimate search) if this is not the case:

$$\text{ORACLE-NODECOMP}(\text{node}_1, \text{node}_2) = \begin{cases} \text{node}_1\text{-better} & \text{if } x^* \in \text{node}_1, \\ \text{node}_2\text{-better} & \text{if } x^* \in \text{node}_2, \\ \text{ESTIMATE-NODECOMP}(\text{node}_1, \text{node}_2) & \text{otherwise.} \end{cases}$$

Since nodes represent a partition of the feasible space, the optimal solution cannot be in the feasible spaces of both nodes simultaneously, so this is well-defined. As this NODECOMP function uses knowledge of the optimal solution, it cannot be used in practice; however, it can be run on training instances by precomputing optimal solutions, and it is worthwhile to try to imitate its decisions without this additional knowledge. To do this, He et al. use DAGGER, an expensive imitation learning algorithm that aims to diversify the states from which the expert is sampled through several rounds of training. We propose a simpler procedure that achieves a similar result with lower computing requirements.

This procedure runs as follows. We first solve the instances using a solver, collecting their optimal solutions. We then solve the instances again, using a plain highest-priority NODESELECT rule. When the solver calls the NODECOMP function, we query the oracle, and if it chooses $\text{node}_1$-better or $\text{node}_2$-better, we collect state information $\mathbf{s}$ and the resulting decision $\mathbf{a}$ as an expert sample $(\mathbf{s}_i, \mathbf{a}_i)$. Next, crucially, we take the opposite decision than the oracle recommends, making a mistake on purpose. This increases the variety of states explored during the sampling phase and makes the state distribution more aligned with the machine learning policy, which is bound to make mistakes. We follow this procedure until the solving is completed.

As a result of this sampling process, we obtain a dataset of expert samples $\mathcal{D} = \{(\mathbf{s}_i, \mathbf{a}_i)\}$ from which to train our machine learning policy. Since we saved samples when the oracle had a preference, the actions can be interpreted as labels 0 or 1 according to whether the first or second node was preferred. Learning the preference of the oracle then becomes a

simple classification task that can be performed by minimizing a cross-entropy loss over our classifier $f$. Since mistakes coming early on in the sampling process can be exponentially costly, we weight the samples during training using an exponentially decreasing scheme, $w = \exp(1 + |d_1 - d_2|)/\min(d_1, d_2))$, where $d_1, d_2$ are the depths of the first and second nodes, respectively. This is similar to the exponential weighting scheme used by He et al.

# 5. Experimental results

We now present experimental results on three NP-hard problems. We evaluate each machine learning method by running it in SCIP with a simple highest-priority rule that falls back to ESTIMATE after two feasible solutions have been found, as we detail in Section 5.4. We also evaluate the default SCIP node selection rule (that is, with both default NODESELECT and NODECOMP). Code for reproducing these experiments can be found at `hhttps://github.com/ds4dm/learn2comparenodes`.

## 5.1. Benchmarks

We evaluate on three NP-hard instance families that are particularly primal-difficult, that is, for which finding feasible solutions is the main challenge. Those are instances for which improved node comparison is likely to have a particularly broad impact, so differences between methods should be clearer. The first benchmark is composed of Fixed Charge Multicommodity Network Flow (FCMCNF) [117] instances, generated from the code of Chmiela et al. [118]. We train and test on instances with $n = 15$ nodes and $m = 1.5 \cdot n$ commodities, and also evaluate on larger transfer instances with $n = 20$ nodes. The second benchmark is composed of Maximum Satisfiability (MAXSAT) instances, generated following the scheme of Béjar et al. [119]. We train and test on instances with a uniformly sampled number of nodes $n \in [60, 70]$ and transfer on instances with $n \in [80-100]$. Finally, our third benchmark is composed of Generalized Independent Set (GISP) [120] instances, generated from the code of Chmiela et al. [118]. We train and test on instances with a uniformly sampled number of nodes $n \in [60, 70]$ and transfer on instances with $n \in [70 - 80]$. All these families require an underlying graph: we use in each case Erdős-Rényi random graphs with the prescribed number of nodes, with edge probability $p = 0.3$ for FCMCNF and $p = 0.6$ for MAXSAT and GISP.

