SCIENCE CHINA
Mathematics

AI Methods for Optimization Problems
• ARTICLES •

June 2024 Vol. 67 No. 6: 1191–1262
https://doi.org/10.1007/s11425-023-2293-3

Learning to optimize: A tutorial for continuous and mixed-integer optimization

Xiaohan Chen, Jialin Liu & Wotao Yin*

Decision Intelligence Lab, Alibaba DAMO Academy, Bellevue, WA 98004, USA
Email: xiaohan.chen@alibaba-inc.com, jialin.liu@alibaba-inc.com, wotao.yin@alibaba-inc.com

Received August 18, 2023; accepted April 10, 2024; published online May 8, 2024

Abstract Learning to optimize (L2O) stands at the intersection of traditional optimization and machine learning, utilizing the capabilities of machine learning to enhance conventional optimization techniques. As real-world optimization problems frequently share common structures, L2O provides a tool to exploit these structures for better or faster solutions. This tutorial dives deep into L2O techniques, introducing how to accelerate optimization algorithms, promptly estimate the solutions, or even reshape the optimization problem itself, making it more adaptive to real-world applications. By considering the prerequisites for successful applications of L2O and the structure of the optimization problems at hand, this tutorial provides a comprehensive guide for practitioners and researchers alike.

Keywords AI for mathematics (AI4Math), learning to optimize, algorithm unrolling, plug-and-play methods, differentiable programming, machine learning for combinatorial optimization (ML4CO)

MSC(2020) 68T07, 90C11, 90C25, 90C59

Citation: Chen X H, Liu J L, Yin W T. Learning to optimize: A tutorial for continuous and mixed-integer optimization. Sci China Math, 2024, 67: 1191–1262, https://doi.org/10.1007/s11425-023-2293-3

1 Introduction

What is learning to optimize (L2O)? Learning to optimize (L2O) is an emerging area where traditional optimization techniques are enhanced or even replaced by machine learning models learned from data. Instead of solely relying on hand-crafted algorithms, L2O employs data to derive optimization strategies using data-driven approaches. The motivation stems from the realization that there are patterns in the way optimization problems are solved, and these patterns can be learned and generalized.

Why bringing learning to optimization? A short answer is that machine learning provides a strong alternative to understand and *utilize* the special structures of data for faster and better optimization, which can be difficult if done analytically. It can be conceptualized as two ends of the same stick:

• On one end, consider a set that contains only one optimization problem, and it is the only single optimization problem that we are interested in. Then the most efficient method is to solve it with a solver and memorize the solution for retrieval when needed.

* Corresponding author

• On the other end, when we are interested in solving all instances in an optimization problem class, classic algorithms with *worst-case* guarantee are the best choices. We simply select the algorithm with the best guarantee.

In real-world scenarios, however, the situation lies between these two ends. In a specific application, the optimization problems of interests often are not arbitrary ones. Instead, they share some *common structures.* Effectively exploiting these special structures can help with accelerating the optimization process or generating better solutions. Sometimes, these common structures can be described in an accurate mathematical language and used for deriving better optimization algorithms.

But in other cases, such structures are too vague to characterize explicitly; hence it is hard to utilize them analytically to improve optimization. Machine learning, especially deep learning, is adept at capturing hidden structures in data by learning from past experience. This capability is endowed by the great capacity of learning-based models and the data-driven learning process. Therefore, when combined properly with classic optimization, learning can help to achieve acceleration or superior performance.

When to consider L2O? The first prerequisite for the success of all learning-based methods, including L2O, is the access to abundant past experience, i.e., in the context of optimization, *many examples* of optimization problems and, often, also their solutions. According to the famous *No Free Lunch* (NFL) theorem [168], we cannot expect an L2O method to work superbly on all optimization problems. Therefore, the example problems must stand as a good representation of the problems to come in the future, or, more formally, are drawn from the same *task distribution.*

There are two scenarios where L2O can improve optimization:

Scenario 1: When we need to solve similar optimization problems repeatedly. In this scenario, the task distribution is concentrated. Because it is challenging to describe the distribution explicitly in a math language, L2O is used to find solution "shortcuts".

Scenario 2: When it is hard to formulate a optimization model. A good example is the term "natural image", whose accurate mathematical characterization is eagerly needed but very difficult to write as a clean formula. In this case, we can use L2O to find better solutions by learning an optimization model or method.

Train offline and then deploy. Unlike traditional optimization techniques, L2O methods typically involve two phases: training and deployment. Training entails refining optimization algorithms or problem structures using data. While this training process might be time-consuming, its outcome is advantageous. After training, the derived optimization algorithms or problem frameworks tend to outperform their conventional counterparts. They either offer significantly quicker results or yield solutions of superior quality, especially when tailored to real-world scenarios.

Organization. Within this tutorial, we categorize L2O into three paradigms, each reflecting a different level of integration between optimization and machine learning:

• Learning to accelerate optimization processes. In this approach, traditional optimization algorithms or solvers are still employed, but certain procedures are replaced with machine learning models to accelerate traditional algorithms, making this a more conservative application of machine learning to the optimization process. Example: Learning for mixed-integer programming in Section 6.

• Learning to generate optimization solutions. This paradigm involves using a machine learning model to directly generate solutions to optimization problems. The aim is to produce solutions far more quickly than traditional algorithms, while maintaining acceptable accuracy. Example: Algorithm unrolling in Section 3.

• Learning to adapt optimization. In this innovative paradigm, the optimization problem itself can even be altered. Recognizing that solving optimization problems is often an approximation of the ideal solution in many real-world scenarios, machine learning models may be employed to tune these problems, making them closer to the desired real-world outcomes. Example: Plug-and-play methods in Section 4 and optimization as a layer in Section 5.

The rest of this paper is organized as follows. First in Section 2, we introduce some preliminaries on machine learning and deep neural networks. Following this, we delve into L2O techniques for continuous optimization in Sections 3, 4, and 5. The order of presentation is determined by its dependency on conventional optimization techniques. Methods in Section 3 aim at quickly finding solutions to certain optimization problems. Sections 4 and 5 are directed at producing solutions that better align with real-world needs, even if these solutions do not solve the original optimization problems. Section 6 turns to mixed-integer programs. Given its distinct methodologies and relative independence from the other sections, we have positioned it after the other sections.

Remarks on notation. The optimization and deep learning communities use two systems of notation that can conflict with each other and cause confusions in some cases. The notation can sometimes differ between different directions of L2O. For example, a vector $\boldsymbol{x}$ is usually used to denote the optimization variable and thus the iterate in an iterative algorithm in the field of optimization. However, in the context of deep learning, $\boldsymbol{x}$ mostly denotes the input to a neural network or to a layer of a network. In this work, we do not strive at creating a new system of notation that is consistent for both fields as it would read unnatural to the audience from either community. Instead, we try to follow the commonly-accepted notations by each field when there is no risk of confusion. When there are potential conflicts, however, we will make it crystal clear the distinction between our notations and the usual ones. We humbly ask caution from the reader for a better and easier understanding of the content of this article.

2 Introduction and deep neural networks

This section commences by providing an introductory overview of deep neural networks and how to train them, along with various adaptations and methodologies that are broadly utilized by the L2O community. We also examine prominent reinforcement learning methods that are broadly applicable in L2O. Our goal is to elucidate the fundamental concepts, advantages, and constraints of those common techniques.

2.1 Preliminaries of machine learning

From a mathematical perspective, many machine learning tasks are about creating a mapping derived from data. Consider the task of image classification. Here, the aim is to establish a mapping that links an image to a category, such as animals or plants. We work with a data set denoted as $\mathbb{D} = \{(\boldsymbol{x}_j, y_j)\}_{j=1}^n$, where $\boldsymbol{x}_j$ from $\mathbb{X}$ denotes an image (with $\mathbb{X}$ being the image space) and $y_j \in \{0, 1\}$ signifies its associated category. Here, $y_j = 0$ indicates that $\boldsymbol{x}_j$ is an animal image, while $y_j = 1$ indicates it is a plant. These images can come from varied sources like photography. Their corresponding labels, y_j usually added by humans, are seen as the definitive answers to the image's category. Machine learning methods typically assume y_j to be the accurate label for $\boldsymbol{x}_j$ and design a mapping that links $\boldsymbol{x}_j$ to y_j for all $j = 1, 2, \ldots, n$. Specifically:

- According to the format and properties of $\boldsymbol{x}$, we select a parameterized mapping $g(\cdot; \boldsymbol{\theta}) : \mathbb{X} \to \{0, 1\}$, where $\boldsymbol{\theta}$ involves all the parameters.
- We look for desirable parameters $\boldsymbol{\theta}^*$ such that $g(\boldsymbol{x}_j; \boldsymbol{\theta}^*) \approx y_j$ for all $j = 1, 2, \ldots, n$.
- With $g(\cdot; \boldsymbol{\theta}^*)$, we can predict the category of a new image $\boldsymbol{x}'$ not in $\mathbb{D}$ with $\hat{y} = g(\boldsymbol{x}'; \boldsymbol{\theta}^*)$.

We clarify the following terminologies:

- The parameterized mapping $g(\cdot; \boldsymbol{\theta})$ is termed a *machine learning model* or, more simply, a *model*.
- The process of finding desirable parameters $\boldsymbol{\theta}^*$ is called *training*. A model equipped with these optimal parameters, $g(\cdot; \boldsymbol{\theta}^*)$, is named a *trained model*.
- Utilizing the trained model to predict the category of a new image is referred to as *inference*.
- The data set $\mathbb{D}$ used during the training phase is termed the *training set*.
- A collection of images used during the inference phase is called a *testing set*.

This methodology is identified as *supervised learning* since each $\boldsymbol{x}_j$ in the training set is paired with a ground truth y_j by an expert, resembling the expert supervising the training phase. Such a supervised learning pipeline can be adapted for other tasks, where $\boldsymbol{x}$ may represent entities other than images, and y might symbolize more than just image categories. Subsequent sections will delve into broader scenarios.

2.2 Deep neural networks

We start by discussing the construction of a machine-learning model, with a focus on neural networks. Neural networks are made of units termed as *layers*, where each layer is a basic function. Similar to function composition, a layer of neural network can also consist of several (sub-)functions. The simplest neural network is a *single-layer perceptron*, a linear binary classifier that makes binary prediction by applying an affine transformation to its input $\boldsymbol{x} \in \mathbb{R}^d$ and then thresholding the output, denoted as $g(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}, \boldsymbol{x} \mapsto h = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$, where h is the binary output of the network and $\sigma : \mathbb{R} \to \{0, 1\}$ is a thresholding function, or *activation function*, yielding 1 if an input is positive or 0 otherwise. Here, the weight $\boldsymbol{w} \in \mathbb{R}$ and bias $b \in \mathbb{R}$ are the *parameters* of the single-layer perceptron; or we say the perceptron is *parameterized* by $\boldsymbol{\theta} = (\boldsymbol{w}, b)$, which can be learned from data. To make it self-evident to distinguish between network input and learnable parameters, we adopt the following notation to represent a single-layer perceptron:

$$g(\boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b). \tag{2.1}$$

A single-layer perceptron can be easily extended to have multi-dimensional outputs by introducing a weight matrix $\boldsymbol{W} \in \mathbb{R}^{d \times d'}$ and a bias vector $\boldsymbol{b} \in \mathbb{R}^{d'}$ with d' being the output dimension. The extended mapping is defined as $\boldsymbol{x} \mapsto \boldsymbol{h} = \sigma(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$. When viewed graphically, each coordinate of either $\boldsymbol{x}$ or $\boldsymbol{h}$ can be thought of as a node. Given a general matrix $\mathbf{W}$, every coordinate of $\boldsymbol{x}$ is linked to every coordinate of $\boldsymbol{h}$. Therefore, such a mapping is usually called a *fully-connected* (FC) layer. Note that the thresholding function in (2.1) is not the sole option for the activation function σ in a fully-connected layer. Below are additional activation functions to consider:

- ReLU σ: Defined as $\boldsymbol{x} \mapsto \max(\boldsymbol{0}, \boldsymbol{x})$, where all operations are executed coordinate-wise.
- Sigmoid σ: Defined as $\boldsymbol{x} \mapsto \boldsymbol{1}/(\boldsymbol{1} + \exp(-\boldsymbol{x}))$, where all operations are executed coordinate-wise.
- Softmax σ: For a vector $\boldsymbol{x}$, its i-th component is transformed as $\sigma(\boldsymbol{x})_i = \exp(x_i)/(\sum_i \exp(x_i))$, where the sum runs over all components of the vector.

Multiple fully-connected layers can be stacked to form a *multi-layer perceptron* (MLP). Given an input $\boldsymbol{x} \in \mathbb{R}^{d_{\text{in}}}$ and a set of L fully-connected layers, each of which is parameterized by $\boldsymbol{\theta}_i = (\boldsymbol{W}_i, \boldsymbol{b}_i)$ and denoted as $\boldsymbol{g}_i(\cdot; \boldsymbol{\theta}_i) : \mathbb{R}^{d^{i-1}} \to \mathbb{R}^{d^i}$ for any $i \in \{1, \dots, L\}$, an MLP is constructed as a composition function, formulated in an iterative manner as

$$\boldsymbol{h}^{(0)} = \boldsymbol{x}, \quad \boldsymbol{h}^{(i)} = \boldsymbol{g}_i(\boldsymbol{h}^{(i-1)}; \boldsymbol{\theta}_i) := \sigma(\boldsymbol{W}_i^\top \boldsymbol{h}^{(i-1)} + \boldsymbol{b}_i), \quad i = 1, \dots, L, \tag{2.2}$$

and finally returns $\boldsymbol{y} = \boldsymbol{h}^{(L)}$. Note that in (2.2), the dimensions of $\boldsymbol{W}_1$ and $\boldsymbol{W}_L$ should be compatible with $\boldsymbol{x}, \boldsymbol{y}$: $d_0 = d_{\text{in}}$ and $d_L = d_{\text{out}}$. The whole of learnable parameters, denoted as $\boldsymbol{\theta} = \{\boldsymbol{\theta}_i\}_{i=1}^L$, can be trained with data to take arbitrary values so that the resulting MLP is able to yield accurate approximations of desired outputs. The power of deep learning lies at the capacity of neural networks to approximate complicated mappings when it uses many layers, i.e., becomes deep [82]. MLPs are simple yet decently effective in many machine learning tasks including L2O. In the subsequent sections, we use the following condensed representation for the entire model as given in (2.2):

$$\boldsymbol{y} = \text{MLP}(\boldsymbol{x}; \boldsymbol{\theta}) \quad \text{or} \quad \boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{2.3}$$

2.3 Training a neural network

Given initial parameters $\boldsymbol{\theta}$, typically with random values, *training* a neural network is to update these parameters so that the neural network can approximate the target unknown mapping. This process is also called learning. Training requires several components: (i) a *dataset* that we learn from; (ii) a quantitative metric for the quality of the approximation on the dataset, called *loss function*; (iii) a mechanism that minimizes the loss function by updating the parameters; (iv) regularization techniques that avoid overfitting. Step (iii) is often, though not always, based on gradient descent, and the gradient is typically, though not always, computed with the *chain rule* and *backpropagation*.

Datasets. A neural network is trained on a finite set of data samples. Each sample consists of the input to the network $\boldsymbol{x} \in \mathbb{R}^{d_{\text{in}}}$ and the desired output $\boldsymbol{y} \in \mathbb{R}^{d_{\text{out}}}$. In this article, we refer to $\boldsymbol{x}$ as the *input* and $\boldsymbol{y}$ as the *label*, and denote the dataset as $\mathbb{D} = \{(\boldsymbol{x}_j, \boldsymbol{y}_j)\}_{j=1}^N$ with N being the size of the dataset. Note that the label $\boldsymbol{y}$ is optional and can be absent in unsupervised and self-supervised tasks.

The format of a sample $(\boldsymbol{x}, \boldsymbol{y})$ depends on the learning task of interest. For image classification, $\boldsymbol{x}$ is an image and $\boldsymbol{y}$ is a scalar indicating the class of $\boldsymbol{x}$. For natural language translation, $\boldsymbol{x}$ is a sequence of tokens in the source language and $\boldsymbol{y}$ is the semantically equivalent sequence of tokens in the target language. However, since L2O is aimed to learn how to solve optimization problems, an input to a network for L2O is what characterizes an optimization problem instance. For example, if a neural network is trained to solve least squares in the form of $\min_{\boldsymbol{z}} \|\boldsymbol{d} - \boldsymbol{A}\boldsymbol{z}\|^2$, an input sample is $\boldsymbol{x} = (\boldsymbol{d}, \boldsymbol{A})$ and its label is the solution $\boldsymbol{y} = \boldsymbol{z}^* = \arg\min_{\boldsymbol{z}} \|\boldsymbol{d} - \boldsymbol{A}\boldsymbol{z}\|^2$ (assuming $\boldsymbol{z}^*$ is the only minimizer).

Loss functions. Given a data set $\mathbb{D}$ of size N, the neural network to be trained $\boldsymbol{g}(\cdot;\boldsymbol{\theta}) : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$ is parameterized by $\boldsymbol{\theta} \in \mathbb{R}^p$ and maps each input $\boldsymbol{x}_j$ from $\mathbb{D}$ to the output $\hat{\boldsymbol{y}}_j(\boldsymbol{\theta}) = \boldsymbol{g}(\boldsymbol{x}_j;\boldsymbol{\theta})$. Here, $\boldsymbol{g}(\cdot;\boldsymbol{\theta})$ denotes an arbitrary parameterized mapping, such as the MLP defined in (2.3). A *loss function* provides a metric to measure the quality of the outputs over the whole set $\{\hat{\boldsymbol{y}}_j(\boldsymbol{\theta})\}_{j=1}^N$, which we need for updating the parameters in the neural network accordingly.

Considering the above-mentioned supervised learning setting where each input $\boldsymbol{x}_j$ comes with a label $\boldsymbol{y}_j$, we first pick a metric function $\ell : \mathbb{R}^{d_{\text{out}}} \times \mathbb{R}^{d_{\text{out}}} \to \mathbb{R}$ that maps a prediction-label pair $(\hat{\boldsymbol{y}}_j, \boldsymbol{y}_j)$ to a scalar $l_j = \ell(\hat{\boldsymbol{y}}_j, \boldsymbol{y}_j)$, which measures the *loss* or *regret* induced by the prediction. For example, the simplest metric function, the mean-squared error (MSE), is defined by $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) := \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|^2$. Cross-entropy (CE) is another commonly-used metric function in machine learning. A loss function can be defined as a function of the parameter $\boldsymbol{\theta}$ over all samples in $\mathbb{D}$:

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{j=1}^N l_j = \sum_{j=1}^N \ell(\hat{\boldsymbol{y}}_j(\boldsymbol{\theta}), \boldsymbol{y}_j). \tag{2.4}$$

In unsupervised or self-supervised learning where the labels are absent, we can adopt a metric function whose value only depends on the output of the network with $l_i = \ell(\hat{\boldsymbol{y}}_j)$.

Backpropagation. To train the neural network, we try to find the parameter $\boldsymbol{\theta}^*$ that nearly minimizes the loss function: $\boldsymbol{\theta}^* \approx \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$. Since neural networks are usually deep and thus overparameterized, the learnable parameter $\boldsymbol{\theta}$ is extremely high-dimensional in most cases. To make the training feasible, it is most common to adopt first-order methods to minimize the loss function, which requires calculating the gradient of the loss function with respect to the parameter, i.e., $\partial\mathcal{L}/\partial\boldsymbol{\theta}$. The chain rule in calculus is leveraged to do so efficiently.

Consider the i-th layer $\boldsymbol{g}^{(i)}(\cdot;\boldsymbol{\theta}^{(i)})$ of the network parameterized by $\boldsymbol{\theta}^{(i)}$. The input to the layer is $\boldsymbol{h}^{(i-1)}$, which is the output of the previous layer, and the output is $\boldsymbol{h}^{(i)} = \boldsymbol{g}^{(i)}(\boldsymbol{h}^{(i-1)};\boldsymbol{\theta}^{(i)})$, which is then fed into the next layer as input. Assuming we are given $\frac{\partial\mathcal{L}}{\partial\boldsymbol{h}^{(i)}}$, the gradient of the loss function with respect to $\boldsymbol{h}^{(i)}$, the gradient with respect to the parameter and input of this layer can be calculated with the chain rule as

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{\theta}^{(i)}} = \frac{\partial\mathcal{L}}{\partial\boldsymbol{h}^{(i)}} \cdot \frac{\partial\boldsymbol{h}^{(i)}}{\partial\boldsymbol{\theta}^{(i)}}, \tag{2.5}$$

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{h}^{(i-1)}} = \frac{\partial\mathcal{L}}{\partial\boldsymbol{h}^{(i)}} \cdot \frac{\partial\boldsymbol{h}^{(i)}}{\partial\boldsymbol{h}^{(i-1)}}, \tag{2.6}$$

where the latter is for propagating the gradient back to previous layers. The second terms on the right-hand sides of both equations depend solely on $\boldsymbol{g}^{(i)}$. Therefore, this procedure can be performed sequentially from the output layer to the input layer. This process is called *backpropagation* in contrast to the *forward propagation* from inputs to outputs, as it essentially propagates the error signals in the backward direction from outputs to inputs. Modern deep learning frameworks, such as TensorFlow and PyTorch, provide out-of-the-box support of the backpropagation for most deep learning layers and modules. Practitioners do not need to code them except when developing new layers.

Training algorithms. Neural networks are mostly trained with first-order methods. For example, we can directly apply gradient descent to minimize the loss function (2.4) over the whole training set. However, since the training set for modern deep learning is very large, the most popular optimization approach is stochastic gradient descent (SGD), which samples a small subset of the entire training set, called a *mini-batch* at each training step. Such a training method is defined in Algorithm 1.

Algorithm 1 Stochastic gradient descent in supervised learning

```
Input: A data set $\mathbb{D} = \{(\boldsymbol{x}_j, \boldsymbol{y}_j)\}_{j=1}^N$, a model $\boldsymbol{g}(\cdot;\boldsymbol{\theta})$ with parameter $\boldsymbol{\theta} \in \mathbb{R}^p$.
1: Determine hyperparameters: the number of epoch $E$, learning rate $\alpha$, mini-batch size $M$.
2: Initialize the values of $\boldsymbol{\theta}$ randomly.
3: for $e = 1$ to $E$ do
4:     Sample a subset $\hat{\mathbb{D}}$ with size $M$: $\hat{\mathbb{D}} \subset \mathbb{D}$.
5:     Calculate the gradient through backpropagation: $\partial\hat{\mathcal{L}}/\partial\boldsymbol{\theta}$, where $\hat{\mathcal{L}} := \sum_{(\boldsymbol{x},\boldsymbol{y})\in\hat{\mathbb{D}}} \ell(\boldsymbol{g}(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y})$.
6:     Update the parameters $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\partial\hat{\mathcal{L}}/\partial\boldsymbol{\theta}$.
7: end for
8: return $\boldsymbol{\theta}$
```

Motivations behind such a mini-batch method are two-fold. Firstly, effective training of heavily overparameterized neural networks requires so many training samples that the device memory cannot hold all of them at once. Secondly, a proper amount of stochasticity helps the training algorithm escape some local minima of $\mathcal{L}$ while too much of it can significantly increase the variance of gradients, slowing down convergence. Although SGD does not guarantee convergence to the global minimum $\boldsymbol{\theta}^*$ due to the non-convexity nature of the loss function, it often yields a parameter $\boldsymbol{\theta}$ corresponding to a reasonably low loss function value under properly chosen hyperparameters. (A hyperparameter tuning method will be presented in the subsequent paragraph "Validation".)

Modifications to SGD have been proposed for more efficient and stable stochastic training, among which Adam [101] and its variants stand out. Adam updates the network parameters with the momentum, the moving average of the gradients over time, which is normalized with the variance of the gradients. Adam implicitly leverages the first-order and second-order information of the loss function and thus is faster than SGD in many applications. SGD and Adam are enough for training neural networks used in most L2O works and are good starting points for new applications of L2O before special needs identified.

Popular training techniques, including SGD and Adam, have been implemented in most modern deep-learning platforms, eliminating the need for manual coding. In the rest of this paper, *we will treat the minimization of loss functions as solvable problems*, shifting our emphasis to other critical subjects.

Validation. Although it is common to manually determine the values of hyperparameters (e.g., E, α, M in Algorithm 1), there is a more systematic approach known as *validation*. This method partitions a complete dataset into two subsets: the training set $\mathbb{D}_{\text{train}}$ and the validation set $\mathbb{D}_{\text{val}}$. Algorithm 1 is employed on $\mathbb{D}_{\text{train}}$. The role of the validation set is to evaluate the model's capability to generalize to new data, enabling the selection of superior training hyperparameters. The validation loss function is defined as

$$\mathcal{L}_{\text{val}}(\boldsymbol{\theta}) := \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathbb{D}_{\text{val}}} \ell(\boldsymbol{g}(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}).$$

Imagine we are comparing two hyperparameter configurations, (E, α, M) and (E', α', M'), and their resulting parameters $\boldsymbol{\theta}(E, \alpha, M)$ and $\boldsymbol{\theta}(E', \alpha', M')$, respectively. If $\mathcal{L}_{\text{val}}(\boldsymbol{\theta}(E, \alpha, M)) < \mathcal{L}_{\text{val}}(\boldsymbol{\theta}(E', \alpha', M'))$, then we conclude that (E, α, M) is better than (E', α', M').

Regularization. While the training of a neural network is conducted on the training set, the ultimate goal is to learn a network that can perform well on samples unseen during training. A neural network with this ability is considered to *generalize* well or have a great *generalization ability*. However, an inappropriate training process can hurt the generalization ability of a neural network by *overfitting* the parameters only to the samples in the training set, resulting in poor performances on unseen samples. Regularization methods aim to address this issue by adding additional constraints or penalties to the learning process, encouraging the model to generalize better.

Here are some commonly-used regularization techniques in deep learning and specifically for L2O: (1) *L*1 *and L*2 *regularization* (*Weight Decay*), which encourages the model to find simpler solutions by shrinking the weights, reducing their impact on the overall loss. (2) *Dropout*, which combats overfitting by randomly disabling a proportion of the neurons during each training iteration. We clarify that no neurons are dropped out during inference though their outputs are scaled down by the dropout probability. (3) *Early stopping*, which forces the training process to exit early when the model's performance on a validation set starts to degrade. Early stopping prevents the model from overfitting by finding a balance between training for too long (leading to overfitting) and stopping too early (resulting in underfitting). (4) *Data augmentation*, which artificially increases the size of the training dataset by applying various transformations to the existing data samples so that the model can learn to be invariant to such variations, thereby improving generalization.

These regularization techniques can be used individually or in combination, depending on the specific problem and characteristics of the dataset. With regularization, deep neural networks can become more robust, generalize better to unseen data, and exhibit improved performance in real-world applications.

2.4 Important variants of neural networks

Recurrent neural networks. A *recurrent neural network* (RNN) is a special class of neural networks designed for processing sequential data. Unlike standard neural networks that work with a single input with a fixed size, RNNs manage an array of sequential inputs, denoted as $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(T)} \in \mathbb{R}^{d_{\text{in}}}$, where the input's dimension is represented by d_{in}. Given that the sequence length T differs between inputs, traditional MLPs fall short, as they are tailored to fixed-size inputs. This variability is commonly seen in applications like machine translation where each $\boldsymbol{x}^{(i)}$ symbolizes a word, necessitating the handling of sentences of diverse lengths.

To cater to this need, RNNs were introduced, with the core idea of *weight sharing*, that is to share the same set of parameters between layers, so that the networks can have arbitrary depths. Moreover, an RNN maintains a *hidden state* vector $\boldsymbol{h}^{(i)} \in \mathbb{R}^{d_{\text{emb}}}$ which is assumed to be a lossy representation or *embedding* of the previous inputs $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(i)}$ with d_{emb} being the dimension of the hidden embeddings. Denoting the output dimension as d_{out}, a generic parameterized RNN $\boldsymbol{g} : \mathbb{R}^{d_{\text{in}}} \times \mathbb{R}^{d_{\text{emb}}} \to \mathbb{R}^{d_{\text{out}}} \times \mathbb{R}^{d_{\text{emb}}}$ can be formulated as

$$\boldsymbol{y}^{(i)}, \boldsymbol{h}^{(i)} = \boldsymbol{g}(\boldsymbol{x}^{(i)}, \boldsymbol{h}^{(i-1)}; \boldsymbol{\theta}). \tag{2.7}$$

Here, the i-th layer or iteration of the RNN yields the *output* of the current layer $\boldsymbol{y}^{(i)} \in \mathbb{R}^{d_{\text{out}}}$, which can be used to calculate losses for training, and the hidden state $\boldsymbol{h}^{(i)}$, which will be sent to the next iteration to pass the previous information. In practice, we set $\boldsymbol{h}^{(0)}$ to all zero or a randomly sampled vector. Note that the operator $\boldsymbol{g}$ and its learnable parameter $\boldsymbol{\theta}$ is time-invariant and thus the network can be applied to sequences with arbitrary lengths.

However, a vanilla RNN suffers from the gradient vanishing or explosion issue when processing very long sequences [126], in which case the backpropagation to early steps involves too many multiplications of the Jacobian matrix of $\boldsymbol{g}$. While gradient explosion is more devastating for failing training, gradient vanishing happens more often and prevents the RNN from learning long-term memories as the result of early iterations receiving near zero gradients. The long short-term memory (LSTM) network was proposed with two core techniques to address this issue. One is the introduction of self-loops so the gradient can backpropagate better for longer sequences. The other is to make the self-loops dependent on the context, i.e., the input sequences so that the network can condition itself to memorize or forget certain parts of an input sequence. LSTM achieves empirical success in many fields such as audio analysis [170] and machine translation [61]. For more details of RNN and LSTM networks and advanced techniques, we refer the reader to [67].

RNN and its variants have been widely adopted by the L2O community. The reason is that numerous optimization problems are solved with iterative algorithms, and the iterative formulation has a natural correspondence with the recurrent structure of an RNN. Consider the gradient descent (GD) algorithm with a fixed step size that tries to minimize a differentiable objective function $f(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}$. One iteration of GD is formulated as $\boldsymbol{x}^{(i)} = \boldsymbol{x}^{(i-1)} - \alpha\nabla f(\boldsymbol{x}^{(i-1)})$. If we think of the gradient $\boldsymbol{d}^{(i-1)} = \nabla f(\boldsymbol{x}^{(i-1)})$ as an input to the current iteration, GD can be written in a similar manner to that in (2.7) as $\boldsymbol{x}^{(i)} = \boldsymbol{g}(\boldsymbol{d}^{(i-1)}, \boldsymbol{x}^{(i-1)}; \alpha)$. Here, the current iterate $\boldsymbol{x}^{(i)}$ corresponds to the hidden state $\boldsymbol{h}^{(i)}$ in (2.7) that carries a summary of all gradients information in the past, and the iteration is parameterized by a single scalar α, the step size in GD. Based on this similarity, many researchers construct RNNs that mimic iterative algorithms and then exploit data-driven learning in expectation of achieving certain levels of acceleration. We will revisit this relation for algorithm unrolling in Section 3 with practical examples.

Convolutional neural networks. *Convolutional neural networks* (CNNs) represent a category of neural networks that excel in processing grid-structured data. This includes 1-D grids like time-series data and 2-D grids such as images. While a vanilla CNN has a structure akin to an MLP as seen in (2.2), the core difference lies in its linear mapping. Instead of the generic linear mapping, $\boldsymbol{x} \mapsto \mathbf{W}^\top \boldsymbol{x} + \boldsymbol{b}$, CNNs deploy a unique linear mapping termed as *convolution*, represented by $\boldsymbol{x} \mapsto \boldsymbol{w} * \boldsymbol{x} + \boldsymbol{b}$, with $\boldsymbol{w}$ being the convolutional kernel. For illustrative purposes, let us focus on the 1-D convolution. In this scenario, the input $\boldsymbol{x} \in \mathbb{R}^{d_{\text{in}}}$ is a discretized variant of a 1-D function $x(t)$. Concurrently, the kernel $\boldsymbol{w} \in \mathbb{R}^K$ mirrors this, where the grid size K is also referred to as the *kernel size*. By denoting $\boldsymbol{x}[i]$ as the i-th component of the vector $\boldsymbol{x}$, the convolution operation, $\boldsymbol{w} * \boldsymbol{x}$, can be defined coordinate-wise as

$$(\boldsymbol{w} * \boldsymbol{x})[i] = \sum_{k=1}^{K} \boldsymbol{w}[k] \cdot \boldsymbol{x}[i + k - 1], \quad \text{for all } i = 1, 2, \dots, d_{\text{in}}. \tag{2.8}$$

Note that the operation formulated in (2.8) is mathematically known as *cross-correlation* but is referred to as convolution in most deep learning papers. Such a convolution can be easily extended to 2-D cases. Contrasted with the generic linear mapping characterized by parameters $(\mathbf{W}, \boldsymbol{b})$, convolutional mapping, defined by parameters $(\boldsymbol{w}, \boldsymbol{b})$, provides a more structured, parameter-efficient approach.

Convolution layers have three key characteristics that are fundamental to the success of CNNs in a wide array of applications. To begin with, when the kernel size K is much smaller than the input dimension d_{in}, the kernel $\boldsymbol{w}$ only locally interacts with part of the input, which endows CNNs with sensitivity to local features. Secondly, the identical kernel is applied across all positions of the input. This design, referred to as *parameter sharing*, serves as an implicit but strong regularization for better generalization and computational efficiency of CNNs. Lastly, convolutions inherently exhibit *equivariance* to translations. This means that the output of a convolution mapping remains unchanged, up to the identical translation, when applied to a translated input. Consequently, this allows CNNs to effectively detect essential patterns even when inputs are shifted, which is considerably challenging for traditional MLPs where the parameters have one-on-one correspondence between input and output coordinates.

In the existing corpus of literature pertaining to L2O, CNNs predominantly find applicability in two principal use cases. The first is to incorporate convolutional layers to imitate certain operations in an optimization process, especially when the optimization involves convolution operations itself. Examples include convolutional compressive sensing, which we will explore in greater detail in Section 3 dedicated for algorithm unrolling. The second use case leverages CNNs as black-box mappings, exploiting their benefits in specific domains. For examples, 2-D CNNs have been observed to excel at natural image denoising. Consequently, researchers have proposed the application of well-trained CNNs as black-box denoisers, intended to be integrated in a plug-and-play manner within a larger optimization framework. Section 4 is dedicated to covering L2O methods in this category.

Graph neural networks. Let us consider an undirected graph represented as $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the collection of vertices and $\mathcal{E}$ is the collection of edges. Each node is indexed by i, and an edge connecting nodes i and j is labeled as the tuple (i, j). The graph-structured data can be described as $\mathcal{G} = (\{\boldsymbol{v}_i\}_{i\in\mathcal{V}}, \{\boldsymbol{e}_{i,j}\}_{(i,j)\in\mathcal{E}})$, in which $\boldsymbol{v}_i$ is the data attached to node i and $\boldsymbol{e}_{i,j}$ is the data attached to edge (i, j). For example, in a social network graph, each node might symbolize a user with the data $\boldsymbol{v}_i$ giving information about them. An edge might signify a connection between two users, and the $\boldsymbol{e}_{i,j}$

describes their relationship. It is worth noting that *enormous optimization problems, especially discrete optimization, can be expressed by graph-structured data* [60, 98]. We defer more details of this type of approach to Section 6.

GNNs map graph-structured data $\mathcal{G}$ to desired outputs $\boldsymbol{y}$. Here we consider a simple one-layer message-passing GNN to present its formal expression. The first step is defined as

$$\boldsymbol{h}_i = \mathrm{MLP}\bigg(\boldsymbol{v}_i, \sum_{j\in\mathcal{N}(i)} \mathrm{MLP}(\boldsymbol{v}_i, \boldsymbol{v}_j, \boldsymbol{e}_{i,j}; \boldsymbol{\theta}_1); \boldsymbol{\theta}_2\bigg). \tag{2.9}$$

Here each node i collects information from its neighbor $\mathcal{N}(i)$, and then we invoke MLP for each node. After the operation of (2.9), we obtain a *hidden state* $\boldsymbol{h}_i$ for each node i. The second step is mapping the hidden states to the output $\boldsymbol{y}$. Based on the structure of $\boldsymbol{y}$, there are two potential mappings. If $\boldsymbol{y}$ illustrates the graph's entire properties, all the states $\boldsymbol{h}_i$ are combined, followed by an MLP invocation for each graph. However, if $\boldsymbol{y}$ consists of labels of all nodes formatted as $\boldsymbol{y} = (\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots)$, an MLP is invoked for each node individually. To elaborate,

$$\text{Graph-level output: } \boldsymbol{y} = \mathrm{MLP}\bigg(\sum_{i\in\mathcal{V}} \boldsymbol{h}_i; \boldsymbol{\theta}_3\bigg), \tag{2.10}$$

$$\text{Node-level output: } \boldsymbol{y}_i = \mathrm{MLP}(\boldsymbol{h}_i; \boldsymbol{\theta}_4). \tag{2.11}$$

For the rest of this paper, the complete GNN model is condensed to

$$\begin{aligned} \boldsymbol{y} &= \mathrm{GNN}(\mathcal{G}; \boldsymbol{\theta}), \quad \text{Graph-level GNN: (2.9), (2.10)},\\ \boldsymbol{y}_i &= \mathrm{GNN}(i, \mathcal{G}; \boldsymbol{\theta}), \quad \text{Node-level GNN: (2.9), (2.11)}. \end{aligned}$$

For a graph-level GNN, the parameters are given by $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3)$, while for a node-level GNN, they are $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_4)$.

The GNN structure outlined earlier presents multiple advantages when dealing with graph-structured data. Firstly, **scalability** is inherent in its design. It is crucial to understand that every MLP employed is consistent across all nodes and edges. As a result, even when a new graph of varying size emerges, the methodology to determine hidden states remains unaltered. Moreover, the same repetitive process can be enacted on this new graph without modifying the MLPs. This implies that this particular message-passing GNN is adaptable to graphs of any dimension. Secondly, the structure ensures **permutation invariance**. Evaluating (2.9), (2.10), and (2.11), it becomes apparent that altering the indices of two nodes does not impact the output of a graph-level GNN. On the other hand, for a node-level GNN, the outputs adjust corresponding to the permutation of inputs. We refer to [92, 172] for more properties and applications of GNN.

2.5 Reinforcement learning

In addition to supervised learning discussed in Subsection 2.1, another important machine learning approach is *reinforcement learning* (*RL*). Instead of relying on vast amounts of data with manually-labeled ground-truth as in supervised learning, RL focuses on learning a decision-making policy through interactions with the underlying system. This subsection introduces basic concepts of RL and provides a concise guide for beginners.

Markov decision process (MDP). At the heart of reinforcement learning lies the Markov decision process. Imagine playing a game where at every step, based on your current position, you choose an action. This action leads to a specific outcome and potentially a reward. The goal is to maximize rewards over time, and this setup is described by an MDP.

Formally, an MDP is a discrete-time stochastic process during time $t = 0, 1, \ldots, T$. At each time t, we have three random variables $\boldsymbol{s}_t, \boldsymbol{a}_t, r_t$. The state variable $\boldsymbol{s}_t$ reflects the system's condition, like a chess board's layout or a car's status (such as position and speed). The action variable $\boldsymbol{a}_t$ represents the decision at time t. The reward variable r_t is a value (typically real) given by the system or set by individuals. An MDP is characterized by a tuple $(\mathbb{S}, \mathbb{A}, p, r)$:

• $\mathbb{S}$ denotes the *state space*, encompassing all possible values of a state variable.

• $\mathbb{A}$ denotes the *action space*, encompassing all possible values of an action variable.

• $p : \mathbb{S} \times \mathbb{S} \times \mathbb{A} \to \mathbb{R}$ defines the probability of *transitioning* from state $\boldsymbol{s}$ to $\boldsymbol{s}'$ after taking action $\boldsymbol{a}$:

$$\Pr(\boldsymbol{s}_{t+1} = \boldsymbol{s}' \mid \boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a}) = p(\boldsymbol{s}, \boldsymbol{s}', \boldsymbol{a}) \quad \text{for all } \boldsymbol{s}, \boldsymbol{s}' \in \mathbb{S},\ \boldsymbol{a} \in \mathbb{A},\ t = 0, 1, 2, \ldots, T.$$

• $r : \mathbb{S} \times \mathbb{A} \to \mathbb{R}$ describes the *reward* received after taking action $\boldsymbol{a}$ at state $\boldsymbol{s}$.

Once $(\mathbb{S}, \mathbb{A}, p, r)$ are specified, the MDP is fully defined. For every $(\boldsymbol{s}, \boldsymbol{a})$, if $p(\boldsymbol{s}, \boldsymbol{s}', \boldsymbol{a}) = 1$ for some $\boldsymbol{s}'$, the transition is deterministic. Depending on the availability of p, the MDP can either be a conventional control problem (known p) or a reinforcement learning problem (unknown p).