## 5.2. Baselines

We compare against the state-of-the-art best estimate node comparison rule [99], [100]. This is the NODECOMP function used by default in SCIP, in conjunction with a diving NODESELECT rule that prioritizes children and siblings of the currently focused node. To disentangle the effect of this NODESELECT rule, we report both results with this rule (default

**Table 1.** Test accuracies of the different machine learning methods in imitating the diving oracle.

| | Test FCMCNF | Test MAXSAT | Test GISP |
|---|---|---|---|
| SVM | 91.5% | 90.6% | 93.0% |
| MLP | **97.8**% | **97.9**% | 95.6% |
| GNN | 95.7% | 97.7% | **97.0**% |

SCIP) and with a plain NODESELECT that always selects the highest-ranked node (ESTIMATE). We also report the performance of the expert we aim to imitate, the diving oracle (ORACLE). This method cheats by having access to the optimal solution ahead of the solving.

In addition, we compare against two competing machine learning approaches: the support vector machine [121] approach of He et al. [101] (SVM) and the RankNet feedforward neural network [122] approach of Song et al. [102] and Yilmaz and Yorke-Smith [103]. The former uses a multilayer perceptron; the latter uses the same, except for one benchmark where they use three hidden layers. For simplicity, we use a multilayer perceptron for all benchmarks (MLP), with a hidden layer of 32 neurons. The features used in the three papers are roughly similar; again, for simplicity, we use the fixed-dimensional features of He et al. for both the SVM and the MLP. All methods except the default SCIP use a plain highest-rank NODESELECT.

## 5.3. Training

We use the training procedure of Section 4.3 for all machine learning models. The SVM model is trained using the scikit-learn [123] library; the MLP and the GNN implemented in PyTorch [124] and optimized using Adam [67] with training batch size of 16. Running the sampling procedure on 1000 training and 100 test instances yielded 16285 training and 3019 test samples for FCMCNF, 41299 training, and 4868 test samples for MAXSAT, and 41299 training and 4868 test samples for GISP. We train/evaluate using an Nvidia® Tesla V100 GPU and an Intel® Xeon Gold 6126 CPU. Test accuracies of the different models can be found in Table 1.

## 5.4. Evaluation

We evaluate each machine learning model by setting SCIP's NODECOMP function be the NODECOMP function associated with the machine learning model (Eq. 4.1). In parallel, the NODESELECT function is designed to use the highest ranked node according to this node comparison function, until two feasible solutions have been obtained. After this state, it falls

**Table 2.** Evaluation of node comparison methods in terms of the 1-shifted geometric mean of the number of nodes and solving time (in seconds) over the instances, with the geometric standard deviation. For each problem, machine learning models are trained on instances of the same size as the test instances, and evaluated on those and the larger transfer instances (50 instances each).

| | Test FCMCNF | | Transfer FCMCNF | | Test MAXSAT | | Transfer MAXSAT | | Test GISP | | Transfer GISP | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Nodes | Time | Nodes | Time | Nodes | Time | Nodes | Time | Nodes | Time | Nodes | Time |
| ORACLE | 15±4 | 3.80±1.5 | 75±4 | 19.9±1.8 | 102±2 | 6.17±1.8 | 160±2 | 8.9±1.5 | 98±3 | 4.18±1.3 | 1062±2 | 22.6±1.5 |
| SCIP | 41±5 | 4.64±1.5 | 178±4 | 26.7±1.9 | 147±2 | 9.26±1.5 | 171±2 | 12.9±1.4 | 184±2 | **4.38**±1.2 | 1533±2 | **19.1**±1.5 |
| ESTIMATE | 21±5 | **4.09**±1.5 | 122±5 | **23.8**±2.0 | 177±2 | 8.16±1.7 | 247±2 | 12.1±1.6 | 218±2 | 4.64±1.3 | 1435±2 | 24.9±1.7 |
| SVM | 20±5 | 4.10±1.5 | 133±5 | 24.8±1.9 | 150±3 | 7.34±1.8 | 225±2 | 10.7±1.6 | 207±3 | 4.57±1.3 | 1295±2 | 23.4±1.6 |
| MLP | 21±5 | 4.15±1.5 | **115**±5 | 24.1±1.9 | 157±3 | 7.76±1.9 | 215±2 | 10.8±1.6 | 209±3 | 4.72±1.3 | 1238±2 | 23.0±1.6 |
| GNN | **19**±5 | 4.14±1.5 | 122±5 | 24.5±1.9 | **117**±3 | **6.66**±1.9 | **171**±2 | **9.1**±1.6 | **170**±3 | 4.64±1.3 | **1203**±2 | 22.8±1.5 |

back to ESTIMATE. This has the effect of prioritizing the learned node comparison during the initial phases of the solving, in a size-independent manner.

We average results over the benchmarks using the 1-shifted geometric mean with geometric standard deviation to measure the average and dispersion of B&B tree size and solving time on our benchmarks. This metric is the standard used in the mixed-integer programming community since it reduces outlier effects from both directions (too easy and too hard instances), as discussed in Appendix A3 of Achterberg [90]. We evaluated on 50 test and 50 transfer instances, as explained in Section 5.1. Table 2 summarizes the results.

## 5.5. Discussion

As can be seen in Table 1, both the MLP and GNN achieve similar accuracies on the datasets, with the SVM lagging a bit more behind. When used in solving, however, the GNN more consistently dominates the other machine learning approaches. More impressively, the model is often competitive with or even better than the SCIP default node strategy, particularly on the MAXSAT problems. In addition, these results generalize to larger instances than those trained on. This is the case despite using a plain NODESELECT rule, which suggests that most of the difficulty in node selection can be reduced by the design of a good NODECOMP function. This is particularly attractive as this is a problem that is naturally amenable to machine learning methods, as described in this work.

A disadvantage of the imitation learning approach we follow is that it is limited by the performance of the expert itself. If the oracle does not beat a baseline, imitating it is unlikely to bring gains. A good example is the largest benchmark, transfer GISP: this is the only family where the oracle does not beat the SCIP default rule in time. Therefore, it is unsurprising that no other machine learning method was able to beat it. Note that

nonetheless, the GNN is the model that manages to come the closest to the performance of the oracle on this benchmark, suggesting strong imitation capabilities.

# 6. Conclusion

This work proposes to train a graph neural network to compare nodes in a branch-and-bound solver for solving mixed-integer linear programs. We represent nodes as bipartite graphs with features and train a neural network to imitate a diving oracle that plunges towards the optimal solution. On three primal-difficult NP-hard benchmarks, our approach outperforms prior machine learning approaches and often even the SCIP default node selection strategy, while generalizing to larger instances than trained on.

An interesting direction for future work would be to combine variable and node selection strategies. Besides the fact that the two problems are tightly linked, good node selection is particularly important in primal-difficult problems, while good branching is particularly useful in dual-difficult problems. Combining the two could thus help outperform current expert-designed strategies on generic problems.

# Conclusion and future work

In this thesis, we first provided two broad concise overviews of operations research and machine learning while deepening more critical notions for this work. Then, a literature review of *data-driven combinatorial optimization* was given, with an extensive study of research in *data-driven node selection in B&B*. Finaly, the article titled *Learning to Compare Nodes in Branch and Bound using Graph Neural Networks* followed, providing our novel contribution to research.

This contribution consists of a new framework for obtaining a problem-adaptive B&B node comparison policy. A simple training regime based on data diversification is proposed, using graph neural networks to directly learn to infer from geometric characterizations of MILPs, instead of solver's pseudostatistics. It is shown that the approach outperforms previous data-driven ones in three NP-hard problems, suggesting its strong imitation potential can be leveraged to learn more complex policies that would transfer even more appropriately on out-of-distribution (typically larger) instances.

The next step for the node comparison problem could be to design an expert for both the diving and the pruning phase in exact solvers. The problem of switching to a handcrafted node selection policy after finding an optimum/almost-optimum solution would then be eliminated. Another direction could be to jointly learn a policy for branching and for searching, which could at the same time boost performance and enable scalability on a larger spectrum of NP-hard problems. In general, we hope to continue pushing for a deeper integration of machine learning methods within combinatorial optimization, moving towards what could be called one day an adaptive combinatorial optimization solver.

# Bibliography


[1] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] J. D. Kececioglu, H.-P. Lenhof, K. Mehlhorn, P. Mutzel, K. Reinert, and M. Vingron, "A polyhedral approach to sequence alignment problems," *Discrete Applied Mathematics*, vol. 104, no. 1, pp. 143–186, 2000.

[3] Y. Pochet and L. A. Wolsey, *Production Planning by Mixed Integer Programming*, 1st. Springer Publishing Company, Incorporated, 2010.