Reinforcement learning. Imagine that we cannot directly access the formula for p and view the MDP as a "black-box". Our interaction with this black-box is iterative: we observe state $\boldsymbol{s}_t$ from it, decide to take action $\boldsymbol{a}_t$, and then receive a reward r_t and a new state $\boldsymbol{s}_{t+1}$ in response. Given these interactions, the challenge is: Can we deduce an optimal policy that gives the probabilities of taking different actions for a particular state? This is the fundamental query addressed by RL. To elucidate, let us define some RL terms:

• Environment: The black-box MDP $(\mathbb{S}, \mathbb{A}, p, r)$ where p is not directly accessible.

• Policy: A parameterized model $\pi(\boldsymbol{a}, \boldsymbol{s}; \boldsymbol{\theta})$ that indicates the probability of selecting action $\boldsymbol{a}$ when in state $\boldsymbol{s}$. Mathematically, $\Pr(\boldsymbol{a}_t = \boldsymbol{a} \mid \boldsymbol{s}_t = \boldsymbol{s}) = \pi(\boldsymbol{a}, \boldsymbol{s}; \boldsymbol{\theta})$ for all $\boldsymbol{a} \in \mathbb{A},\ \boldsymbol{s} \in \mathbb{S},\ t = 0, 1, \ldots, T$.

• Trial: With a given environment and policy, the distribution of random variables $\boldsymbol{s}_t, \boldsymbol{a}_t$ are well-defined. Concatenating all these variables together, it is denoted as $\boldsymbol{\tau} := \{(\boldsymbol{s}_t, \boldsymbol{a}_t)\}_{t=0}^{T}$. The joint probability distribution of $\boldsymbol{\tau}$, induced by the MDP $(\mathbb{S}, \mathbb{A}, p, r)$ and policy $\pi(\boldsymbol{a}, \boldsymbol{s}; \boldsymbol{\theta})$, is denoted with $P_{\boldsymbol{\theta}}(\boldsymbol{\tau})$ and is given by

$$P_{\boldsymbol{\theta}}(\boldsymbol{\tau}) := \Pr(\boldsymbol{s}_0, \boldsymbol{a}_0, \ldots, \boldsymbol{s}_T, \boldsymbol{a}_T) = \Pr(\boldsymbol{s}_0)\Bigg(\prod_{t=0}^{T-1} \Pr(\boldsymbol{a}_t \mid \boldsymbol{s}_t)\Pr(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t)\Bigg)\Pr(\boldsymbol{a}_T \mid \boldsymbol{s}_T). \tag{2.12}$$

A realization of $\boldsymbol{\tau}$ is called a *trial*. A trial manifests when the policy interacts with the environment, logging the resulting data.

The goal of RL is to minimize the loss function defined by the expected cumulative reward over time:

$$\underset{\boldsymbol{\theta}}{\text{minimize}}\ \mathcal{L}_T(\boldsymbol{\theta}) := -\mathbb{E}_{\boldsymbol{\tau} \sim P_{\boldsymbol{\theta}}(\boldsymbol{\tau})} \sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t). \tag{2.13}$$

Policy gradient. The policy gradient is a technique of computing the gradient $\partial \mathcal{L}_T / \partial \boldsymbol{\theta}$, allowing the use of first-order optimization methods like SGD or Adam, as discussed in Subsection 2.3. Specifically, the gradient can be expressed as

$$\begin{aligned}
\nabla \mathcal{L}_T(\boldsymbol{\theta}) &= \nabla \int_{\boldsymbol{\tau} \in (\mathbb{S} \times \mathbb{A})^{T+1}} P_{\boldsymbol{\theta}}(\boldsymbol{\tau}) \Bigg(\sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t)\Bigg) d\boldsymbol{\tau} \\
&= \int_{\boldsymbol{\tau} \in (\mathbb{S} \times \mathbb{A})^{T+1}} \frac{\partial}{\partial \boldsymbol{\theta}} P_{\boldsymbol{\theta}}(\boldsymbol{\tau}) \Bigg(\sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t)\Bigg) d\boldsymbol{\tau} \\
&= \int_{\boldsymbol{\tau} \in (\mathbb{S} \times \mathbb{A})^{T+1}} P_{\boldsymbol{\theta}}(\boldsymbol{\tau}) \frac{\partial}{\partial \boldsymbol{\theta}} \log(P_{\boldsymbol{\theta}}(\boldsymbol{\tau})) \Bigg(\sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t)\Bigg) d\boldsymbol{\tau} \\
&= \mathbb{E}_{\boldsymbol{\tau} \sim P_{\boldsymbol{\theta}}(\boldsymbol{\tau})} \Bigg[\frac{\partial}{\partial \boldsymbol{\theta}} \log(P_{\boldsymbol{\theta}}(\boldsymbol{\tau})) \Bigg(\sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t)\Bigg)\Bigg].
\end{aligned}$$

Using the joint distribution formula,

$$\log(P_{\boldsymbol{\theta}}(\boldsymbol{\tau})) = \log(\Pr(\boldsymbol{s}_0)) + \left(\sum_{t=0}^{T-1} \log(\Pr(\boldsymbol{a}_t \mid \boldsymbol{s}_t)) + \log(\Pr(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t))\right) + \log(\Pr(\boldsymbol{a}_T \mid \boldsymbol{s}_T)).$$

Note that $\Pr(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t)$ and $\Pr(\boldsymbol{s}_0)$ are independent of $\boldsymbol{\theta}$. Therefore,

$$\frac{\partial}{\partial \boldsymbol{\theta}} \log(P_{\boldsymbol{\theta}}(\boldsymbol{\tau})) = \sum_{t=0}^{T} \frac{\partial}{\partial \boldsymbol{\theta}} \log(\Pr(\boldsymbol{a}_t \mid \boldsymbol{s}_t)) = \sum_{t=0}^{T} \frac{\partial}{\partial \boldsymbol{\theta}} \log(\pi(\boldsymbol{a}_t, \boldsymbol{s}_t; \boldsymbol{\theta})).$$

Typically, a deep neural network is selected as the policy model π. Thus, in the above equation, $\partial \log(\pi)/\partial \boldsymbol{\theta}$ can be computed using backpropagation, as mentioned in Subsection 2.3. To solve (2.13), one can employ Algorithm 1: interact with the environment M times, record a trial $\boldsymbol{\tau}_i$ each time, consider a set of trials $\{\boldsymbol{\tau}_i\}_{i=1}^{M}$ a mini-batch in Algorithm 1, and update $\boldsymbol{\theta}$ similarly to Algorithm 1.

Q-learning. Let us examine the scenario where $T = \infty$. To ensure that the cumulative reward is not infinite, we introduce a *discount factor* $\gamma \in (0, 1)$ and define a new loss function as

$$\mathcal{L}_{\gamma}(\pi) := -\mathbb{E}_{\boldsymbol{\tau} \sim P_{\pi}(\boldsymbol{\tau})} \sum_{t=1}^{\infty} \gamma^{t} r(\boldsymbol{s}_t, \boldsymbol{a}_t).$$

Instead of the $P_{\boldsymbol{\theta}}$ in (2.13), here we use a more general joint distribution P_{π}, where π represents any potential policy with $\pi(\boldsymbol{a}, \boldsymbol{s}) = \Pr(\boldsymbol{a}_t = \boldsymbol{a} \mid \boldsymbol{s}_t = \boldsymbol{s})$. Based on this, we define a Q-function $Q_{\pi} : \mathbb{A} \times \mathbb{S} \to \mathbb{R}$ as

$$Q_{\pi}(\boldsymbol{a}, \boldsymbol{s}) := \mathbb{E}_{\boldsymbol{\tau} \sim P_{\pi}(\boldsymbol{\tau})} \left[\sum_{t'=t}^{\infty} \gamma^{t'-t} r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'}) \,\middle|\, \boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a} \right].$$

The Q-function measures the expected reward for the action $\boldsymbol{a}_t = \boldsymbol{a}$ as $\boldsymbol{s}_t = \boldsymbol{s}$. Given the nature of MDPs, this function remains time-independent. Hence, the Q-function can act as an evaluation metric for policy π: the higher its value, the better the policy. The best policy's Q-function, denoted by $Q_* = \max_{\pi} Q_{\pi}$, should satisfy the Bellman equation

$$Q_*(\boldsymbol{a}, \boldsymbol{s}) = \mathbb{E}\left[r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \max_{\boldsymbol{a}' \in \mathbb{A}} Q_*(\boldsymbol{s}_{t+1}, \boldsymbol{a}') \,\middle|\, \boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a} \right]. \tag{2.14}$$

The expectation here relies only on the transition probability $\Pr(\boldsymbol{s}_{t+1} = \boldsymbol{s}' \mid \boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a})$, and is independent of the policy π. If the optimal Q-function Q_* is available, an optimal deterministic policy is easy to derive as $\boldsymbol{a}_t = \max_{\boldsymbol{a} \in \mathbb{A}} Q_*(\boldsymbol{a}, \boldsymbol{s}_t)$, assuming the size of $\mathbb{A}$ is moderate. Consequently, Q-learning's primary objective is to *identify a Q-function that aligns with the Bellman equation.*

For Q-learning, we adopt a parameterized machine-learning model to represent the Q-function $Q(\boldsymbol{a}, \boldsymbol{s}; \boldsymbol{\theta})$. Initially, we run the MDP, gathering data in tuples like $(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')$. Here, $\boldsymbol{s}$ represents any state during the current run, $\boldsymbol{a}$ is the action suggested by $\arg\max_{\boldsymbol{a}} Q(\boldsymbol{a}, \boldsymbol{s}; \boldsymbol{\theta})$, $r = r(\boldsymbol{s}, \boldsymbol{a})$ is the reward value, and $\boldsymbol{s}'$ represents the next state. Multiple such data tuples can be extracted from a single run. After sufficient runs and ample data collection, the objective becomes to train the Q-function, so that it reflects the Bellman equation. This is achieved by minimizing the loss:

$$\underset{\boldsymbol{\theta}}{\text{minimize}}\, \mathcal{L}_Q(\boldsymbol{\theta}) := \sum_{(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')} (y - Q(\boldsymbol{a}, \boldsymbol{s}; \boldsymbol{\theta}))^2, \quad \text{where } y = r + \max_{\boldsymbol{a}' \in \mathbb{A}} Q(\boldsymbol{a}', \boldsymbol{s}'; \boldsymbol{\theta}_{\text{prev}}). \tag{2.15}$$

In this scenario, we update $\boldsymbol{\theta}$ while keeping $\boldsymbol{\theta}_{\text{prev}}$ constant. Through backpropagation, we can compute the gradient of $\mathcal{L}_Q$, enabling us to apply Algorithm 1 to update $\boldsymbol{\theta}$. After adequate updates of $\boldsymbol{\theta}$, we synchronize $\boldsymbol{\theta}_{\text{prev}}$ and $\boldsymbol{\theta}$. This cycle of updating $\boldsymbol{\theta}$ continues until the computation budget is reached or $\boldsymbol{\theta}_{\text{prev}}$ and $\boldsymbol{\theta}$ converge to the same value.

A start-up recipe. It is essential to understand that many of the widely adopted reinforcement learning algorithms, like the policy gradient method and Q-learning, are already accessible through modern deep learning platforms. Hence, there is no pressing need to build these algorithms from scratch. When integrating these algorithms into your projects, the primary elements one must define in their code include:

- a state space;
- an action space;
- a reward function.

Based on these components, one also configure:

- a stochastic policy model $\pi(\boldsymbol{a}, \boldsymbol{s}; \boldsymbol{\theta})$ for the policy gradient method; or
- a parameterized Q function $Q(\boldsymbol{a}, \boldsymbol{s}; \boldsymbol{\theta})$ for Q-learning.

3 Algorithm unrolling

In this section, we present an important branch of L2O methods called *algorithm unrolling*, which expresses a classic iterative algorithm as a neural network of a certain depth and replaces certain components of the algorithm with learnable parameters. We start with a simple example of unrolling a projected gradient descent algorithm (PGD) into a neural network. Consider a non-negative least squares optimization problem

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{d} - \boldsymbol{A}\boldsymbol{x}\|_2^2 \quad \text{s.t. } \boldsymbol{x} \geqslant \boldsymbol{0}, \tag{3.1}$$

where $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ is given, and $\boldsymbol{d} \in \mathbb{R}^n$ is the observation we have for the regression. We say $\boldsymbol{d}$ is the input to the algorithm, which characterizes an instance of non-negative least squares along with $\boldsymbol{A}$. PGD for (3.1) is the iteration

$$\boldsymbol{x}^{(i)} = P_+(\boldsymbol{x}^{(i-1)} - \alpha\nabla f(\boldsymbol{x}^{(i-1)})), \quad i = 1, 2, \ldots, \tag{3.2}$$

where $P_+(\cdot)$ projects its input to the non-negativity orthant, $\nabla f(\boldsymbol{x}) = \boldsymbol{A}^\top(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{d})$, and α is the step size. $P_+(\cdot)$ coincides with the ReLU activation function, which is ubiquitously used in deep learning. If we treat the iterate $\boldsymbol{x}^{(i)}$ as a state vector of the i-th iteration and $(\alpha, \boldsymbol{A})$ as the parameters, with some rearrangement, we can re-write (3.2) as a recurrent system

$$\boldsymbol{x}^{(i)} = P_+((\alpha\boldsymbol{A}^\top)\boldsymbol{d} + (\boldsymbol{I}_n - \alpha\boldsymbol{A}^\top\boldsymbol{A})\boldsymbol{x}^{(i-1)}) =: \boldsymbol{g}(\boldsymbol{d}, \boldsymbol{x}^{(i-1)}; (\alpha, \boldsymbol{A})), \tag{3.3}$$

where $\boldsymbol{I}_n \in \mathbb{R}^{n\times n}$ is the identity matrix and =: defines its right-hand side by its left-hand side. We can see the correspondence between PGD (3.3) and RNN (2.7), with $P_+(\cdot)$ being the activation function and $\boldsymbol{d}$ the constant input to each iteration.[1)] This similarity essentially motivates the researchers to actually "relax" the algorithm into an RNN by *parameterizing* part of the algorithm to be learnable with data, e.g., $(\alpha, \boldsymbol{A})$ in (3.3). Note that there are many ways of parameterization other than simply plain conversion. We will discuss in more details later in this section.

Once the parameterization is done, we are left with a neural network that is expected to achieve acceleration on a certain type of optimization problems after proper data-driven training. The original motivation of algorithm unrolling [68] is that the conversion from an iterative algorithm to a neural network enables data-driven learning for solving a specific type of optimization problem. At the cost of convergence guarantee for worst cases, the resulting networks are observed to accelerate when applied to **unseen** problems of the same type, as shown in Figure 1. Since its emergence, algorithm unrolling has found broad applications, including but not limited to (natural) image restoration and enhancement [185], medical and biological imaging [105], and wireless communication [13]. Algorithm unrolling is also referred to as *algorithm unfolding* in the literature.

1) Note there is a difference in notation, where the state vector is denoted as $\boldsymbol{x}^{(i)}$ in (3.3) while it is denoted as $\boldsymbol{h}^{(i)}$ in (2.7).

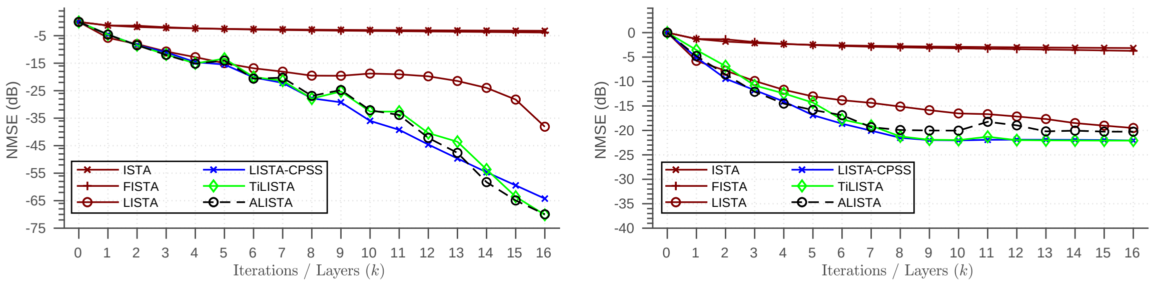


Figure 1 (Color online) Impressive acceleration can be achieved by algorithm unrolling methods (LISTA, LISTA-CPSS, TiLISTA and ALISTA) compared with the classic iterative algorithm ISTA and its variant FISTA accelerated with momentum. Algorithm unrolling uses orders of magnitude fewer iterations than ISTA/FISTA to achieve the same precision. Left figure: noiseless case. Right figure: noisy case (SNR = 20). X-axis is the number of iterations; Y-axis is the normalized mean squared error (lower is better). *Plot source*: Figure 1 of [108]

In this section, we strive to present a comprehensive guideline of how to unroll an algorithm in hand, the major design elements to consider and useful techniques in practice. We first set up the basics and notations for algorithm unrolling and introduce the key motivations behind it. Then we use learned ISTA (LISTA) [68] for a case study to lay out the general routine of unrolling an iterative algorithm into a neural network. After discussions on the key design elements and advanced techniques for algorithm unrolling, we will conclude this section with the introduction to mathematical analyses behind algorithm unrolling in the existing literature.

3.1 Basics and motivations

Generic formulation. We consider optimization problems in the form of

$$\operatorname*{minimize}_{\boldsymbol{x}\in\mathbb{X}(\boldsymbol{d})} f(\boldsymbol{x};\boldsymbol{d}),$$

where $\boldsymbol{x}$ is the variable to optimize and $\boldsymbol{d}$ characterizes an instance of such type of problems. We first select a general iterative algorithm that targets regressing the optimal solution from the observation $\boldsymbol{d}$ with the updating form

$$\boldsymbol{x}^{(i)} = \boldsymbol{g}(\boldsymbol{d}, \boldsymbol{x}^{(i-1)}; \boldsymbol{\theta}), \quad i = 1, 2, \ldots, \infty. \tag{3.4}$$

In this case, we say $\boldsymbol{d}$ is the *input* to the algorithm and $\boldsymbol{\theta}$ is the parameter of the algorithm that consists of the knowledge of the optimization problem and the hyperparameters selected by users. We hope that $\boldsymbol{x}^{(i)}(\boldsymbol{d}) \to \boldsymbol{x}^{\star}(\boldsymbol{d})$, where $\boldsymbol{x}^{\star}(\boldsymbol{d}) := \arg\min_{\boldsymbol{x}\in\mathbb{X}(\boldsymbol{d})} f(\boldsymbol{x};\boldsymbol{d})$ for all $\boldsymbol{d}$ of interest.

Algorithm unrolling unfolds the optimization iteration (3.4) into a neural network of a certain depth and replacing certain parameters of the algorithm, i.e., $\boldsymbol{\theta}$ in (3.4), into learnable parameters. The new updates take the form

$$\boldsymbol{x}^{(i)} = \boldsymbol{g}(\boldsymbol{d}, \boldsymbol{x}^{(i-1)}; \boldsymbol{\theta}^{(i)}), \quad i = 1, 2, \ldots, T, \tag{3.5}$$

where T is the depth of the neural network, and $\boldsymbol{\theta}^{(i)}$ are the learnable parameters in the i-th layer. If we use depth-invariant parameters, i.e., $\boldsymbol{\theta}^{(i)} \equiv \boldsymbol{\theta}$ for all i, we will get an RNN, which is the case of the first network derived using algorithm unrolling in [68]. We also say that the parameter $\boldsymbol{\theta}$ is *shared* or *tied* across layers in this case. Many works that follow [68] propose to use time-variant parameters for a larger model capacity and update them independently during training [29, 39]. Other works also show the benefits of a hybrid parameterization scheme that partially share parameters over time [1, 108]. We will discuss about this design choice in Subsection 3.3. The process of unrolling an iterative algorithm into a neural network is illustrated in Figure 2.

The next step is to train the resulting neural network on a training set $\mathbb{D}_{\text{train}}$ of size N. Consider a simple supervised learning setting where the target signal is accessible by users. Each sample in the training set is an input-label pair $(\boldsymbol{d}_j, \boldsymbol{x}_j^*) \in \mathbb{D}_{\text{train}}$. Denote the resulting network as a end-to-end mapping

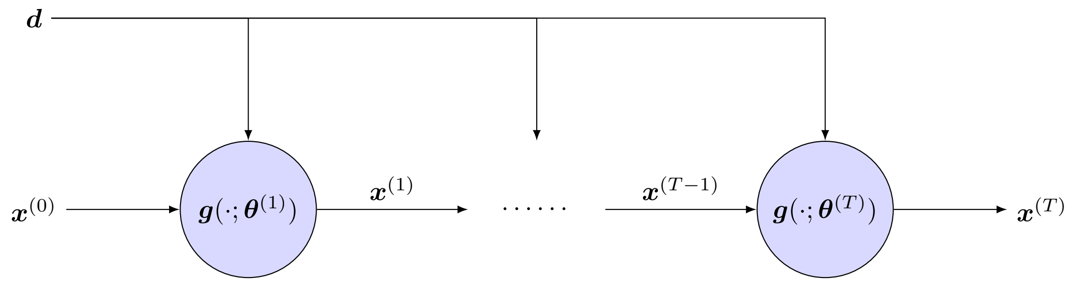


Figure 2 (Color online) An illustration of a generic unrolling process that turns an iterative algorithm into truncated neural network. The observation $\boldsymbol{d}$ can be seen as the constant input to all iterations. The parameter $\boldsymbol{\theta}^{(i)}$ can be either temporally varying or constant, corresponding to a recurrent or feedforward system, respectively

$G:\mathbb{R}^m\to\mathbb{R}^n$ that maps input $\boldsymbol{d}_j$ to output $\boldsymbol{x}_j^{(T)}=G(\boldsymbol{d}_j;\boldsymbol{\Theta})$, where $\boldsymbol{\Theta}=\{\boldsymbol{\theta}^{(i)}\}_{i=1}^T$. With a proper metric function $\ell:\mathbb{R}^n\times\mathbb{R}^n\to\mathbb{R},(\boldsymbol{x}_j^{(T)},\boldsymbol{x}_j^*)\mapsto l_j=\ell(\boldsymbol{x}_j^{(T)},\boldsymbol{x}_j^*)$, we train $\boldsymbol{\theta}$ by minimizing a loss function

$$\min_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\Theta})=\sum_{j=1}^N c_j\cdot l_j=\sum_{j=1}^N c_j\cdot\ell(G(\boldsymbol{d}_j;\boldsymbol{\Theta}),\boldsymbol{x}_j^*),\tag{3.6}$$

where $c_j\geqslant 0$ is the coefficient of the weighted sum with $\sum_{j=1}^N c_j=1$. When $\boldsymbol{x}_j^*$ is not available, we can adopt an unsupervised metric function such as the objective function itself.

Motivations. Introduced to bridge the gap between classic optimization methods and black-box deep learning approaches, algorithm unrolling combines benefits from both domains. By unrolling an iterative optimization algorithm into a fixed-depth neural network, it can be trained to solve a specific distribution of optimization problems, often achieving significant acceleration over the original algorithm on similar problems. This learning ability allows the unrolled algorithm to adapt to the problem class and learn optimal parameters for faster convergence.

At the same time, algorithm unrolling inherits the mathematical structures and domain knowledge embedded in the classic optimization methods developed by experts. This inherited knowledge acts as a strong regularization for the neural networks, constraining the search space and guiding the learning process. Hence, the unrolled algorithms can be trained more efficiently, requiring fewer training samples compared with generic deep neural networks.

Why can algorithm unrolling be effective? We consider algorithm unrolling to be effective if the resulting network can, compared with the original algorithm, accelerate convergence to a desired solution or generate a solution with better quality. The key motivations for the effectiveness are two-fold.

In one scenario, the optimization problems in a specific application are usually drawn from a concentrated distribution. For example, the authors of [68] studied sparse-coding-based image reconstruction on the MNIST dataset [104]. A common set of basis vectors is used for all images from MNIST; hence the reconstruction problems are the same but have different input images. The classic algorithms for these problems such as gradient-projection iterations work for a much larger class of problems and are not tailored to the specific reconstruction problems where the images to reconstruct share a common linear basis. Algorithm unrolling can learn from the training data about the commonality and yield a neural network that takes a fast "shortcut" to the solution not known by human experts.

The other scenario arises where the optimization formulations derived by human experts fail to fully capture the intricacies and complexities of real-world applications. Consequently, the exact minimizer of such an optimization problem may yield suboptimal solutions. A prime example of this limitation is the use of total variation regularization in image processing, which removes noise in flat regions while preserving sharp edges but also introduces undesirable staircase artifacts in the resulting image. Total variation regularization and subsequent improvements only partially model the inherent properties of natural images. By applying algorithm unrolling, which significantly relaxes the original algorithm, we can discover effective corrections to the original formulation's algorithms and thus capture the nuances of data-specific information, which in turn yield high-quality solutions in practical applications, overcoming the limitations of traditional, analytic formulations.

3.2 A recipe for algorithm unrolling — Learned ISTA as an example

Given a specific task, we lay out a general outline of unrolling an iterative algorithm into a neural network for solving a specific type of optimization problems for the task:

Step 1. Set up a proper modeling of the task as an optimization problem and identify a classic iterative algorithm for solving the modeled optimization.

Step 2. Unroll the iterative algorithm over time into the form of a recurrent system as in (3.3) and truncate it a finite number of steps.

Step 3. Parameterize the unrolled recurrent system, that is to make part of the recurrent update form learnable by directly replacing them with learnable parameters or generating them with a neural network with learnable parameters.

Step 4. Train the unrolled and parameterized network with data.

The first and second steps are intuitive in most cases, especially when the original goal is to accelerate a specified algorithm for a specific task, such as accelerating ISTA for sparse coding problems in [68] and projected gradient descent for Multiple-Input-Multiple-Output (MIMO) detection in [137].

The fourth step can play an important role in the empirical success of a neural network derived with algorithm unrolling. To train the network derived effectively, one needs a proper training objective or loss function that is suitable for the task of interest, an effective training algorithm with necessary regularization, and other training techniques that helps to stabilize or accelerate the training process. We will present the basic training scheme and part of the techniques via the case study in this subsection and have a more detailed discussion in Subsection 3.3.

The third step, i.e., the parameterization step, is the essential yet the most challenging step in the methodology of algorithm unrolling. A proper parameterization is fundamental to the easiness of training later and to the empirical performance and acceleration that can be expected during inference, but at the same time, requires careful consideration of a number of design elements. These design elements directly influence the architecture of the resulting network and thus its empirical performance. There is no one "correct" way of parameterization, but it should be adaptive to the task of interest, including its objective, data format, and common structures shared among the data.

For example, *iterative shrinkage thresholding algorithm* (ISTA), which was first unrolled in [68], now has more than ten versions with different parameterizations and the resulting neural networks. These variants have been proposed for varying purposes with adaptation for specific tasks. Some variants have special parameter representation for improved trainability and interpretability [2,39,122]; some variants introduce additional structures or modules so that the resulting network can generalize to a larger range of data without re-training [1,19,40]; and other variants replace certain parts of the unrolled algorithms with more advanced deep learning modules [182]. For its richness in the literature, ISTA is a perfect example for algorithm unrolling to explain the multiple aspects of network design.

A case study with learned ISTA. We adopt the original *learned ISTA* (LISTA) network [68] as a case study to go over the four steps laid out in the recipe. We choose LISTA [68] as the case study example because it is the first algorithm unrolling work and the most impactful one in the sense that the algorithm unrolled in [68], ISTA, has been investigated most frequently in the literature.

Step 1. Set up the optimization model and algorithm. The first step is to set up the optimization model and identify a classic iterative algorithm for solving it. In [68] the authors strove at learning a fast approximation to sparse coding problems by unrolling ISTA. Given an observation $\boldsymbol{d} \in \mathbb{R}^m$ as input, the sparse coding problem is usually formulated as finding the optimal sparse vector $\boldsymbol{x}^* \in \mathbb{R}^n$ that minimizes the LASSO objective function, which is a weighted sum of the squared reconstruction error and an ℓ_1

sparsity regularization, formulated as

$$\text{Find} \quad \boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \frac{1}{2}\|\boldsymbol{d} - \boldsymbol{A}\boldsymbol{x}\|_2^2 + \lambda\|\boldsymbol{x}\|_1. \tag{3.7}$$

Here, $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ is the given dictionary that is often overcomplete $(m < n)$ and λ is a hyperparameter that controls the regularization strength, which is usually tuned by hand on a set of sparse coding problems. ISTA algorithm is an iterative algorithm for solving (3.7). In each iteration, ISTA essentially takes a gradient descent step with respect to the squared reconstruction error term with a step size of α and then shrink the non-zero entries towards zero through a so-called soft-thresholding function $\eta(\cdot;\rho)$ with a positive threshold ρ,

$$\begin{aligned} \text{Classic ISTA [17]:} \quad & \boldsymbol{x}^{(i)} = \eta(\boldsymbol{x}^{(i-1)} - \alpha\boldsymbol{A}^\top(\boldsymbol{A}\boldsymbol{x}^{(i-1)} - \boldsymbol{d}); \rho), \quad i = 1, \dots, \infty, \\ & \boldsymbol{x}^{(0)} = \boldsymbol{0}, \end{aligned} \tag{3.8}$$

where $\eta(z;\rho) = \mathrm{sign}(z)(|z| - \rho)_+$ is called *soft-thresholding function* which is applied to the input vector in a coordinatewise manner. With a proper selection of the step size and the threshold, ISTA algorithm is provably converging to $\boldsymbol{x}^*$ at a sublinear speed of $\mathcal{O}(1/i)$. [2)]

Step 2. Unroll the iterative algorithm and truncate. In [68], the authors rearranged and merged terms in (3.8) and truncate the iteration to a fixed number steps, say T, turning the algorithm into the following form:

$$\begin{aligned} \text{LISTA [68]:} \quad & \boldsymbol{x}^{(i)} = \eta(\boldsymbol{W}_1\boldsymbol{x}^{(i-1)} + \boldsymbol{W}_2\boldsymbol{d}; \rho), \quad i = 1, \dots, T, \\ & \boldsymbol{W}_1 = \boldsymbol{I}_n - \alpha\boldsymbol{A}^\top\boldsymbol{A}, \quad \boldsymbol{W}_2 = \alpha\boldsymbol{A}^\top, \quad \boldsymbol{x}^{(0)} = \boldsymbol{0}. \end{aligned} \tag{3.9}$$

Step 3. Parameterize the unrolled system into a neural network. With the unrolled and truncated system in (3.9), the authors in [68] directly converted the $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ matrices and the threshold ρ into learnable parameters that are shared across all layers, thus turning ISTA into a recurrent neural network. We denote the learnable parameters of the resulting RNN as the concatenation of the two matrices and the threshold $\boldsymbol{\theta} = (\boldsymbol{W}_1, \boldsymbol{W}_2, \rho)$. Note that while we allow the parameters to be updated through data-driven training, we can still initialize them in the same way as we set them in standard ISTA. This initialization scheme provides a good starting point that is already a good approximation to ISTA in the first place, which can help the training process in the next step.

Besides directly converting $\boldsymbol{W}_1$, $\boldsymbol{W}_2$ and ρ into learnable parameters and sharing them across layers, there are many other ways of parameterization resulting in different variants of LISTA. For example, a simple extension to the original recurrent version of LISTA in [68] is to *untie* $\boldsymbol{W}_1$, $\boldsymbol{W}_2$ and ρ and instead learn layer-dependent parameters $(\boldsymbol{W}_1^{(i)}, \boldsymbol{W}_2^{(i)}, \rho^{(i)})$ for all $i \in \{1, \dots, T\}$. This untied parameterization strategy is widely adopted in following works such as improved LISTA for sparse inverse recovery [39] and algorithm unrolling for other algorithms [29]. The main benefit of this strategy is that it radically increases the number of learnable parameters in the resulting network, lifting the *complexity of parameterization* from $\mathcal{C}(m(m+n))$ in [68] to $\mathcal{C}(Tm(m+n))$.[3)] The increased complexity in parameterization enables the neural network to learn more complicated mappings and thus expect potentially better acceleration, but it also makes the training process more difficult and requires special techniques for a stabilized training process to actually achieve that expected acceleration. We will have a thorough discussion in the next subsection about the trade-offs between various parameterization strategies.

Step 4. Train the resulting neural network. To set up a proper training process, we first need to generate a training set. In [68], the authors assumed that all sparse coding problems of their interest are *drawn from the same distribution.* To be specific, the inputs, $\{\boldsymbol{d}_j\}_{j=1}^N$, are N image patches extracted

2) Typically for ISTA, we set the step size $\alpha = 1/L$, where L is the largest eigenvalue of $\boldsymbol{A}\boldsymbol{A}^\top$ and threshold $\rho = \lambda/L$.

3) We use a different notation of $\mathcal{C}(\cdot)$ to represent the complexity of parameterization to distinguish it from the time complexity of computation for an algorithm or a neural network, which is denoted with $\mathcal{O}(\cdot)$.

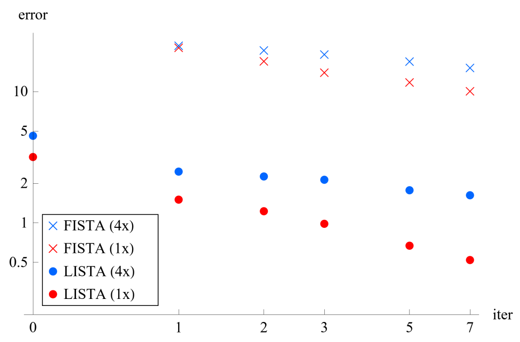


Figure 3 (Color online) Comparison of LISTA an FISTA for solving sparse coding problems on the testing set. It takes 18 iterations of FISTA to reach the error for LISTA with just one iteration for $n = 100$, and 35 iteration for $n = 400$. *Plot source*: Figure 3 of [68]

from one dataset. All sparse coding problems share the same dictionary matrix $\boldsymbol{A}$, which is learned beforehand on the same dataset with an online dictionary learning algorithm.

As we elaborated in Step 1, the objective of the original LISTA is to learn a fast approximation to optimal solutions of sparse coding problems, i.e., the minimizers of the LASSO objective $\boldsymbol{x}^*$ in (3.7). Therefore, the loss function for training targets at regressing towards $\boldsymbol{x}^*$. The authors of [68] trained LISTA networks in a supervised setting, where they adopted a coordinate descent algorithm to solve the sparse coding problem (3.7) exactly to generate the optimal sparse code $\boldsymbol{x}_j^*$ given each input $\boldsymbol{d}_j$. In this way, a training set of N input-label pairs $\mathbb{D}_{\text{train}} = \{(\boldsymbol{d}_j, \boldsymbol{x}_j^*)\}_{j=1}^N$ is generated. LISTA networks are then trained on this training set to minimize the mean squared error (MSE) between the approximated sparse codes and the ground-truth sparse codes,

$$\min_{\boldsymbol{\Theta}} \mathcal{L}(\boldsymbol{\Theta}) = \sum_{j=1}^{N} \|\boldsymbol{x}_j^{(T)} - \boldsymbol{x}_j^*\|_2^2, \tag{3.10}$$

where $\boldsymbol{\Theta} = \{(\boldsymbol{W}_1^{(i)}, \boldsymbol{W}_2^{(i)}, \rho^{(i)})\}_{i=1}^T$ involves all the parameters to learn and $\boldsymbol{x}_j^{(T)}$ is dependent on $\boldsymbol{\Theta}$ since it is computed using (3.9) with $\boldsymbol{d}_j$ being the input. In practice, the network is trained with stochastic optimization algorithms such as SGD and Adam, where training data are fed in mini-batches randomly sampled from the whole training set to calculate the loss function values and the gradients with respect to the parameters through backpropagation.

Deploy and evaluate. After the offline training is finished, the final act is the deployment of the trained network for solving new optimization problems. To evaluate the empirical performance of a trained LISTA network from the last step, we first prepare a testing set in the same way as we do for the training set — all problems in the testing set share the same dictionary but the input vectors are different yet unseen during training. Figure 3 (from Figure 3 in [68]) shows the evaluation of LISTA on the Berkeley image database, where the input vectors are 10×10 image patches and two dictionary matrices are learned: one dictionary is $1\times$ complete, i.e., $m = n = 100$, presented in red in the figure; the other is $4\times$ complete, i.e., $n = 4m = 400$, presented in blue. LISTA is compared with FISTA [17], an extension to ISTA accelerated with momentum, in terms of how fast the prediction error decreases as iterations go on. It is reported in [68] and we quote here that *it takes* 18 *iterations of FISTA to reach the error for LISTA with just one iteration for* $n = 100$, *and* 35 *iteration for* $n = 400$. *Hence one can say that LISTA is roughly* 20 *times faster than FISTA for approximate solutions.*

3.3 Design elements, techniques and applications

In this subsection, we overview the various design elements, techniques, and applications that were developed to meet specific practical requirements. Researchers have created a rich body of literature by developing diverse design elements. We will explore the key design elements featured in the literature, discussing their origins, the impact of different design choices, and their advantages and disadvantages. To help improve the understanding of these design elements, we will establish connections with practical examples from the literature. We will also discuss the training techniques known to be effective for algorithm unrolling.

Share parameters across unrolled iterations? The first design element to consider when unrolling an iterative algorithm is: *do we share the parameters across layers in the resulting network*? Although it is trained and tested with a fixed number of iterations, the original LISTA [68] unrolls ISTA into an RNN where all iterations share the same set of learnable parameters. In this case, we also say that the network has *tied weights* in the literature. As the authors stated in [68], the recurrent system derived with unrolling, if truncated to a fixed number of iterations for all inputs, can also be viewed as a feedforward neural network of a fixed number of layers with tied weights. Based on this perspective, most work that followed [68] proposed to unroll algorithms into feedforward networks by breaking the parameter sharing design, which means the learnable parameters in different layers are *untied* and learned independently [29, 39, 122]. The untying process converts the recurrent formulation (3.9) into an MLP-type architecture:

$$\text{LISTA untied:}\quad \begin{aligned} &\boldsymbol{x}^{(i)} = \eta(\boldsymbol{W}_1^{(i)}\boldsymbol{x}^{(i-1)} + \boldsymbol{W}_2^{(i)}\boldsymbol{d}; \rho^{(i)}), \quad i = 1, \dots, T, \\ &\boldsymbol{x}^{(0)} = \boldsymbol{0}. \end{aligned} \tag{3.11}$$

In this feedforward formulation, we denote the set of learnable parameters as $\boldsymbol{\Theta} = \{(\boldsymbol{W}_1^{(i)}, \boldsymbol{W}_2^{(i)}, \rho^{(i)})\}_{i=1}^{T}$. The two types of networks are both valid choices for algorithm unrolling but they have their pros and cons depending on the scenarios and should be decided accordingly to make the most out of L2O.

It is worth noting that networks derived with algorithm unrolling actually process non-sequential inputs in most L2O applications despite them being RNNs. For example, in the case of unrolling gradient descent that we formulated in (3.3) and the original LISTA [68], the input to the network is a single vector $\boldsymbol{d}$, which is the input to the corresponding optimization problem. This means that the resulting RNNs only exploit the idea of parameter sharing and the recurrent structure, and there is no temporal correlation in inputs for them to capture. In some applications, however, the recurrent parameterization still has certain advantages over the non-recurrent counterpart.

To be more specific, an RNN and a feedforward network derived with algorithm unrolling differ in different aspects:

• *Capacity.* An RNN uses shared parameters across time and thus is compact in terms of the overall number of learnable parameters. In contrast, a feedforward network can have different parameters in each layer, independently learned from data. The total number of learnable parameters is thus proportionally increasing as the network goes deeper. This discrepancy in the compactness of parameterization results in differences in the *capacity* of the resulting networks, by which we mean the ability of the network to approximate complex mappings when the learnable parameters take proper values [82]. With a larger of amount of parameters, feedforward networks have greater capacity to learn more complex mappings when compared with RNNs.

On the other hand, however, the compact parameterization of an RNN brings benefits at the hardware level, e.g., lower storage and memory requirements for containing the trained network and for the computations. The hardware efficiency allows RNNs to be deployed on resource-constrained platforms such as edge devices [73, 190] and distributed or Internet-of-Things (IoT) systems [191].

• *Flexibility.* An essential benefit of the idea of parameter sharing in RNNs is the *flexibility* of applying them to inputs of varying lengths and generalize among them. While an RNN is truncated to a fixed number of iterations during training in [68] and most algorithm unrolling works, it is straightforward to apply it for an arbitrary number of iterations for inference on new optimization problems. In fact, there is a natural need for such flexibility in L2O networks because some optimization problems are intrinsically more difficult than others and thus require more iterations to converge well. On the other hand, feedforward networks obviously lack such flexibility. Once the number of layers is decided during unrolling, the number of parameters is also fixed, and there is not a trivial way to properly apply parameters in previous layers to later layers. A simple extension that allows a trained feedforward network to process longer iterations is to append iterations after the last layer using a classic iterative algorithm [78].