[4] N. Mouha, Q. Wang, D. Gu, and B. Preneel, "Differential and linear cryptanalysis using mixed-integer linear programming," in *Inscrypt*, 2011.

[5] Y. Wei, D. Rideout, and T. Hall, "Toward efficient management of large fires: A mixed integer programming model and two iterative approaches," *Forest Science*, vol. 57, pp. 435–447, Oct. 2011.

[6] K. Pang, Y.-W. Wan, W. Choi, *et al.*, "Combinatorial therapy discovery using mixed integer linear programming," *Bioinformatics (Oxford, England)*, vol. 30, Jan. 2014.

[7] Routing, M. Fischetti, John-Josef, Borchersen, and A. Bech, "A mixed-integer linear programming approach to wind farm layout and inter-array cable," 2015.

[8] H. L. Beyer, Y. Dujardin, M. E. Watts, and H. P. Possingham, "Solving conservation planning problems with integer linear programming," *Ecological Modelling*, vol. 328, pp. 14–22, 2016.

[9] G. Dantzig, *Linear programming and extensions*, ser. Rand Corporation Research Study. Princeton, NJ: Princeton Univ. Press, 1963, XVI, 625.

[10] N. Karmarkar, "A new polynomial-time algorithm for linear programming-ii," *Combinatorica*, vol. 4, pp. 373–395, Dec. 1984.

[11] R. Gomory, "Outline of an algorithm for integer solutions to linear programs," *Bulletin of the American Mathematical Society*, vol. 64, pp. 275–278, Sep. 1958.

[12] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[14] J. H. Holland, "Genetic algorithms," *Scientific American*, vol. 267, no. 1, pp. 66–73, 1992.

[15] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers & Operations Research*, vol. 13, no. 5, pp. 533–549, 1986, Applications of Integer Programming.

[16] J. P. Walser, *Integer Optimization by Local Search: A Domain-Independent Approach*. Berlin, Heidelberg: Springer-Verlag, 1999.

[17] D. Avis, D. Bremner, and R. Seidel, "How good are convex hull algorithms?" *Computational Geometry*, vol. 7, no. 5, pp. 265–301, 1997, 11th ACM Symposium on Computational Geometry, ISSN: 0925-7721. DOI: https://doi.org/10.1016/S0925-7721(96)00023-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925772196000235.

[18] G. Gamrath, D. Anderson, K. Bestuzheva, *et al.*, "The SCIP Optimization Suite 7.0," Zuse Institute Berlin, ZIB-Report 20-10, Mar. 2020.

[19] Gurobi Optimization LLC, *Gurobi Optimizer Reference Manual*, 2020.

[20] I. I. Cplex, "V12. 1: User's manual for cplex," *International Business Machines Corporation*, vol. 46, no. 53, p. 157, 2009.

[21] A. Lodi, "The heuristic (dark) side of mip solvers," in *Hybrid Metaheuristics*, 2013.

[22] "A new combinatorial branch-and-bound algorithm for the knapsack problem with conflicts," *European Journal of Operational Research*, vol. 289, no. 2, pp. 435–455, 2021.

[23] "Exact algorithms for maximum independent set," *Information and Computation*, vol. 255, pp. 126–146, 2017.

[24] K. Salimifard and S. Bigharaz, "The multicommodity network flow problem: State of the art classification, applications, and solution methods," *Operational Research*, vol. 22, Mar. 2022.

[25] J. Ryu and S. Park, "A branch-and-price algorithm for the robust single-source capacitated facility location problem under demand uncertainty," *EURO Journal on Transportation and Logistics*, vol. 11, p. 100 069, 2022.

[26] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, no. 2, pp. 405–421, 2021.

[27] T. M. Mitchell, *Machine learning, International Edition*, ser. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[28] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[29] L. Deng and X. Li, "Machine learning paradigms for speech recognition: An overview," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 21, pp. 1060–1089, 2013.

[30] J. Kober, J. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, pp. 1238–1274, Sep. 2013.

[31] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 229–256, May 1992.

[32] C. Szepesvari, *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers, 2010.

[33] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, and J. Peters, "An algorithmic perspective on imitation learning," *Foundations and Trends in Robotics*, vol. 7, no. 1-2, pp. 1–179, 2018.