• *Trainability.* RNNs are generally considered harder to train compared with feedforward networks, especially when they are trained with long sequences or a large number of iterations in the case of algorithm unrolling [126]. This difficulty of training is caused by gradient vanishing (the more common case in practice) and explosion issues. Consider an RNN whose recurrent layer has an ill-conditioned Jacobian matrix. Gradients will be scaled downwards or upwards when backpropagated to earlier time steps with the Jacobian matrix multiplied to the gradients repeatedly. Practically, the gradient explosion issue can be mitigated with gradient clipping. However, alleviating the gradient vanishing issue for an RNN usually requires modifications to the architecture, which is impractical in algorithm unrolling because the architecture is decided by the original algorithm. In contrast, feedforward networks are generally easier to train but certain training tricks are also needed in some cases, especially when the resulting networks are deep and thus have too many learnable parameters. We will have a further discussion on training techniques later.

How to parameterize the resulting network? This is the core question that practitioners need to answer when adopting the algorithm unrolling methodology. The answer to this question decides the architecture of the resulting network, thus its capacity, trainability and empirical performance achievable in the end. To find the better way of parameterization for the task of interest, we need to carefully consider of a number of factors. While it is impractical and overwhelming to list every piece of work in the literature and describe how its parameterization is done, we list three major factors that have driven different designs of parameterization: (i) data structure, (ii) task goal, and (iii) interpretability. Here, we limit the discussion to algorithm unrolling methods that strictly follow the formula of the original iterative algorithms. Variants with more advanced and more heavily parameterized deep learning extensions are elaborated in discussions of other design elements below.

– *Data structure* describes the inherent complexity of the data distribution of the task that we strive to tackle using algorithm unrolling. It has direct influence on the decision of parameterization because the resulting network should at least provide the capacity to cope with that data complexity. The parameterization design of the original LISTA [68] already reflects such consideration, where the authors proposed to use the same learnable matrices $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ for all sparse coding problems in the dataset. The rationale behind this parameterization strategy is rooted at the assumption that all sparse coding problems share the same dictionary $\boldsymbol{A}$, which decides the value of $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ in classic ISTA iterations as formulated in (3.9). Therefore, it is reasonable to learn the same $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ for all problems so that some *common shortcut* can be utilized through data-driven learning from those optimization problems.

However, if the data structure is more complex, e.g., the optimization problems use varying dictionaries, the original LISTA might fail to generalize. To deal with this increased data complexity, the authors of [1] proposed Ada-LISTA (standing for Adaptive LISTA) which *augments* ISTA by replacing the input dictionary $\boldsymbol{A}$ in each iteration with two matrices $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ that are linearly transformed from $\boldsymbol{A}$ with two learnable matrices $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$:

$$\begin{aligned}\text{Ada-LISTA [1]:}\quad &\boldsymbol{x}^{(i)} = \eta((\boldsymbol{I}_n - \alpha^{(i)}\boldsymbol{G}_1^\top\boldsymbol{G}_1)\boldsymbol{x}^{(i-1)} + \alpha^{(i)}\boldsymbol{G}_2^\top\boldsymbol{d};\ \rho^{(i)}),\quad i=1,\dots,T,\\ &\boldsymbol{G}_1 = \boldsymbol{W}_1\boldsymbol{A},\quad \boldsymbol{G}_2 = \boldsymbol{W}_2\boldsymbol{A},\quad \boldsymbol{x}^{(0)} = \boldsymbol{0}.\end{aligned} \tag{3.12}$$

Here, the parameter matrices $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ are shared across layers but the layerwise learnable step sizes and thresholds are untied, similar to (3.11). It is worth noting that in the formulation of Ada-LISTA the input to the network now is an observation-dictionary pair $(\boldsymbol{d}, \boldsymbol{A})$. This augmentation endows Ada-LISTA enough capacity to learn from and generalize among sparse coding problems with varying dictionaries. But this capacity is still limited. The authors showed theoretically and empirically that Ada-LISTA can adapt to variants of a clean dictionary perturbed with column permutations or (small) random noises. If we expect the dataset contains more general sparse coding problems, further improvement to the network is necessary.

– *Task goal* is the target that we expect the resulting network to yield. In the original LISTA [68] and many following works [1, 2, 122], the task goal is to learn fast approximation to the minimizer of the LASSO objective $\boldsymbol{x}^*$ as we denoted in (3.7). In some applications, the goal is to recover an unknown signal, which we denote as $\bar{\boldsymbol{x}}^* \in \mathbb{R}^n$ to distinguish it from the minimizer of an objective function. The recovery is based on its observation generated with a forward model. A simple example of a forward model is a linear mapping perturbed with noise:

$$\boldsymbol{d} = \boldsymbol{A}\bar{\boldsymbol{x}}^* + \epsilon, \tag{3.13}$$

where $\boldsymbol{A}$ represents the forward linear mapping and ϵ is the random noise. When $\bar{\boldsymbol{x}}^*$ is sparse, the minimizer of LASSO $\boldsymbol{x}^*$ is a decent approximation to $\bar{\boldsymbol{x}}^*$ when the ℓ_1 penalty level λ is properly selected. However, $\boldsymbol{x}^*$ is not a perfect recovery of $\bar{\boldsymbol{x}}^*$ due to the bias introduced by the uniform ℓ_1 penalty on all coordinates [57]. Therefore, the two task goals are subtly different.

The difference in task goal sometimes requires a change of the parameterization strategy. For example, instead of minimizing LASSO, the authors of [39] unrolled ISTA into a feedforward network, aiming at solving the linear sparse recovery problems formulated in (3.13) with the sparsity assumption on $\bar{\boldsymbol{x}}^*$. They theoretically showed that the $\boldsymbol{W}_1^{(i)}$ and $\boldsymbol{W}_2^{(i)}$ matrices in LISTA (3.11) need to asymptotically satisfy a coupling relationship, i.e., $\boldsymbol{W}_1^{(i)} - (I - \boldsymbol{W}_2^{(i)}A) \to \boldsymbol{0}$ when $i \to \infty$, so that the output of the network converges to $\bar{\boldsymbol{x}}^*$ as the number of layers increases. With this result, the authors proposed to sustain this coupling structure throughout all layers, leading to the following parameterization: 4)

$$\text{LISTA-CP [39]:} \quad \begin{aligned} &\boldsymbol{x}^{(i)} = \eta(\boldsymbol{x}^{(i-1)} + {\boldsymbol{W}^{(i)}}^\top(\boldsymbol{d} - \boldsymbol{A}\boldsymbol{x}^{(i-1)});\ \rho^{(i)}), \quad i = 1, \dots, T, \\ &\boldsymbol{x}^{(0)} = \boldsymbol{0}. \end{aligned} \tag{3.14}$$

In contrast, it is shown in [2] that, however, if the resulting network is expected to converge to the LASSO minimizer, it must *tend towards ISTA iterations* up to a learnable step size $\alpha^{(i)}$ in each layer. The threshold of the soft-thresholding function is set to $\rho^{(i)} = \alpha^{(i)}\lambda$ accordingly:

$$\text{Step-LISTA [2]:} \quad \begin{aligned} &\boldsymbol{x}^{(i)} = \eta(\boldsymbol{x}^{(i-1)} + \alpha^{(i)}\boldsymbol{A}^\top(\boldsymbol{d} - \boldsymbol{A}\boldsymbol{x}^{(i-1)});\ \alpha^{(i)}\lambda), \quad i = 1, \dots, T, \\ &\boldsymbol{x}^{(0)} = \boldsymbol{0}. \end{aligned} \tag{3.15}$$

We refer the reader to [37] for more works in the literature targeted at the two different task goals.

– *Interpretability* is demanded in the field of algorithm unrolling for understanding how the acceleration is achieved via data-driven learning or for a more transparent computation process than black-box deep learning. The derivation of an interpretable network might bring up an architecture with special structures as a by-product and thus a new way of parameterization.

For example, the authors of [122] explained the acceleration of LISTA from the perspective of matrix factorization on the Gram matrix of the dictionary $\boldsymbol{A}$. This perspective can derive a re-parameterized variant of the original LISTA which achieves similar convergence rate. In the case of linear sparse recovery, the authors of [108] theoretically showed that all layers in the coupled LISTA formulated in (3.14) can use the same matrix $\tilde{\boldsymbol{W}}$ up to a learnable step size. Instead of being trained with data, $\tilde{\boldsymbol{W}}$ is analytically generated by solving a normalized mutual coherence minimization with respect to the dictionary $\boldsymbol{A}$. This analytical formulation leaves only the layerwise step sizes and thresholds for learning: 4)

$$\text{ALISTA [108]:} \quad \begin{aligned} &\boldsymbol{x}^{(i)} = \eta(\boldsymbol{x}^{(i-1)} + \alpha^{(i)}\tilde{\boldsymbol{W}}^\top(\boldsymbol{d} - \boldsymbol{A}\boldsymbol{x}^{(i-1)});\ \rho^{(i)}), \quad i = 1, \dots, T, \\ &\boldsymbol{x}^{(0)} = \boldsymbol{0}. \end{aligned} \tag{3.16}$$

When to stop the iteration? While it is most standard to have an algorithm unrolled and truncated to a fixed number of iterations, it is usually needed to have a more sophisticated and adaptive strategy to decide how many layers or iterations are needed for a specific input. This need is well-motivated by

4) We omit the usage of the advanced soft-thresholding technique called *support selection* in [39] for simplicity.

the fact that optimization problems are not difficult to the same level. For problems that are easier to solve, it is possible to approximate the solution with only a few layers, while more iterations are needed for other more difficult problem instances. Similar motivation stirred the dynamic inference direction in the field of deep learning, where the researchers also want the network learn to *exit early* if the input is easy [155]. This direction is also related to the optimal stopping time in control theory.

Another approach is to design a dynamic mechanism for algorithm unrolling to achieve this goal. For example, the authors of [189] designed a stopping policy based on reinforcement learning (RL) that dynamically decides the stopping time of a recurrent process for image restoration. The stopping policy is learned with Deep Q-Learning [120]. A similar RL-based approach is adopted in a plug-and-play method [165] for solving inverse problems in the image domain, where the authors learn a CNN-based policy network that simultaneously predicts the stopping probability and the parameters of the outer iterative framework. The network is trained using the actor-critic framework [151]. While it is designed for a plug-and-play method, we believe the dynamic stopping policy in [165] is still valuable for the practitioners in algorithm unrolling to develop their own dynamic strategy.

Besides RL-based methods, the authors of [38] proposed a dynamic stopping strategy implemented with a *variantional auto-encoder* (VAE) that learns when to terminate iterations early. The authors interpreted their proposed framework from a variantional Bayes perspective and connected it with RL. However, their method is still based on a neural network truncated to fixed *max length* and only learns how to exit early, resulting in the lack of flexibility to extend to longer iterations.

Domain-specific designs. When algorithm unrolling comes to certain domains, we can do more than simply strictly following the original formulation of the iterative algorithm when unrolling and parameterizing it. Instead, we can customize parts of the resulting network to satisfy specific needs emerging in those domains. While the customization comes with the cost of losing the strict correspondence between the original algorithm and the neural network derived, which slightly undermines its interpretability, we can expect much better empirical performance.

One scenario where special designs are needed is in the field of wireless communication systems and scientific imaging where we need processing real-valued data and optimization problems. However, most modern deep learning frameworks do not support processing complex-valued input out of the box. To make algorithm unrolling still applicable in such applications, for example, the authors of [13] used real-value decomposition to transform the original problem formulation so that a project gradient descent algorithm can be unrolled for solving MIMO detection problems in wireless communication systems.

Another important and ubiquitous example is the exploitation of convolutional neural networks in algorithm unrolling. CNNs have been proved to be extremely effective for image-based tasks thanks to the great properties of convolutional layers that we discussed in Subsection 2.2. Therefore, when algorithm unrolling is applied to such tasks, it is well-motivated to combine it with CNNs. For example, in [182], the authors unrolled ISTA algorithm for natural image compressive sensing. The optimization-based compressive sensing minimizes a LASSO-type objective but assumes the image signal has a sparse representation under *some linear transformation*, say Ψ, such as DCT and wavelet transformations. However, this assumption only approximately holds and barely captures the complexity of natural images. Therefore, the authors of [182] proposed to insert two separate two-layer CNNs, $\mathcal{F}^{(i)}$ and $\tilde{\mathcal{F}}^{(i)}$, into each layer of the network, with $\mathcal{F}^{(i)}$ replacing the Ψ transformation and $\tilde{\mathcal{F}}^{(i)}$ replacing the backward transformation, which is usually $\Psi^\top$ in classic algorithms. For better regularization and interpretability, the authors also proposed to penalize $\|\tilde{\mathcal{F}}^{(i)} \circ \mathcal{F}^{(i)}(\boldsymbol{x}^{(i)}) - \boldsymbol{x}^{(i)}\|$ during training so that the learned $\mathcal{F}^{(i)}$ and $\tilde{\mathcal{F}}^{(i)}$ are encouraged to follow the symmetry constraint.

This combination of algorithm unrolling and more advanced CNN modules has been empirically proved to be successful on the task of natural image compressive sensing [182], and been later applied to many other image-domain tasks, especially the field of biomedical imaging, e.g., CT imaging [70, 96], photo-acoustic tomography (PAT) [74], magnetic resonance imaging (MRI) [129], ultrasound imaging [145] and Shepp-Logan phantom and human phantoms [4]. Another case of introducing CNNs to algorithm unrolling is where the optimization problems are themselves modeled with convolutional operators, such as convolutional sparse coding [150], which is later extended in [108].

Instance adaptivity. Neural networks derived from algorithm unrolling typically learn parameters corresponding to components in the original algorithm, such as step sizes in gradient descent and thresholds in ISTA. Once the training is completed, the set of learned parameters is fixed. However, the optimal choices for these components are often instance-specific. To address this, some works propose adaptively generating these components for each instance using information from the current iteration or past optimization trajectory, a property known as *instance adaptivity*. Two main methodologies exist for instance-specific generation.

One way is to adopt neural network-based augmentations. The authors of [19] proposed an extension to ALISTA (3.16) that generates step sizes and thresholds using an LSTM module. The LSTM network takes the ℓ_1 norm of the residual of the current iterate and the ℓ_1 norm of the residual transformed with the $\tilde{\boldsymbol{W}}$ matrix analytically generated as in [108]. The outputs of the LSTM network are two scalars. This method is based on the authors' observation on the correlation between the ℓ_1 norm of the residual and the ℓ_1 norm of the difference between the current iterate to the ground truth signal in each iteration. A recent work [109] also utilizes an LSTM network for generating the diagonal matrix that preconditions the gradient step, the additive bias term, and the step size of the momentum term. The authors derived the basic mathematical conditions that successful update rules commonly satisfy. The overall framework derived in this way shows great adaptivity and even the ability to generalize cross datasets and tasks.

Another method for generating the instance-specific components for iterations is via analytical derivation which guarantees theoretical convergence. The authors of [40] introduced a new ALISTA parameterization augmented with momentum acceleration for the sparse recovery task, and then analytically derived the formula of the instance-optimal key components of the ALISTA algorithm (3.16), i.e., the step size, soft-thresholding threshold, step size of the momentum term and size of the selected support. This formulation leaves reduces the number of learnable parameters in ALISTA to only tuning three scalars but endows the resulting network with great adaptivity.

Training techniques. Here we have a brief introduction to the selection of training objective and regularization techniques in algorithm unrolling training.

– *Training objective* mainly depends on the task goal, which we have discussed about its relationship with the design of parameterization. Task goal also decides the training objective and the loss function utilized for training. We still take LISTA as an example, which can be trained to either (i) approximate the LASSO minimizers, or (ii) recover unknown signals.

We generally have two ways to achieve the first goal. One approach is to generate optimal solutions of the optimization problems in the training set using a third-party solver, which for sure induces computation overheads during dataset generation and requires the precision of the solver. The neural network is then trained to regress the generated solutions using a regression loss function, e.g., the mean squared error (MSE). The second approach is to directly adopt the LASSO objective function as the loss function and to train the network to minimize the objective value at the output of the whole network. In this case, however, extra caution is required for potential numerical stability issues, especially when the objective includes logarithmic operations. In the case of LASSO, we have observed numerical issues when the ℓ_1 penalty level λ is large (around or larger than 10^{-1}). Our conjecture is that the absolute value function consistently yield gradients with magnitude 1, which is likely to be scaled upwards and to cause gradient explosion during backpropagation.

For the second goal, it is straightforward to train the neural network in a supervised regression manner as long as the ground truth signals are accessible for training. However, this accessibility is forbidden in certain real-world applications. Two workarounds for this limitation are: (1) we can change the formulation of the optimization problems so that the optimal solutions approximate the desired signals better, and then fall back to the previous case; or (2) we can train the neural network on synthesized problems whose ground truth signals are generated instead of solved. The latter approach requires the neural network to have good generalization ability to generalize from synthesized problems to those in real-world scenarios.

– *Regularization techniques* are designed for more stabilized training process and better generalization ability of deep neural networks. Firstly, most of standard deep learning regularization techniques introduced in Subsection 2.3 can also be used for algorithm unrolling training, such as early stopping when the network performance stops improving on a hold-out validation set. Secondly, customized regularization terms can be added to the training loss function to encourage sustaining special structure in the neural network derived. A good example is the regularization term added for training ISTA-Net [182] which enforces the composition of the forward and backward convolutional modules to approximate identity mapping, as we discussed earlier in the Domain-specific designs paragraph. Moreover, special training schemes can also serve as implicit regularization on the training process. For example, a stage-wise training strategy with learning rate decaying was adopted in [29, 39] for training resulting networks with untied parameters, which was empirically shown to help with stabilizing the training and achieving better performance.

Applications. The application of algorithm unrolling extends across a diverse array of fields, where its implementation has been markedly successful. Particularly in the area of image restoration and reconstruction, a burgeoning literature has emerged, reflecting its widespread applicability. Within this field, algorithm unrolling has been utilized in classical low-level restoration tasks, such as denoising [185], deblurring [48], inpainting [1], super-resolution [63] and video background subtraction [34]. It also plays a crucial role in image artifacts removal tasks like JPEG artifact reduction [164]. Furthermore, reconstruction tasks, e.g., natural image compressive sensing [116, 182], have also benefited from the application of this method. The scope of algorithm unrolling is not restricted to the aforementioned areas but also encompasses scientific imaging domains, including seismic, medical, and biological imaging. An exhaustive and well-integrated review of algorithm unrolling in the context of biomedical imaging can be found in [105]. Additionally, this technique has found resonance in wireless communication systems, offering solutions to various challenges such as resource management [153], channel estimation and signal detection [76], and LDPC coding [162]. A seminal survey that elaborates on the methods of algorithm unrolling within communication systems can be referred to in [13].

3.4 Mathematics behind algorithm unrolling

In algorithm unrolling techniques, iterative optimization algorithms, or their generalized variants, are treated as deep neural networks. This raises many interesting mathematical problems, some of which are still open or not well-studied. In this subsection, we present some of them. For more aspects on related topics, readers can refer to [37, 121, 138].

Model selection. Let us recall the recipe introduced above. The first step is to set up the optimization model and its associated iterative scheme. The optimization problem is naturally determined by the user's interest, while determining the specific formula of the iterative scheme is an art. Within the general framework presented in Subsection 3.1, the challenge lies in specifying the operator $\boldsymbol{g}$ in (3.5). Clearly, not all operators are suitable. Some must be dismissed, such as:

- $\boldsymbol{g}(\boldsymbol{d}, \boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{x} + \boldsymbol{1}$. This operator does not yield a fixed point.
- $\boldsymbol{g}(\boldsymbol{d}, \boldsymbol{x}; \boldsymbol{\theta}) = 2\boldsymbol{x}$. Iteration through this operator cannot converge unless the starting point is $\boldsymbol{x} = \boldsymbol{0}$.

A very recent study [109] marks an initial step in this direction. The authors study some basic conditions that such an operator $\boldsymbol{g}$ should satisfy. Based on these conditions, they proposed a structured iterative scheme that exhibits superior performance on convex problems.

Expressivity. Assuming the optimization model and associated iterative algorithm are determined, given an operator $\boldsymbol{g}$, consider the following two sequences:

- Sequence 1: $\{\boldsymbol{x}^{(i)}\}_{i=1}^{T}$ generated by (3.4).
- Sequence 2: $\{\boldsymbol{x}^{(i)}\}_{i=1}^{T}$ generated by (3.5).

This raises a question: are there parameters $\{\boldsymbol{\theta}^{(i)}\}_{i=1}^{T}$ in (3.5) such that Sequence 2 is significantly better than Sequence 1, provided that the parameters $\boldsymbol{\theta}$ in (3.4) is properly chosen? If not, algorithm unrolling may not be a wise option, as it induces extra cost of training without surpassing traditional algorithms. Fortunately, many works have demonstrated and quantified the potential advantages of algorithm unrolling over traditional methods. For example, in [39], the authors show in the context of LISTA that, a conventional algorithm typically converges in a sub-linear rate, while there exist parameters such that algorithm unrolling yields a linear rate. For more results, readers may refer to [1, 2, 40, 108, 152, 171, 174, 176, 180]. Note that these results only show the existence of desirable parameters, but do not guarantee such parameters can be obtained by training. In machine learning, such results are categorized as the expressive power of machine learning models, regarding unrolled algorithms as a deep neural network.

Training. Given the existence of parameters such that algorithm unrolling surpasses conventional methods, the question arises: Are there algorithms to compute such parameters with theoretical assurance? Specifically, one needs to solve minimization problems like (3.6), which is essentially a bi-level optimization. The lower-level optimization is defined by (3.5) and the upper-level optimization is given by (3.6). Although complete results with the optimality guarantee still lack, some studies like [31] identify some theoretical properties of the gradient of the loss function. Further references to related topics can be found in Subsection 5.2.

Generalization. Note that, even if problem (3.6) can be perfectly solved, there might still be a gap between the trained and ideal models. The gap results from the differences between the training set $\mathbb{D}_{\text{train}}$ (defined before (3.6)) and the upcoming new instances not in $\mathbb{D}_{\text{train}}$. To quantify this gap, one needs to create a testing set $\mathbb{D}_{\text{test}} = \{(\boldsymbol{d}_j, \boldsymbol{x}_j^\star)\}_j$, where all instances are independent of those in the training set. Then one can define the loss function on that testing set as

$$\mathcal{L}_{\text{test}}(\boldsymbol{\Theta}) = \sum_{(\boldsymbol{d},\boldsymbol{x}^\star)\in\mathbb{D}_{\text{test}}} \ell(G(\boldsymbol{d};\boldsymbol{\Theta}), \boldsymbol{x}^\star).$$

If a relation like

$$\mathcal{L}_{\text{test}}(\boldsymbol{\Theta}_\star) \leqslant \mathcal{L}(\boldsymbol{\Theta}_\star) + \delta, \quad \text{where } \boldsymbol{\Theta}_\star := \arg\min_{\boldsymbol{\Theta}} \mathcal{L}(\boldsymbol{\Theta})$$

can be established with a reasonable $\delta > 0$, the trained model is said to have robust generalization performance on instances not found in the training set. Luckily, tools in statistical learning such as Rademacher complexity analysis can confirm such results. Refer to [18, 41, 94, 102, 140] for more details.

4 Plug-and-play methods: DNN-assisted optimization model

In this section, we present the *plug-and-play* (**PnP**) framework with DNN assistance, which incorporates deep learning-based image denoisers into classic iterative algorithms. More precisely, this is achieved by substituting an analytic expression with a trained neural network (usually an image denoiser), followed by the immediate deployment of the modified algorithm to optimize the subjects derived from the identical task distribution. Additional training of the modified algorithm is optional.

4.1 An introduction to plug-and-play methods

Here, we provide a demonstration of the PnP framework, integrating a neural network with a classical optimization algorithm, the alternating direction method of multipliers (ADMM). Consider the optimization problem: $\text{minimize}_{\boldsymbol{x}\in\mathbb{R}^d} f(\boldsymbol{x})+g(\boldsymbol{x})$. While a direct minimization of $f+g$ can be challenging, minimizing f and g separately might be considerably simpler. Hence, one can alternately tackle f and g in an iterative manner. This is facilitated by introducing an auxiliary variable $\boldsymbol{y}$:

$$\underset{\boldsymbol{x},\boldsymbol{y}}{\text{minimize}}\, f(\boldsymbol{y}) + g(\boldsymbol{x}), \quad \text{subject to } \boldsymbol{x} = \boldsymbol{y}. \tag{4.1}$$

Subsequently, the ADMM is employed to solve (4.1) in an iterative fashion:

$$\boldsymbol{x}^{(i)} = \text{Prox}_{\beta g}(\boldsymbol{y}^{(i-1)} - \boldsymbol{u}^{(i-1)}) := \arg\min_{\boldsymbol{x}} \left\{ \beta g(\boldsymbol{x}) + \frac{1}{2}\|\boldsymbol{x} - (\boldsymbol{y}^{(i-1)} - \boldsymbol{u}^{(i-1)})\|^2 \right\}, \tag{4.2a}$$

$$\boldsymbol{y}^{(i)} = \mathrm{Prox}_{\alpha f}(\boldsymbol{x}^{(i)} + \boldsymbol{u}^{(i-1)}) := \arg\min_{\boldsymbol{y}} \left\{ \alpha f(\boldsymbol{y}) + \frac{1}{2}\|\boldsymbol{y} - (\boldsymbol{x}^{(i)} + \boldsymbol{u}^{(i-1)})\|^2 \right\}, \tag{4.2b}$$

$$\boldsymbol{u}^{(i)} = \boldsymbol{u}^{(i-1)} + \boldsymbol{x}^{(i)} - \boldsymbol{y}^{(i)}. \tag{4.2c}$$

In the above, (4.2a) and (4.2b) are the proximal operators for $f(\boldsymbol{y})$ and $g(\boldsymbol{x})$, where α and β are step sizes. (4.2c) updates the dual variable $\boldsymbol{u}$. Under mild assumptions on f and g, the global convergence of ADMM can be proved. Specifically, from any starting point $\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)}, \boldsymbol{u}^{(0)}$, the sequences $\boldsymbol{x}^{(i)}$ and $\boldsymbol{y}^{(i)}$ will converge to the optimal solutions of (4.1). One can refer to [30] for more details of ADMM.

Now, let us revisit the optimization problem (4.1) and ADMM in the context of image processing. For example, imagine a scenario where the goal is to recover an image, denoted as $\boldsymbol{x}^\star$, from its observations $\boldsymbol{b}$. The connection between $\boldsymbol{b}$ and $\boldsymbol{x}^\star$ can be expressed as

$$\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}^\star + \varepsilon,$$

where $\boldsymbol{A}$ models the observation physics, such as CT, and ε represents the noise during observation. For simplicity, we assume $\boldsymbol{A}$ is a known linear operator. To recover $\boldsymbol{x}^\star$ from $\boldsymbol{b}$, one can solve an optimization problem

$$\operatorname*{minimize}_{\boldsymbol{x}\in\mathbb{R}^d} \underbrace{(1/2)\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|^2}_{f(\boldsymbol{x})} + \underbrace{\lambda \mathrm{TV}(\boldsymbol{x})}_{g(\boldsymbol{x})}. \tag{4.3}$$

The function $f(\boldsymbol{x})$ measures the consistency of its input $\boldsymbol{x}$ with the data we observe. The function $g(\boldsymbol{x})$ serves as a regularization term that measures the noise level in $\boldsymbol{x}$ and the "TV" within $g(\boldsymbol{x})$ stands for total-variation[5] [134]. Therefore, the roles of the proximal operators can be summarized as

$$\begin{array}{rrcl} \mathrm{Prox}_{\beta g}: & \text{noisy image} & \mapsto & \text{less noisy image,} \\ \mathrm{Prox}_{\alpha f}: & \text{less consistent image} & \mapsto & \text{more consistent image with observation.} \end{array} \tag{4.4}$$

Note that the function f is determined by data, while the function g is designed manually. The design of the regularization function g reflects our understanding on the structural property we wish an ideal solution $\boldsymbol{x}^\star$ to possess. Given that g only provides a simple approximation to the complexity of real-world image structures, it raises the question: *why not supersede g or* $\mathrm{Prox}_{\beta g}$ *with a more effective denoiser* that might not originate from optimization? This leads us to the essence of the "plug-and-play" strategy: **plug** a stand-alone denoising operator into (4.2a):

$$\boldsymbol{x}^{(i)} = \mathrm{Denoiser}_\sigma(\boldsymbol{y}^{(i-1)} - \boldsymbol{u}^{(i-1)}), \tag{4.2a$'$}$$

and play as (4.2a$'$), (4.2b), and (4.2c). In (4.2a$'$), σ plays a similar role of λ in (4.3), denoting the estimated noise level in the input to the denoiser, which is accepted by most modern denoisers as a hyperparameter. The higher the noise level σ, the more aggressive denoising is performed.

Specialized denoisers, such as BM3D [49], is more sophisticated than TV, and thus is more promising to yield visually better results than solving (4.3). Today, one can consider a denoiser based on a well-trained deep neural network, such as DnCNN [184], within (4.2a), and witness remarkable performance in image-related tasks.

4.2 Histories, motivations and a recipe for DNN-assisted plug-and-play

History. The PnP methodology dates back to the work [160], where the authors creatively replaced one of the two proximal operators in ADMM with a *manually-designed denoiser* (such as non-local means [33], K-SVD [6], and BM3D [49]). The resulting ADMM, detailed in eqrefeq:pnp-admm-1, (4.2b), and (4.2c), exhibited greater performance than the original ADMM.

At the beginning, PnP had not been explicitly linked to deep learning or deep neural networks. It was not until the emergence of [118, 133, 185] that the notion of "learning operators" was integrated into

[5] For a 1D signal represented as $\boldsymbol{x} \in \mathbb{R}^d$, the total variation is given by $\sum_{i=1}^{d-1} |\boldsymbol{x}[i+1] - \boldsymbol{x}[i]|$. In the case of a 2D image denoted by $\boldsymbol{x} \in \mathbb{R}^{d_1\times d_2}$, the TV is formulated as $\sum_{i=1}^{d-1}\sum_{j=1}^{d-1}\sqrt{(\boldsymbol{x}[i+1,j] - \boldsymbol{x}[i,j])^2 + (\boldsymbol{x}[i,j+1] - \boldsymbol{x}[i,j])^2}$.

the PnP framework. Instead of using a manually designed denoiser, these studies replaced the proximal operator by a deep neural network that is trained with data. The empirical efficacy of this approach significantly surpassed previous implementations of PnP.

Motivations. Focusing narrowly on image denoising, deep neural networks, especially convolutional networks, have been the state of the art. The concept of residual learning [77] and techniques such as batch normalization [91] make network training significantly more effective, leading to trained networks with greater capacity and flexibility to process more real-world images. *DnCNN* [184] and more advanced variants [114, 183, 186] integrate these techniques with convolutional neural networks and are vastly superior than analytical denoisers such as BM3D.

Therefore, it is well-motivated to use CNN denoisers in the PnP framework. On top of the great network capacity brought by stacking up more layers, the intuitions behind the success of CNN denoisers might also lie at the convolutional operation itself. Paper [158] shows convolutional operations naturally generate *self-similarity* in its output due to its inherent parameter-sharing structure, which makes CNNs very suitable for image restoration tasks, even if they have random parameters. See more discussion at the end of the next subsection.

A recipe for DNN-assisted plug-and-play methods. Procedures for deriving a DNN-assisted PnP method are more straightforward than other L2O frameworks as the derivation involves a minimal modification to the original optimization scheme:

Step 1. Based on an optimization formulation, select an iterative solution method.

Step 2. Identify one or more parts of the method that can be replaced with DNN denoisers.

Step 3. Train the DNN denoisers on proper datasets.

Step 4. *Plug* the well-trained DNN operators into the original optimization algorithm, and *play*.

4.3 Design elements and techniques of plug-and-play algorithms

Options and selection of plugged-in denoisers. Among the initial attempts of CNN-based denoising, *DnCNN* [184] stands as a significant development, which is grounded in VGG networks [143]. DnCNN set the new state of the art for image denoising through the incorporation of residual learning concepts [77] and the employment of batch normalization layers [77]. DnCNN is used for PnP in [135]. Another representative CNN denoiser was proposed in [114], which devised an encoder-decoder architectural complemented by skip connections.

It is noteworthy, however, that these denoisers are configured to address inputs with a singular, fixed noise level. In scenarios with the presence of multiple noise levels within images, the approach necessitates the training of distinct CNNs, each tailored to the specific noise levels encountered. Consequently, the application of these models becomes contingent on the precise noise characteristics of the inputs, thus underscoring a limitation in their generalizability across diverse noise conditions.

In an effort to overcome the aforementioned limitation pertaining to flexibility and adaptivity, a novel CNN denoiser, termed FFDNet, was introduced by the authors of [186]. It enables the handling of inputs contaminated with noises across a spectrum of levels through the utilization of a single network. The unique implementation accepts a noise level map as an auxiliary input, thereby fostering adaptivity to varying noise intensities.

Such adaptability is vital for PnP. As the iterative process of PnP advances, the quality of the denoiser's input also improves (i.e., less noisy). Consequently, an optimal strategy entails a gradual decrease of the noise level corresponding with these improvements. This is a capability uniquely afforded by FFDNet [179], making it a strong candidate in contexts demanding adaptive responsiveness to varying noise conditions.

In a more recent PnP work [183], the authors proposed a new CNN-based denoiser that essentially combines FFDNet and the U-Net architecture [110]. U-Net is effective in image-to-image translation, because The skip connections introduced in U-Net provide the CNN-based denoisers the ability to synthesize features at different scales, thus generating more visually vivid features for better image restoration performances.

Parameter tuning in PnP methods. Using learning in plug-and-play (PnP) methods is not just about finding the right denoising tool or helper. It can also help figure out the best settings for the PnP process itself. Although denoisers (including the DNN-based), are integrated into PnP frameworks in a hot-plugged fashion, each iterative step necessitates the determination of particular parameters, such as the step size of the other proximal step and the noise level specific to the current iteration. The calibration of these elements demands coordination in a sense for optimal performance of PnP. Intuitively, the step size of the other proximal step influences the "noisiness" of the input to the denoiser.

Recognizing the complexity of this interaction, the authors of [165] proposes to learn a policy network using reinforcement learning, to meticulously tune these parameters and even decide the appropriate stopping time of the PnP iterative process. The new optimization method can get results as good as those from using perfect settings, which are usually based on ground truth we cannot normally access. However, the introduction of reinforcement induces an extra step of learning a policy network besides pre-training the plugged-in denoiser.

Discussion on denoisers as image prior. A key motivation of adopting CNN-based denoisers in the PnP frameworks besides the network capacity is that convolutional neural networks serve as better image prior. But why do CNNs capture the features of natural images so well? In Subsection 2.2 we have discussed the three key characteristics of a CNN that are fundamental to its empirical success: local interaction with inputs, parameter sharing and the equivariance to translations. Here we provide another perspective that is slightly out of the context of PnP but related. The authors of [158] showed that even a randomly-initialized CNN can be used as a handcrafted prior with excellent results in standard inverse problems including denoising. The network is trained to fit a degraded measurement by mapping a randomly generated but fixed 3D tensor without any other explicit regularization. They called this phenomenon deep image prior. In [158], the authors attribute it to the convolutional filters themselves, which naturally impose self-similarity in the outputs. When combined with skip connections such as in U-Net, such self-similarity structures can be sustained at multiple scales in the image, which is essential for the visual quality of the restored images.

4.4 Mathematical analysis on the convergence of plug-and-play methods

Concerning the convergence of PnP-ADMM, theoretical guarantees were initially attempted in [149], based on the unrealistic assumption that the denoiser's derivative was doubly stochastic. Next, [36] proves the convergence of PnP-ADMM through the implementation of a more pragmatic assumption pertaining to "bounded denoisers".

Enforceable convergence conditions for PnP-FBS and PnP-ADMM were given in [135], using the fixed-point analysis. Their assumption is a particular Lipschitz condition imposed on the denoisers, which can be achieved by applying normalization when the denoisers are trained. A more recent study is [156].

The aforementioned fixed-point framework was extended in [46] to the RED-PRO framework. Independently, [79] introduces a technique of learning projection operators onto the compact manifolds of true data. Under specific assumptions regarding the manifold and the adequate representation by the sampled data, they establish that their method for the synthesis of a PnP operator can, with statistical assurance, approximate the projection onto a low-dimensional manifold of data.

Enforcing special structures in denoisers. The integration of PnP with L2O has been revealed to confer theoretical advantages. Most of the existing theoretical insights into the convergence of PnP towards the desired solution depend on specific conditions associated with the denoisers. Despite the intrinsic difficulty in furnishing such guarantees for manually crafted denoisers, such as K-SVD and BM3D, the adoption of learning-based methods affords a flexibility that facilitates the convergence conditions.

Normalization or regularization strategies on the Lipschitz continuity and its variants of DNNs are useful tools for such purpose. Notably, spectral constraints have been applied to both DNNs [15,16,124] and generative adversarial networks (GANs) [32,119]. Regularizing Lipschitz continuity serves to stabilize the training process and increase the network's robustness to adversarial attacks [131,166].

In a specific illustration, the researchers in [119] innovated a strategy to normalize all weights to be congruent with unit spectral norms, thereby circumscribing the Lipschitz constant of the entire network to a maximum value of one. Concurrently, the authors of [135] introduced a normalization technique explicitly tailored for the training of deep learning-based denoisers, ensuring compliance with the Lipschitz condition, further substantiating the theoretical strengths of the amalgamation of PnP with L2O.

5 End-to-end learning and optimization as a layer

"Predict then optimize" is a process used in decision-making processes, especially when predictions about future data are necessary for defining how to optimize and their constraints. The technique starts with training a machine learning model, then applies to model to predict future data, and finally optimizes the decision based on these predictions. Typical examples include predicting travel times for route optimization, predicting sales for inventory optimization, and predicting returns and volatility for portfolio optimization.

Suppose we have a set of decision variables $\boldsymbol{x}$ and a random variable $\mathbf{Y}$, which represents some future events so is currently unknown. The relationship between the decision $\boldsymbol{x}$ and the random variable $\mathbf{Y}$ is represented by a loss function $f(\boldsymbol{x},\boldsymbol{y})$, which measures the performance of decision $\boldsymbol{x}$ given the realization $\boldsymbol{y}$ of $\mathbf{Y}$. In the predict-then-optimize technique, the first step involves building a machine learning model $\boldsymbol{s}_{\boldsymbol{\theta}}$ to predict $\mathbf{Y}$ based on some features $\boldsymbol{\beta}$. This yields a prediction $\hat{\boldsymbol{y}} = \boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{\beta})$. The next step is to solve the optimization problem according to the model prediction $\hat{\boldsymbol{y}}$: $\min_{\boldsymbol{x}} f(\boldsymbol{x},\hat{\boldsymbol{y}})$. In a conventional predict-then-optimize pipeline, the learning and optimization are separated. In the training phase, we collect data pairs in the form of $\{\boldsymbol{\beta}_i,\boldsymbol{y}_i^\star\}_{i=1}^n$, and train a machine learning model $\boldsymbol{s}_{\boldsymbol{\theta}}$ that maps each $\boldsymbol{\beta}_i$ to $\boldsymbol{y}_i^\star$. Here $\boldsymbol{y}_i^\star$ is an "ideal prediction" under the input $\boldsymbol{\beta}$, which should be built upon domain knowledge. In the prediction phase, we make decisions $\boldsymbol{x}$ according to the model prediction under the upcoming input $\boldsymbol{\beta}$. Specifically, it is

$$\text{Learning: } \underset{\boldsymbol{\theta}}{\text{minimize}} \sum_{i=1}^{n} \ell(\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{\beta}_i),\boldsymbol{y}_i^\star); \quad \text{Predict-then-Optimize: } \underset{\boldsymbol{x}}{\text{minimize}}\, f(\boldsymbol{x},\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{\beta})). \tag{5.1}$$

Here ℓ measures the difference between $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{\beta}_i)$ and $\boldsymbol{y}_i^\star$, which might be taken as the squared error, cross-entropy, etc.

Note that the quality of the decision $\boldsymbol{x}$ obtained from this optimization problem is typically measured by its actual loss $f(\boldsymbol{x},\boldsymbol{y})$, rather than the predicted loss $f(\boldsymbol{x},\hat{\boldsymbol{y}})$. This introduces a discrepancy between the process of making the decision, which is based on the predicted loss, and the evaluation of the decision, which is based on the actual loss. This discrepancy is a key challenge in the predict-then-optimize technique. To mitigate this discrepancy, different approaches have been proposed and explored:

• **Stochastic optimization.** This approach uses the samples and other prior knowledge of $\mathbf{Y}$, given the contextual information $\boldsymbol{\beta}$, to model its randomness by a distribution $D(\boldsymbol{\beta})$. Then, the decision is made by minimizing the expected loss:

$$\underset{\boldsymbol{x}}{\text{minimize}}\, \mathbb{E}_{\boldsymbol{y}\sim D(\boldsymbol{\beta})}[f(\boldsymbol{x},\boldsymbol{y})].$$

• **(Distributional) robust optimization.** This approach takes into consideration the uncertainty in the predictions. It finds the decision that performs best under the worst-case scenario within a certain uncertainty set around $\hat{\boldsymbol{y}}$, say $B(\hat{\boldsymbol{y}})$, leading to the following optimization problem:

$$\underset{\boldsymbol{x}}{\text{minimize}} \max_{\boldsymbol{y}\in B(\hat{\boldsymbol{y}})} f(\boldsymbol{x},\boldsymbol{y}).$$

Distributional robust optimization combines stochastic optimization and robust optimization by introducing an ambiguity set $\mathcal{D}$:

$$\underset{\boldsymbol{x}}{\text{minimize}} \max_{D\in\mathcal{D}(\boldsymbol{\beta})} \mathbb{E}_{\boldsymbol{y}\sim D}[f(\boldsymbol{x},\boldsymbol{y})].$$

5.1 End-to-end learning

More recently, the approach known as end-to-end learning or decision-focus learning [24, 53, 54, 97, 167] integrates the optimization step into the learning process, which serves as a straightforward approach to address the discrepancy induced by (5.1). It does not rely on an intermediate prediction step in the traditional approaches but instead attempts to learn a direct mapping from features $\boldsymbol{\beta}$ to decisions $\boldsymbol{x}^\star$. Specifically, in the training phase, we collect data pairs in the form of $\{\boldsymbol{\beta}_i, \boldsymbol{x}_i^\star\}_{i=1}^n$, and search parameters such that the final decision $\boldsymbol{x}_i$ matches the ideal solution $\boldsymbol{x}_i^\star$:

$$\underset{\boldsymbol{\theta}}{\text{minimize}} \sum_{i=1}^{n} \bar{\ell}(\boldsymbol{x}_i, \boldsymbol{x}_i^\star), \quad \text{where } \boldsymbol{x}_i = \arg\min_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{\beta}_i)).$$

Here, $\bar{\ell}$ directly measures the difference between the model-based decision $\boldsymbol{x}_i$ and the ideal decision $\boldsymbol{x}_i^\star$. Once such a training process is done, the predict-then-optimize format remains the same with (5.1).

A general form. In the context of end-to-end learning, we no longer depend on the intermediate variable $\boldsymbol{y}$. Therefore, in this subsection, we can rewrite the function $f(\boldsymbol{x}, \boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{\beta}))$ into a more general form $f_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta})$ and introduce some explicit constraints that the decisions $\boldsymbol{x}$ must satisfy for reasons such as safety, regulations, and physical feasibility:

$$\underset{\boldsymbol{x}}{\text{minimize}}\, f_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}) \qquad \text{subject to } \boldsymbol{g}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}) = \boldsymbol{0},\ \boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}) \leqslant \boldsymbol{0}. \tag{5.2}$$

Finding the optimal parameter $\boldsymbol{\theta}$ in (5.2) can be expressed explicitly in the bi-level optimization problem:

$$\underset{\boldsymbol{\theta}}{\text{minimize}}\ \frac{1}{n} \sum_{i=1}^{n} \bar{\ell}(\boldsymbol{x}_i, \boldsymbol{x}_i^\star), \quad \text{subject to } \boldsymbol{x}_i = \arg\min_{\boldsymbol{x}} \{f_\theta(\boldsymbol{x}, \boldsymbol{\beta}_i) : g_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}_i) = \boldsymbol{0}, h_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}_i) \leqslant \boldsymbol{0}\}. \tag{5.3}$$

For simplicity, we ignore the case when the lower-level problem may have multiple solutions. This approach yields the mapping

$$\boldsymbol{x}_{\boldsymbol{\theta}} :\ \boldsymbol{\beta} \mapsto \arg\min_{\boldsymbol{x}} \{f_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}) : \boldsymbol{g}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}) = \boldsymbol{0}, \boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}) \leqslant \boldsymbol{0}\}.$$

Pros and cons. In the traditional approaches (5.1), errors from the standalone prediction step $\delta \boldsymbol{y} := \boldsymbol{y}^\star - \hat{\boldsymbol{y}}$ can propagate to the optimization solution, and the error can be difficult to control since, even in linear programming, arbitrary small errors in data can lead to an arbitrarily large error in the solution. End-to-end learning avoids this issue by optimizing the decision variables directly with respect to the actual loss. Additionally, end-to-end learning does not explicitly model the randomness in the data, but instead tries to directly learn a good decision mapping from data. So, it may perform better than stochastic optimization when the distribution of the data is complex and difficult to model

On the other hand, end-to-end learning does not explicitly consider the worst-case scenario, making it less suitable than robust optimization for cases where robustness is crucial.

In addition, the solution mappings of most optimization are generally not differentiable, so they prevent the gradients easily flow back through in the backpropagation algorithm. We address this challenge below.

5.2 Differentiation through solution and KKT system

In order to learn a mapping $\boldsymbol{x}_{\boldsymbol{\theta}}$, a common approach is to approximate the gradient

$$\frac{d}{d\boldsymbol{\theta}} \bar{\ell}(\boldsymbol{x}_{\boldsymbol{\theta}}(\boldsymbol{\beta}_i), \boldsymbol{x}_i^\star)$$

for every training data point i, and then employ an SGD-like algorithm, such as Adam [101], to find a desirable $\boldsymbol{\theta}$. The key obstacle is to "differentiating through" the solution, i.e., to compute

$$\frac{\partial}{\partial \boldsymbol{\theta}} \boldsymbol{x}_{\boldsymbol{\theta}}(\boldsymbol{\beta}_i) \tag{5.4}$$

with $\boldsymbol{x}_{\boldsymbol{\theta}}(\boldsymbol{\beta}_i)$ being the solution of an optimization problem. Even when the underlying optimization problem is a linear program, the differentiation is ill-defined. Specifically, a linear program may have multiple solutions, and even when the solution is unique, it is locally piece-wise constant with respect to the coefficients of the objective and those in the constraints. In this case, (5.4) is either zero or undefined. To obtain an informative gradient with respect to $\boldsymbol{\theta}$, modifications to the optimization formulation is necessary.

The Karush-Kuhn-Tucker (KKT) system is a set of equalities and inequalities that describe what optimal primal and dual solutions should satisfy. Under proper constraint qualifications, the KKT system is a necessary and sufficient optimality condition. Specifically, we introduce dual variables $\boldsymbol{\lambda}$ and $\boldsymbol{\nu}$ for constraints $\boldsymbol{g}_{\boldsymbol{\theta}}$ and $\boldsymbol{h}_{\boldsymbol{\theta}}$ respectively, and the KKT system is then defined for the primal-dual solution tuple $(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\nu})$ as follows:

$$\mathcal{G}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}, \boldsymbol{\beta}) = \boldsymbol{0}, \quad \boldsymbol{\nu} \geqslant \boldsymbol{0}, \quad \boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}) \leqslant \boldsymbol{0}, \tag{5.5}$$

where the operator $\mathcal{G}_{\boldsymbol{\theta}}$ is defined by

$$\mathcal{G}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}, \boldsymbol{\beta}) = \begin{bmatrix} \frac{\partial f_{\boldsymbol{\theta}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{\beta}) + (\frac{\partial \boldsymbol{g}_{\boldsymbol{\theta}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{\beta}))^\top \boldsymbol{\lambda} + (\frac{\partial \boldsymbol{h}_{\boldsymbol{\theta}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{\beta}))^\top \boldsymbol{\nu} \\ \boldsymbol{g}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}) \\ \boldsymbol{\nu} \circ \boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{\beta}) \end{bmatrix}.$$

Since the above equation (5.5) describes the optimality condition of $(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\nu})$, the decision $\boldsymbol{x}_{\boldsymbol{\theta}}(\boldsymbol{\beta})$ in (5.4) must satisfy the KKT system above, serving as the $\boldsymbol{x}$ in (5.5). This method enables one to determine the differential of $\boldsymbol{x}_{\boldsymbol{\theta}}(\boldsymbol{\beta})$ with respect to $\boldsymbol{\theta}$. Furthermore, to illustrates the dependency of $\boldsymbol{\lambda}, \boldsymbol{\nu}$ on $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$, we use the notions $\boldsymbol{\lambda}_{\boldsymbol{\theta}}(\boldsymbol{\beta}), \boldsymbol{\nu}_{\boldsymbol{\theta}}(\boldsymbol{\beta})$. By assuming the smoothness of f, $\boldsymbol{g}$, $\boldsymbol{h}$ and applying the implicit function theorem, one can derive $\partial \boldsymbol{x}_{\boldsymbol{\theta}} / \partial \boldsymbol{\theta}$ from (5.5) as

$$\frac{d}{d\boldsymbol{\theta}} \mathcal{G}_{\boldsymbol{\theta}}(\boldsymbol{x}_{\boldsymbol{\theta}}(\boldsymbol{\beta}), \boldsymbol{\lambda}_{\boldsymbol{\theta}}(\boldsymbol{\beta}), \boldsymbol{\nu}_{\boldsymbol{\theta}}(\boldsymbol{\beta}), \boldsymbol{\beta}) = \boldsymbol{0} \tag{5.6}$$

$$\Longrightarrow \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{x}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}} + \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\lambda}} \frac{\partial \boldsymbol{\lambda}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}} + \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\nu}} \frac{\partial \boldsymbol{\nu}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}} + \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}} = \boldsymbol{0} \tag{5.7}$$

$$\Longrightarrow \left[\left(\frac{\partial \boldsymbol{x}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}}\right)^\top \left(\frac{\partial \boldsymbol{\lambda}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}}\right)^\top \left(\frac{\partial \boldsymbol{\nu}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}}\right)^\top \right]^\top = -\left[\frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{x}} \ \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\lambda}} \ \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\nu}} \right]^{-1} \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}}. \tag{5.8}$$

When the system is not differentiable at a certain solution, the Jacobian matrix $J\mathcal{G}_{\boldsymbol{\theta}} := [\frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{x}} \ \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\lambda}} \ \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\nu}}]$ is not invertible and, thus, the above step (5.8) fails to hold. In subsequent paragraphs, we will review different means to address this problem.

Additionally, it is worth noting that solving the system defined in (5.7) or calculating the inverse of the Jacobian matrix $J\mathcal{G}_{\boldsymbol{\theta}}$ is computationally expensive. Suppose $\boldsymbol{x} \in \mathbb{R}^{d_1}$, $\boldsymbol{\lambda} \in \mathbb{R}^{d_2}$, $\boldsymbol{\nu} \in \mathbb{R}^{d_3}$. The complexity of calculating $\partial \boldsymbol{x}_{\boldsymbol{\theta}} / \partial \boldsymbol{\theta}$ via (5.8) will be $\mathcal{O}((d_1 + d_2 + d_3)^3)$. In the entire training process (i.e., solving (5.3)), typically thousands of such gradients have to be evaluated, resulting in considerable computational demands. To address this issue, various techniques, such as Neumann approximation [62] and Jacobian-free training [59], have been proposed and explored. An earlier work [8] applies the KKT approach to certain quadratic programs.

Heuristics. When the solution mapping is nondifferentiable and even discontinuous, one often applies heuristics [69]. Take, for example, a situation with a singular Jacobian matrix. Instead of directly solving the system as in (5.8), one can compute the least squares solution for system (5.7). Specifically, we solve the following quadratic optimization problem as an approximation to (5.7):

$$\underset{\boldsymbol{d}}{\text{minimize}} \left\| \left[\frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{x}} \ \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\lambda}} \ \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\nu}} \right] \boldsymbol{d} + \frac{\partial \mathcal{G}_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}} \right\|^2 + \lambda \|\boldsymbol{d}\|^2,$$

where the last term $\lambda \|\boldsymbol{d}\|^2$ serves as a regularizer to ensure the magnitude of $\boldsymbol{d}$ does not become excessively large. The scalar $\lambda \in \mathbb{R}_+$ controls the intensity of this regularization. Iterative methods, like the conjugate gradient approach, can be employed to solve this optimization problem. The number of iterations is determined by the desired accuracy specified by the user.

Adding regularizer. To address the non-differentiable issue of the solution mapping, when the underlying optimization problem (5.2) is convex, a common approach from [**?**] is to add a small quadratic function, $\frac{\alpha}{2}\|\boldsymbol{x}\|_2^2$ and, therefore, yielding the new objective function $f_\theta(\boldsymbol{x},\beta,\alpha)$. This makes the problem *strongly convex* and the solution not only unique but also differentiable. Another choice of regularizer is the logarithmic barrier function [113], which is often used on conic optimization.

Perturbation. Gaussian smoothing or Gaussian perturbation is a standard technique used to make the differentiation of nonsmooth or discontinuous functions possible. This technique can be particularly useful in optimization problems where standard gradient-based methods are unable to operate due to the non-differentiability of the objective function. We leverage the methodology illustrated in [21] as a prime example: perturb the parameter in the solution mapping $\boldsymbol{x}_{\boldsymbol{\theta}}$ and take expectation as follows:

$$\boldsymbol{x}_{\boldsymbol{\theta},\varepsilon}:\ \boldsymbol{\beta}\mapsto \mathbb{E}_{\boldsymbol{n}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}[\boldsymbol{x}_{\boldsymbol{\theta}+\varepsilon\boldsymbol{n}}(\boldsymbol{\beta})].$$

Assuming the dimension of $\boldsymbol{\theta}$ is n, the smoothed solution mapping can be explicitly expressed as

$$\begin{aligned}\boldsymbol{x}_{\boldsymbol{\theta},\varepsilon}(\boldsymbol{\beta}) &= \frac{1}{\sqrt{(2\pi\varepsilon^2)^n}}\int_{\boldsymbol{n}\in\mathbb{R}^n}\exp\bigg(-\frac{1}{2}\|\boldsymbol{n}\|^2\bigg)\boldsymbol{x}_{\boldsymbol{\theta}+\varepsilon\boldsymbol{n}}(\boldsymbol{\beta})d\boldsymbol{n}\\ &= \frac{1}{\sqrt{(2\pi\varepsilon^2)^n}\varepsilon}\int_{\boldsymbol{\theta}'\in\mathbb{R}^n}\exp\bigg(-\frac{1}{2\varepsilon^2}\|\boldsymbol{\theta}'-\boldsymbol{\theta}\|^2\bigg)\boldsymbol{x}_{\boldsymbol{\theta}'}(\boldsymbol{\beta})d\boldsymbol{\theta}'.\end{aligned}$$

According to the standard real analysis, the mapping $\boldsymbol{x}_{\boldsymbol{\theta},\varepsilon}$, as a convolution of $\boldsymbol{x}_{\boldsymbol{\theta}}$ with a smooth mollifier, must be smooth as long as $\boldsymbol{x}_{\boldsymbol{\theta}}$ is measurable. Its gradient can be derived by

$$\begin{aligned}\frac{\partial}{\partial\boldsymbol{\theta}}\boldsymbol{x}_{\boldsymbol{\theta},\varepsilon}(\boldsymbol{\beta}) &= \frac{1}{\sqrt{(2\pi\varepsilon^2)^n}\varepsilon}\int_{\boldsymbol{\theta}'\in\mathbb{R}^n}\frac{\partial}{\partial\boldsymbol{\theta}}\exp\bigg(-\frac{1}{2\varepsilon^2}\|\boldsymbol{\theta}'-\boldsymbol{\theta}\|^2\bigg)\otimes\boldsymbol{x}_{\boldsymbol{\theta}'}(\boldsymbol{\beta})d\boldsymbol{\theta}'\\ &= \frac{1}{\sqrt{(2\pi\varepsilon^2)^n}\varepsilon}\int_{\boldsymbol{\theta}'\in\mathbb{R}^n}\exp\bigg(-\frac{1}{2\varepsilon^2}\|\boldsymbol{\theta}'-\boldsymbol{\theta}\|^2\bigg)\frac{\partial}{\partial\boldsymbol{\theta}}\bigg(-\frac{1}{2\varepsilon^2}\|\boldsymbol{\theta}'-\boldsymbol{\theta}\|^2\bigg)\otimes\boldsymbol{x}_{\boldsymbol{\theta}'}(\boldsymbol{\beta})d\boldsymbol{\theta}'\\ &= \frac{1}{\sqrt{(2\pi\varepsilon^2)^n}\varepsilon}\int_{\boldsymbol{\theta}'\in\mathbb{R}^n}\exp\bigg(-\frac{1}{2\varepsilon^2}\|\boldsymbol{\theta}'-\boldsymbol{\theta}\|^2\bigg)\bigg(\frac{1}{\varepsilon^2}(\boldsymbol{\theta}'-\boldsymbol{\theta})\bigg)\otimes\boldsymbol{x}_{\boldsymbol{\theta}'}(\boldsymbol{\beta})d\boldsymbol{\theta}'\\ &= \mathbb{E}_{\boldsymbol{n}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}\bigg[\frac{1}{\varepsilon}\boldsymbol{n}\otimes\boldsymbol{x}_{\boldsymbol{\theta}+\varepsilon\boldsymbol{n}}(\boldsymbol{\beta})\bigg].\end{aligned}$$

Here, $\otimes$ means the outer product. With proper assumptions on $\boldsymbol{f}_{\boldsymbol{\theta}}$, $\boldsymbol{g}_{\boldsymbol{\theta}}$, $\boldsymbol{h}_{\boldsymbol{\theta}}$, it can be shown that $\partial\boldsymbol{x}_{\boldsymbol{\theta},\varepsilon}/\partial\boldsymbol{\theta}\to\partial\boldsymbol{x}_{\boldsymbol{\theta}}/\partial\boldsymbol{\theta}$ as $\varepsilon\to 0$. To calculate the smoothed gradient in practice, we can use the Monte-Carlo method: sample enough points $\{\boldsymbol{n}_i\}_{i=1}^M$ that yields $\boldsymbol{n}_i\sim\mathcal{N}(\mathbf{0},\mathbf{I})$ and estimate the gradient via

$$\frac{\partial}{\partial\boldsymbol{\theta}}\boldsymbol{x}_{\boldsymbol{\theta},\varepsilon}(\boldsymbol{\beta})\approx\frac{1}{M}\sum_{i=1}^M\frac{1}{\varepsilon}\boldsymbol{n}_i\otimes\boldsymbol{x}_{\boldsymbol{\theta}+\varepsilon\boldsymbol{n}_i}(\boldsymbol{\beta}).$$

5.3 Fixed-point layer

The KKT system is not the only optimality condition. Convex optimization problems with two or more components, as well as certain nonconvex optimization problems, also have another form of optimality condition of the fixed-point equation. To quickly introduce it, we use the following problem as an example:

$$\operatorname*{minimize}_{\boldsymbol{x}\in\mathbb{R}^n}\ \frac{1}{2}\|\mathbf{A}\boldsymbol{x}-\boldsymbol{b}\|^2+\|\mathbf{D}\boldsymbol{x}\|_1\qquad\text{subject to } \boldsymbol{l}\leqslant\boldsymbol{x}\leqslant\boldsymbol{u}, \tag{5.9}$$

where

$$\mathbf{D}=\begin{bmatrix}-1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & & \\ & & & -1 & 1\end{bmatrix}.$$

The function $\|\mathbf{D}\boldsymbol{x}\|_1$ is also known as one-dimensional total variation. This problem includes the following components:

- the least squares term, $\frac{1}{2}\|\mathbf{A}\boldsymbol{x}-\boldsymbol{b}\|^2$, which is Lipschitz differentiable;
- the ℓ_1-norm, $\|\cdot\|_1$, whose proximal operator is simple and closed-form;
- the finite-difference operator, $\mathbf{D}$, is a simple convolution-like matrix, and so is its adjoint $\mathbf{D}^*$;
- the box constraint, $\boldsymbol{l}\leqslant\boldsymbol{x}\leqslant\boldsymbol{u}$, is easy to project onto.

By applying the technique of operator splitting Condat-Vu splitting [47,161], $\boldsymbol{x}^\star$ is a solution of problem (5.9) if and only if there exists $\boldsymbol{y}^\star$ such that $\boldsymbol{z}^\star=[\boldsymbol{x}^\star;\boldsymbol{y}^\star]$ satisfies

$$\boldsymbol{z}^\star=\mathcal{T}\boldsymbol{z}^\star,$$

where $\mathcal{T}$ is called a fixed-point operator constructed with the following individual operators: $\nabla_{\boldsymbol{x}}\frac{1}{2}\|\mathbf{A}\boldsymbol{x}-\boldsymbol{b}\|^2=\mathbf{A}^\top(\mathbf{A}\boldsymbol{x}-\boldsymbol{b})$, $\mathbf{D}$, $\mathbf{D}^*$, $\mathrm{prox}_{\|\cdot\|_1}$, and $\mathrm{proj}_{[\boldsymbol{l},\boldsymbol{u}]}$. With $\boldsymbol{z}=[\boldsymbol{x}^\top\boldsymbol{u}^\top]^\top$, $\boldsymbol{z}^{k+1}=\mathcal{T}\boldsymbol{z}^k$ is given by

$$\begin{aligned}\boldsymbol{x}^{k+1}&=\mathrm{proj}_{[\boldsymbol{l},\boldsymbol{u}]}(\boldsymbol{x}^k-\alpha\mathbf{D}^*\boldsymbol{u}^k-\alpha\mathbf{A}^\top(\mathbf{A}\boldsymbol{x}^k-\boldsymbol{b})),\\ \boldsymbol{u}^{k+1}&=\mathrm{prox}_{\beta\|\cdot\|_1^*}(\boldsymbol{u}^k+\beta\mathbf{A}(2\boldsymbol{x}^{k+1}-\boldsymbol{x}^k)),\end{aligned}$$

where $\mathrm{prox}_{\beta\|\cdot\|_1^*}$ can be computed via the Moreau identity $\mathrm{prox}_{\beta\|\cdot\|_1^*}(\boldsymbol{x})+\beta\mathrm{prox}_{\beta^{-1}\|\cdot\|_1}(\beta^{-1}\boldsymbol{x})=\boldsymbol{x}$. When the solution is not unique, every solution $\boldsymbol{x}^\star$ must satisfy this fixed-point equation along with some $\boldsymbol{y}^\star$.

In general, let $\boldsymbol{d}$ denote all the data that define an optimization problem, and consider the operator $\mathcal{T}(\cdot;\boldsymbol{d})$. We call $\boldsymbol{z}^\star$ a fixed point of $\mathcal{T}(\cdot;\boldsymbol{d})$ if $\boldsymbol{z}^\star=\mathcal{T}(\boldsymbol{z}^\star;\boldsymbol{d})$. We say that the fixed-point problem

$$\mathrm{solve}_{\boldsymbol{z}}\quad \boldsymbol{z}=\mathcal{T}(\boldsymbol{z};\boldsymbol{d})$$

is equivalent to the optimization problem if every optimization solution $\boldsymbol{x}^\star$ forms a fixed point $\boldsymbol{z}^\star$ (possibly along with another subvector in $\boldsymbol{z}^\star$) of $\mathcal{T}(\cdot;\boldsymbol{d})$ and the converse also holds. Below we briefly review two common fixed-point operators. More fixed-point operators are covered in [136].

A **fixed-point layer**, introduced also as the deep equilibrium model [10], given by $\mathcal{T}$, is the mapping:

$$\boldsymbol{d}\mapsto\boldsymbol{z}^\star\quad\text{or}\quad\boldsymbol{d}\mapsto\boldsymbol{x}^\star,$$

where $\boldsymbol{z}^\star$ is a fixed point of $\mathcal{T}(\cdot;\boldsymbol{d})$ and $\boldsymbol{x}^\star$ is an optimization solution that can be obtained directly from $\boldsymbol{z}^\star$. In other words, $\boldsymbol{d}$ is the input and $\boldsymbol{z}^\star$ or $\boldsymbol{x}^\star$ is the output. Computing the output can be done by either iterating the fixed-point operator or otherwise solving the original optimization problem.

When the input $\boldsymbol{d}$ depends on θ, the fixed-point $\boldsymbol{z}^\star$ is also a function of θ. In addition, it is possible for $\mathcal{T}$ to also include tunable parameters θ; when this happens, we write $\mathcal{T}_\theta$.

Relationship with RNN. A fixed-point iteration, especially when the operator $\mathcal{T}_\theta$ involves trainable parameters θ, is reminiscent of RNN in that the iteration maintains a memory and gradually refines its estimate of the solution. However, fixed-point iteration is equipped with a rich, solid mathematical theory regarding its convergence behavior. As we will see below, it gives rise to cheaper approaches to obtain a gradient or a descent direction than RNN.

Example: Davis-Yin fixed-point operator [50]. Consider

$$\underset{\boldsymbol{x}\in\mathbb{R}^n}{\mathrm{minimize}}\quad f(\boldsymbol{x})+g(\boldsymbol{x})+h(\boldsymbol{x}),$$

where f, g, h are proper, closed, convex functions and h is differentiable. Assume total duality [136]. Then $\boldsymbol{x}^\star$ is a solution if and only if it satisfies

$$0\in\partial f(\boldsymbol{x}^\star)+\partial g(\boldsymbol{x}^\star)+\nabla h(\boldsymbol{x}^\star),$$

where ∂f and ∂g are subdifferentiables of f and g, respectively. They are possibly set-valued. Hence, solving for $\boldsymbol{x}^\star$ is equivalent to the problem:

$$\mathrm{find}_{\boldsymbol{x}\in\mathbb{R}^n}\quad 0\in(\mathbb{A}+\mathbb{B}+\mathbb{C})\boldsymbol{x}$$

with $\mathbb{A} = \partial f$, $\mathbb{B} = \partial g$, and $\mathbb{C} = \nabla h$. The Davis-Yin splitting refers to the equivalent problem: for $\alpha > 0$,

$$\text{solve}_{\boldsymbol{v} \in \mathbb{R}^n} \quad \boldsymbol{z} = \underbrace{(I - \mathbb{J}_{\alpha\mathbb{B}} + \mathbb{J}_{\alpha\mathbb{A}}(2\mathbb{J}_{\alpha\mathbb{B}} - I - \alpha\mathbb{C}\mathbb{J}_{\alpha\mathbb{B}}))}_{=:\mathcal{T}_{\mathrm{DYS}}} \boldsymbol{z},$$

where $\mathbb{J}_{\alpha\mathbb{B}} := (I + \alpha\mathbb{B})$ is the resolvent of $\mathbb{B}$ and, with $\mathbb{B} = \partial g$, equals $\operatorname{prox}_{\alpha g}$. The last two problems are related through

$$\boldsymbol{x}^\star = \mathbb{J}_{\alpha\mathbb{B}} \boldsymbol{z}^\star.$$

When ∇h is L_h-Lipschitz and $\mathcal{T}$ has at least one fixed point, the iteration

$$\boldsymbol{z}^{k+1} = \mathcal{T}_{\mathrm{DYS}} \boldsymbol{z}^k$$

converges to a fixed point $\boldsymbol{z}^\star = \mathcal{T}_{\mathrm{DYS}} \boldsymbol{z}^\star$, and $\boldsymbol{x}^k = \mathbb{J}_{\alpha\mathbb{B}} \boldsymbol{z}^k$ converges to a solution $\boldsymbol{x}^\star$.

When $g = 0$, we have $\mathbb{J}_{\alpha\mathbb{B}} = I$ and $\mathcal{T}_{\mathrm{DYS}}$ degenerating to the forward-backward splitting. When $h = 0$, we have $\mathbb{C} = 0$ and $\mathcal{T}_{\mathrm{DYS}}$ degenerating to the Douglas-Rachford splitting. Their corresponding optimization iterations are widely known as the proximal-gradient method and the ADMM.

Differentiation through a fixed-point layer. To run a backpropagation algorithm through a fixed-point layer, there are three approaches: traditional, implicit, and Jacobian-free.

In the traditional approach, the forward computation iterates $\boldsymbol{z}^{k+1}(\theta) = \mathcal{T}(\boldsymbol{z}^k(\theta); \boldsymbol{d}(\theta))$ for a finite number of times, $k = 0, 1, \ldots, K-1$. (The choice of K is either fixed and adaptive to the need of approximation accuracy.) Then, backpropagation is based on the differentiation of that formula

$$\frac{\partial}{\partial\theta} \boldsymbol{z}^{k+1}(\theta) = J_{\boldsymbol{z}} \mathcal{T}(\boldsymbol{z}^k(\theta); \boldsymbol{d}(\theta)) \frac{\partial}{\partial\theta} \boldsymbol{z}^k(\theta) + J_{\boldsymbol{d}} \mathcal{T}(\boldsymbol{z}^k(\theta); \boldsymbol{d}(\theta)) \frac{\partial}{\partial\theta} \boldsymbol{d}(\theta), \tag{5.10}$$

which establishes the relationships among successive $\frac{\partial}{\partial\theta} \boldsymbol{z}^k(\theta)$, $k = 0, \ldots, K-1$, and $\frac{\partial}{\partial\theta} \boldsymbol{d}(\theta)$.

The implicit differentiation approach is based on the fixed-point property:

$$\begin{aligned}
&\boldsymbol{z}^\star(\theta) = \mathcal{T}(\boldsymbol{z}^\star(\theta), \boldsymbol{d}(\theta)) \\
&\implies \quad \frac{\partial}{\partial\theta} \boldsymbol{z}^\star(\theta) = J_{\boldsymbol{z}} \mathcal{T}(\boldsymbol{z}^\star(\theta); \boldsymbol{d}(\theta)) \frac{\partial}{\partial\theta} \boldsymbol{z}^\star(\theta) + J_{\boldsymbol{d}} \mathcal{T}(\boldsymbol{z}^\star(\theta); \boldsymbol{d}(\theta)) \frac{\partial}{\partial\theta} \boldsymbol{d}(\theta) \\
&\implies \quad \frac{\partial}{\partial\theta} \boldsymbol{z}^\star(\theta) = (I - J_{\boldsymbol{z}} \mathcal{T}(\boldsymbol{z}^\star; \boldsymbol{d}))^{-1} J_{\boldsymbol{d}} \mathcal{T}(\boldsymbol{z}^\star; \boldsymbol{d}) \frac{\partial}{\partial\theta} \boldsymbol{d}(\theta),
\end{aligned}$$

which establishes the relationship between $\frac{\partial}{\partial\theta} \boldsymbol{z}^\star(\theta)$ and $\frac{\partial}{\partial\theta} \boldsymbol{d}(\theta)$. There is no restriction on the choice of the forward computation method. Suppose the forward method generates $\hat{\boldsymbol{z}}$, an approximate of $\boldsymbol{z}^\star$. Since $\boldsymbol{z}^\star$ is unavailable, the backprapogation algorithm is left to use the approximate formula

$$\frac{\partial}{\partial\theta} \hat{\boldsymbol{z}}(\theta) = (I - J_{\boldsymbol{z}} \mathcal{T}(\hat{\boldsymbol{z}}; \boldsymbol{d}))^{-1} J_{\boldsymbol{d}} \mathcal{T}(\hat{\boldsymbol{z}}; \boldsymbol{d}) \frac{\partial}{\partial\theta} \boldsymbol{d}(\theta).$$

The Jacobian-free approach [59] allows any forward computation method to generate $\hat{\boldsymbol{z}}(\theta)$, and, starting from $\boldsymbol{z}^0 := \hat{\boldsymbol{z}}(\theta)$, it performs one fixed-point iteration $\boldsymbol{z}^1(\theta) = \mathcal{T}(\boldsymbol{z}^0(\theta); \boldsymbol{d}(\theta))$. It uses $\boldsymbol{z}^1(\theta)$ as the output. To apply backpropagation, the Jacobian-free approach uses

$$\frac{\partial}{\partial\theta} \boldsymbol{z}^1(\theta) = J_{\boldsymbol{d}} \mathcal{T}(\boldsymbol{z}^0(\theta); \boldsymbol{d}(\theta)) \frac{\partial}{\partial\theta} \boldsymbol{d}(\theta),$$

which is free of computing the Jacobian, $J_{\boldsymbol{z}} \mathcal{T}(\boldsymbol{z}^\star; \boldsymbol{d})$, and any matrix inversion.

A variant of the Jacobian-free approach [27] performs more than one fixed-point iteration. Specifically, from $\boldsymbol{z}^0 = \hat{\boldsymbol{z}}(\theta)$, it performs $K \geqslant 2$ fixed-point iterations: $\boldsymbol{z}^{k+1}(\theta) = \mathcal{T}(\boldsymbol{z}^k(\theta); \boldsymbol{d}(\theta))$, $k = 0, 1, \ldots, K-1$ to return $\boldsymbol{z}^K(\theta)$. For backpropagation, use (5.10).

Comparisons of different approaches. The forward computations of all the approaches return an approximate fixed-point $\boldsymbol{z}^\star$ or an approximate solution $\boldsymbol{x}^\star$. The traditional approach, as its backpropagation uses (5.10), must record all the fixed-point iterates $\boldsymbol{z}^0, \boldsymbol{z}^1, \ldots, \boldsymbol{z}^{K-1}$. The other approaches do not need to record the fixed-point iterates. The variant of the Jacobian-free approach needs to record only the last few iterates, and the number .

For gradient backpropagation, all the approaches need to compute $J_{\boldsymbol{d}}\mathcal{T}$. The main differences lie in the Jacobian $J_{\boldsymbol{d}}\mathcal{T}$. In the traditional approach, the Jacobian is computed K times. In the implicit differentiation approach, it is computed only once, but the inversion cost is generally cubic of the dimension n. The inversion can be approximated by the Neumann approximation [62] at a lower cost, which grows with the number of terms used in the approximation. The Jacobian-free approach [59], though completely avoids the Jacobian and inversion, does not produce a gradient. The authors [27, 59, 117], under different assumptions such as contraction and linear independence constraint qualification, show that the obtained directions are descent direction despite not equaling gradients.

6 AI-guided mixed-integer optimization

In this section, we turn to mixed-integer programming (MIP), an important class of optimization problems where some or all of the variables are constrained to be integers. MIP finds extensive use across diverse fields such as supply chain management, transportation, energy systems, finance, and more. Unfortunately, MIP is generally considered to be NP-hard due to its integer constraints. It might be quite challenging to solve a fairly large MIP instance. To efficiently solve a specific type of MIP, one has to dive deeply into the theory and practice related to that MIP type. Only then is it possible to design a customized solver that surpasses general solvers on that particular problem class.

Given the multitudinous types of MIP, this process can be quite laborious. However, thanks to recent progress in machine learning, it has become possible to develop customized solvers built on a general solver and a dataset, without requiring too much expertise. This section will present some typical uses of machine learning for mixed-integer linear programming (MILP) and some associated practical and theoretical open questions.

Remarks on notation. In Section 6, we adopt the following conventions for notation. Boldface lowercase symbols, such as $\boldsymbol{x}$, denote vectors. Typically, x_j signifies the j-th element of vector $\boldsymbol{x}$. It is important to distinguish between $\boldsymbol{x}_j$ and x_j; the former refers to a vector indexed by j, the specifics of which are determined by context. On the other hand, bold uppercase symbols, like $\mathbf{A}$, indicate matrices. By default, $A_{i,j}$ is the value at the matrix's i-th row and j-th column, while $A_{i,:}$ signifies the entire row.

6.1 Preliminaries

A general instance of an MILP problem is represented as

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} \ \boldsymbol{c}^\top \boldsymbol{x}, \quad \text{s.t. } \boldsymbol{x} \in \mathbb{X}_{\text{MILP}} = \{\boldsymbol{x} : \mathbf{A}\boldsymbol{x} \circ \boldsymbol{b},\ \boldsymbol{l} \leqslant \boldsymbol{x} \leqslant \boldsymbol{u},\ x_j \in \mathbb{Z},\ \forall\, j \in \mathbb{I}\}. \tag{6.1}$$

Here, $\mathbf{A} \in \mathbb{Q}^{m\times n}$, $\boldsymbol{c} \in \mathbb{Q}^n$, and $\boldsymbol{b} \in \mathbb{Q}^m$. The symbols $\boldsymbol{l}$ and $\boldsymbol{u}$ signify the lower and upper bounds of the variables, respectively, and are coordinate-wise selected from $l_j \in \mathbb{Q} \cup \{-\infty\}, u_j \in \mathbb{Q} \cup \{+\infty\}$. The dot $\circ$ represents the type of constraints, chosen element-wise from $\circ_j \in \{\leqslant, =, \geqslant\}$. The index set $\mathbb{I} \subset \{1, 2, \ldots, n\}$ includes those indices j where the variable x_j is constrained to be an integer. Each MILP instance can be described as a tuple $(\boldsymbol{c}, \mathbb{X}_{\text{MILP}})$.

Basic concepts. The set $\mathbb{X}_{\text{MILP}}$ that describes all the constraints in (6.1) is termed the feasible set. An MILP instance is considered infeasible if $\mathbb{X}_{\text{MILP}} = \emptyset$ and feasible otherwise. For feasible MILPs, $\inf\{\boldsymbol{c}^\top \boldsymbol{x} : \boldsymbol{x} \in \mathbb{X}_{\text{MILP}}\}$ is referred to as the optimal objective value. If there exists $\boldsymbol{x}^* \in \mathbb{X}_{\text{MILP}}$ such that $\boldsymbol{c}^\top \boldsymbol{x}^* \leqslant \boldsymbol{c}^\top \boldsymbol{x}$, $\forall\, \boldsymbol{x} \in \mathbb{X}_{\text{MILP}}$, then $\boldsymbol{x}^*$ is designated as an optimal solution. There can be a situation where the objective value is arbitrarily low, meaning that for any $R > 0$, the inequality $\boldsymbol{c}^\top \boldsymbol{x}^* < -R$ is satisfied for some $\boldsymbol{x}^* \in \mathbb{X}_{\mathrm{MILP}}$. In such scenarios, the MILP is said to be unbounded, or its optimal objective value is $-\infty$.

Approaches and challenges via ML. Addressing an MILP involves determining its feasibility and boundedness, as well as providing an optimal solution when possible. A direct approach to applying ML techniques on MILP, and indeed the one adopted in the early research literature, involves training a neural network to map an MILP to its results. For example, suppose we consider the scenario where MILPs are feasible and bounded. In this context, we attempt to approximate the optimal solution of an MILP through a trained neural network:

$$\boldsymbol{x}^* \approx \mathrm{NeuralNetwork}(\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}}).$$

Assuming the availability of sufficient data, this approach is theoretically doable. Given a collection of MILP instances and their corresponding solution $\{(\boldsymbol{c}_k, \mathbb{X}_{\mathrm{MILP}_k}, \boldsymbol{x}_k^*)\}_{k=1}^K$, the neural network with parameters $\boldsymbol{\theta}$ can then be trained using the following minimization problem:

$$\min_{\boldsymbol{\theta}} \sum_k \|\boldsymbol{x}_k^* - \mathrm{NeuralNetwork}(\boldsymbol{c}_k, \mathbb{X}_{\mathrm{MILP}_k}; \boldsymbol{\theta})\|$$

Here, the objective is to minimize the sum of the normed differences between the optimal solutions and the outputs of the neural network for each given MILP instance. However, this approach presents two fundamental challenges:

• Ensuring feasibility: The outcomes of an NN are not inherently constrained by the stringent requirements of an MILP problem. Unlike in the case of continuous optimizations where constraint violations may be bounded by a small upper limit, there are no guarantees that the results from an NN will be feasible in the context of an MILP problem.

• Verifying optimality: Even if the NN's output is an optimal solution, validating its optimality is nontrivial. Unlike in continuous optimizations where conditions like the KKT can validate optimality, there is no straightforward method to confirm the optimality of a solution generated by a neural network.

These issues highlight the limitations of a purely data-driven approach and point towards the necessity of incorporating algorithmic approaches. The conventional MILP solver is instrumental in overcoming these limitations by serving as a safeguard, ensuring the optimality of the solutions. Concurrently, the NN can be integrated into the solver to leverage its strengths. The intuition offered by the NN can facilitate adaptive and efficient problem-solving, providing initial estimates, or "warm starts", to guide the solver.

To summarize, a more effective practice of machine learning for MILP problems is utilizing neural networks to *guide traditional solvers, rather than replacing these solvers entirely with neural networks.* In the remaining subsections, we will dive into various components of MILP solvers, outlining how machine learning methodologies can be incorporated to enhance their performance.

6.2 Improving branch and bound by ML

The branch and bound (BnB) algorithm is a widely employed method for solving general MILP problems and forms the foundation of modern MILP solvers. It operates on the principle of "divide and conquer", and is guaranteed to provide an optimal solution when it exists. Throughout the BnB process, numerous decisions need to be made, such as variable selection and node selection. While these choices do not affect the final result, they substantially impact the algorithm's efficiency. Unfortunately, there are no theoretical guidelines on making the best decisions for general MILP. Therefore, these decision-making strategies often rely on practitioners' expertise and experience. Machine learning (ML) can be a helpful tool in developing these strategies. Before introducing ML for branching, we will first present some key components in BnB: LP relaxation, branching, and bounding.