[34] D. A. Pomerleau, "Efficient training of artificial neural networks for autonomous navigation," *Neural computation*, vol. 3, no. 1, pp. 88–97, 1991.

[35] D. A. Pomerleau, "Alvinn: An autonomous land vehicle in a neural network," in *Proceedings of (NeurIPS) Neural Information Processing Systems*, D. Touretzky, Ed., Morgan Kaufmann, Dec. 1989, pp. 305–313.

[36] F. Torabi, G. Warnell, and P. Stone, "Behavioral cloning from observation," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI'18, Stockholm, Sweden: AAAI Press, 2018, pp. 4950–4957, ISBN: 9780999241127.

[37] D. Silver, A. Huang, C. J. Maddison, *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, jan 2016, ISSN: 0028-0836. DOI: 10.1038/nature16961.

[38] A. Kanervisto, J. Pussinen, and V. Hautamäki, *Benchmarking end-to-end behavioural cloning on video games*, 2020. DOI: 10.48550/ARXIV.2004.00981. [Online]. Available: https://arxiv.org/abs/2004.00981.

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

[40] 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*, JMLR Workshop and Conference Proceedings, 2011, pp. 627–635.

[41] D. A. Roberts, S. Yaida, and B. Hanin, *The Principles of Deep Learning Theory*. Cambridge University Press, 2022.

[42] C. Gambella, B. Ghaddar, and J. Naoum-Sawaya, "Optimization problems for machine learning: A survey," *European Journal of Operational Research*, vol. 290, no. 3, pp. 807–828, May 2021.

[43] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[44] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–44, May 2015.

[45] M. Malik, M. Malik, K. Mehmood, and I. Makhdoom, "Automatic speech recognition: A survey," *Multimedia Tools and Applications*, vol. 80, pp. 1–47, Mar. 2021.

[46] R. Dabre, C. Chu, and A. Kunchukuttan, "A survey of multilingual neural machine translation," *ACM Comput. Surv.*, vol. 53, no. 5, Sep. 2020.

[47] M. Birjali, M. Kasri, and A. Beni-Hssane, "A comprehensive survey on sentiment analysis: Approaches, challenges and trends," *Knowledge-Based Systems*, vol. 226, p. 107 134, 2021.

[48] C. Dong, Y. Li, H. Gong, *et al.*, *A survey of natural language generation*, Dec. 2021.

[49] L. Jiao, F. Zhang, F. Liu, *et al.*, "A survey of deep learning-based object detection," *IEEE Access*, vol. 7, 2019.

[50] S. Minaee, P. Luo, Z. Lin, and K. Bowyer, *Going deeper into face detection: A survey*, 2021.

[51] J. Su, B. Xu, and H. Yin, "A survey of deep learning approaches to image restoration," *Neurocomputing*, vol. 487, pp. 46–65, 2022.

[52] G. Ciaparrone, F. L. Sánchez, S. Tabik, L. Troiano, R. Tagliaferri, and F. Herrera, "Deep learning in video multi-object tracking: A survey," *Neurocomputing*, vol. 381, pp. 61–88, Mar. 2020.

[53] G. Palm, "Warren mcculloch and walter pitts: A logical calculus of the ideas immanent in nervous activity," in *Brain Theory*, G. Palm and A. Aertsen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 229–230.

[54] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, ser. Cornell Aeronautical Laboratory. Report no. VG-1196-G-8. Spartan Books, 1962.

[55] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[56] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997.

[57] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14, Montreal, Canada: MIT Press, 2014, pp. 3104–3112.

[58] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," in *Shape, Contour and Grouping in Computer Vision*, 1999.

[59] C. Morris, M. Ritzert, M. Fey, *et al.*, "Weisfeiler and leman go neural: Higher-order graph neural networks," in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'19/IAAI'19/EAAI'19, Honolulu, Hawaii, USA: AAAI Press, 2019.

[60] M. L. Smith, L. N. Smith, and M. F. Hansen, "The quiet revolution in machine vision - a state-of-the-art survey paper, including historical review, perspectives, and future directions," *Computers in Industry*, vol. 130, p. 103 472, 2021.

[61] P. Ma, C. Li, M. M. Rahaman, *et al.*, *A state-of-the-art survey of object detection techniques in microorganism image analysis: From classical methods to deep learning approaches*, 2021.