6.2.1 *Branch and bound*

LP relaxation. Removing all the integer constraints in (6.1) (i.e., $x_j, j \in \mathbb{I}$ is no longer required to be an integer) results in a linear program (LP), known as the LP relaxation of (6.1). The resulting feasible set is denoted as $\mathbb{X}_{\mathrm{LP}}$, and the LP relaxation is formulated as $\min_{\boldsymbol{x}\in\mathbb{X}_{\mathrm{LP}}} \boldsymbol{c}^\top\boldsymbol{x}$. The LP relaxation plays a pivotal role in solving the MILP. If the LP relaxation is infeasible, one can confidently assert the infeasibility of the original MILP. If the LP relaxation is lower unbounded, one can deduce the unboundedness of the MILP, given $\mathbf{A}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{l}$, $\boldsymbol{u}$ are all rational data. If the LP relaxation produces an optimal solution denoted as $\underline{\boldsymbol{x}}$, this solution provides a lower bound for the MILP as

$$\boldsymbol{c}^\top\underline{\boldsymbol{x}} \leqslant \boldsymbol{c}^\top\boldsymbol{x}, \quad \text{for all } \boldsymbol{x}\in\mathbb{X}_{\mathrm{MILP}}.$$

If $\underline{\boldsymbol{x}}$ fulfills all the integer constraints, then it can be concluded that $\underline{\boldsymbol{x}}$ is the optimal solution to the MILP. However, it might be quite challenging to identify such an $\underline{\boldsymbol{x}}$ if the LP relaxation has multiple optimal solutions. More commonly, the LP relaxation may not present a solution that aligns with all the integer constraints. To measure this, we define the set of indices that $\underline{\boldsymbol{x}}$ violates the integer constraints:

$$\mathrm{frac}(\underline{\boldsymbol{x}}) := \{j\in\mathbb{I} : \underline{x}_j \notin \mathbb{Z}\}.$$

Branching. Suppose $\mathrm{frac}(\underline{\boldsymbol{x}}) \neq \emptyset$. To obtain a solution that has less violation than $\underline{\boldsymbol{x}}$, we select an index $j\in\mathrm{frac}(\underline{\boldsymbol{x}})$ and generate two sub-MILPs:

$$\begin{aligned}
&\mathrm{MILP}_{\mathrm{up}}: \quad \min_{\boldsymbol{x}} \boldsymbol{c}^\top\boldsymbol{x} \ \text{ s.t. } \ \boldsymbol{x}\in\mathbb{X}_{\mathrm{MILP}_{\mathrm{up}}} = \{\boldsymbol{x} : \boldsymbol{x}\in\mathbb{X}_{\mathrm{MILP}}, x_j \geqslant \lceil \underline{x}_j\rceil\},\\
&\mathrm{MILP}_{\mathrm{down}}: \quad \min_{\boldsymbol{x}} \boldsymbol{c}^\top\boldsymbol{x} \ \text{ s.t. } \ \boldsymbol{x}\in\mathbb{X}_{\mathrm{MILP}_{\mathrm{down}}} = \{\boldsymbol{x} : \boldsymbol{x}\in\mathbb{X}_{\mathrm{MILP}}, x_j \leqslant \lfloor \underline{x}_j\rfloor\}.
\end{aligned}$$

Since $\mathbb{X}_{\mathrm{MILP}} = \mathbb{X}_{\mathrm{MILP}_{\mathrm{up}}} \cup \mathbb{X}_{\mathrm{MILP}_{\mathrm{down}}}$, the feasibility of the original MILP can be determined by the feasibilities of the two sub-MILPs, and the optimal solution, if it exists, must be the best among the two optimal solutions of $\mathrm{MILP}_{\mathrm{up}}$ and $\mathrm{MILP}_{\mathrm{down}}$. Thus, solving the original problem can be reduced to solving two sub-MILPs. To address $\mathrm{MILP}_{\mathrm{up}}$ and $\mathrm{MILP}_{\mathrm{down}}$, we relax their integer constraints, yielding LP relaxations denoted as $\mathrm{LP}_{\mathrm{up}}$ and $\mathrm{LP}_{\mathrm{down}}$, respectively. The LP optimal solutions are denoted as $\underline{\boldsymbol{x}}_{\mathrm{up}}$ and $\underline{\boldsymbol{x}}_{\mathrm{down}}$, respectively. Compared with the original LP solution $\underline{\boldsymbol{x}}$, $\underline{\boldsymbol{x}}_{\mathrm{up}}$ and $\underline{\boldsymbol{x}}_{\mathrm{down}}$ are closer to the MILP's optimal solution in two respects:

• (Feasibility) Both $\underline{\boldsymbol{x}}_{\mathrm{up}}$ and $\underline{\boldsymbol{x}}_{\mathrm{down}}$ are more likely to yield integer solutions. Typically, in the cases where integer variables are binary, $\mathrm{LP}_{\mathrm{up}}$ fixes x_j as 1 and $\mathrm{LP}_{\mathrm{down}}$ fixes x_j as 0, satisfying $x_j\in\mathbb{Z}$. Even in the general cases, $\mathrm{LP}_{\mathrm{up}}$ and $\mathrm{LP}_{\mathrm{down}}$ help narrow down the search space by eliminating non-integer values in $(\lfloor\underline{x}_j\rfloor, \lceil\underline{x}_j\rceil)$.

• (Lower bounds) The solutions $\underline{\boldsymbol{x}}_{\mathrm{up}}$ and $\underline{\boldsymbol{x}}_{\mathrm{down}}$ provide tighter lower bounds than $\underline{\boldsymbol{x}}$:

$$\boldsymbol{c}^\top\underline{\boldsymbol{x}} \leqslant \min(\boldsymbol{c}^\top\underline{\boldsymbol{x}}_{\mathrm{up}}, \boldsymbol{c}^\top\underline{\boldsymbol{x}}_{\mathrm{down}}) \leqslant \boldsymbol{c}^\top\boldsymbol{x}, \quad \text{for all } \boldsymbol{x}\in\mathbb{X}_{\mathrm{MILP}}. \tag{6.2}$$

This is because the union of feasible sets of $\mathrm{LP}_{\mathrm{up}}$ and $\mathrm{LP}_{\mathrm{down}}$ forms a strict subset of the original LP relaxation:

$$\mathbb{X}_{\mathrm{LP}} \supsetneq (\mathbb{X}_{\mathrm{LP}}\cap\{\boldsymbol{x} : x_j \geqslant \lceil\underline{x}_j\rceil\}) \cup (\mathbb{X}_{\mathrm{LP}}\cap\{\boldsymbol{x} : x_j \leqslant \lfloor\underline{x}_j\rfloor\}).$$

If $\mathrm{frac}(\underline{\boldsymbol{x}}_{\mathrm{up}}) = \emptyset$, we conclude that $\underline{\boldsymbol{x}}_{\mathrm{up}}$ must be an optimal solution to $\mathrm{MILP}_{\mathrm{up}}$, and *at least a feasible solution to the original MILP*. Its optimality will be determined after solving $\mathrm{MILP}_{\mathrm{down}}$. Otherwise, we conclude that $\underline{\boldsymbol{x}}_{\mathrm{up}}$ is still not a feasible solution to the original MILP, and we repeat the above process and branch $\mathrm{MILP}_{\mathrm{up}}$ into two sub-problems. We handle $\mathrm{MILP}_{\mathrm{down}}$ with a similar approach.

Bounding. If we view each sub-MILP as a node, these nodes together make up a binary tree known as *the BnB tree*. In this context, the original MILP is regarded as the *root node* of the entire BnB tree, and $\mathrm{MILP}_{\mathrm{up}}$ and $\mathrm{MILP}_{\mathrm{down}}$ are seen as the two *children* of the root, and subsequent sub-problems of $\mathrm{MILP}_{\mathrm{up}}$ are considered children of $\mathrm{MILP}_{\mathrm{up}}$, and so on. Expanding this tree fully can be time-consuming. While in practice, some nodes might be pruned. To discuss the pruning method, we first need to understand two basic concepts:

• (Upper bound) During the branching process, as soon as we find a feasible solution for the original MILP, we store it in memory. It remains there until a solution with a lower objective value is found and replaces it. This stored solution, denoted as $\overline{\boldsymbol{x}}$, is known as the *incumbent* solution and serves as an upper limit for the optimal objective value:

$$\boldsymbol{c}^\top \boldsymbol{x}^* \leqslant \boldsymbol{c}^\top \overline{\boldsymbol{x}},$$

where $\boldsymbol{x}^*$ represents any optimal solutions of the original MILP. The upper bound of the optimal objective value, represented as $\overline{f} := \boldsymbol{c}^\top \overline{\boldsymbol{x}}$, is known as the *global upper bound* or the *primal bound*.

• (Lower bound) As mentioned before, the LP relaxations of the MILP or sub-MILPs provide lower bounds of the objective value. After obtaining $\underline{\boldsymbol{x}}$, we get a lower bound $\underline{f} := \boldsymbol{c}^\top \underline{\boldsymbol{x}}$, denoted as the *global lower bound* or the *dual bound*. After $\mathrm{LP}_{\mathrm{up}}$ and $\mathrm{LP}_{\mathrm{down}}$ are solved, the dual bound will be updated with a higher value $\min(\boldsymbol{c}^\top \underline{\boldsymbol{x}}_{\mathrm{up}}, \boldsymbol{c}^\top \underline{\boldsymbol{x}}_{\mathrm{down}})$. This process continues throughout the BnB procedure.

As we progress through this procedure, the primal bound decreases while the dual bound increases until they converge. Based on these two bounds, we can prune certain nodes in the BnB tree. If the optimal objective value of the LP relaxation of a sub-MILP exceeds the global upper bound, we can prune this sub-MILP (or node) as neither it nor its children can produce a solution better than the incumbent solution. It is important to note that this includes instances where the LP relaxation is infeasible since any children of an infeasible sub-MILP must also be infeasible. A full description of BnB is provided in Algorithm 2.

Algorithm 2 Branch and bound algorithm

```
Input: An MILP instance $(\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}})$; Variable selection rule; Node selection rule
Output: Label "Infeasible", Label "Infeasible or Unbounded", or an optimal solution $\overline{\boldsymbol{x}}$;
 1: Active node set $\mathbb{S} \Leftarrow \{\}$. {The set $\mathbb{S}$ consists of tuples structured as $(d, f, \mathbb{X})$. Each of these tuples symbolizes a node
    within the BnB tree. Here, $d$ stands for the depth of the node, $f$ signifies a *local lower bound* of the node - equivalent
    to the node's parent's optimal LP objective value, while $\mathbb{X} \subset \mathbb{X}_{\mathrm{MILP}}$ characterizes the feasible set of the sub-MILP
    corresponding to the node.}
 2: The global lower bound $\underline{f} \Leftarrow -\infty$, the global upper bound $\overline{f} \Leftarrow +\infty$.
 3: Push a node with data $(0, \underline{f}, \mathbb{X}_{\mathrm{MILP}})$ into $\mathbb{S}$.
 4: while $\mathbb{S}$ is not empty do
 5:   Pop a node $(d, f, \mathbb{X})$ from $\mathbb{S}$ with the node selection rule.
 6:   if The parent's objective exceeds the upper bound: $f \geqslant \overline{f}$ then
 7:     continue
 8:   end if
 9:   Update the global lower bound: $\underline{f} \Leftarrow \min_{(d,f,\mathbb{X})\in\mathbb{S}} f$.
10:   if The lower and upper bound converge $\overline{f} = \underline{f}$ then
11:     return $\overline{\boldsymbol{x}}$.
12:   end if
13:   Solve $(\boldsymbol{c}, \mathrm{Relax}(\mathbb{X}))$ and denote its optimal solution as $\underline{\boldsymbol{x}}$ if feasible. {Here $(\boldsymbol{c}, \mathrm{Relax}(\mathbb{X}))$ represents the LP relaxation
      of $(\boldsymbol{c}, \mathbb{X})$.}
14:   if $(\boldsymbol{c}, \mathrm{Relax}(\mathbb{X}))$ is infeasible or its objective exceeds the upper bound: $\boldsymbol{c}^\top \underline{\boldsymbol{x}} \geqslant \overline{f}$ then
15:     continue
16:   else if $(\boldsymbol{c}, \mathrm{Relax}(\mathbb{X}))$ is lower unbounded then
17:     return Infeasible or Unbounded.
18:   else
19:     if $\underline{\boldsymbol{x}}$ satisfies integer constraints: $\mathrm{frac}(\underline{\boldsymbol{x}}) = \emptyset$ then
20:       if $\underline{\boldsymbol{x}}$ has better objective $\boldsymbol{c}^\top \underline{\boldsymbol{x}} < \overline{f}$ then
21:         Update the upper bound and the incumbent: $\overline{f} \Leftarrow \boldsymbol{c}^\top \underline{\boldsymbol{x}}$, $\overline{\boldsymbol{x}} \Leftarrow \underline{\boldsymbol{x}}$.
22:         if The lower and upper bound converge $\overline{f} = \underline{f}$ then
23:           return $\overline{\boldsymbol{x}}$.
24:         end if
25:       end if
26:     else
27:       Pick an index $j \in \mathrm{frac}(\underline{\boldsymbol{x}})$ with the variable selection rule.
28:       Create two branches: $\mathbb{X}_{\mathrm{up}} = \mathbb{X} \cap \{\boldsymbol{x} : x_j \geqslant \lceil \underline{x}_j \rceil\}$, $\mathbb{X}_{\mathrm{down}} = \mathbb{X} \cap \{\boldsymbol{x} : x_j \leqslant \lfloor \underline{x}_j \rfloor\}$.
29:       Add $(d+1, \boldsymbol{c}^\top \underline{\boldsymbol{x}}, \mathbb{X}_{\mathrm{up}})$ and $(d+1, \boldsymbol{c}^\top \underline{\boldsymbol{x}}, \mathbb{X}_{\mathrm{down}})$ into $\mathbb{S}$.
30:     end if
31:   end if
32: end while
33: return Infeasible if $\overline{f} = +\infty$ else return $\overline{\boldsymbol{x}}$
```

Efficiency analysis. The BnB approach is known to terminate in finite steps when applied to MILP problems with rational parameters, regardless of the sequence in which variables and nodes are chosen. However, the effectiveness of the BnB method significantly depends on the strategies used for variable selection and node selection. As demonstrated in Figure 4, certain examples show a case of how varying variable selection rules result in final BnB trees of different sizes. The same conclusion is observed in Figure 5 with various node selection rules.

These examples prompt us to question: *What are the optimal strategies for variable and node selection for a specific type of MILP instances*? Unfortunately, there are no definitive answers to this inquiry. It is still typical for practitioners to develop these strategies based on a *trial and error* approach. By testing many different strategies on a specific type of MILP (e.g., scheduling, bin-packing, vehicle routing), practitioners can gain domain-specific knowledge that informs the development of new strategies. However, this iterative process is problematic due to the following:

- Its labor-intensive nature.
- The absence of a systematic methodology. The acquired domain knowledge is specific to the type of MILP and does not easily translate to other domains.

Hence, it would be advantageous to develop a systematic approach to autonomously create or discover effective variable and node selection rules. The remarkable advancements in machine learning techniques in recent years lead us to ask: *Can we leverage machine learning to extract powerful strategies from data*? In the following subsections, we will show how to learn variable and node selection rules with machine learning.

6.2.2 *Variable selection*

The variable selection rule, also known as the *branching rule*, determines which variable to branch on when multiple fractional values are present in the current LP relaxation solution. To identify the ideal variable to branch on, we collect information related to all potential variables and make decisions based on the collected data, which may include:

- the original MILP: $\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}}$;
- the current MILP and its LP relaxation: $\mathbb{X}$, $\mathrm{Relax}(\mathbb{X})$ and $\underline{\boldsymbol{x}}$ (defined in Algorithm 2, Line 13);
- data generated during BnB: $\underline{f}$, $\overline{f}$, $\overline{\boldsymbol{x}}$, $\mathbb{S}$.

All the aforementioned information is accessible during the BnB procedure. Now, let us present the formal formulation of a branching rule:

$$s_j = \mathcal{V}(j, \cdot), \quad j^* = \mathop{\arg\max}_{j \in \mathrm{frac}(\underline{\boldsymbol{x}})} s_j. \tag{6.3}$$

In this formulation, $\cdot$ represents the collected information, and $\mathcal{V}$ is a mapping function that assigns a real number, which is named as a score, to a variable. Subsequently, the index with the top score is selected for branching. In case of a tie, lexicographical order is usually adopted. The mapping $\mathcal{V}$ forms the cornerstone of this rule and is undoubtedly the main component we aim to design or learn. Some typical handcrafted branching rule is listed here:

- (Most infeasible branching) We use the fractionality as the score: $s_j = \min(\underline{x}_j - \lfloor \underline{x}_j \rfloor, \lceil \underline{x}_j \rceil - \underline{x}_j)$.
- (Strong Branching) A primary aim of branching is to improve the lower bound. The strong branching (SB) rule seeks to achieve this by testing each variable $j \in \mathrm{frac}(\underline{\boldsymbol{x}})$ to see which one significantly tightens the lower bound the most. Suppose there are n_{frac} elements in $\mathrm{frac}(\boldsymbol{x})$, a full strong branching constructs $2n_{\mathrm{frac}}$ sub-MILPs with

$$\mathbb{X}_{j,\mathrm{up}} = \mathbb{X} \cap \{\boldsymbol{x} : x_j \geqslant \lceil \underline{x}_j \rceil\}, \quad \mathbb{X}_{j,\mathrm{down}} = \mathbb{X} \cap \{\boldsymbol{x} : x_j \leqslant \lfloor \underline{x}_j \rfloor\}, \quad j \in \mathrm{frac}(\boldsymbol{x}).$$

Subsequently, we solve the LP relaxations of these MILPs $\{(\boldsymbol{c}, \mathbb{X}_{j,\mathrm{up}}), (\boldsymbol{c}, \mathbb{X}_{j,\mathrm{down}})\}_{j \in \mathrm{frac}(\boldsymbol{x})}$. The solutions of these are represented as $\{\underline{\boldsymbol{x}}_{j,\mathrm{up}}, \ \underline{\boldsymbol{x}}_{j,\mathrm{down}}\}_{j \in \mathrm{frac}(\underline{\boldsymbol{x}})}$ respectively. The improvement in the objective of the two sub-MILPs, obtained by branching along x_j, is computed as $\delta_{j,\mathrm{up}} = \boldsymbol{c}^\top \underline{\boldsymbol{x}}_{j,\mathrm{up}} - \boldsymbol{c}^\top \underline{\boldsymbol{x}}$ and $\delta_{j,\mathrm{down}} = \boldsymbol{c}^\top \underline{\boldsymbol{x}}_{j,\mathrm{down}} - \boldsymbol{c}^\top \underline{\boldsymbol{x}}$. Given a small number ε (say $\varepsilon = 10^{-8}$), the SB score of x_j is derived by

$$s_{j,\mathrm{SB}} = \mathcal{V}_{\mathrm{SB}}(j, \boldsymbol{c}, \mathbb{X}, \underline{\boldsymbol{x}}, \varepsilon) = \max(\delta_{j,\mathrm{up}}, \varepsilon) \times \max(\delta_{j,\mathrm{down}}, \varepsilon). \tag{6.4}$$

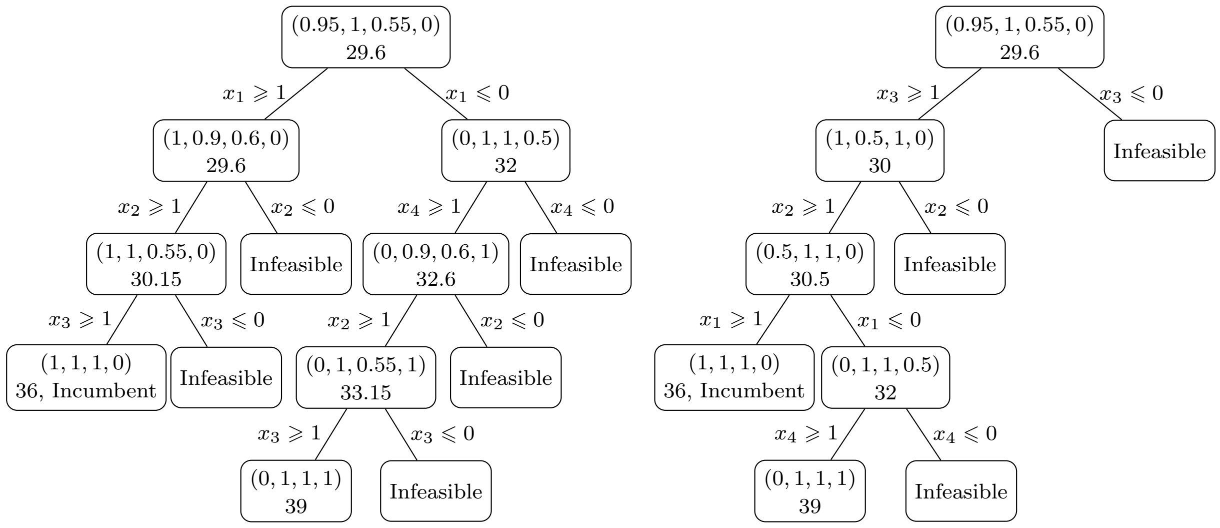


Figure 4 Variable selection can influence the size of the BnB tree. Consider the MILP: $\min 11x_1 + 12x_2 + 13x_3 + 14x_4$ subject to $x_1 + x_2 + x_3 + x_4 \geqslant 2.5$, $x_2 + 2x_3 \geqslant 2.1$ and all the variables must be binary. The root node represents an optimal solution $\underline{\boldsymbol{x}} = (0.95, 1, 0.55, 0)$ to the LP relaxation and its corresponding objective $\boldsymbol{c}^\top \underline{\boldsymbol{x}} = 29.6$. In $\underline{\boldsymbol{x}}$, two elements are fractional: x_1 and x_3. We have to decide which one to branch on. If we adopt the first approach—branching based on *the fractional element with the smallest index*—we will choose x_1 to branch on, leading to the tree shown in the left figure. On the other hand, if we employ a different strategy—branching based on *the element with the largest fractionality*—the choice is determined by $\arg\max_j \min(\underline{x}_j - \lfloor \underline{x}_j \rfloor, \lceil \underline{x}_j \rceil - \underline{x}_j)$. In this case, x_3 is selected, resulting in the tree in the right figure. Continuing this process in either case will yield different BnB trees. Note that, in this case, the node selection strategy will not influence the BnB tree, and you can verify this with any node selection method

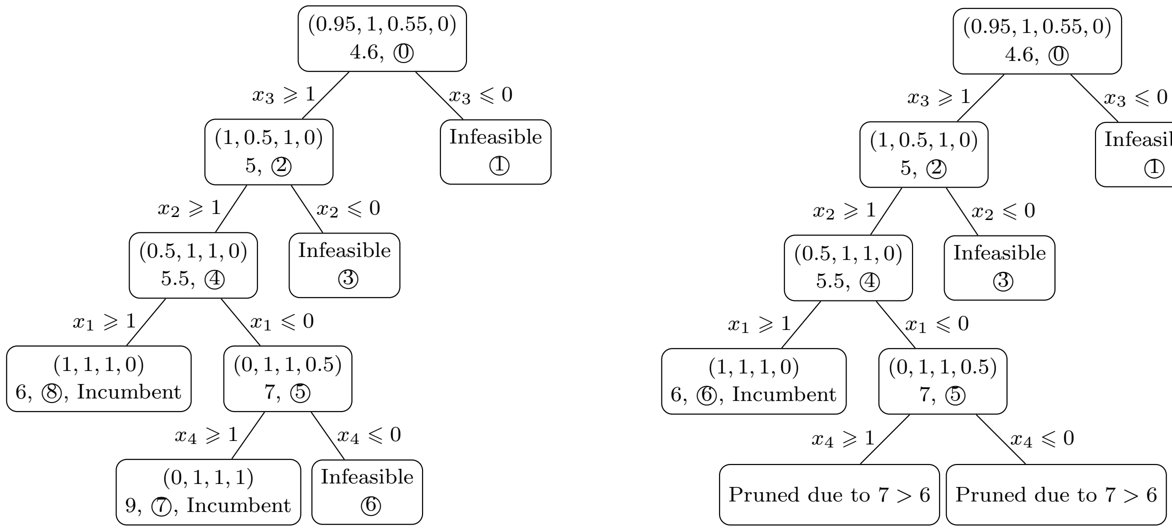


Figure 5 Node selection can influence the size of the BnB tree. Consider the MILP: $\min x_1 + 2x_2 + 3x_3 + 4x_4$ subject to $x_1 + x_2 + x_3 + x_4 \geqslant 2.5$, $x_2 + 2x_3 \geqslant 2.1$ and all the variables must be binary. We consistently branch based on the element with the highest fractionality but employ different strategies for node selection. In the figure on the left, we employ a *depth-first* order for node selection, which results in the need to solve a total of 9 LP problems. The order to access nodes are: ⓪, ①, ②, ③, ④, ⑤, ⑥, ⑦, ⑧. Conversely, the figure on the right demonstrates the use of a *breadth-first* order for node selection, which leads to pruning 2 LPs. The order to access nodes are: ⓪, ①, ②, ③, ④, ⑤, ⑥

• (Pseudocost branching) The pseudocost (PC) branching rule is essentially an SB approximation that relies on historical data. It observes that a single variable is often branched multiple times. For example, in the left tree in Figure 4, x_2 and x_3 are branched two times. Therefore, historical objective improvements can be stored and averaged for each variable, providing an estimate for predicting the SB score. Specifically, we maintain two quantities $\delta f_{j,\mathrm{up}}$ and $\delta f_{j,\mathrm{down}}$ for each variable x_j and they are initialized by zero. Every time we branch along x_j, we calculate the "objective improvement per unit" and store its historical average in $\delta f_{j,\mathrm{up}}$ and $\delta f_{j,\mathrm{down}}$ respectively:

$$\frac{\boldsymbol{c}^\top \underline{\boldsymbol{x}}_{j,\mathrm{up}} - \boldsymbol{c}^\top \underline{\boldsymbol{x}}}{\lceil \underline{x}_j \rceil - \underline{x}_j}, \quad \frac{\boldsymbol{c}^\top \underline{\boldsymbol{x}}_{j,\mathrm{down}} - \boldsymbol{c}^\top \underline{\boldsymbol{x}}}{\underline{x}_j - \lfloor \underline{x}_j \rfloor}.$$

With the two maintained quantities, we can calculate estimates for objective improvements: $\delta_{j,\mathrm{up}} = (\lceil \underline{x}_j \rceil - \underline{x}_j)\delta f_{j,\mathrm{up}}$ and $\delta_{j,\mathrm{down}} = (\underline{x}_j - \lfloor \underline{x}_j \rfloor)\delta f_{j,\mathrm{down}}$. Plugging these estimates into the right-hand side of (6.4), we will obtain an approximated SB score, named the PC score $s_{j,\mathrm{PC}}$.

(Imitating SB) Simple branching rules, such as the first fractional rule and the most fractional rule presented in Figure 4, are straightforward but may lack effectiveness due to their short-sighted nature. The strong branching rule (6.4), while leading to potentially higher lower bound increases, requires solving multiple LPs, and hence is computationally expensive. While the pseudo-cost rule reduces the overhead compared with SB but is often found to be less effective in improving node efficiency. *A powerful yet efficient approximation of SB without solving large amounts of LPs would be greatly beneficial.* Motivated by this point, we are developing efficient branching rules that imitate SB while the complexity is controlled by a budget B (informal form):

$$\min_{\mathcal{V}} \sum_{(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}})\in\mathbb{M}} \sum_{\text{BnB node}} \sum_{j} \|s_{j,\mathrm{SB}} - \mathcal{V}(j,\cdot)\| \quad \text{s.t.} \quad \mathrm{Complexity}(\mathcal{V}) \leqslant B. \tag{6.5}$$

(Evaluating branching rules) To evaluate the performance of a mapping $\mathcal{V}$ for a given MILP instance, it is necessary to use a consistent node selection rule, for example, Depth-First Search (DFS), commonly adopted in contemporary MILP solvers. This enables the evaluation of the BnB performance given the mapping $\mathcal{V}$ and its resulting variable selection strategy. In particular, it is recommended to embed a timer in the BnB algorithm and track the upper bound $\overline{f}$ and the lower bound $\underline{f}$ in real-time. These quantities at a specific moment t can be denoted as $\overline{f}(t)$ and $\underline{f}(t)$. Assuming the BnB begins solving $(\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}})$ at time $t = 0$ and finishes at T_{end}, a typical metric can be expressed as follows:

$$I(\mathcal{V}, (\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}}), \alpha) = \alpha \int_0^{T_{\mathrm{end}}} (\overline{f}(t) - \underline{f}(t))\mathrm{d}t + T_{\mathrm{end}}. \tag{6.6}$$

In BnB, the upper bound $\overline{f}$ decreases over time t while the lower bound $\underline{f}$ increases. When the gap $\overline{f}(t) - \underline{f}(t)$ vanishes, we obtain an optimal solution of the MILP. Hence, we aim for the primal-dual gap to decrease as rapidly as possible. This drives the first term in (6.6). In addition, a scalar α balances the two terms in (6.6). In contrast to the compromise goal (6.5), a more ambitious goal is to seek a variable selection rule $\mathcal{V}$ that minimizes the measure across a set of MILP instances $\mathbb{M}$:

$$\min_{\mathcal{V}} \sum_{(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}})\in\mathbb{M}} I(\mathcal{V}, (\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}}), \alpha). \tag{6.7}$$

6.2.3 *Learning to branch*

Utilizing modern machine learning tools and optimization solvers, it is possible to quickly implement a pipeline targeting (6.5). This pipeline can be universally applicable to any data set $\mathbb{M}$, regardless of the need for specialized knowledge related to the specific form of the MILP instances in $\mathbb{M}$. Although incorporating domain knowledge into machine learning may enhance performance, we are suggesting here the feasibility of an elementary model that operates without such knowledge. The pipeline consists of the following critical stages:

• (Parameterization) Initially, one needs to define the input $\cdot$ of $\mathcal{V}$ and establish the explicit form of $\mathcal{V}$. There should be a sufficient number of parameters in the form of $\mathcal{V}$ to allow a search for parameters that best fit (6.5). Neural networks, due to their strong representation capacity, are a good choice to express $\mathcal{V}$. (For further comprehension on parameterizing a mapping with a neural network, refer to Subsection 2.2.) One has the flexibility to choose from various neural network structures or sizes (in terms of the number of parameters). Once the neural network's structure and size have been chosen, the complexity of invoking the neural network is limited by an upper bound, which satisfies the constraint in (6.5).

• (Data collection) Following this, it is necessary to collect data for training during the BnB process. At each BnB node, we collect "(*feature, label*)" pairs. The *feature* refers to the specific values of the input $\cdot$ under the current circumstances, while the *label* corresponds to the target we want the neural network's output to match. In this case, we can use the SB score as label.

• (Training and validation) This stage follows a standard supervised learning procedure. Given those "(feature, label)" pairs, we can train a neural network to fit the strong branching. In the training process, there might be some hyperparameters that should be tuned by hand. We can use a separate validation set to help us choose the best hyperparameters.

• (BnB with trained models) Upon completion of the training, the trained neural network can be applied during the BnB process. Specifically, one can invoke the neural network to assign a score to each branching candidate in $\mathrm{frac}(\underline{\boldsymbol{x}})$ in Algorithm 2, and select the candidate with the highest score to branch on, just as in (6.3).

The entire pipeline develops branching rules with machine learning, and is named *learning to branch* (L2B). In the subsequent paragraphs, we will individually discuss the components of this pipeline.

Parameterization and complexity. We discuss several neural network-based scoring rules and compare their complexity with traditional score calculations like SB and PC.

Approach 1: Separate models. The most straightforward parameterization method treats each variable as an independent entity. Features of variables are gathered individually and processed separately using a shared neural network. More specifically, the feature vector for each variable may include elements like $\boldsymbol{f}_j = (c_j, l_j, u_j, z_j, \underline{x}_j, s_{j,\mathrm{PC}})$, where $c_j,\ l_j,\ u_j$ are as defined in (6.1), z_j signifies if x_j is mandated to be an integer (where $z_j = 1$ if $j \in \mathbb{I}$ and $z_j = 0$ otherwise), $\underline{x}_j$ is the j-th component of $\underline{\boldsymbol{x}}$ as defined in Line 13 of Algorithm 2, and $s_{j,\mathrm{PC}}$ represents the PC score. The ultimate goal is to build a machine learning model, such as an MLP (2.3), that can map this feature vector to the strong branching score:

$$\mathcal{V}_{\mathrm{Sepa}}(j, \boldsymbol{c}, \boldsymbol{l}, \boldsymbol{u}, \boldsymbol{z}, \underline{\boldsymbol{x}}, \boldsymbol{s}_{\mathrm{PC}}; \boldsymbol{\theta}) := \mathrm{MLP}(\boldsymbol{f}_j; \boldsymbol{\theta}) \approx s_{j,\mathrm{SB}}, \tag{6.8}$$

where $\boldsymbol{\theta}$ denotes the parameters in that neural network.

Graph representations. To enrich the previous model, we will consider the relationship between variables and constraints in Approaches 2 and 3. To clearly describe this point, we adopt the language of graphs. We start by demonstrating how to model an MILP instance using a *bipartite* graph, as shown in Figure 6. Variables are represented as nodes in the "$\mathbb{V}$" group, while constraints are represented as nodes in the "$\mathbb{C}$" group. No edges exist within each group. An edge with weight $A_{i,j}$ connects the i-th constraint to the j-th variable if $A_{i,j} \neq 0$. Each variable node carries a feature vector $\boldsymbol{f}_j$, and each constraint node carries a feature vector $\boldsymbol{g}_i$ that may include quantities related to that specific constraint, like $\boldsymbol{g}_i = (\circ_i, b_i)$. (Recall (6.1) for the definition of "$\circ, \boldsymbol{b}$".) For each variable node $j \in \{1, \ldots, n\}$, $\mathcal{N}(j)$ is denoted as the set of its adjacent constraints. Conversely, $\mathcal{N}(i)$ denotes the set of variables involved in the i-th constraint "$\mathbf{A}_{i,:}\boldsymbol{x} \circ_i b_i$".

Approach 2: Features augmented with graph information. We can enrich $\boldsymbol{f}_j$ by incorporating the relationships between variables and constraints. Rather than independently gathering features for each variable, we collect additional features that represent the neighborhood of x_j. These features can include statistical information about $\{\boldsymbol{g}_i\}_{i \in \mathcal{N}(j)}$, such as the distribution of constraints with $\circ_i$ being equal to "$\geqslant$","$=$" and "$\leqslant$", respectively, and the mean and standard deviation of the set $\{b_i\}_{i \in \mathcal{N}(j)}$. We

$$\begin{aligned}\min_{\boldsymbol{x}}\ & x_1+2x_2+3x_3\\ \text{s.t. }& x_1+x_2\geqslant 1\\ & x_2+2x_3\leqslant 2\\ & x_1,x_2,x_3\in\{0,1\}\end{aligned}$$

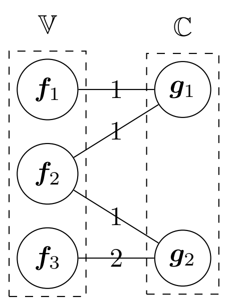
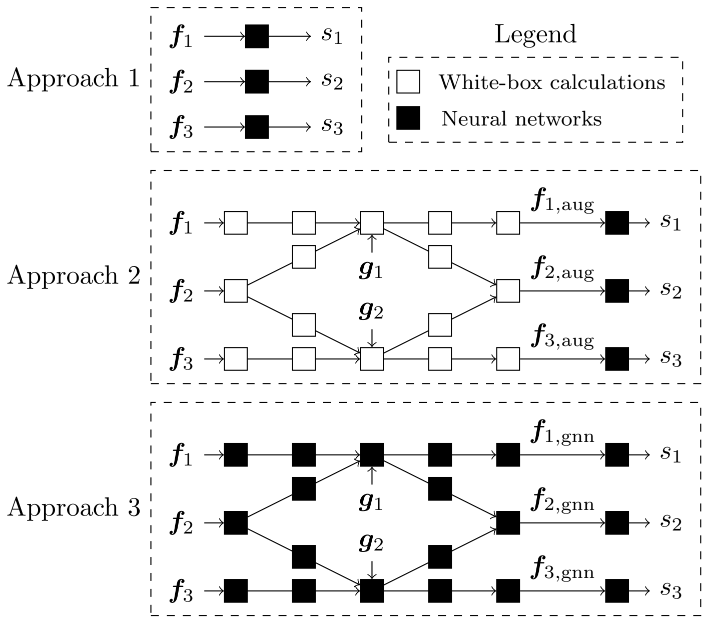


Figure 6 An MILP instance represented by a bipartite graph



Approach 1
$\boldsymbol{f}_1$ s_1
$\boldsymbol{f}_2$ s_2
$\boldsymbol{f}_3$ s_3
Legend
☐ White-box calculations
■ Neural networks
Approach 2
$\boldsymbol{f}_{1,\text{aug}}$
$\boldsymbol{f}_{2,\text{aug}}$
$\boldsymbol{f}_{3,\text{aug}}$
$\boldsymbol{g}_1$
$\boldsymbol{g}_2$
Approach 3
$\boldsymbol{f}_{1,\text{gnn}}$
$\boldsymbol{f}_{2,\text{gnn}}$
$\boldsymbol{f}_{3,\text{gnn}}$

Figure 7 Illustration of three methods to represent $\mathcal{V}(j,\cdot)$

can also consider more complex features including information about variables that are adjacent to x_j's neighbors, like the mean and standard deviation of these variables' objective coefficients, lower bounds, and upper bounds. These values can be appended to $\boldsymbol{f}_j$ to create an enhanced feature vector $\boldsymbol{f}_{j,\text{aug}}$, which provides a more comprehensive view of a variable. Using these enhanced features, we can more accurately predict the SB score:

$$\mathcal{V}_{\text{Aug}}(j,\boldsymbol{c},\boldsymbol{l},\boldsymbol{u},\boldsymbol{z},\underline{\boldsymbol{x}},\boldsymbol{s}_{\text{PC}},\mathbf{A},\circ,\boldsymbol{b};\boldsymbol{\theta}):=\text{MLP}(\boldsymbol{f}_{j,\text{aug}};\boldsymbol{\theta})\approx s_{j,\text{SB}}. \tag{6.9}$$

Approach 3: Graph neural network. Graph neural networks (GNNs) offer a way to represent and capture the essential information about the relationships between variables and constraints, without manual intervention. Specifically, we alternate between passing information from variables to constraints and vice versa, iteratively updating the feature vectors:

$$\begin{aligned}\boldsymbol{g}_i&\leftarrow \text{MLP}\Bigg(\boldsymbol{g}_i,\sum_{j\in\mathcal{N}(i)}\text{MLP}(\boldsymbol{g}_i,\boldsymbol{f}_j,A_{i,j};\boldsymbol{\theta}_1);\boldsymbol{\theta}_2\Bigg),\\ \boldsymbol{f}_j&\leftarrow \text{MLP}\Bigg(\boldsymbol{f}_j,\sum_{i\in\mathcal{N}(j)}\text{MLP}(\boldsymbol{g}_i,\boldsymbol{f}_j,A_{i,j};\boldsymbol{\theta}_3);\boldsymbol{\theta}_4\Bigg).\end{aligned} \tag{6.10}$$

After several iterations, the final resulting variable and node features are denoted as $\{\boldsymbol{f}_{j,\text{gnn}}\}_{j=1}^n$ and $\{\boldsymbol{g}_{i,\text{gnn}}\}_{i=1}^m$. An independent model is applied to each $\boldsymbol{f}_{j,\text{gnn}}$ to predict the SB score: $\text{MLP}(\boldsymbol{f}_{j,\text{gnn}};\boldsymbol{\theta}_5)\approx s_{j,\text{SB}}$. For more information about the merits of GNNs, such as scalability and permutation invariance, please refer to Subsection 2.4. The complete model can be abbreviated as

$$\mathcal{V}_{\text{GNN}}(j,\boldsymbol{c},\boldsymbol{l},\boldsymbol{u},\boldsymbol{z},\underline{\boldsymbol{x}},\boldsymbol{s}_{\text{PC}},\mathbf{A},\circ,\boldsymbol{b};\boldsymbol{\theta}):=\text{GNN}(j,\mathcal{G};\boldsymbol{\theta})\approx s_{j,\text{SB}}. \tag{6.11}$$

Here $\mathcal{G}=(\mathbf{A},\{\boldsymbol{f}_j\}_{j=1}^n,\{\boldsymbol{g}_i\}_{i=1}^m)$ represents the entire input to a GNN and $\boldsymbol{\theta}=(\boldsymbol{\theta}_1,\boldsymbol{\theta}_2,\boldsymbol{\theta}_3,\boldsymbol{\theta}_4,\boldsymbol{\theta}_5)$ includes all parameters in the GNN. A comparison between these three methods can be found in Figure 7. Note that this diagram only displays one iteration of message passing (from variables to constraints and back), while in practice, multiple rounds might be employed.