[62] A. Elngar, M. Arafa, A. Fathy, *et al.*, "Image classification based on cnn: A survey," *Journal of Cybersecurity and Information Management*, PP. 18–50, Jan. 2021.

[63] W. Yin, K. Kann, M. Yu, and H. Schütze, *Comparative study of cnn and rnn for natural language processing*, 2017.

[64] W. Wang and J. Gang, "Application of convolutional neural network in natural language processing," in *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE)*, 2018, pp. 64–70.

[65] J. Zhou, G. Cui, S. Hu, *et al.*, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[66] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković, *Combinatorial optimization and reasoning with graph neural networks*, 2021.

[67] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014.

[68] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[69] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul. 2011.

[70] O. Vinyals, M. Fortunato, and N. Jaitly, *Pointer networks*, 2015.

[71] N. Christofides, "Worst-case analysis of a new heuristic for the traveling salesman problem," *Carnegie Mellon University*, vol. 3, p. 10, Mar. 2022.

[72] I. Bello, H. Pham, Q. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," Nov. 2016.

[73] E. B. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, *Learning combinatorial optimization algorithms over graphs*, 2017.

[74] H. Dai, B. Dai, and L. Song, *Discriminative embeddings of latent variable models for structured data*, 2016.

[75] M. Riedmiller, "Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method," in *Proceedings of the 16th European Conference on Machine Learning*, ser. ECML'05, Porto, Portugal: Springer-Verlag, 2005, pp. 317–328.

[76] Z. Li, Q. Chen, and V. Koltun, *Combinatorial optimization with graph convolutional networks and guided tree search*, 2018.

[77] R. Thapa, "A survey on deep learning-based methodologies for solving combinatorial optimization problems," Aug. 2020.

[78] R. Baltean-Lugojan, P. Bonami, R. Misener, and A. Tramontani, "Selecting cutting planes for quadratic semidefinite outer-approximation via trained neural networks," 2018.

[79] Y. Tang, S. Agrawal, and Y. Faenza, "Reinforcement learning for integer programming: Learning to cut," in *Proceedings of the 37th International Conference on Machine Learning*, H. D. III and A. Singh, Eds., ser. Proceedings of Machine Learning Research, vol. 119, PMLR, Jul. 2020, pp. 9367–9376.

[80] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, *Attention is all you need*, 2017.

[81] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," *arXiv*, Mar. 2017.

[82] U. H. S. Franz Wesselmann, "Implementing cutting plane management and selection techniques," University of Paderborn, Tech. Rep., Dec. 2012.

[83] Z. Huang, K. Wang, F. Liu, *et al.*, *Learning to select cuts for efficient mixed-integer programming*, 2021.

[84] "Solving the multiple instance problem with axis-parallel rectangles," *Artificial Intelligence*, vol. 89, no. 1, pp. 31–71, 1997.

[85] M. B. Paulus, G. Zarpellon, A. Krause, L. Charlin, and C. J. Maddison, *Learning to cut by looking ahead: Cutting plane selection via imitation learning*, 2022.

[86] D. L. Applegate, R. E. Bixby, V. Chvatál, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[87] M. Gasse, D. Chetelat, N. Ferroni, L. Charlin, and A. Lodi, "Exact combinatorial optimization with graph convolutional neural networks," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 15 580–15 592.

[88] A. Lodi and G. Zarpellon, "On learning and branching: A survey," *TOP*, vol. 25, pp. 1–30, Jun. 2017.

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

[90] T. Achterberg, "Constraint integer programming," Ph.D. dissertation, ZIB, Berlin, 2007.

[91] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, Apr. 2006.

[92] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, Feb. 2016.

[93] T. Joachims, "Optimizing search engines using clickthrough data," ser. KDD '02, Edmonton, Alberta, Canada: Association for Computing Machinery, 2002, pp. 133–142.

[94] C. Hansknecht, I. Joormann, and S. Stiller, *Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem*, 2018.

[95] C. Burges, "From ranknet to lambdarank to lambdamart: An overview," *Learning*, vol. 11, Jan. 2010.

[96] G. Zarpellon, J. Jo, A. Lodi, and Y. Bengio, "Parameterizing branch-and-bound search trees to learn branching policies," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 3931–3939.

[97] T. Achterberg, "Scip: Solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, pp. 1–41, Jul. 2009.