Complexity analysis. Initially, we will discuss the complexity of SB and PC. In SB, the primary computational overhead lies in solving $2n_{\text{frac}}$ LPs. There is still no polynomial-time algorithm for finding an exact LP solution. The simplex method, often used for LPs in BnB, may exhibit exponential

Table 1 Summary of the complexities of branching rules

SB	PC	Approach 1	Approach 2	Approach 3
$\mathcal{O}(\mathcal{C}_{\text{LP}}\ n_{\text{frac}})$	$\mathcal{O}(n_{\text{frac}})$	$\mathcal{O}(\mathcal{C}_{\text{MLP}}\ n_{\text{frac}})$	$\mathcal{O}(L\ \text{nnz}(\mathbf{A}))+\mathcal{O}(\mathcal{C}_{\text{MLP}}\ n_{\text{frac}})$	$\mathcal{O}(L\ \mathcal{C}_{\text{MLP}}\ \text{nnz}(\mathbf{A}))$

complexity $\mathcal{O}(2^n)$ in the worst case [25]. Despite typically not reaching this upper limit, the LP solving overhead is still substantial in real-world applications, with its complexity being somewhat unpredictable due to these potential worst-case outcomes. In summary, the cost of calculating the SB score is $\mathcal{O}(\mathcal{C}_{\text{LP}}\ n_{\text{frac}})$, where $\mathcal{C}_{\text{LP}}$ denotes the complexity of solving a single LP. In contrast, the cost of PC is significantly lower. It consists of two parts: updating $\delta f_{j,\text{up}}, \delta f_{j,\text{down}}$ for the chosen j and calculating $\delta_{j,\text{up}}, \delta_{j,\text{down}}$ for all $j \in \text{frac}(\underline{\boldsymbol{x}})$. The first part is dominated by standard BnB computations, and the second part is of $\mathcal{O}(n_{\text{frac}})$. Hence, the overall complexity of PC is simply $\mathcal{O}(n_{\text{frac}})$. To simplify the analysis, we will use $\mathcal{C}_{\text{MLP}}$ to denote the complexity of each neural network used in ML-based strategies. In Approach 1, we use an individual model for each candidate in $\text{frac}(\underline{\boldsymbol{x}})$, resulting in a complexity of $\mathcal{O}(\mathcal{C}_{\text{MLP}}\ n_{\text{frac}})$. In Approach 2, when passing information from variables to constraints, the complexity for each constraint is proportional to its number of neighbors. Adding all these complexities, the total is $\mathcal{O}(\text{nnz}(\mathbf{A}))$. The complexity of passing information from constraints to variables is the same. Thus, the complexity of Approach 2 is $\mathcal{O}(L\ \text{nnz}(\mathbf{A}))+\mathcal{O}(\mathcal{C}_{\text{MLP}}\ n_{\text{frac}})$, where L is the number of message-passing rounds. Using similar logic, the complexity of Approach 3 is $\mathcal{O}(L\ \mathcal{C}_{\text{MLP}}\ \text{nnz}(\mathbf{A}))+\mathcal{O}(\mathcal{C}_{\text{MLP}}\ n_{\text{frac}})$. Here, the first term can obviously dominate the second one due to $\text{nnz}(\mathbf{A})) \geqslant n \geqslant n_{\text{frac}}$, so it can be expressed as $\mathcal{O}(L\ \mathcal{C}_{\text{MLP}}\ \text{nnz}(\mathbf{A}))$. A summary of all complexities is provided in Table 1.

Note that $\mathcal{C}_{\text{MLP}}$ is solely determined by the architecture and size of the neural network and can be fully controlled by the user. On the other hand, $\mathcal{C}_{\text{LP}}$ depends on the properties of the LP instance and the solver used, making it less straightforward for the user to control. *This complexity explains our claim that neural networks can fulfill the constraint presented in* (6.5). As we progress from Approach 1 to Approach 3, both the computational load and the power to fit the SB score increase. In practice, the choice of method should make a balance between efficiency and effectiveness.

Data collection. Regardless of the approach chosen, it is necessary to collect sufficient data to train the chosen ML models. The collected data must correspond with the input structure of the ML model. Now let us take Approach 1 as an example. We maintain the data set $\mathbb{D}_{\text{Sepa}}$ in memory and initialize it as empty. Assume we have a set of MILP instances $\mathbb{M}$ available. During data collection, we iterate over all instances in $\mathbb{M}$ and execute BnB (Algorithm 2) for each instance. When Algorithm 2 invokes the variable selection rule, we perform the following steps:

- Compute $\boldsymbol{f}_j$ for all branching candidates $j \in \text{frac}(\underline{\boldsymbol{x}})$.
- Run SB and calculate the SB score $s_{j,\text{SB}}$ for $j \in \text{frac}(\underline{\boldsymbol{x}})$.
- Append these newly gathered data to $\mathbb{D}_{\text{Sepa}}$:

$$\mathbb{D}_{\text{Sepa}} \leftarrow \mathbb{D}_{\text{Sepa}} \cup \{(\boldsymbol{f}_j, s_{j,\text{SB}})\}_{j\in\text{frac}(\underline{\boldsymbol{x}})}.$$

Finally, the set $\mathbb{D}_{\text{Sepa}}$ becomes the training data set. The data collection procedure for Approach 2 aligns with that of Approach 1, with the only modification being the replacement of $\boldsymbol{f}_j$ with $\boldsymbol{f}_{j,\text{aug}}$ and the re-naming of $\mathbb{D}_{\text{Sepa}}$ to $\mathbb{D}_{\text{Aug}}$. The procedure for Approach 3 slightly differs as the GNN's input requires features from the entire MILP, not just those of variables with fractional values. For Approach 3, the data set is labeled as $\mathbb{D}_{\text{GNN}}$, and the following steps are taken when BnB calls the variable selection rule:

- Compute $\boldsymbol{f}_j$ for all variables in the current MILP $j \in \{1,\dots,n\}$.
- Compute $\boldsymbol{g}_i$ for all constraints in the current MILP $i \in \{1,\dots,m\}$.
- Run SB and compute the SB score $s_{j,\text{SB}}$ for $j \in \text{frac}(\underline{\boldsymbol{x}})$.
- Define $\mathcal{G} = (\mathbf{A}, \{\boldsymbol{f}_j\}_{j=1}^n, \{\boldsymbol{g}_i\}_{i=1}^m)$, where $\mathbf{A}$ corresponds to the current MILP. Append these newly collected data to $\mathbb{D}_{\text{GNN}}$:

$$\mathbb{D}_{\text{GNN}} \leftarrow \mathbb{D}_{\text{GNN}} \cup \{(\mathcal{G}, \text{frac}(\underline{\boldsymbol{x}}), \boldsymbol{s}_{\text{SB}})\}.$$

Note that the data form presented here are simply the basic format. *Users can enhance this framework by adding more features.* For example, in variable-related features $\boldsymbol{f}_j$, the reduced cost (an essential concept in LP) can be included; in constraint-related features $\boldsymbol{g}_i$, the dual variable can be added to indicate if the corresponding constraint is active. Such advanced feature engineering can be found in the relevant literature [181]. Another important trick is *invoking SB with a probability.* Triggering SB at each BnB node might be extremely time-consuming. As such, we can call SB and gather data with a small likelihood, say 10%, and in other instances, resort to less expensive branching rules like PC. This tactic is discussed in [130].

Model training and validation. Once we have the datasets $\mathbb{D}_{\text{Sepa}}$, $\mathbb{D}_{\text{Aug}}$, or $\mathbb{D}_{\text{GNN}}$ ready, we can proceed to the model training phase using standard machine learning procedures. For the first two approaches, the machine learning models can be trained by minimizing

$$\min_{\boldsymbol{\theta}} \mathcal{L}_{\text{Sepa}}(\mathbb{D}_{\text{Sepa}}, \boldsymbol{\theta}) := \sum_{(\boldsymbol{f},s)\in\mathbb{D}_{\text{Sepa}}} \ell(s, \text{MLP}(\boldsymbol{f};\boldsymbol{\theta})), \tag{6.12}$$

$$\min_{\boldsymbol{\theta}} \mathcal{L}_{\text{Aug}}(\mathbb{D}_{\text{Aug}}, \boldsymbol{\theta}) := \sum_{(\boldsymbol{f},s)\in\mathbb{D}_{\text{Aug}}} \ell(s, \text{MLP}(\boldsymbol{f};\boldsymbol{\theta})). \tag{6.13}$$

In these equations, ℓ represents the loss function, which should be manually chosen. A typical loss function might be the squared error: $\ell(a,b) = (a-b)^2$. For the GNN model, the loss function for training can be formulated as

$$\min_{\boldsymbol{\theta}} \mathcal{L}_{\text{GNN}}(\mathbb{D}_{\text{GNN}}, \boldsymbol{\theta}) := \sum_{(\mathcal{G},\mathbb{J},\boldsymbol{s})\in\mathbb{D}_{\text{GNN}}} \sum_{j\in\mathbb{J}} \ell(s_j, \text{GNN}(j,\mathcal{G};\boldsymbol{\theta})). \tag{6.14}$$

All these minimization formulas can be seen as specific forms of (6.5). Given the complexity of neural networks can be directly controlled as demonstrated in Table 1, the constraints in (6.5) can be met. Furthermore, minimization over the parameter space can be executed with "back-propagation" which is built into many modern machine learning tools. See Subsection 2.3 for a comprehensive introduction on training machine learning models. Note that certain hyperparameters, like the size of the neural networks and the learning rate, should be manually chosen during the training procedure. These hyperparameters are typically chosen through the *validation* process. In addition to the MILP set used for training $\mathbb{M}$, we prepare an independent MILP set $\mathbb{M}'$, where we execute the same data collection procedure to result in the *validation set* $\mathbb{D}'_{\text{Sepa}}$, $\mathbb{D}'_{\text{Aug}}$, or $\mathbb{D}'_{\text{GNN}}$. Using the validation set, we compute the loss function values $\mathcal{L}_{\cdot}(\cdot, \boldsymbol{\theta}_*)$ with $\boldsymbol{\theta}_*$ as the parameters trained via (6.12), (6.13), or (6.14) with varying hyperparameters. The hyperparameters that yield the lowest loss function value on the validation set are selected. This is a standard validation procedure in machine learning. The model validated with a separate set $\mathbb{M}'$ typically exhibits robust generalization ability, meaning it performs well on instances not included in the training set $\mathbb{M}$.

Branching with training models. Once the model has been trained, it is straightforward to apply it. Given an MILP instance, we run BnB and trigger the trained models when BnB calls for the variable selection rule. Subsequently, each branching candidate is assigned a score, and we choose the candidate with the highest score for branching, as indicated by (6.3).

Exploring new rules with reinforcement learning. Instead of trying to imitate the strong branching, a more ambitious question arises: *Can we develop a branching rule that directly aims to minimize the primal-dual gap integral* (6.7)? In comparison to fitting the strong branching, this method offers two potential advantages:

- The possibility to outperform strong branching. Despite the thoughtful design behind the strong branching rule, it may not be optimal in terms of (6.7). Directly devising branching rules based on (6.7) may lead to a more powerful and innovative rule.
- Freedom from reliance on the resource-intensive strong branching during the data collection phase. The strong branching can be exceedingly time-consuming on large-scale instances. Even a single BnB node might take longer than fully solving the MILP using economic branching rules like the pseudocost. This

can result in inefficient data collection, while a training process that is independent of strong branching can remedy this issue.

Although the supervised learning pipelines discussed earlier in this section are not applicable to solve a minimization problem like (6.7) due to complex dependencies between the branching rule $\mathcal{V}$ and the objective function $I(\cdot)$, this problem fits perfectly within the scope of *reinforcement learning* (RL). First, we demonstrate that the branch and bound method (Algorithm 2) is a well-defined, albeit complex, Markov decision process (MDP), assuming we use a fixed node selection rule like depth-first search (DFS). The key elements of the MDP can be defined as follows (for introduction of MDP and RL, refer to Subsection 2.5):

• State: All the data available when BnB calls for the branching rule: $\boldsymbol{c}$, $\mathbb{X}_{\text{MILP}}$, $\mathbb{X}$, $\underline{\boldsymbol{x}}$, $\underline{f}$, $\overline{\boldsymbol{x}}$, $\overline{f}$, $\mathbb{S}$.

• Action: The action space consists of $\text{frac}(\underline{\boldsymbol{x}})$ and the action is the variable selected for branching.

• Transition: Given the current state and action, execute BnB until it invokes the branching rule again. The resulting new state is the transitioned state and thus the transition probability is defined.

• Reward r: If BnB does not stop, the reward is the negative of the primal-dual gap $r = -(\overline{f} - \underline{f})$ at that point; if BnB stops, the reward is the negative of the total solving time $r = -T_{\text{end}}$. This definition will accurately reconstruct (6.6) when integrated over time.

Using these definitions, we can deploy RL to achieve the minimization outlined in (6.7). Please refer to Subsection 2.5 for an introduction to RL. Contrary to the supervised learning methods (6.12)–(6.14), where we first gather data using the strong branching rule in the BnB framework and then train ML models to fit the SB, RL approaches interact with the BnB: we obtain the state from BnB, make a decision, and send that action to BnB. BnB then returns a reward, indicating whether the lower bound has become tighter or whether BnB has stopped. Using this reward as a guide, we train the ML model. Both basic RL methods, value-based methods like Q-learning and policy-based methods like policy gradient, can be applied here, and we will delve into how these two methods can be utilized in learning to branch.

Q-Learning. The Q-Learning algorithm aims at learning a scoring function, also known as the *Q function*, which maps state-action pairs to scores. Given our definition of GNN model from (6.11), we can consider the input $\mathcal{G}$ as the state and j as the action. Thus, $\text{GNN}(j, \mathcal{G}; \boldsymbol{\theta})$ can be seen as a parameterized Q function. Instead of imitating strong branching as in (6.14), Q-Learning seeks to make the Q function conform to the *Bellman optimality equation*, which is an optimality condition for (6.7):

$$\min_{\boldsymbol{\theta}} \sum_{(\mathcal{G},j,r,\mathcal{G}')} (y - \text{GNN}(j, \mathcal{G}; \boldsymbol{\theta}))^2, \quad \text{where } y = r + \max_{j'} \text{GNN}(j', \mathcal{G}'; \boldsymbol{\theta}_{\text{prev}}). \tag{6.15}$$

Here, $\mathcal{G}'$ represents the subsequent state given the current state $\mathcal{G}$ and action j, and $\boldsymbol{\theta}_{\text{prev}}$ denotes the parameters established before the current training round. For example, one can randomly initialize $\boldsymbol{\theta}_{\text{prev}}$, execute the BnB, and compile a collection of 4-tuples $\{(\mathcal{G}, j, r, \mathcal{G}')\}$. These are then trained using (6.15) to derive $\boldsymbol{\theta}$. This new $\boldsymbol{\theta}$ then serves as $\boldsymbol{\theta}_{\text{prev}}$ in the next training round, eventually yielding $\boldsymbol{\theta}_{\text{next}}$, and so on. Upon obtaining the final parameters $\boldsymbol{\theta}_*$, the corresponding GNN model $\text{GNN}(\cdot, \cdot; \boldsymbol{\theta}_*)$ can be deployed as a branching score within BnB. When the BnB invokes the branching rule, the action $j_* = \max_{j \in \text{frac}(\underline{\boldsymbol{x}})} \text{GNN}(j, \mathcal{G}; \boldsymbol{\theta}_*)$ can be returned as the selected index to branch on.

Policy Gradient In contrast to Q-Learning, policy-based methods utilize a conditional probability distribution as the policy. This policy distribution is state-dependent and proposes a probability for all potential actions. The goal is to train the policy distribution to suggest actions reliably. By making a small modification, the GNN model from (6.11) can be treated as such a policy distribution: If we denote the outcomes of the GNN as s_j for $j \in \{1, \dots, n\}$, we can combine the scores of all branching candidates into a vector $(s_j)_{j \in \text{frac}(\underline{\boldsymbol{x}})}$ and apply a softmax function to that vector. The resulting output vector can then be seen as a conditional probability distribution and each entry in that vector represents the probability to select a specific index for branching. Such a probabilistic policy based on GNN is denoted as $p_{\text{GNN}}(j|\mathcal{G}; \boldsymbol{\theta})$. Now, we use this probabilistic policy as the branching rule and execute the BnB on multiple MILP instances, resulting in a collection of trajectories. We gather the states, actions, and rewards from these trajectories. With this data at hand, we can compute the gradient of the function

(6.6) with respect to the parameters in the policy and use this gradient to update the parameters, just like Subsection 2.5. After training, suppose the final parameters are $\boldsymbol{\theta}_*$. We then use $p_{\mathrm{GNN}}(j \mid \mathcal{G}; \boldsymbol{\theta}_*)$ as the branching rule: sample an index from this distribution and branch along that index.

Bibliographical notes. The methodology of approximating SB via machine learning models can be traced back to 2014 [115]. Initially, research on this topic yielded Approaches 1 and 2 presented in this paper [7, 99]. From 2019 onward, innovative representations of MILP, including bipartite [60] or tripartite representations [52], as well as their related GNNs, have been introduced. Since then, such GNNs (described as Approach 3 in this paper) have been a fundamental model for ML applications in MILP. Recent research has explored alternative neural network architectures, such as transformers [106] and hierarchical structures [181]. More recent advances can be found in [71, 72, 123]. The exploration of reinforcement learning for crafting branching rules emerged later. Preliminary studies showcased that the performance of rules based on RL faced challenges in matching up to those derived by imitating SB. However, recent developments have significantly propelled this domain forward, making RL not only viable but also often outpacing imitation learning in certain contexts. Note that the methods presented in this section are only foundational setups that illustrate the potential of reinforcement learning for branching rules. To achieve competitive results, we recommend referring to recent literature [35, 55, 125, 132, 139, 188], which provides more advanced techniques.

6.2.4 *Node selection and learning to search*

Revisiting Algorithm 2 for the BnB method, we can see that at Line 5, the BnB calls the node selection rule (also referred to as the *searching rule*). This rule selects a node from the set of all active nodes $\mathbb{S}$ (active nodes being those that have not been solved or pruned) and pops it from $\mathbb{S}$. As illustrated in Figure 5, this searching rule can impact the efficiency of BnB. We outline some standard searching strategies used in MILP solvers below.

• Best-first search. This strategy selects the node with the smallest local lower bound. Looking at the global lower bound formula in Line 9 of Algorithm 2: the global lower bound equals the smallest of all local lower bounds from nodes in $\mathbb{S}$. By removing nodes with smaller local lower bounds from $\mathbb{S}$ earlier, the global lower bound can be improved quickly.

• Breadth-first search. This method selects the node with the least depth. Intuitively, nodes with lower depth usually possess a smaller local lower bound. Therefore, breadth-first search can also improve the global lower bound. However, best-first search is a more direct approach, and the breadth-first search technique offers little benefit over it.

• Depth-first search. This method selects the next node candidate at the maximum depth in the tree. Unlike the best-first search, the depth-first search is more likely to produce feasible MILP solutions, often early in the search process, allowing for a faster improvement of the upper bound.

• Hybrid search. This strategy involves switching between different search methods. For example, one might initially employ the depth-first search to find a feasible solution quickly, and then transition to the best-first search to improve the dual bound.

To summarize, a searching rule in BnB is formulated as

$$(d, f, \mathbb{X}) = \operatorname*{arg\,max}_{(d', f', \mathbb{X}') \in \mathbb{S}} \mathcal{S}((d', f', \mathbb{X}'), \cdot). \tag{6.16}$$

In this equation, the mapping $\mathcal{S}$ assigns a score to each node, relying on its data $(d', f', \mathbb{X}')$ and other beneficial information "$\cdot$", such as the original MILP $(\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}})$. Then the node with the highest score $(d, f, \mathbb{X})$ is chosen for exploration, as outlined in Line 5 of Algorithm 2.

Target analysis. The ultimate goal of the node selection rule is the same with the variable selection rule: to minimize the primal-dual gap as quickly as possible and reduce the size of the BnB tree. Formally, assuming a fixed variable selection rule such as the pseudocost rule, we seek the optimal node selection rule over a set of MILP instances $\mathbb{M}$:

$$\min_{\mathbf{S}} \sum_{(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}})\in\mathbb{M}} I(\mathcal{S},(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}}),\alpha) := \alpha\int_0^{T_{\mathrm{end}}} \overline{f}(t) - \underline{f}(t)\mathrm{d}t + T_{\mathrm{end}}. \tag{6.17}$$

Here, $\underline{f}(t)$, $\overline{f}(t)$, and T_{end} have the same definitions as those in (6.6). To further clarify the ambitious objective in (6.17), we discuss "lowering $\overline{f}(t)$" and "boosting $\underline{f}(t)$" separately.

• Improving the dual bound. Initially, we discuss how to improve the dual bound as rapidly as possible by choosing nodes in a suitable order. It can be demonstrated that *the best-first search, previously discussed, is indeed optimal under this criterion.* Assume we have two nodes $(d,f,\mathbb{X})$ and $(d',f',\mathbb{X}')$ in the set $\mathbb{S}$ with $f<f'$. The best-first search proposes to explore $(d,f,\mathbb{X})$ first. If we decide to explore the other node first, even when $(d',f',\mathbb{X}')$ yields an incumbent $\overline{\boldsymbol{x}}$, the node $(d,f,\mathbb{X})$ cannot be pruned because $f<f'\leqslant \boldsymbol{c}^\top\overline{\boldsymbol{x}}$. This implies that investigating $(d,f,\mathbb{X})$ cannot be skipped under any circumstances. The dual bound can only be enhanced after the exploration of $(d,f,\mathbb{X})$, thereby demonstrating that the best-first search is indeed optimal under this measure.

• Decreasing the primal bound. Contrary to improving the dual bound, no node selection rule that guarantees optimal primal bound reduction exists. None of the strategies introduced previously can assure the production of a feasible solution to the MILP and a consequent reduction of the primal bound. However, this presents us with an opportunity for improvement: one can design an effective strategy based on the domain-specific knowledge of a certain type of MILP, or develop a rule from data using machine learning techniques.

Inspired by the above analysis, we will develop a strategy to reduce the primal bound rapidly. *Intuitively, if we can identify the node in* $\mathbb{S}$ *containing the optimal solution of MILP* $\boldsymbol{x}^*$ (*i.e., finding the node* $(d,f,\mathbb{X})\in\mathbb{S}$ *such that* $\boldsymbol{x}^*\in\mathbb{X}$), *selecting that node will be a logical choice.* Branching along this node may yield incumbents $\overline{\boldsymbol{x}}$ with a lower objective function and will ultimately provide the best primal bound because that node comprises the feasible solution with the smallest objective. Therefore, an ideal node scorer can be defined as

$$\mathcal{S}_{\mathrm{ref}}((d,f,\mathbb{X}),\boldsymbol{x}^*) = \begin{cases} 1, & \text{if } \boldsymbol{x}^*\in\mathbb{X},\\ 0, & \text{otherwise.}\end{cases} \tag{6.18}$$

However, this mapping is not directly applicable as a node scorer in BnB due to its dependence on the optimal solution $\boldsymbol{x}^*$. Therefore, it can be advantageous to develop an efficient mapping that approximates $\mathcal{S}_{\mathrm{ref}}$ without relying on $\boldsymbol{x}^*$. Formally, this can be written as

$$\min_{\mathcal{S}} \sum_{(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}})\in\mathbb{M}}\ \sum_{(d,f,\mathbb{X})\in\mathbb{S}} \|\mathcal{S}((d,f,\mathbb{X}),(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}})) - \mathcal{S}_{\mathrm{ref}}((d,f,\mathbb{X}),\boldsymbol{x}^*)\| \quad \text{s.t.} \quad \mathrm{Complexity}(\mathcal{S})\leqslant B. \tag{6.19}$$

Here, $\mathbb{S}$ denotes the active node set generated during the BnB process on the MILP instance $(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}})$ and $\boldsymbol{x}^*$ is one of the optimal solutions to $(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}})$.

Upon examining the formula in (6.19), one might ask: *Is* (6.19) *essentially the same as solving an MILP*? We argue that finding the mapping $\mathcal{S}$ in (6.19) is significantly less challenging than directly finding a solution to an MILP, especially in the early stage of BnB. To illustrate this, consider the scenario where we have just explored the root node of the BnB tree. Based on Algorithm 2, there will be two child nodes in $\mathbb{S}$: $(1,\boldsymbol{c}^\top\underline{\boldsymbol{x}},\mathbb{X}_{\mathrm{up}})$ and $(1,\boldsymbol{c}^\top\underline{\boldsymbol{x}},\mathbb{X}_{\mathrm{down}})$. Now we must decide which node to explore first. As discussed earlier, it would be beneficial to explore the node containing the final optimal solution $\boldsymbol{x}^*$. The goal of $\mathcal{S}$ in (6.19) is merely to predict whether $\boldsymbol{x}^*$ is in $\mathbb{X}_{\mathrm{up}}$ or $\mathbb{X}_{\mathrm{down}}$, without needing to determine the exact value of $\boldsymbol{x}^*$. In common scenarios, both $\mathbb{X}_{\mathrm{up}}$ and $\mathbb{X}_{\mathrm{down}}$ are large sets containing many feasible solutions because BnB is in the early stage, *so determining* $\boldsymbol{x}^*$ *in either of these sets can only provide a rough estimate of* $\boldsymbol{x}^*$*, far from determining the precise location.* Thus, the mapping $\mathcal{S}$ in (6.19) is much less complex than solving an MILP, yet useful in BnB node selection. In the following paragraph, we will discuss how (6.19) can be effectively implemented using machine learning.

Learning to search. Developing a machine learning model to act as a searching rule is referred to as *learning to search* (L2S). Similar to L2B, as discussed in Subsection 6.2.3, L2S follows the same pipeline:

parameterization, data collection, model training and validation, and the application of the trained model within the BnB process.

Parameterization. Here we present a method to express all the inputs of $\mathcal{S}$ "$(d, f, \mathbb{X}), (\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}})$" in a format compatible with a GNN. This enables us to parameterize $\mathcal{S}$ with a GNN. To start, consider the original MILP $(\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}})$. As depicted in Figure 6, each variable and each constraint can be regarded as a vertex, thus conceptualizing the MILP as a bipartite graph. The feasible set $\mathbb{X}_{\mathrm{MILP}}$ is determined by $(\mathbf{A}, \circ, \boldsymbol{b}, \boldsymbol{l}, \boldsymbol{u}, \boldsymbol{z})$, where $\boldsymbol{z} \in \{0,1\}^n$ denotes the variable type (i.e., $z_j = 1$ for $j \in \mathbb{I}$ and $z_j = 0$ otherwise). Here, $(\boldsymbol{c}, \boldsymbol{l}, \boldsymbol{u}, \boldsymbol{z})$ can be viewed as features to characterize a variable, and $\circ, \boldsymbol{b}$ characterizes a constraint. The connections between variables and constraints are expressed by the matrix $\mathbf{A}$. Compared with the original feasible set $\mathbb{X}_{\mathrm{MILP}}$, the feasible set corresponding to a BnB node $\mathbb{X}$ only differs in terms of $\boldsymbol{l}$ and $\boldsymbol{u}$, as BnB only tightens the lower or upper bound of a variable (see Line 28 in Algorithm 2). We denote the new variable bounds as $\boldsymbol{l}'$ and $\boldsymbol{u}'$. Finally, the variable feature is defined with $\boldsymbol{f}_j = (c_j, l_j, u_j, z_j, l'_j, u'_j)$ and the constraint feature is defined with $\boldsymbol{g}_i = (\circ_i, b_i)$. We then apply the message passing procedure defined in (6.10) for several rounds to obtain updated feature vectors $\{\boldsymbol{f}_{j,\mathrm{gnn}}\}_{j=1}^n$ and $\{\boldsymbol{g}_{i,\mathrm{gnn}}\}_{i=1}^m$ and propagate them together, applying an MLP as $\mathrm{MLP}(\sum_j \boldsymbol{f}_{j,\mathrm{gnn}}, \sum_i \boldsymbol{g}_{i,\mathrm{gnn}}, d, f; \boldsymbol{\theta}_6)$, of which the output is a real number. This result can be regarded as the probability of $\boldsymbol{x}^* \in \mathbb{X}$. We use $\mathcal{G} = (\mathbf{A}, \{\boldsymbol{f}_j\}_{j=1}^n, \{\boldsymbol{g}_i\}_{i=1}^m, d, f)$ to represent the input of the GNN. The entire model can be denoted as

$$\mathcal{S}_{\mathrm{GNN}}(\mathcal{G}; \boldsymbol{\theta}) := \mathrm{MLP}\left(\sum_{j=1}^n \boldsymbol{f}_{j,\mathrm{gnn}}, \sum_{i=1}^m \boldsymbol{g}_{i,\mathrm{gnn}}, d, f; \boldsymbol{\theta}_6\right) \approx \mathcal{S}_{\mathrm{ref}}((d, f, \mathbb{X}), \boldsymbol{x}^*). \tag{6.20}$$

In the equation above, $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3, \boldsymbol{\theta}_4, \boldsymbol{\theta}_6)$, with $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3, \boldsymbol{\theta}_4$ being defined in (6.10). This is a graph-level GNN rather than a node-level GNN. (For the definition of types of GNN, please refer to Subsection 2.4.)

Data collection and model training. The processes of data collection, model training, and validation closely follow the pipeline outlined for L2B in Subsection 6.2.3. For each MILP instance in the training set, we run BnB, and when BnB calls on the node selection rule, we gather data in the form of $\mathcal{G}$. This data is then appended to the dataset $\mathbb{D}_{\mathrm{Input}}$, which is initialized as an empty set:

$$\mathbb{D}_{\mathrm{Input}} \leftarrow \mathbb{D}_{\mathrm{Input}} \cup \{\mathcal{G}\}.$$

In $\mathbb{D}_{\mathrm{Input}}$, each element corresponds to a node in the BnB tree. Once BnB has been completed for an MILP instance and an optimal solution $\boldsymbol{x}^*$ is obtained, we can determine whether $\boldsymbol{x}^*$ is included in each BnB node, enabling us to assign labels to each node. If the node corresponding to $\mathcal{G}$ includes $\boldsymbol{x}^*$ (i.e., $\mathcal{S}_{\mathrm{ref}}((d, f, \mathbb{X}), \boldsymbol{x}^*) = 1$), we set $s_{\mathcal{G}} = 1$, otherwise $s_{\mathcal{G}} = 0$. The dataset for training a search strategy is then defined as

$$\mathbb{D}_{\mathrm{Search}} = \{(\mathcal{G}, s_{\mathcal{G}})\}_{\mathcal{G} \in \mathbb{D}_{\mathrm{Input}}}.$$

With the above data set, we train the GNN with

$$\min_{\boldsymbol{\theta}} \sum_{(\mathcal{G}, s_{\mathcal{G}}) \in \mathbb{D}_{\mathrm{Search}}} |\mathcal{S}_{\mathrm{GNN}}(\mathcal{G}; \boldsymbol{\theta}) - s_{\mathcal{G}}|.$$

For information regarding model validation and hyperparameter tuning, please refer to Subsection 6.2.3.

ML-based node selection. Once the training phase is complete, the trained GNN model, denoted as $\mathcal{S}_{\mathrm{GNN}}(\mathcal{G}; \boldsymbol{\theta})$, assigns a score to each BnB node. When BnB calls upon the node selection rule, we select the node with the highest score. If $\mathcal{S}_{\mathrm{GNN}}$ is a good approximation of $\mathcal{S}_{\mathrm{ref}}$, as previously discussed, this node selection strategy is advantageous for reducing the primal bound and can be employed during the early stages of BnB. After several iterations of BnB, it may be beneficial to transition to the best-first search strategy, as previously mentioned.

Bibliographical notes. The presented L2S framework serves as a basic configuration to demonstrate its potential applicability. In practice, to achieve more competitive outcomes, neural networks are typically employed to compare two given BnB nodes rather than assigning a score to a single node [75,100]. For other efforts on ML-based node selection, please refer to the designated reference [103, 178].

6.3 Cutting plane methods

Besides the branch-and-bound method, the *cutting plane method* is another important algorithm in solving MILP. Rather than employing the BnB approach that generates and solves multiple sub-MILPs, the cutting plane method progressively incorporates linear constraints into the root LP relaxation. We first introduce some basic concepts in the cutting plane method and then present how to utilize machine learning in it.

6.3.1 *Introduction to cutting plane methods*

Cuts and separators. Here we adopt the notation outlined in Algorithm 2. For a given BnB node $(d, f, \mathbb{X})$, we solve the LP relaxation $(\boldsymbol{c}, \mathrm{Relax}(\mathbb{X}))$. For simplicity, *we assume such an LP is feasible and bounded*, and hence we can achieve its optimal solution $\underline{\boldsymbol{x}}$. Suppose $\underline{\boldsymbol{x}}$ violates the integer constraint: $\mathrm{frac}(\underline{\boldsymbol{x}}) \neq \emptyset$ and therefore $\underline{\boldsymbol{x}} \notin \mathbb{X}$. If there exist $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \mathbb{Q}^n$ and $\beta \in \mathbb{Q}$ such that

$$\boldsymbol{\alpha}^\top \boldsymbol{x} \leqslant \beta, \quad \forall\, \boldsymbol{x} \in \mathbb{X}, \quad \boldsymbol{\alpha}^\top \underline{\boldsymbol{x}} > \beta,$$

then we term $(\boldsymbol{\alpha}, \beta)$ as a *cut* or a *separator* of this BnB node. Since $\underline{\boldsymbol{x}} \notin \mathbb{X}$, adding the additional constraint $\boldsymbol{\alpha}^\top \boldsymbol{x} \leqslant \beta$ to $\mathbb{X}$ will cut off certain points (including at least $\underline{\boldsymbol{x}}$) from $\mathrm{Relax}(\mathbb{X})$, while preserving the MILP feasible set $\mathbb{X}$:

$$\mathbb{X} = \mathbb{X} \cap \{\boldsymbol{x} : \boldsymbol{\alpha}^\top \boldsymbol{x} \leqslant \beta\}, \quad \mathrm{Relax}(\mathbb{X}) \subsetneq \mathrm{Relax}(\mathbb{X}) \cap \{\boldsymbol{x} : \boldsymbol{\alpha}^\top \boldsymbol{x} \leqslant \beta\}.$$

Let us denote $\underline{\boldsymbol{x}}'$ as the optimal solution to the LP relaxation after adding the cut. Surely it holds that $\boldsymbol{c}^\top \underline{\boldsymbol{x}}' \geqslant \boldsymbol{c}^\top \underline{\boldsymbol{x}}$. Therefore, such a cut can tighten the LP relaxation and hopefully improve the lower bound. Intuitively, a cut separates $\underline{\boldsymbol{x}}$ from $\mathbb{X}$ and Figure 8 visually demonstrates this concept.

Generating cuts. Indeed, the literature reveals an extensive array of potential cut candidates applicable to an MILP. In this context, we introduce the Chvtal-Gomory cut (CG cut) as a representative example. Consider a scenario in which we deal with a pure integer program where all variables are required to be non-negative integers. In the context of equation (6.1), we assume $\mathbb{I} = \{1, \ldots, n\}$ and $\boldsymbol{l} = \boldsymbol{0}, \boldsymbol{u} \in \mathbb{Z}_+^n$. Let us assume that the i-th constraint of the MILP takes the form $\boldsymbol{a}_i^\top \boldsymbol{x} \leqslant b_i$, with $\boldsymbol{a}_i$ representing the i-th row of the matrix $\mathbf{A}$ and b_i being the i-th element in $\boldsymbol{b}$. From this constraint, we can derive a cut through the following rounding technique:

$$\lfloor \boldsymbol{a}_i \rfloor^\top \boldsymbol{x} \leqslant \lfloor b_i \rfloor. \tag{6.21}$$

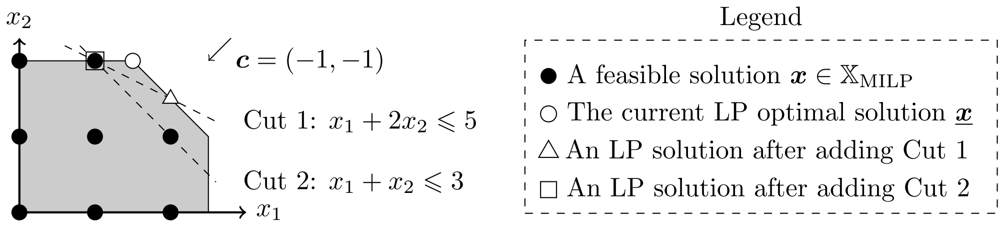

x_2
x_1
$\boldsymbol{c} = (-1, -1)$
Cut 1: $x_1 + 2x_2 \leqslant 5$
Cut 2: $x_1 + x_2 \leqslant 3$
Legend
● A feasible solution $\boldsymbol{x} \in \mathbb{X}_{\mathrm{MILP}}$
○ The current LP optimal solution $\underline{\boldsymbol{x}}$
△ An LP solution after adding Cut 1
□ An LP solution after adding Cut 2

Figure 8 An example of cuts. The MILP feasible set is defined with $\mathbb{X}_{\mathrm{MILP}} = \{(x_1, x_2) : x_1, x_2 \in \mathbb{Z}, 0 \leqslant x_1 \leqslant 2.5, 0 \leqslant x_2 \leqslant 2, x_1 + x_2 \leqslant 3.5\}$. Suppose we run the simplex algorithm and obtain an LP optimal solution $\underline{\boldsymbol{x}} = (1.5, 2)$. To tighten the LP relaxation, one can introduce an extra constraint. In this example, both Cut 1 and Cut 2 can fulfill this purpose, as they both separate $\underline{\boldsymbol{x}}$ from all feasible solutions $\boldsymbol{x} \in \mathbb{X}_{\mathrm{MILP}}$. In other words, both cuts will successfully eliminate $\underline{\boldsymbol{x}}$ while preserving the feasible set $\mathbb{X}_{\mathrm{MILP}}$. However, the efficiency of these two cuts differs. Using Cut 1, the LP solution becomes $(2, 1.5)$, leading to a lower bound of -3.5. With Cut 2, an LP solution is $(1, 2)$ and the lower bound is -3. Clearly, Cut 2 is better as it provides a greater enhancement to the lower bound

Here, $\lfloor \boldsymbol{a}_i \rfloor$ signifies the element-wise floor operation. Clearly, any feasible integer solution $\boldsymbol{x} \in \mathbb{X}$ must satisfy (6.21): $\lfloor \boldsymbol{a}_i \rfloor^\top \boldsymbol{x} \leqslant \boldsymbol{a}_i^\top \boldsymbol{x} \leqslant b_i$ due to the non-negativity of $\boldsymbol{x}$, and $\lfloor \boldsymbol{a}_i \rfloor^\top \boldsymbol{x}$ must be an integer, which implies (6.21). Thus, adding the constraint (6.21) to $\mathbb{X}$ will not cut off any desirable solutions. Given $\underline{\boldsymbol{x}}$, it is straightforward to examine if $\lfloor \boldsymbol{a}_i \rfloor^\top \underline{\boldsymbol{x}} > \lfloor b_i \rfloor$ is true. If this condition is met, (6.21) will serve as a valid cut that separates $\underline{\boldsymbol{x}}$ from $\mathbb{X}$. Through this methodology, one may potentially create as many as m cuts (equivalent to the number of constraints), a significant amount. Furthermore, there exists a wide variety of practical techniques for generating cuts, such as strong CG-cut, MIR-cut, lift-and-project cut, and so on [169]. Each of these can give rise to a substantial quantity of cuts applicable to an MILP. While this tutorial refrains from diving into the vast scope of papers on this subject, it may offer a typical number of cuts as a guide for newcomers. Given an MILP of size $\mathbf{A} \in \mathbb{Q}^{500\times1000}$, the number of all the potential cut candidates might exceed 10^4.

Selecting cuts. Given the large number of potential cuts, it is impractical to incorporate all potential cutting plane candidates. The reason is two-fold: (1) the resulting LP relaxation might be significantly enlarged and subsequently challenging to solve; (2) many cut candidates cannot significantly improve the lower bound. The first reason is relatively straightforward, while the second one can be illustrated by Figure 8, where Cut 1 is obviously undesirable as it cannot improve the lower bound. Therefore, it becomes vital to select part of the cut candidates and then resolve the subsequent LP.

Cutting plane methods. The procedure detailed above can be executed iteratively. Specifically, we first generate cuts, then select from among them, add the chosen cuts to $\mathbb{X}$, and resolve the LP relaxation. This sequence of actions is repeated continuously until either a satisfactory solution is discovered or the designated budget is reached. This entire iterative approach is known as the *cutting plane method* or *separation method*, which is summarized in Algorithm 3.

Algorithm 3 Cutting plane algorithm

```
Input: A feasible and bounded (sub-)MILP $(\boldsymbol{c}, \mathbb{X})$; Cut selection rule;
Output: An optimal solution to $(\boldsymbol{c}, \mathbb{X})$ or a tightened feasible set $\mathbb{X}'$;
 1: Initialize $\mathbb{X}' \leftarrow \mathbb{X}$;
 2: while The time budget is not reached do
 3:     Solve $(\boldsymbol{c}, \mathrm{Relax}(\mathbb{X}'))$ to obtain its optimal solution $\underline{\boldsymbol{x}}'$;
 4:     if $\underline{\boldsymbol{x}}' \in \mathbb{X}$ (i.e., all the integer constraints are satisfied) then
 5:         return $\underline{\boldsymbol{x}}'$ as the optimal solution;
 6:     end if
 7:     Generate a bag of cuts $\{(\boldsymbol{\alpha}_i, \beta_i)\}_{i=0}^{m'}$ such that $\boldsymbol{\alpha}_i^\top \boldsymbol{x} \leqslant \beta_i$ for all $\boldsymbol{x} \in \mathbb{X}$, and $\boldsymbol{\alpha}_i^\top \underline{\boldsymbol{x}} > \beta_i$;
 8:     Select a subset of cuts from $\{(\boldsymbol{\alpha}_i, \beta_i)\}_{i=0}^{m'}$ according to the cut selection rule; {The selected cuts form a new set
        $\{(\boldsymbol{\alpha}_i, \beta_i)\}_{i\in\mathbb{I}'}$.}
 9:     Update $\mathbb{X}'$: $\mathbb{X}' \leftarrow \mathbb{X}' \cap (\cap_{i\in\mathbb{I}'} \{\boldsymbol{x} : \boldsymbol{\alpha}_i^\top \boldsymbol{x} \leqslant \beta_i\})$;
10: end while
11: return $\mathbb{X}'$
```

If this algorithm is directly applied to the original MILP $(\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}})$, and an ample computational budget is allocated, integer programs with rational data can theoretically be solved within a finite number of steps, provided that the cuts are appropriately selected (as demonstrated by Gomory [65,66]). However, a pure cutting plane method usually requires an extremely large number of iterations to produce the optimal solution in practice, making it far from efficient compared with the BnB method. Such a situation remained unchanged until Balas et al. [11] showed that the cutting plane method is efficient when combined with the BnB. State-of-the-art MIP solvers typically combine BnB and cutting planes with one of the following approaches [3]:

- Cut and branch. We first apply Algorithm 3 on the original MILP $(\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}})$ with a controlled budget and obtain a tightened feasible set $\mathbb{X}'$. Afterwards, the problem $(\boldsymbol{c}, \mathbb{X}')$ is solved with BnB (Algorithm 2). With the global lower bound already improved through the cutting planes, this cut-and-branch procedure often concludes earlier than the standard BnB method, without the necessity of traversing the entire BnB tree.

• Branch and cut. The MILP is solved with BnB, while all BnB nodes might be tightened by the cutting plane method. The cuts included at the root node (the original MILP) are considered global cuts, whereas those integrated into a sub-MILP are viewed as local cuts. Local cuts, valid solely for the sub-MILP and its subsequent child nodes, must be discarded when the exploration extends beyond that specific subtree.

In this paper, we only consider cut-and-branch for the sake of simplicity. However, this narrowed scope does not significantly deviate from the practices commonly employed in real-world applications. Modern MILP solvers utilizing the branch-and-cut method typically allocate more computational resources for cuts in the original MILP (or the root node) compared with those in a sub-MILP, due to the global applicability of the cuts added at the root node.

6.3.2 *Learning to select cuts*

Similar to the rules for selecting branching variables and BnB nodes, the rule for cut selection can be stated as: assign a score to every potential cut and choose the topmost $\gamma\%$ of them. Generally, the score assigned to a cut is influenced not only by the cut $(\boldsymbol{\alpha}_i, \beta_i)$ itself, but also by the MILP data $(\boldsymbol{c}, \mathbb{X})$. The scoring function can be expressed as

$$s_{i,\text{cut}} = \mathcal{C}((\boldsymbol{c}, \mathbb{X}), (\boldsymbol{\alpha}_i, \beta_i)). \tag{6.22}$$

Based on this equation, we want to discover a mapping $\mathcal{C}$ that assigns high ratings to effective cuts and low ratings to ineffective ones. It is worth noting that the cut selection proportion $\gamma\%$ is another critical aspect that can be optimized using machine learning. However, in this context, we treat it as a manually tunable hyperparameter for simplicity's sake. Like the variable selection rule stated in Subsection 6.2.3, the cut selection strategy can also be parameterized by neural networks and trained via both supervised and reinforcement learning approaches.

Supervised learning. We present an exemplary application of supervised learning for the selection of cuts, drawing inspiration from the principles of strong branching. In SB, rather than immediately proceeding with a branch, the solver simulates branching on various candidates to assess their potential effect on the objective function. Similarly, in cut selection, the most impactful cut, in terms of objective improvement, is a good choice. By incorporating a cut of the form $\boldsymbol{\alpha}_i^\top \boldsymbol{x} \leqslant \beta_i$ into the set $\mathbb{X}$ and evaluating the modified LP relaxation, the resulting shift in the objective value provides a valuable measure of the cut's effectiveness. Specifically, this idea can be formulated as

$$s_{i,\text{cut}} := \frac{\Delta_i}{\Delta_{i_*}}, \quad \text{where } \Delta_i = \min_{\boldsymbol{x} \in \text{Relax}(\mathbb{X}) \cap \{\boldsymbol{x} : \boldsymbol{\alpha}_i^\top \boldsymbol{x} \leqslant \beta_i\}} \boldsymbol{c}^\top \boldsymbol{x} - \boldsymbol{c}^\top \underline{\boldsymbol{x}} \ \text{ and } i_* = \arg\max \Delta_i. \tag{6.23}$$

Here, $s_{i,\text{cut}}$ represents a score assigned to the i-th cut $\boldsymbol{\alpha}_i^\top \boldsymbol{x} \leqslant \beta_i$, quantifying its effectiveness. Such a "look-ahead" cut-scoring function maximizes the dual bound improvement greedily, but it is computationally demanding as it requires solving multiple LPs — one for each proposed cut. Considering that the number of potential cuts even might exceed the count of linear constraints in the original MILP, the computational load due to LP evaluations is notably expensive. Still, there is potential to *leverage neural networks as fast approximators of the scoring function* delineated in (6.23). This mirrors the strategy where neural networks have been utilized to imitate SB, as expounded in Subsection 6.2.3.

Parameterization. Here we present a GNN-based method to parameterize the scoring mapping, $\mathcal{C}$, that maps $(\boldsymbol{c}, \mathbb{X}), (\boldsymbol{\alpha}_i, \beta_i)$ to $s_{i,\text{cut}}$. Drawing inspiration from Figure 6 in Subsection 6.2.3, we consider each variable of the MILP as a node. Similarly, every constraint, including the additional constraints $\{\boldsymbol{\alpha}_i^\top \boldsymbol{x} \leqslant \beta_i\}_{i=1}^{m'}$, is also treated as a node. For variable nodes, we define their feature vector as $\boldsymbol{f}_j = (c_j, l_j, u_j, z_j, \underline{x}_j)$. For constraint nodes, the associated feature vector is captured by $\boldsymbol{g}_i = (\circ_i, b_i)$ for the range $1 \leqslant i \leqslant m$ and $\boldsymbol{g}_i = (\leqslant, \beta_{i-m})$ for the range $m+1 \leqslant i \leqslant m+m'$. Given the additional constraints, we must expand the matrix $\mathbf{A} \in \mathbb{Q}^{m \times n}$ to $\mathbf{A}_{\text{cut}} \in \mathbb{Q}^{(m+m') \times n}$. The initial m rows of $\mathbf{A}_{\text{cut}}$ are identical to $\mathbf{A}$. The succeeding m' rows are defined by $\mathbf{A}_{i,:,\text{cut}} := \boldsymbol{\alpha}_{i-m}$ for all $m+1 \leqslant i \leqslant m+m'$. With the concepts defined above, we can apply the message passing rule (6.10) for several rounds and obtain the outcomes $\{\boldsymbol{f}_{j,\mathrm{gnn}}\}_{j=1}^{n}$ and $\{\boldsymbol{g}_{i,\mathrm{gnn}}\}_{i=1}^{m+m'}$. Finally, a dense neural network is employed individually on $\{\boldsymbol{g}_{i,\mathrm{gnn}}\}_{i=m+1}^{m+m'}$ to fit the cut scores in (6.23): specifically, $\mathrm{MLP}(\boldsymbol{g}_{i,\mathrm{gnn}};\boldsymbol{\theta}_8)\approx s_{i-m,\mathrm{cut}}$. Summarizing, the entire GNN model can be represented as

$$\mathcal{C}_{\mathrm{GNN}}(i,\mathcal{G}_{\mathrm{cut}};\boldsymbol{\theta}):=\mathrm{MLP}(\boldsymbol{g}_{i+m,\mathrm{gnn}};\boldsymbol{\theta}_8)\approx s_{i,\mathrm{cut}},\quad 1\leqslant i\leqslant m'.$$

Here, $\mathcal{G}_{\mathrm{cut}}=(\mathbf{A}_{\mathrm{cut}},\{\boldsymbol{f}_j\}_{j=1}^{n},\{\boldsymbol{g}_i\}_{i=1}^{m+m'})$ serves as the GNN input while $\boldsymbol{\theta}=(\boldsymbol{\theta}_1,\boldsymbol{\theta}_2,\boldsymbol{\theta}_3,\boldsymbol{\theta}_4,\boldsymbol{\theta}_8)$ represents all the parameters in the GNN, with $\boldsymbol{\theta}_1,\boldsymbol{\theta}_2,\boldsymbol{\theta}_3,\boldsymbol{\theta}_4$ being defined in (6.10).

Data collection and training. For each MILP instance in the training set, we collect $\mathcal{G}_{\mathrm{cut}}$ and calculate $\{s_{i,\mathrm{cut}}\}_{i=1}^{m'}$ based on (6.23), each tuple $(\mathcal{G}_{\mathrm{cut}},\{s_{i,\mathrm{cut}}\}_{i=1}^{m'})$ is treated as an individual sample. Following this, we identify the top $\gamma\%$ of cuts based on their scores, integrate them into the MILP, and thereby generate a tightened MILP instance. For this new instance, another sample is collected. By iterating this procedure across multiple cycles, we accumulate a series of training samples. After processing all the MILP instances in the training dataset and consolidating the gathered samples, our compiled training data is denoted as: $\mathbb{D}_{\mathrm{cut}}$. Utilizing this dataset, our aim is to optimize the GNN in order to best predict the cut scores:

$$\min_{\boldsymbol{\theta}}\sum_{(\mathcal{G}_{\mathrm{cut}},\{s_{i,\mathrm{cut}}\}_{i=1}^{m'})\in\mathbb{D}_{\mathrm{cut}}}\sum_{i=1}^{m'}\ell(\mathcal{C}(i,\mathcal{G}_{\mathrm{cut}};\boldsymbol{\theta}),s_{i,\mathrm{cut}}).$$

Within this context, the loss function, denoted as ℓ, might be the mean squared error or perhaps the cross-entropy as will be seen in (6.29). Other procedures, like model validation and the tuning of hyperparameters, align with conventional machine learning methodologies. Please refer to Subsection 6.2.3 for details. Upon successful training, the trained GNN serves as the surrogate for the mapping $\mathcal{C}$ as presented in (6.22), equipping us with a GNN-driven strategy for cut selection.

Reinforcement learning. Here we present an alternate method utilizing reinforcement learning (RL) for cut selection. We start by outlining the motivation behind the application of RL. While the cut selector, derived from (6.23), might appear optimal for a single round of cutting planes, it is actually a local optimization strategy. Such a greedy approach might not necessarily produce the globally optimal cut selection strategy as defined by

$$\max_{\mathcal{C}}\sum_{(\boldsymbol{c},\mathbb{X})\in\mathbb{M}}I(\mathcal{C},(\boldsymbol{c},\mathbb{X})):=\int_0^{T_{\mathrm{end}}}\underline{f}(t)\mathrm{d}t. \tag{6.24}$$

Here $\mathbb{M}$ aggregates all conceivable (sub-)MILPs encountered during cutting plane methods, with $\underline{f}(t)$ representing the lower bound at a specific time t, and T_{end} marking the termination of Algorithm 3. Termination can result from achieving optimality or due to budget control. To obtain an effective cut selection rule in the sense of (6.24), one can adopt reinforcement learning. By presuming the selection of the single best cut in every cycle, Algorithm 3 can naturally be formulated as an MDP:

- State: The previously defined $\mathcal{G}_{\mathrm{cut}}$ describes the current state.
- Action: The cut to be selected. We use $i\in\{1,\ldots,m'\}$ to denote its index.
- Transition: Upon selecting a cut, it is added into the MILP, resulting in the tightened feasible set $\mathbb{X}'$ as Line 9 in Algorithm 3.
- Reward: This is dictated by the dual bound, with higher values being more desirable.

With these definitions, methods such as Q-learning and policy gradient methods, as highlighted in Subsection 6.2.3, can be conveniently adopted for enhancing cut selection. For example, when using Q-learning, one can simply replace j with i and the policy $\mathrm{GNN}(j,\mathcal{G};\boldsymbol{\theta})$ with $\mathcal{C}_{\mathrm{GNN}}(i,\mathcal{G}_{\mathrm{cut}};\boldsymbol{\theta})$ in (6.15). To adopt the policy gradient, one may treat $\mathcal{C}_{\mathrm{GNN}}$ as a stochastic policy rule, where the output corresponds to the likelihood of selecting a particular cut candidate. However, it is worth noting that *the action space for cut selection is much larger than that of variable selection* so that its training is also much more challenging. To provide context: for an MILP instance described by $\mathbf{A} \in \mathbb{Q}^{500\times 1000}$, the action space for variable selection consists of all the fractional variables, typically of sizes $20 \sim 50$. While in the cut selection, the action space involves all the cut candidates, typically of size that may exceed 10^4 as we mentioned before. Expanding the number of cuts in a given round further inflates this action space. For example, the size of the action space will be $\binom{10^4}{10}$ if we select 10 cuts each time.

Bibliographical notes. The incorporation of machine learning into the cutting plane method emerged relatively late compared with the evolution of learning to branch. Several factors contribute to this delay. Firstly, unlike the straightforward criterion of strong branching for variable selection, cut selection lacks a universally accepted standard. Secondly, as previously discussed, the action space for cut selection is considerably larger. Nevertheless, with the development of deep reinforcement learning, there has been a growing interest in learning for cut selection. The discussions on supervised learning in this subsection are mainly inspired by [88, 128], notable for their analogy to strong-branching imitation. In parallel, the reinforcement learning aspects are informed by studies such as [154, 163]. Other noteworthy contributions to this field include [23, 93, 157]. For a comprehensive review on this topic, readers may refer to [51].

6.4 Primal heuristics

In addition to branch-and-bound and cutting plane methods, *primal heuristics* are also an important technique in solving MILP. Although heuristics lack the optimality guarantees that BnB offers, they are typically more computationally efficient and able to promptly reduce the primal bound. Modern solvers usually first resort to heuristic methods before implementing BnB, treating BnB as a final solution. This subsection will discuss primal heuristics and how machine learning can enhance them.

6.4.1 *Introduction to heuristics*

Any procedure aimed at swiftly creating a feasible solution $\overline{\boldsymbol{x}} \in \mathbb{X}_{\mathrm{MILP}}$ without promising optimality is termed a *primal heuristic*. The motivation behind primal heuristics is to rapidly reduce the primal bound $\overline{f} = \boldsymbol{c}^\top \overline{\boldsymbol{x}}$. If a desirable primal bound can be obtained in the early stage of BnB, it is plausible to prune a substantial portion of nodes in the BnB tree. (Refer to Lines 6 and 14 in Algorithm 2 for pruning.) The following are some typical primal heuristics:

Rounding methods. The aim of rounding methods is to construct a solution that satisfies the integer constraint from a fractional solution, such as an optimal solution to the root LP relaxation $\underline{\boldsymbol{x}}$. If $\mathrm{frac}(\underline{\boldsymbol{x}}) \neq \emptyset$, some fractional values may be rounded to an integer value. A simple rounding method is as follows:

$$x'_j = \begin{cases} \lfloor \underline{x}_j \rfloor, & \text{if } c_j > 0, \\ \lceil \underline{x}_j \rceil, & \text{if } c_j < 0, \\ [\underline{x}_j], & \text{if } c_j = 0, \end{cases} \tag{6.25}$$

where $[x]$ denotes the nearest integer to x. This rounding method may generate a solution $\boldsymbol{x}'$ that violates the linear constraint $\mathbf{A}\boldsymbol{x} \circ \boldsymbol{b}$. However, if the generated solution $\boldsymbol{x}'$ satisfies all the constraints, we consider a feasible solution has been discovered and mark $\boldsymbol{x}'$ as $\overline{\boldsymbol{x}}$ and update $\overline{f} = \boldsymbol{c}^\top \overline{\boldsymbol{x}}$.

Diving methods. Essentially, diving methods are *incomplete BnB methods*. In BnB branching, only one child node may be generated instead of two, as prescribed in Algorithm 2. A common diving method is the *fractionality diving*, running BnB with a branching strategy that selects the fractional element with the smallest fractionality

$$j_* = \mathop{\arg\min}_{j \in \mathrm{frac}(\underline{\boldsymbol{x}})} \max\{\underline{x}_j - \lfloor \underline{x}_j \rfloor, \lceil \underline{x}_j \rceil - \underline{x}_j\}$$

and generates a single child node defined as the following rule until either infeasibility is reached or a feasible solution $\overline{\boldsymbol{x}}$ is obtained:

$$\mathbb{X}_{\mathrm{child}} = \mathbb{X} \cap \{\boldsymbol{x} : x_j = [\underline{x}_j]\}. \tag{6.26}$$

This differs from the typical most fractionality rule that we might adopt in a complete BnB; a diving heuristic does not aim to find the optimal solution and improve the dual bound, but instead seeks to quickly identify a feasible solution by selecting the element with the smallest fractionality. Given that such a method resembles diving into the BnB tree, it is aptly termed as a diving method.

Neighborhood search. Given an estimated solution $\hat{\boldsymbol{x}}$, one may explore integer solutions within the neighborhood of $\hat{\boldsymbol{x}}$. For example, a subset of the values in $\boldsymbol{x}$ can be fixed to match $\hat{\boldsymbol{x}}$, forming a "sub-MIP" with the remaining unfixed variables. The sub-MIP typically has fewer variables than the original MILP and is hence likely to be solved in less time. Formally, an index set $\mathbb{J}$ is determined to create the sub-MIP as follows:

$$\min_{\boldsymbol{x}} \boldsymbol{c}^\top \boldsymbol{x} \quad \text{s.t. } \boldsymbol{x} \in \mathbb{X}_{\text{MILP}}, \quad x_j = \hat{x}_j \text{ for } j \in \mathbb{J}. \tag{6.27}$$

If $\hat{\boldsymbol{x}}$ is not feasible, a feasible solution in its neighborhood is sought; if it is already feasible, the goal is to find a feasible solution better than $\hat{\boldsymbol{x}}$. If this goal is achieved, the optimal solution to (6.27) is designated as $\overline{\boldsymbol{x}}$, and $\overline{f}$ is accordingly updated.

Note that many heuristics do not guarantee the generation of a feasible solution within a given time limit. One strategy may be to allocate a time budget for each heuristic and try different heuristics either prior to or during the BnB process. If a feasible solution with a superior primal bound is discovered, BnB will update the incumbent $\overline{\boldsymbol{x}}$ and the primal bound $\overline{f}$. Refer to [22] for more practical heuristics.

6.4.2 *Learning to predict a solution*

Given that the primary aim of primal heuristics is to discover a feasible solution with as low a primal bound as possible, *it is always beneficial if we can train a neural network to predict the MILP solution.* Even though the predicted solution might not be fully accurate, it can be immediately employed in rounding methods and neighborhood searches. Moreover, it can support diving methods by suggesting the branching variable. Applications of a machine learning-based solution predictor in primal heuristics will be discussed in the following subsection. This subsection will focus on how to train a solution predictor. The training process adheres to the previously-mentioned pipeline: parameterization, data collection, model training, and validation.

Assumptions. To streamline notation and highlight the principal concept, we presume all the integer variables to be binary: $l_j = 0, u_j = 1$ for all $j \in \mathbb{I}$, and our solution predictor only predicts the values of these binary variables. **This assumption is applicable for Subsections 6.4.2, 6.4.3, and 6.4.4.**

Parameterization. Here our goal is to build a parameterized mapping that maps $(\boldsymbol{c}, \mathbb{X}_{\text{MILP}})$ to the predicted solution $\hat{\boldsymbol{x}}$. The three parameterization approaches, (6.8), (6.9) and (6.11) detailed in Subsection 6.2.3, are all applicable for solution prediction. We will now illustrate the structure of the GNN for solution prediction, bearing in mind that the other two apporaches can be adapted similarly. The solution predictor's input is $(\boldsymbol{c}, \mathbb{X}_{\text{MILP}})$, which can be represented as

$$\mathcal{G} = (\mathbf{A}, \{\boldsymbol{f}_j\}_{j=1}^n, \{\boldsymbol{g}_i\}_{i=1}^m)$$

to suit the GNN, where $\boldsymbol{f}_j = (c_j, l_j, u_j, z_j)$ and $\boldsymbol{g}_i = (\circ_i, b_i)$. We then apply (6.10) on $\mathcal{G}$ for multiple rounds to obtain the outcomes $\{\boldsymbol{f}_{j,\text{gnn}}\}_{j=1}^n$ and $\{\boldsymbol{g}_{i,\text{gnn}}\}_{i=1}^m$. Finally, a binary classifier is applied on $\{\boldsymbol{f}_{j,\text{gnn}}\}_{j\in\mathbb{I}}$ to predict a probability distribution of the binary variable values,

$$\text{GNN}(j, \mathcal{G}; \boldsymbol{\theta}) := \text{MLP}(\boldsymbol{f}_{j,\text{gnn}}; \boldsymbol{\theta}_7) = p_j, \tag{6.28}$$

where p_j signifies the probability that $\hat{x}_j = 1$, and $1 - p_j$ represents the probability that $\hat{x}_j = 0$. Given the above GNN, to derive a predicted solution from the GNN outcome p_j, one can either sample $\hat{x}_j$ from the Bernoulli distribution with probability p_j or decide the values directly by setting $\hat{x}_j = 0$ if $p_j \leqslant 0.5$ and $\hat{x}_j = 1$ otherwise. The parameters of the GNN, denoted by $\boldsymbol{\theta}$, include $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3, \boldsymbol{\theta}_4, \boldsymbol{\theta}_7)$, where $\boldsymbol{\theta}_1,\ \boldsymbol{\theta}_2,\ \boldsymbol{\theta}_3,\ \boldsymbol{\theta}_4$ are as defined in (6.10).

Data collection. The process of data collection in this context is straightforward: collecting pairs of MILPs and their corresponding solutions in the format $(\mathcal{G}, \boldsymbol{x}^*)$. Given a set of MILP instances for training, one should execute BnB on each instance to obtain its optimal solution. By aggregating these pairs, a dataset denoted as $\mathbb{D}_{\mathrm{Sol}}$ can be established.

Model training and validation. This phase follows the standard end-to-end training procedure and employs the binary cross-entropy loss function to determine the parameters in GNN:

$$\min_{\boldsymbol{\theta}} \sum_{(\mathcal{G},\boldsymbol{x}^*)\in\mathbb{D}_{\mathrm{Sol}}} \sum_{j\in\mathbb{I}} -x_j^* \log(p_j) - (1-x_j^*)\log(1-p_j). \tag{6.29}$$

This type of cross-entropy loss is typically employed in binary classification scenarios. To understand cross-entropy, consider that a perfect fit implies $p_j = 1$ for all $j \in \mathbb{I} : x_j^* = 1$ and $p_j = 0$ for all $j \in \mathbb{I} : x_j^* = 0$. The tuning of hyperparameters and model validation follows the same process outlined in Subsection 6.2.3.

6.4.3 *Incorporating solution predictors into heuristics*

Having trained a GNN solution predictor as per (6.28), the question arises: How can it be effectively utilized to assist with MILP solving? Using the predicted solution $\hat{\boldsymbol{x}}$ directly as $\overline{\boldsymbol{x}}$ might entail risks, given that *such a solution predicted by the GNN might be not just suboptimal, but also infeasible.* Nonetheless, incorporating this predicted solution with the three categories of heuristics previously introduced can yield significant benefits. In this subsection, we outline some methods for merging neural network predictions with primal heuristics.

Rounding. A straightforward method to incorporate the GNN in rounding is to directly use the outcome of the GNN, p_j, as the $\underline{x}_j$ in (6.25). A safer alternative would be to set an acceptance threshold for p_j. Let us say the threshold is denoted as $\xi \in (0,1)$, only variables with a probability of being 0 or 1 that exceeds ξ are accepted: $x'_j = 1$ if $p_j > \xi$ and $x'_j = 0$ if $1 - p_j > \xi$. For all other cases, we use (6.25) to create values of $\boldsymbol{x}'$. Please note that neither of these rounding methods can guarantee feasibility. However, due to their minimal computational cost, it is still worth attempting these methods. If a new feasible solution is discovered, it will be highly beneficial; if not, the rounding results can be discarded.

Diving. The output of the GNN can potentially enhance a diving method. Given the GNN outputs $\{p_j\}_{j\in\mathbb{I}}$, one can simply use p_j as the $\underline{x}_j$ in the fractionality diving (6.26). This leads to the branching rule for diving being defined as

$$j_* = \arg\min_j \max(p_j, 1-p_j).$$

The other aspects of diving remain unchanged and follow the fractionality diving approach. If the GNN's output is closer to the optimal solution of the MILP $\boldsymbol{x}^*$, such an ML-based diving method can potentially yield a better solution compared with the conventional fractionality diving.

Neighborhood search. Based on the idea of "fixing some variables and solving the sub-MIP" introduced in (6.27), an ML-enhanced neighborhood search approach can be formulated: variables exceeding a certain acceptance threshold ξ are fixed, with the remaining variables left unfixed, and the resulting sub-MILP is then solved:

$$\min_{\boldsymbol{x}\in\mathbb{X}_{\mathrm{MILP}}} \boldsymbol{c}^\top\boldsymbol{x} \quad \text{s.t. } x_j = 1 \text{ for } j \in \{j\in\mathbb{I} : p_j > \xi\}, \quad x_j = 0 \text{ for } j \in \{j \in \mathbb{I} : p_j < 1-\xi\}. \tag{6.30}$$

Let $\mathbb{J}(\xi) = \{j\in\mathbb{I} : p_j > \xi\} \cup \{j\in\mathbb{I} : p_j < 1-\xi\}$ represent the set of variables to be fixed. If $\mathbb{J}(\xi)$ is large, the sub-MIP is easier to solve due to its smaller size but the possibility of causing infeasibility increases due to the GNN-output not guaranteeing feasibility. If $\mathbb{J}(\xi)$ is small, the opposite scenario is expected. By adjusting ξ, the size of $\mathbb{J}(\xi)$ can be controlled. However, to ensure feasibility when high-confidence prediction errors are present, one might have to use a very small fixing set and cannot fully leverage the results of the GNN. Figure 9 illustrates an example of this, a typical situation in practice. To address

p_j	0.1	0.9	0.9	0.2	0.8	0.3	0.7	0.3	0.3	0.7	0.6	0.6
Confidence	0.9	0.9	0.9	0.8	0.8	0.7	0.7	0.7	0.7	0.7	0.6	0.6
$\hat{x}_j$	0	1	1	0	1	0	1	0	0	1	-	-
x_j^*	0	1	1	1	0	0	1	0	0	1	0	1
Correctness	T	T	T	F	F	T	T	T	T	T	-	-

$\xi = 0.85$ $\xi = 0.65$

Figure 9 Determining the threshold in neighborhood search. In this example, we are working with 16 binary variables. For simplicity, we have sorted these variables according to the confidence measure $\max(p_j, 1-p_j)$, accepting and fixing only those variables that exhibit sufficient confidence. With the methods in (6.30), we must set $\xi > 0.8$ to maintain correctness, consequently fixing only the first three variables. However, if we utilize (6.31) with $\tau = 2$, thereby allowing a maximum of two elements to be incorrect, we can set $\xi = 0.65$ and fix an additional five variables

this, the constraints in (6.30) may be relaxed, permitting a small error ratio within the set $\mathbb{J}(\xi)$:

$$\min_{\boldsymbol{x}\in\mathbb{X}_{\mathrm{MILP}}} \boldsymbol{c}^\top\boldsymbol{x} \quad \text{s.t.} \left(\sum_{j\in\{j\in\mathbb{I}:p_j>\xi\}}(1-x_j)+\sum_{j\in\{j\in\mathbb{I}:p_j<1-\xi\}}x_j\right)\leqslant\tau. \tag{6.31}$$

In this equation, τ represents the tolerance for errors in $\mathbb{J}(\xi)$. If $\tau = 0$, (6.31) becomes (6.30). If $\tau = 1$, the constraint in (6.31) allows for a single element in $\mathbb{J}(\xi)$ to differ from the GNN's prediction. This method, known as *local branching* [58] in the MILP community, offers considerably more flexibility than (6.30).

6.4.4 *Improvement heuristics*

The methods described earlier aim to produce a feasible solution $\overline{\boldsymbol{x}} \in \mathbb{X}_{\mathrm{MILP}}$ for an MILP from its potentially infeasible estimate $\hat{\boldsymbol{x}}$. Additionally, some heuristics can improve an existing feasible solution $\overline{\boldsymbol{x}}$, which are usually named *improvement heuristics.* For example, the technique of local branching, as outlined previously, can be employed to discover an improved feasible solution $\overline{\boldsymbol{x}}'$ from a given $\overline{\boldsymbol{x}}$ through

$$\overline{\boldsymbol{x}}' = \arg\min_{\boldsymbol{x}\in\mathbb{X}_{\mathrm{MILP}}} \boldsymbol{c}^\top\boldsymbol{x} \quad \text{s.t.} \left(\sum_{j\in\{j\in\mathbb{I}:\overline{x}_j=1\}}(1-x_j)+\sum_{j\in\{j\in\mathbb{I}:\overline{x}_j=0\}}x_j\right)\leqslant\tau. \tag{6.32}$$

This constraint ensures that $\overline{\boldsymbol{x}}$ and $\overline{\boldsymbol{x}}'$ differ in at most τ coordinates, effectively limiting the search to a "neighborhood" around $\overline{\boldsymbol{x}}$. If $\boldsymbol{c}^\top\overline{\boldsymbol{x}}' = \boldsymbol{c}^\top\overline{\boldsymbol{x}}$, it indicates that $\overline{\boldsymbol{x}}$ is the optimal solution within this neighborhood. In this case, one may enlarge τ to explore potentially better feasible solutions within a larger neighborhood. Otherwise, if $\boldsymbol{c}^\top\overline{\boldsymbol{x}}' < \boldsymbol{c}^\top\overline{\boldsymbol{x}}$, an improved solution has been found. In such cases, the process repeats with $\overline{\boldsymbol{x}}'$ as the new starting point, potentially leading to a sequence of increasingly better solutions $\overline{\boldsymbol{x}}'', \overline{\boldsymbol{x}}''', \ldots$ This iterative process is subject to computational constraints, emphasizing the heuristic nature of quickly finding feasible solutions without guarantees of optimality.

While local branching confines the search to a neighborhood, the actual scope of this neighborhood can be vast due to the unknown specific coordinates where $\overline{\boldsymbol{x}}$ and $\overline{\boldsymbol{x}}'$ differ. Totally there are $\binom{n}{\tau}$ variations, where n is the number of variables. The complexity provides a chance for machine-learning approaches to *predict these specific coordinates, thereby simplifying the MILP by reducing the variable number to* τ, with the remaining variables fixed. In other words, the goal of this approach is to *imitate local branching* through ML models.

A straightforward implementation involves the following steps:

• Collecting data. Iteratively apply local branching to obtain a series of solutions $\overline{\boldsymbol{x}}', \overline{\boldsymbol{x}}'', \overline{\boldsymbol{x}}''', \ldots$ until a computational limit is reached or no further improvements are found. Compute difference vectors to indicate changes between successive solutions:

$$\boldsymbol{y}' = |\overline{\boldsymbol{x}} - \overline{\boldsymbol{x}}'|, \quad \boldsymbol{y}'' = |\overline{\boldsymbol{x}}' - \overline{\boldsymbol{x}}''|, \quad \boldsymbol{y}''' = |\overline{\boldsymbol{x}}'' - \overline{\boldsymbol{x}}'''|, \quad \ldots$$

Here $|\cdot|$ denotes the coordinate-wise absolute value, and $|\overline{\boldsymbol{x}}-\overline{\boldsymbol{x}}'|\in\{0,1\}^n$ describes an indicator vector that measures whether $\overline{\boldsymbol{x}}$ and $\overline{\boldsymbol{x}}'$ are different. We want to train a neural network that maps $(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}},\overline{\boldsymbol{x}})$ to $\boldsymbol{y}'$, maps $(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}},\overline{\boldsymbol{x}}')$ to $\boldsymbol{y}''$, and maps $(\boldsymbol{c},\mathbb{X}_{\mathrm{MILP}},\overline{\boldsymbol{x}}'')$ to $\boldsymbol{y}'''$, etc. The data is collected in the form of

$$\begin{aligned}
\mathcal{G}&=(\mathbf{A},\{\boldsymbol{f}_j'\}_{j=1}^n,\{\boldsymbol{g}_i\}_{i=1}^m), \quad \boldsymbol{f}_j=(c_j,l_j,u_j,z_j,\overline{x}_j),\\
\mathcal{G}'&=(\mathbf{A},\{\boldsymbol{f}_j'\}_{j=1}^n,\{\boldsymbol{g}_i\}_{i=1}^m), \quad \boldsymbol{f}_j'=(c_j,l_j,u_j,z_j,\overline{x}_j'),\\
\mathcal{G}''&=(\mathbf{A},\{\boldsymbol{f}_j''\}_{j=1}^n,\{\boldsymbol{g}_i\}_{i=1}^m), \quad \boldsymbol{f}_j''=(c_j,l_j,u_j,z_j,\overline{x}_j''),\\
&\ \ \vdots
\end{aligned}$$

Repeat the process for all MILPs in the training set and collect all the data together.

• Training. The GNN structure follows (6.28) and training follows (6.29).

• Inference. After training, the model is then capable of identifying which variables should remain unfixed. Given a new MILP with a feasible solution, one can structure the MILP data into the previously-mentioned format $\mathcal{G}$ and apply the ML model, which outcomes a $\boldsymbol{y}$. For each variable j, y_j might be a continuous value, with higher values suggesting a greater likelihood that the corresponding variable x_j would change to achieve a better feasible solution $\overline{\boldsymbol{x}}'$. Consequently, by ranking the elements of $\boldsymbol{y}$ and selecting the top τ variables, denoted as $\mathbb{J}_\tau$, to remain unfixed, and fixing the values of the remaining variables to their current values in $\overline{\boldsymbol{x}}$. A sub-MILP is then formulated:

$$\min_{\boldsymbol{x}\in\mathbb{X}_{\mathrm{MILP}}} \boldsymbol{c}^\top\boldsymbol{x} \quad \text{s.t.} \ \ x_j=\overline{x}_j \ \ \text{for } j\notin\mathbb{J}_\tau.$$

Solving this reduced MILP may lead to a better feasible solution. If no better solution is obtained, one may enlarge the neighborhood size τ and retry. Repeat the above process until the time budget is reached.

This pipeline represents a basic approach, and achieving competitive results may require incorporating advanced techniques from recent research [86,148,173]. We provide some references for other learning heuristics in the next paragraph.

Bibliographical notes. The success of AlphaGo [142] showed the promising combination of machine learning techniques and exact searching methods. Machine learning excels at swiftly delivering intuitive high-level guidance. Conversely, search methods, like the Monte Carlo tree search utilized in AlphaGo, offer a refined, precise solution grounded on the foresight provided by machine learning. This framework closely mirrors the ML-enhanced heuristics described earlier: the ML model yields a rough solution, while exact mathematical procedures refine and produce an MILP feasible solution based on ML predictions. Within ML4CO (machine learning for combinatorial optimization), heuristic learning has become one of the mainstream approaches. While the ML-guided rounding explored in this subsection is straightforward, the inspirations for ML-assisted diving and neighborhood search in this paper are drawn from the works of [141] and [52,85] respectively. The following references are highlighted for other aspects on this subject: [45,56,80,84,86,87,107,111,123,127,146,148,173].

6.5 Configurations

Modern MILP solvers integrate the aforementioned BnB, cutting plane, and heuristics, along with other techniques not detailed in this note, like presolving. These solvers provide a collection of hyperparameters to allow users flexibility in controlling specific components. Here, we highlight some of these hyperparameters:

• (LP) Which algorithm should be employed for the LP relaxation: simplex or the interior-point method?

• (Cut) The budget for the cutting plane method and the order of trying different kinds of cuts.

• (Heuristics) The budget for heuristics and the order of trying different heuristic approaches.

• (BnB) The chosen branching and searching strategies. It is worth noting that numerous manually-designed strategies are not covered in this note.

Let us represent the k-th hyperparameter as h_k. By consolidating all the hyperparameters into a vector $\boldsymbol{h} = (h_1, \ldots, h_K)$, we define a *configuration*. The performance of an MILP solver greatly depends on this configuration. Typically, an MILP solver offers a default configuration $\boldsymbol{h}_{\text{default}}$. Developers derive this configuration from diverse and general-purpose datasets to ensure its reliability. For datasets catering to specific MILP instances, $\boldsymbol{h}_{\text{default}}$ might be conservative and not the optimal choice. To obtain the optimal configuration for a specific MILP set $\mathbb{M}$, one would need to solve:

$$\min_{\boldsymbol{h}} \sum_{(\boldsymbol{c}, \mathbb{X}_{\text{MILP}}) \in \mathbb{M}} \mathcal{M}((\boldsymbol{c}, \mathbb{X}_{\text{MILP}}), \boldsymbol{h}), \tag{6.33}$$

where $\mathcal{M}$ measures the solver's performance on a specific MILP instance $(\boldsymbol{c}, \mathbb{X}_{\text{MILP}})$ under a given configuration $\boldsymbol{h}$. This performance can be assessed based on time consumption, the BnB tree size, among other metrics. While in this note we only consider the sum over all MILP instances in $\mathbb{M}$, other possible methods include the maximum (for robust hyperparameter optimization), the geometric mean, etc. Compared with the learning methods discussed in Subsections 6.2, 6.3, and 6.4, the hyperparameter optimization approach treats the MILP solver as a black box, without diving into specific components like heuristics, cuts, and branching. This methodology is also referred to as *black-box optimization* or *algorithm configuration.*

Hyperparameter tuning. Solving (6.33) is usually challenging due to several factors: the non-differentiability of $\mathcal{M}$, its non-convex nature, and the fact that some hyperparameters are categorical rather than continuous. Yet, while pinpointing the global optimum of (6.33) can be challenging, finding reasonably good sub-optimal hyperparameters is tractable, with ample research available in the field. Strategies like random search [20], grid search, Bayesian optimization techniques [144], and meta-heuristic methods [64] have been explored. For a comprehensive overview, see [177]. For hyperparameter tuning in the context of MILP challenges, refer to [14, 81, 89, 90, 187]. This paper will not delve into specifics of any single hyperparameter optimization technique. Instead, we will treat problems in the form of (6.33) as solvable modules and focus on leveraging machine learning techniques to enhance configuration.

Adaptive configuration. Assigning a uniform configuration to all instances in $\mathbb{M}$ as per (6.33) lacks flexibility and typically does not yield the optimal outcome. For example, let us imagine $\mathbb{M}$ consists of two distinct MILP types. For one type, allocating a substantial computational budget to heuristics might drastically shrink the primal bound and BnB tree size. However, for the other MILP variety, heuristics might prove ineffective, warranting a minimal allocation. Using a strategy like (6.33) would struggle to produce a universally efficient configuration for these diverse MILP types. Rather than settling for a one-size-fits-all solution, adaptive configuration offers a more dynamic alternative: *configuring each instance uniquely based on its needs.* One direct method to achieve this is to tune the hyperparameters for each specific MILP $(\boldsymbol{c}, \mathbb{X}_{\text{MILP}})$ to pinpoint its optimal configuration $\boldsymbol{h}_*$. Subsequently, a neural network can be trained to map each $(\boldsymbol{c}, \mathbb{X}_{\text{MILP}})$ to its respective $\boldsymbol{h}_*$ within all instances in $\mathbb{M}$. However, this method might be too ambitious and rarely adopted in practice. A more practical strategy is partitioning the instances in $\mathbb{M}$ into several clusters, designating a unique configuration for each cluster, and then training a classification model based on these clusters. Once set up, when we meet a new MILP instance beyond $\mathbb{M}$, we can invoke the classifier and assign a configuration to this MILP based on its classification result. Here, we delve deeper into this workflow.

Clustering. The purpose of clustering is to divide the entire set $\mathbb{M}$ into several subsets, represented by $\mathbb{M} = \bigcap_{c=1}^{C} \mathbb{M}_c$, with each subset referred to as a *cluster*. The goal is to make the instances within each cluster similar enough that a universally effective configuration can be applied to them. The criteria for clustering can be quite diverse and may include:

- (Data source) In real-world scenarios, $\mathbb{M}$ might be gathered from various origins, such as different teams across various businesses. Even though all these problems can be formulated as MILPs, treating instances from distinct sources separately can be more effective.