[98] R. Ebendt, W. Gunther, and R. Drechsler, "Combining ordered best-first search with branch and bound for exact bdd minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1515–1529, 2005.

[99] M. Bénichou, J.-M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent, "Experiments in mixed-integer linear programming," *Mathematical Programming*, vol. 1, no. 1, pp. 76–94, 1971.

[100] J. Forrest, J. Hirst, and J. A. Tomlin, "Practical solution of large mixed integer programming problems with umpire," *Management Science*, vol. 20, no. 5, pp. 736–773, 1974.

[101] H. He, H. Daume III, and J. M. Eisner, "Learning to search in branch and bound algorithms," *Advances in neural information processing systems*, vol. 27, 2014.

[102] J. Song, R. Lanka, A. Zhao, A. Bhatnagar, Y. Yue, and M. Ono, "Learning to search via retrospective imitation," *arXiv*, 2018.

[103] K. Yilmaz and N. Yorke-Smith, "A study of learning search approximation in mixed integer branch and bound: Node selection in scip," *Ai*, vol. 2, no. 2, pp. 150–178, 2021.

[104] C. Bayliss, G. De Maere, J. A. Atkin, and M. Paelinck, "A simulation scenario based mixed integer programming approach to airline reserve crew scheduling under uncertainty," *Annals of Operations Research*, vol. 252, no. 2, pp. 335–363, 2017.

[105] M. Lombardi, M. Milano, and A. Bartolini, "Empirical decision model learning," *Artificial Intelligence*, vol. 244, pp. 343–367, 2017.

[106] J. Abrache, T. G. Crainic, M. Gendreau, and M. Rekik, "Combinatorial auctions," *Annals of Operations Research*, vol. 153, no. 1, pp. 131–164, 2007.

[107] C. A. Floudas and X. Lin, "Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications," *Annals of Operations Research*, vol. 139, no. 1, pp. 131–162, 2005.

[108] A. H. Land and A. G. Doig, "An automatic method for solving discrete programming problems," in *50 Years of Integer Programming 1958-2008*, Springer, 2010, pp. 105–132.

[109] T. Achterberg, T. Berthold, S. Heinz, T. Koch, and K. Wolter, "Constraint integer programming: Techniques and applications," 2008.

[110] P. Gupta, M. Gasse, E. Khalil, P. Mudigonda, A. Lodi, and Y. Bengio, "Hybrid models for learning to branch," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, 2020, pp. 18 087–18 097.

[111] V. Nair, S. Bartunov, F. Gimeno, *et al.*, "Solving mixed integer programs using neural networks," 2020.

[112] M. Etheve, Z. Alès, C. Bissuel, O. Juan, and S. Kedad-Sidhoum, "Reinforcement learning for variable selection in a branch and bound algorithm," in *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Springer, 2020, pp. 176–185.

[113] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE international joint conference on neural networks*, vol. 2, 2005, pp. 729–734.

[114] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[115] R. J. Dakin, "A tree-search algorithm for mixed integer programming problems," *The Computer Journal*, vol. 8, no. 3, pp. 250–255, 1965.

[116] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," *Advances in Neural Information Processing Systems*, vol. 6, 1993.

[117] M. Hewitt, G. Nemhauser, and M. Savelsbergh, "Combining exact and heuristic approaches for the capacitated fixed-charge network flow problem," *INFORMS Journal on Computing*, vol. 22, pp. 314–325, May 2010.

[118] A. Chmiela, E. Khalil, A. Gleixner, A. Lodi, and S. Pokutta, "Learning to schedule heuristics in branch and bound," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[119] R. Béjar, A. Cabiscol, F. Manyà, and J. Planes, "Generating hard instances for maxsat," in *2009 39th International Symposium on Multiple-Valued Logic*, 2009, pp. 191–195.

[120] M. Colombi, R. Mansini, and M. Savelsbergh, "The generalized independent set problem: Polyhedral analysis and solution approaches," *European Journal of Operational Research*, vol. 260, no. 1, pp. 41–55, 2017.

[121] V. Vapnik, *The Nature of Statistical Learning Theory*. Springer science & business media, 1999.

[122] C. Burges, T. Shaked, E. Renshaw, *et al.*, "Learning to rank using gradient descent," in *Proceedings of the 22nd International Conference on Machine learning*, 2005, pp. 89–96.

[123] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, 2011.

[124] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.