• (Domain knowledge) Utilizing background information about the MILPs in $\mathbb{M}$ can facilitate clustering. Take the vehicle routing problem as an example; MILPs derived from weekday data and weekend data might be clustered and configured separately, based on such insights.

• (Trial and error) A more generic clustering approach is trial-and-error. Imagine a scenario where only one binary hyperparameter h is considered: the LP relaxation algorithm. The value of h represents which method to use, with 0 for the simplex method and 1 for the interior-point method. By testing both options across all instances in $\mathbb{M}$, it is possible to divide $\mathbb{M}$ into two clusters: $\mathbb{M}_0$, where the simplex method outperforms the interior-point method, and $\mathbb{M}_1$, encompassing all remaining instances. For cases involving multiple hyperparameters, this standard trial-and-error approach might be expensive, and practitioners may choose to eliminate certain undesirable configurations based on experience.

For each identified cluster $\mathbb{M}_c$, hyperparameter tuning akin to (6.33) can be employed to derive the optimal configuration $\boldsymbol{h}_c$ tied to that cluster. Every MILP instance is tagged with a label to signify its cluster affiliation. If an MILP is part of $\mathbb{M}_c$, the corresponding label is $\boldsymbol{p}_* \in \mathbb{R}^C$, with the c-th component set to 1 and the rest marked as 0.

Training. With the defined clusters, our next step is to establish a mapping from a given MILP instance, represented as $(\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}})$, to its corresponding cluster indicator, $\boldsymbol{p}_*$. To accomplish this, one can utilize a parameterized model like Approach 2 (Graph-augmented neural networks) or Approach 3 (GNNs) as detailed in Subsection 6.2.3. Here, for demonstration, we use Approach 3. By employing the method outlined in Subsection 6.2.3, data in the form of $(\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}})$ can be converted into GNN-friendly inputs denoted as $\mathcal{G} = (\mathbf{A}, \{\boldsymbol{f}_j\}_{j=1}^n, \{\boldsymbol{g}_i\}_{i=1}^m)$. Following this, we apply the message passing mechanism (6.10) to $\mathcal{G}$. The results are labeled as $\{\boldsymbol{f}_{j,\mathrm{gnn}}\}_{j=1}^n$ and $\{\boldsymbol{g}_{j,\mathrm{gnn}}\}_{j=1}^m$. By aggregating these results, we derive two feature vectors, $\sum_j \boldsymbol{f}_{j,\mathrm{gnn}}$ and $\sum_i \boldsymbol{g}_{j,\mathrm{gnn}}$, that characterize the nature of the MILP instance $(\boldsymbol{c}, \mathbb{X}_{\mathrm{MILP}})$. An MLP is then applied to these vectors. The neural network's output is formatted as a C-dimensional vector, depicting the likelihood of the MILP instance's association with a particular cluster. This complete model can be represented as

$$\mathcal{H}_{\mathrm{GNN}}(\mathcal{G};\boldsymbol{\theta}) := \mathrm{MLP}\left(\sum_{j=1}^n \boldsymbol{f}_{j,\mathrm{gnn}}, \sum_{i=1}^m \boldsymbol{g}_{j,\mathrm{gnn}}; \boldsymbol{\theta}_9\right) = (p_1, p_2, \ldots, p_C) \in \mathbb{R}^C.$$

In this context, p_c indicates the probability of the MILP aligning with cluster $\mathbb{M}_c$, and $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3, \boldsymbol{\theta}_4, \boldsymbol{\theta}_9)$ embodies all the GNN's parameters, with the definitions of $\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$, $\boldsymbol{\theta}_3$, $\boldsymbol{\theta}_4$ presented in (6.10). To effectively train this GNN, we collect data represented as pairs $(\mathcal{G}, \boldsymbol{p}_*)$, compiling them into $\mathbb{D}_{\mathrm{config}}$. Then we fit the GNN's output with $\boldsymbol{p}_*$. Formally, this is described as

$$\min_{\boldsymbol{\theta}} \sum_{(\mathcal{G},\boldsymbol{p}_*)\in\mathbb{D}_{\mathrm{config}}} \ell(\boldsymbol{p}_*, \mathcal{H}_{\mathrm{GNN}}(\mathcal{G};\boldsymbol{\theta})),$$

where $\ell(\boldsymbol{p}, \boldsymbol{q}) = \sum_{c=1}^C p_c \log(q_c)$ is taken as the cross entropy, a generalized form of the binary cross entropy defined in (6.29).

Inference. Once the GNN has been trained, it can be utilized to categorize a specific MILP, even if it is outside the training set $\mathbb{M}$. Assuming the output from the GNN is $\hat{\boldsymbol{p}}$, and the classification aligns the MILP to cluster c_* (where $c_* = \arg\max_c \hat{p}_c$), we can then apply the configuration $\boldsymbol{h}_{c_*}$ to the presented MILP based on this classification.

Bibliographical notes. The concept of instance-adaptive or instance-aware algorithm configuration is far from new, with initial implementations tracing back to works such as [95, 175], to the best of our knowledge. This direction has gained arising interest in recent years [9, 28, 83]. The methodology and pipeline presented within this paper draw inspirations from two state-of-the-art MIP configuration works [147, 159]. For a comprehensive review, please refer to [112].

6.6 Numerical results

In this subsection, we present numerical results of learning to branch, learning to select cuts, learning heuristics, and learning configurations on a common type of MILP problem, namely *Set Covering*. This classic problem in combinatorial optimization aims to find the minimum number of subsets needed to cover a given set of elements. Formally, let $U = \{1, 2, \ldots, m\}$ denote a set of m elements, and let $S = \{S_1, S_2, \ldots, S_n\}$ be a collection of n subsets of U. The objective is to select the smallest number of these subsets such that every element in U is included in at least one of the selected subsets. This can be formulated as the following MILP problem:

$$\min_{\boldsymbol{x}\in\{0,1\}^n} \sum_{j=1}^{n} x_j \quad \text{s.t.} \sum_{j:i\in S_j} x_j \geqslant 1, \quad \forall\, i \in U,$$

where x_j is a binary decision variable that equals 1 if subset S_j is selected and 0 otherwise. The objective function minimizes the total number of subsets selected, while the constraints ensure that every element in the universe U is covered by at least one of the chosen subsets.

In our experiments, we randomly generate Set Covering instances with a Python software package named Ecole [130]. The generation scheme follows the convention in [12], which has been recognized and followed by recent work such as [60]. Set Covering instances for training, validation and testing are generated independently.

Our experiment serves as a numerical showcase on a common ground of the effectiveness that can be expected out of different L2O methodologies for mixed-integer optimization. We select works to report based on consideration of both representativeness and reproducibility.

Environments. The experiments were conducted on a high-performance workstation equipped with an Intel(R) Xeon(R) Platinum 8163 CPU and 8 NVIDIA Tesla V100 GPUs, utilizing SCIP 8.0.1 [26] as the baseline MILP solver and PyTorch 1.8.1 for neural network training. Additionally, Ecole 0.8.1 was deployed to generate problem instances and to facilitate the extraction of bipartite graph representations from MILP instances. This setup also included the use of Ecole's provided demonstrations for learning branching strategies and solver configurations.

ML-based branching, cut selection, and configuration. We reproduced numerical results of three existing methods, one for each direction. We implemented [60] as a representative for ML-based branching methods, [163] for ML-based cut selection and [159] for ML-based solver configuration. [60,159] are based on graph neural networks (GNNs) that take bipartite representations of MILP instances as inputs. [163] exploits reinforcement learning to learn a cut selection policy. Note that here we reduce the configuration space of [159] to only 16 possible candidates to make the label generation time manageable.[6)]

All three methods are trained and evaluated on a common dataset of Set Covering problems released by the authors of [163].[7)] The dataset contains a training partition of 10,000 instances of Set Covering problem, a validation partition of 2,000 instances and a testing partition of 100 instances. During training, we follow the exact hyperparameter settings (e.g., learning rate) used in the original implementation.

Results of the average solving times and primal-dual integrals are reported in Table 2, which clearly show that all three ML-based methods outperform SCIP 8.0.1 with default configuration by significant margins in both metrics. Moreover:

• While the ML-based branching method [60] achieves the best performance, it requires an expensive process of collecting high-quality branching examples to facilitate supervised learning to imitate the full strong branching policy. This characteristic limits its scalability to larger problems as the complexity of example collection becomes prohibitive.

• The ML-based configuration method [159] runs a similar process to collect training labels for each instance-configuration pairs (primal-dual integrals within a fixed time limit in its case). The time cost

6) We set heuristics and separating parameters among ['DEFAULT', 'AGGRESSIVE', 'FAST', 'OFF'], resulting in a configuration space of $4 \times 4 = 16$ elements. Other solver settings are set to default.

7) Please refer to https://github.com/MIRALab-USTC/L2O-HEM-Torch.

Table 2 Results of ML-based branching, cut selection and configuration on Set Covering problems with 500 constraints and 1,000 variables. Numbers reported here are the average on 100 testing instances

Method	Time (seconds)		Primal Dual Integral	
	Value	Improvement	Value	Improvement
SCIP default [26]	11.82	-	103.13	-
ML-based branching [60]	2.03	+82.83%	24.76	+75.99%
ML-based configuration [159]	4.24	+64.13%	44.18	+57.16%
ML-based cut selection [163]	3.12	+73.60%	59.66	+42.15%

grows linearly with respect to the size of configuration space and exponentially with respect to the number of solver parameters considered. Therefore, it is essential to exclude non-promising configurations early, which is one of main contributions of [159].

• In contrast, the RL-based cut selection method [163] is free of such data collection process and thus is more scalable to larger-scale mixed-integer programming problems.

Learning heuristics. For the heuristic learning component, we incorporated methodologies from the research presented in [86], available at https://github.com/facebookresearch/CL-LNS.git. This approach is grounded in the strategies discussed in Subsection 6.4.4, but it further integrates sophisticated ML models such as graph attention networks (GAT) along with an enhanced training protocol known as contrastive learning. In this experiment, we leveraged Ecole to create two distinct sets of problem instances: one set involving smaller problems with dimensions $n = 1,000$ and $m = 500$, and another set consisting of larger problems with dimensions $n = 4,000$ and $m = 5,000$. Each set involved $1,000$ MILP instances for training and validation, alongside additional 100 instances for testing.

In contrast to the evaluation metrics applied to ML-based branching, cut selection, and configuration where the emphasis is on the duration taken to achieve a certain primal-dual gapour heuristic performance assessment employs a different criterion. We evaluate the effectiveness of heuristics based on *the quality of the best solutions they can generate within a specified time limit.* This alternative metric is commonly used in relevant literature.[8] Specifically, we employ the following two metrics to measure the heuristics's performance:

• The first metric is a curve that tracks the best achievable objective value over time, $\boldsymbol{c}^\top \overline{\boldsymbol{x}}(t)$, with $\overline{\boldsymbol{x}}(t)$ representing the optimal feasible solution at time t. This value is computed as an average across all 100 test instances.

• The second metric assesses the comparative performance at a specific time t across the test instances, detailing the number of instances where the ML-based heuristics outperform, the instances where SCIP's provided heuristics win, and the instances where both approaches result in a tie.

While the first metric provides a broad overview of average performance, the second offers a deeper analysis, examining outcomes in an instance-by-instance manner. In our analysis, we benchmark our findings against the heuristics provided by SCIP. It is important to note that SCIP employs various manually crafted heuristic strategies rather than relying on a single method. However, using SCIP's default configuration is unfair here. This is because SCIP's default mode is not solely focused on primal heuristics; it also dedicates significant computational effort to branching and cutting processes. Therefore, we use the aggressive heuristic mode in SCIP, encouraging SCIP to allocate more resources towards heuristic processes and minimizing the focus on other computations.

The results using the first evaluation metric can be seen in Figures 10 and 11, while those corresponding to the second metric are detailed in Table 3. It is worth mentioning that our implementation involved a scaled-down version of the GAT network and a reduced batch size during training, diverging from the

[8] The rationale behind this metric lies in the nature of heuristics as somewhat independent components within an MILP solver. Evaluating heuristics based on the optimal objective achievable within a fixed time frame aligns with their fundamental purpose: to quickly generate feasible solutions with potentially lower objective values. Conversely, branching strategies, cut selection, and solver configuration are deeply tied with specific solvers, making it impractical to isolate and evaluate these modules as standalone entities. Consequently, their performance is inherently linked to the overall execution time of the solver where they are integrated.

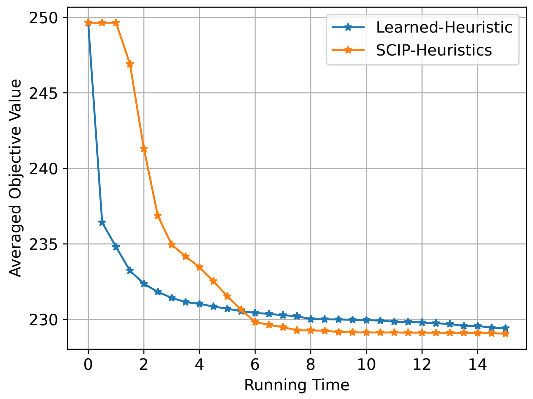


Figure 10 (Color online) Set Covering problems with size 500 × 1000

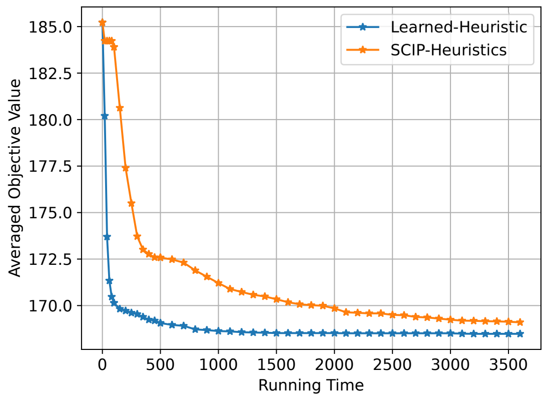


Figure 11 (Color online) Set Covering problems with size 5000 × 4000

Table 3 Comparative victory count across test instances for various heuristic approaches

Performance on small-scale set-covering problems (size: 500 × 1000)						
Running time (secs)	1	2	4	6	8	12
Learned-Heuristic wins	98	72	24	5	4	4
SCIP-Heuristics win	0	17	36	38	33	28
Tied	2	11	40	57	63	68
Performance on large-scale set-covering problems (size: 5000 × 4000)						
Running time (secs)	100	200	400	800	1600	3200
Learned-Heuristic wins	99	93	88	79	67	50
SCIP-Heuristics win	0	3	5	8	17	26
Tied	1	4	7	13	16	24

configurations recommended in [86], due to the memory limit of our hardware.[9] The numerical results clearly show that: *Initially, the ML-based heuristic demonstrates an overwhelming advantage*; *however, as the computation progresses, this advantage diminishes and might even fall behind the heuristics provided by SCIP*. This phenomenon can be attributed to the *intuitive nature* of machine learning models in contrast to the precision of mathematical algorithms employed by SCIP. Consequently, leveraging ML techniques to swiftly derive a feasible solution is promising, particularly in outperforming SCIP in terms of quickly finding superior feasible solutions within a *limited timeframe*.

6.7 Summaries

The above-introduced techniques may leave readers questioning how these methods are integrated within an MILP solver, and what connections exist between the ML-based approaches and the MILP solver itself. Figure 12 offers an illustrative answer. Here, we highlight the motivations of the aforementioned ML-based modules:

• (Learning to branch) Branching rules have a significant impact on the size of the BnB tree. While the SB rule is effective in controlling the BnB tree, it can be computationally expensive. Training a neural network to quickly approximate SB has proven effective, and recent research suggests that novel branching rules beyond SB can be discovered.

• (Learning to search) Quick discovery of BnB nodes containing the optimal solution may lead to better feasible solutions and a substantial reduction in the primal bound. Neural networks can be trained to predict whether a BnB node contains the optimal solution, thereby guiding the order in which nodes are processed.

[9] Specifically, we set the embedding size in GAT as 32 rather than 64, and set the batch size as 8 rather than 32.

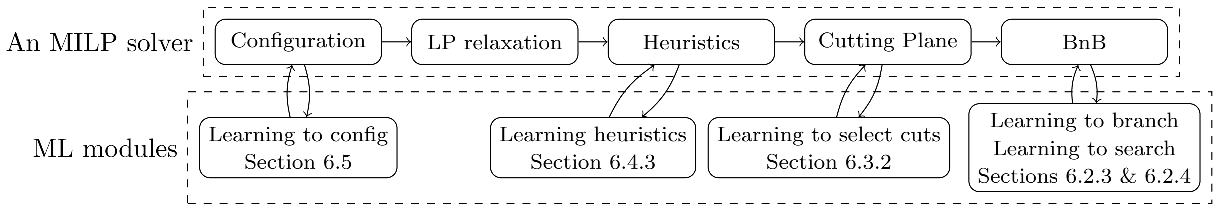


Figure 12 A recipe of machine learning in solving MILP. Here each ML module interacts with a certain component in the MILP solver: obtaining the training data from the solver and returning actions to guide the solver. Note that this is a simplified diagram. In modern MILP solvers, some heuristics might be employed before the root LP relaxation or following the cutting plane stage, and specific cuts might be considered before heuristics or during the BnB phase

• (Learning to select cuts) Properly adding cuts to the LP relaxation can greatly enhance the dual bound. However, indiscriminately adding all possible cuts can lead to computational challenges. Neural networks can be used to predict the impact of potential cuts on the dual-bound, guiding the selection of priority cut candidates.

• (Learning heuristics) Before initiating BnB, swiftly finding a feasible solution with a low objective value can reduce both the primal bound and the overall BnB tree. Neural networks can help predict the solution values for MILPs. While the predicted solutions might be infeasible, they can still serve as a foundation for constructing feasible solutions.

• (Learning to config) Since MILP instances may not all require the same optimal configuration, clustering them into distinct categories allows for specialized hyperparameter tuning. Neural networks can predict the category an MILP instance fits into, facilitating the assignment of adaptive configurations. While this overview may inspire the start of hands-on projects, questions about the selection of the right techniques or theoretical foundations might still arise. Others may wish to delve into the open challenges in this field. In the following, we will provide insights into these aspects.

Practical questions. This paragraph aims to guide the reader on how to choose the appropriate method from the many techniques detailed previously. Below are specific questions and answers to assist in this decision-making process:

• **Which solver component should be improved using machine learning?** The answer depends on the specific MILP instances. We recommend employing a "*ground truth observation*" strategy. For example, when considering heuristics, to identify if the MILPs of interest are suitable for learning heuristics, one might first solve an MILP instance to retrieve its optimal solution. If we hypothetically had this optimal solution (ground truth) prior to running the solver, it can be used as the predicted solution $\hat{\boldsymbol{x}}$ in the heuristics detailed in Subsection 6.4.3. The aim is to see if this optimal-solution-driven heuristic can reduce the solving time. Here are potential outcomes:

– If the solving time is not notably shortened using these heuristics, it indicates learning heuristics for this instance is useless because a learned model cannot surpass the performance of the optimal solution.

– On the other hand, if there is a noticeable benefit, it demonstrates the possibility of learning heuristics, assuming the neural network can predict the solution accurately.

This same approach can be applied to other modules. To check the viability of learning to config, one can tune hyperparameters for each MILP instance separately and observe the effect of this optimal configuration (the ground truth). In learning to branch, one can use SB score as the ground truth; in learning to search, one can use (6.18); in learning to select cuts, one can use (6.23). Using this ground truth observation method allows for quick decisions on which components should be boosted using ML. If the ground truth does not significantly improve the solver, pursuing that approach must be useless. Otherwise, it merits consideration.

• **What sequence should one follow when trying different ML modules?** We suggest the following sequence: start with learning to configure, then move to learning heuristics, followed by learning to select cuts, and finally, delve into learning to branch and search in BnB. The reason for prioritizing configuration is its minimal need for interaction with the solver and its relative independence. If a simpler method is already efficient, it negates the need for more complicated ones. The sequence of other modules is then determined by their dependency on solvers and frequency of interaction.

• **Among the three parameterization methods in Subsection 6.2.3, which should one opt for?** We recommend the following sequence: Approach 1, followed by Approach 2, and lastly, Approach 3. The reasoning aligns with previous points: starting with the simplest to implement and progressing to the most complex.

• **Should one utilize supervised learning or reinforcement learning?** Supervised learning is suggested to begin with, owing to its straightforward training process. Moreover, recent studies [132, 142] indicate that expert strategies in supervised learning can initialize the policies in reinforcement learning. Reinforcement learning becomes an option when: (1) Supervised learning has been executed, but there is a quest for an even superior model. (2) Determining the supervised learning ground truth becomes intractable, necessitating reinforcement learning. For example, if certain MILP instances have more than 10^5 variables, then acquiring full strong branching at a single BnB node can demand thousands of LP evaluations. Clearly, it is impractical to employ full strong branching to determine the supervised learning ground truth.

Theoretical questions. In this paragraph, we introduce theoretical aspects concerning the application of ML to MILP, while highlighting recent literature contributions. Common theoretical concerns in general machine learning contexts include the expressive capacity of specific ML models (e.g., GNNs), the methodology behind training ML models and its theoretical convergence assurance, and how trained models generalize to instances absent from the training dataset. Let us discuss these within the context of MILP tasks.

• Expressivity. The expressive power measures the ability of given ML models to fit certain mappings. For example, within the framework of learning heuristics (refer to Subsections 6.4.2 and 6.4.3), the GNN's expressiveness can be quantified using the metric:

$$\mathcal{L}(\boldsymbol{\theta},\mathbb{D}_{\mathrm{Sol}}) := \sup_{(\mathcal{G},\boldsymbol{x}^*)\in\mathbb{D}_{\mathrm{Sol}}} \sum_{j\in\mathbb{I}} |x_j^* - \lfloor p_j \rceil|, \quad \text{where } p_j = \mathrm{GNN}(j,\mathcal{G};\boldsymbol{\theta}). \tag{6.34}$$

Here, GNN refers to the GNN detailed in (6.28), and $\lfloor\cdot\rceil$ indicates nearest neighbor rounding. If $\inf_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta},\mathbb{D}_{\mathrm{Sol}}) = 0$, it implies that this class of GNNs has *strong expressive power*. For any $\varepsilon > 0$, a corresponding $\boldsymbol{\theta}$ exists such that the GNN's predicted solution is proximate to the optimal one, within a distance of ε. Otherwise, employing GNNs to approximate MILP solutions inherently introduces errors. In such cases, GNN may not be universally adept as a solution approximator for every instance in $\mathbb{D}_{\mathrm{Sol}}$. Existing research, such as [42–44], has made strides in understanding the expressiveness of GNNs concerning MILPs. A key takeaway is that while standard GNNs are universally solution approximators for all LPs, they falter for all MILPs. Instances of MILP with pronounced symmetry may exhibit inherent representational inaccuracies, prompting practitioners to break this symmetry using methods like random features.

• Training. Possessing robust expressiveness merely confirms the existence of a $\boldsymbol{\theta}$ that minimizes error. This arises a question: Are there methods to calculate such a $\boldsymbol{\theta}$ within a time limit? While algorithms, such as Adam [101], can address (6.34) and often yield a satisfactory $\boldsymbol{\theta}$ in practice, the theoretical foundation to ensure the derivation of a $\boldsymbol{\theta}$ so that $\mathcal{L}(\boldsymbol{\theta},\mathbb{D}_{\mathrm{Sol}}) < \varepsilon$ within a complexity limit remains elusive.

• Generalization. Even if one successfully identifies $\boldsymbol{\theta}_* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta},\mathbb{D}_{\mathrm{Sol}})$, there still exists a significant question: Is $\boldsymbol{\theta}_*$ applicable to an independent dataset $\mathbb{D}'_{\mathrm{Sol}}$? Unfortunately, to the best of our knowledge, such theoretical results still lack, although the generalization ability of ML models is numerically tested in almost all papers focusing on machine learning for MILP.

Although we define the above concepts in the context of learning heuristics, they can be naturally extended to other ML modules presented in this paper.

7 Conclusions

This tutorial has provided an in-depth exploration of L2O, a promising frontier that integrates the strengths of machine learning with traditional optimization techniques. By examining numerous optimization cases, we have demonstrated that L2O has the potential to reshape the way optimization problems are approached and solved dramatically. Whether it is about accelerating established algorithms, directly producing solutions, or even reconfiguring the optimization problem itself, L2O offers a range of methodologies to cater to various application scenarios. While L2O thrives in environments with ample past experience, its application must be carefully selected, keeping in mind the nature of the optimization problems and the availability of representative data. Although most approaches introduced in this paper remain at a research stage and are not yet ready to be used in commercial solvers, we firmly believe that the incorporation between machine learning and optimization has just begun and will continue to evolve.

References

1 Aberdam A, Golts A, Elad M. Ada-LISTA: Learned solvers adaptive to varying models. IEEE Trans Pattern Anal Mach Intell. 2021, 44:9222-9235

2 Ablin P, Moreau T, Massias M, et al. Learning step sizes for unfolded sparse coding. In: Proceedings of the 33rd Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2019, 32

3 Achterberg T. Constraint Integer Programming. Berlin-Heidelberg: Springer, 2007

4 Adler J, Öktem O. Learned primal-dual reconstruction. IEEE Trans Medical Imag, 2018, 37: 1322–1332

5 Agrawal A, Amos B, Barratt S, et al. Differentiable convex optimization layers. In: Proceedings of the 33rd Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2019, 32

6 Aharon M, Elad M, Bruckstein A. K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. IEEE Trans Signal Process, 2006, 54:4311–4322

7 Alvarez A M, Louveaux Q, Wehenkel L. A machine learning-based approximation of strong branching. INFORMS J Comput, 2017, 29: 185–195

8 Amos B, Kolter J Z. OptNet: Differentiable optimization as a layer in neural networks. In: Proceedings of the 34th International Conference on Machine Learning International Conference on Machine Learning. Ann Arbor: PMLR, 2017, 70

9 Anstegui C, Gabàs J, Malitsky Y, et al. MaxSAT by improved instance-specific algorithm configuration. Artificial Intelligence, 2016, 235: 26–39

10 Bai S, Kolter J Z, Koltun V. Deep equilibrium models. In: Proceedings of the 33rd Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2019, 32

11 Balas E, Ceria S, Cornuejols G, et al. Gomory cuts revisited. Oper Res Lett, 1996, 19: 1–9

12 Balas E, Ho A. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study. In: Combinatorial Optimization. Berlin-Heidelberg: Springer, 1980, 37–60

13 Balatsoukas-Stimming A, Studer C. Deep unfolding for communications systems: A survey and some new directions. In Proceedings of the 2019 IEEE International Workshop on Signal Processing Systems. San Francisco: IEEE, 2019, 266–271

14 Balcan M-F, Dick T, Sandholm T, et al. Learning to branch. In: Proceedings of the 35th International Conference on Machine Learning. Ann Arbor: PMLR, 2018, 80

15 Bansal N, Chen X, Wang Z. Can we gain more from orthogonality regularizations in training deep networks? In: Proceedings of the 32nd Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2018, 31

16 Bartlett P L, Foster D J, Telgarsky M J. Spectrally-normalized margin bounds for neural networks. In: Proceedings of the 31st Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2017, 30

17 Beck A, Teboulle M. A fast iterative shrinkage-thresholding algorithm with application to wavelet-based image deblurring. In: Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing. San Francisco: IEEE, 2009, 693–696

18 Behboodi A, Rauhut H, Schnoor E. Compressive sensing and neural networks from a statistical learning perspective. In: Compressed Sensing in Information Processing. Cham: Springer, 2022, 247–277

19 Behrens F, Sauder J, Jung P. Neurally augmented ALISTA. In: Proceedings of the 8th International Conference on Learning Representations. New Orleans: OpenReview.net, 2020

20 Bergstra J, Bengio Y. Random search for hyper-parameter optimization. J Mach Learn Res, 2012, 13: 281–305

21 Berthet Q, Blondel M, Teboul O, et al. Learning with differentiable pertubed optimizers. In: Proceedings of the 34th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2020, 33: 9508–9519
22 Berthold T. Primal heuristics for mixed integer programs. PhD Thesis. Berlin: Technischen Universität, 2006
23 Berthold T, Francobaldi M, Hendel G. Learning to use local cuts. arXiv:2206.11618, 2022
24 Bertsimas D, Kallus N. From predictive to prescriptive analytics. Manag Sci, 2020, 66: 1025–1044
25 Bertsimas D, Tsitsiklis J N. Introduction to Linear Optimization. Belmont: Athena Scientific, 1997
26 Bestuzheva K, Besançon M, Chen W K, et al. The SCIP optimization suite 8.0. arXiv:2112.08872, 2021
27 Bolte J, Pauwels E, Vaiter S. One-step differentiation of iterative algorithms. In: Proceedings of the 37th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2024, 36
28 Bonami P, Lodi A, Zarpellon G. A classifier to decide on the linearization of mixed-integer quadratic problems in CPLEX. Oper Res, 2022, 70: 3303–3320
29 Borgerding M, Schniter P, Rangan S. AMP-inspired deep networks for sparse linear inverse problems. IEEE Trans Signal Process, 2017, 65: 4293–4308
30 Boyd S, Parikh N, Chu E, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. Found Trends Mach Learn, 2011, 3: 1–22
31 Brauer C, Breustedt N, De Wolff T, et al. Learning variational models with unrolling and bilevel optimization. arXiv:2209.12651, 2022
32 Brock A, Donahue J, Simonyan K. Large scale GAN training for high fidelity natural image synthesis. In: Proceedings of the 6th International Conference on Learning Representations. New Orleans: OpenReview.net, 2018,
33 Buades A, Coll B, Morel J M. A non-local algorithm for image denoising. In: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision And Pattern Recognition. San Francisco: IEEE, 2005, 60–65
34 Cai H, Liu J, Yin W. Learned robust PCA: A scalable deep unfolding approach for high-dimensional outlier detection. In: Proceedings of the 35th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2021, 34: 16977–16989
35 Cappart Q, Moisan T, Rousseau L M, et al. Combining reinforcement learning and constraint programming for combinatorial optimization. In Proceedings of the 35th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2021, 3677–3687
36 Chan S H, Wang X, Elgendy O A. Plug-and-play ADMM for image restoration: Fixed-point convergence and applications. IEEE Trans Comput Imaging, 2016, 3: 84–98
37 Chen T, Chen X, Chen W, et al. Learning to optimize: A primer and a benchmark. J Mach Learn Res, 2022, 23: 1–59
38 Chen X, Dai H, Li Y, et al. Learning to stop while learning to predict. In: Proceedings of International Conference on Machine Learning. Ann Arbor: PMLR, 2020, 1520–1530
39 Chen X, Liu J, Wang Z, et al. Theoretical linear convergence of unfolded ISTA and its practical weights and thresholds. In: Proceedings of the 32nd Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2018, 31
40 Chen X, Liu J, Wang Z, et al. Hyperparameter tuning is all you need for LISTA. In: Proceedings of the 35th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2021, 34: 11678–11689
41 Chen X, Zhang Y, Reisinger C, et al. Understanding deep architecture with reasoning layer. In: Proceedings of the 34th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2020, 33: 1240–1252
42 Chen Z, Liu J, Chen X, et al. Rethinking the capacity of graph neural networks for branching strategy. arXiv:2402.07099, 2024
43 Chen Z, Liu J, Wang X, et al. On representing linear programs by graph neural networks. In: Proceedings of the 11th International Conference on Learning Representations. New Orleans: OpenReview.net, 2023
44 Chen Z, Liu J, Wang X, et al. On representing mixed-integer linear programs by graph neural networks. In: Proceedings of the 11th International Conference on Learning Representations. New Orleans: OpenReview.net, 2023
45 Chmiela A, Khalil E, Gleixner A, et al. Learning to schedule heuristics in branch and bound. In: Proceedings of the 35th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2021, 34: 24235–24246
46 Cohen R, Elad M, Milanfar P. Regularization by Denoising via Fixed-Point Projection (RED-PRO). SIAM J Imaging Sci, 2021, 14: 1374–1406
47 Condat L. A primalCdual splitting method for convex optimization involving Lipschitzian, proximable and linear composite terms. J Optim Theo Appl, 2013, 158: 460–479
48 Corbineau M C, Bertocchi C, Chouzenoux E, et al. Learned image deblurring by unfolding a proximal interior point algorithm. In: Proceedings of the 2019 IEEE International Conference on Image Processing. San Francisco: IEEE, 2019, 4664–4668
49 Dabov K, Foi A, Katkovnik V, et al. Image denoising by sparse 3-D transform-domain collaborative filtering. IEEE Trans Image Process, 2007, 16: 2080–2095

50 Davis D, Yin W. A three-operator splitting scheme and its optimization applications. Set-Valued Var Anal, 2017, 25: 829–858

51 Deza A, Khalil E B. Machine learning for cutting planes in integer programming: A survey. arXiv:2302.09166, 2023

52 Ding J Y, Zhang C, Shen L, et al. Accelerating primal solution findings for mixed integer programs based on solution prediction. In Proceedings of the 34th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2020, 1452–1459

53 Donti P, Amos B, Kolter J Z. Task-based end-to-end model learning in stochastic optimization. In: Proceedings of the 31st Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2017, 30

54 Elmachtoub A N, Grigas P. Smart "predict, then optimize". Manag Sci, 2022, 68: 9–26

55 Etheve M, Als Z, Bissuel C, et al. Reinforcement learning for variable selection in a branch and bound algorithm. In: Proceedings of the 17th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research. Cham: Springer, 2020, 176–185

56 Falkner J K, Thyssens D, Schmidt-Thieme L. Large neighborhood search based on neural construction heuristics. arXiv:2205.00772, 2022

57 Fan J, Li R. Variable selection via nonconcave penalized likelihood and its oracle properties. J Amer Statist Assoc, 2001, 96: 1348–1360

58 Fischetti M, Lodi A. Local branching. Math Program, 2003, 98: 23–47

59 Fung S W, Heaton H, Li Q, et al. JFB: Jacobian-free backpropagation for implicit networks. In Proceedings of the 36th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2022, 6648–6656

60 Gasse M, Chtelat D, Ferroni N, et al. Exact combinatorial optimization with graph convolutional neural networks. In: Proceedings of the 33rd Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2019, 32

61 Gehring J, Auli M, Grangier D, et al. A convolutional encoder model for neural machine translation. arXiv:1611.02344, 2016

62 Geng Z, Zhang X Y, Bai S, et al. On training implicit models. In: Proceedings of the 35th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2021, 34: 24247–2460

63 Giryes R, Eldar Y C, Bronstein A M, Sapiro G. Tradeoffs between convergence speed and reconstruction accuracy in inverse problems. IEEE Trans Signal Process, 2018, 66: 1676–1690

64 Gogna A, Tayal A. Metaheuristics: Review and application. J Exp Theoret Artificial Intell, 2013, 25: 503–526

65 Gomory R E. An Algorithm for Integer Solutions to Lmear Programs. Princeton-IBM Mathematics Research Project Technical Report 1. Princeton: Princeton University, 1958

66 Gomory R E. Solving linear programming problems in integers. Combin Anal, 1960, 10: 211–215

67 Goodfellow I, Bengio Y, Courville A. Deep Learning. Cambridge: MIT Press, 2016

68 Gregor K, LeCun Y. Learning fast approximations of sparse coding. In: Proceedings of the 27th International Conference on Machine Learning. Ann Arbor: PMLR, 2010, 399–406

69 Griewank A, Walther A. Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Philadelphia: SIAM, 2008

70 Gupta H, Jin K H, Nguyen H Q, et al. CNN-based projected gradient descent for consistent CT image reconstruction. IEEE Trans Medical Imag, 2018, 37: 1440–1453

71 Gupta P, Gasse M, Khalil E, et al. Hybrid models for learning to branch. In: Proceedings of the 34th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2020, 33: 18087–18097

72 Gupta P, Khalil E B, Chetlat D, et al. Lookback for learning to branch. arXiv:2206.14987, 2022

73 Han S, Fu R, Wang S, et al. Online adaptive dictionary learning and weighted sparse coding for abnormality detection. In: Proceedings of the 2013 IEEE International Conference on Image Processing. San Francisco: IEEE, 2013, 151–155

74 Hauptmann A, Lucka F, Betcke M, et al. Model-based learning for accelerated, limited-view 3-D photoacoustic tomography. IEEE Trans Medical Imag, 2018, 37: 1382–1393

75 He H, Daume III H, Eisner JM. Learning to search in branch and bound algorithms. In: Proceedings of the 28th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2014, 27

76 He H, Wen C K, Jin S, et al. Model-driven deep learning for MIMO detection. IEEE Trans Signal Process, 2020, 68: 1702–1715

77 He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. San Francisco: IEEE, 2016, 770–778

78 Heaton H, Chen X, Wang Z, et al. Safeguarded learned convex optimization. In Proceedings of the 37th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2023, 7848–7855

79 Heaton H, Fung S W, Lin A T, et al. Wasserstein-based projections with applications to inverse problems. SIAM J Math Data Sci, 2022, 4: 581–603

80 Hendel G. Adaptive large neighborhood search for mixed integer programming. Math Program Comput, 2022, 14: 185–221

81 Himmich I, El Hachemi N, El Hallaoui I, et al. MPILS: An automatic tuner for MILP solvers. Comput Oper Res, 2023, 159: 106344

82 Hornik K, Stinchcombe M, White H. Multilayer feedforward networks are universal approximators. Neural Netw, 1989, 2: 359–366

83 Hosny A, Reda S. Automatic MILP solver configuration by learning problem similarities. Ann Oper Res, 2024, in press

84 Hottung A, Tierney K. Neural large neighborhood search for the capacitated vehicle routing problem. arXiv:1911.09539, 2019

85 Huang L, Chen X, Huo W, et al. Improving primal heuristics for mixed integer programming problems based on problem reduction: A learning-based approach. In: Proceedings of the 17th International Conference on Control, Automation, Robotics and Vision. San Francisco: IEEE, 2022, 181–186

86 Huang T, Ferber A M, Tian Y, et al. Searching large neighborhoods for integer linear programs with contrastive learning. In: Proceedings of the 40th International Conference on Machine Learning. Ann Arbor: PMLR, 2023, 13869–13890

87 Huang T, Li J, Koenig S, et al. Anytime multi-agent path finding via machine learning-guided large neighborhood search. In Proceedings of the 36th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2022, 9368–9376

88 Huang Z, Wang K, Liu F, et al. Learning to select cuts for efficient mixed-integer programming. Pattern Recog, 2022, 123: 108353

89 Hutter F, Hoos H H, Leyton-Brown K. Sequential model-based optimization for general algorithm configuration. In: Proceedings of the 5th International Conference on Learning and Intelligent Optimization. Berlin-Heidelberg: Springer, 2011, 507–523

90 Hutter F, Hoos H H, Leyton-Brown K, et al. ParamILS: An automatic algorithm configuration framework. J Artificial Intell Res, 2009, 36: 267–306

91 Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Proceedings of the 32nd International Conference on Machine Learning. Ann Arbor: PMLR, 2015, 448–456

92 Jegelka S. Theory of graph neural networks: Representation and learning. In: Proceedings of the 2022 International Congress of Mathematicians.

93 Jia H, Shen S. Benders cut classification via support vector machines for solving two-stage stochastic programs. INFORMS J Optim, 2021, 3: 278–297

94 Joukovsky B, Mukherjee T, Van Luong H, et al. Generalization error bounds for deep unfolding RNNs. In: Uncertainty in Artificial Intelligence PMLR. Ann Arbor: PMLR, 2021, 1515–1524

95 Kadioglu S, Malitsky Y, Sellmann M, et al. ISACCinstance-specific algorithm configuration. In: Proceedings of the 19th European Conference on Artificial Intelligence. Amsterdam: IOS Press, 2010, 751–756

96 Kang E, Chang W, Yoo J, et al. Deep convolutional framelet denosing for low-dose CT via wavelet residual network. IEEE Trans Medical Imag, 2018, 37: 1358–1369

97 Kao Y H, Roy B, Yan X. Directed regression. In: Proceedings of the 23rd Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2009, 22

98 Khalil E B, Dai H, Zhang Y, et al. Learning combinatorial optimization algorithms over graphs. In: Proceedings of the 31st Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2017, 30

99 Khalil E B, Le Bodic P, Song L, et al. Learning to branch in mixed integer programming. In Proceedings of the 30th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2016

100 Khalil E B, Morris C, Lodi A. MIP-GNN: A data-driven framework for guiding combinatorial solvers. In: Proceedings of the 36th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2022, 10219–10227

101 Kingma D P, Ba J. Adam: A method for stochastic optimization. arXiv:1412.6980, 2014

102 Kouni V, Panagakis Y. DECONET: An unfolding network for analysis-based compressed sensing with generalization error bounds. IEEE Trans Signal Process, 2023, 71: 1938–1951

103 Labassi A G, Chtelat D, Lodi A. Learning to compare nodes in branch and bound with graph neural networks. In: Proceedings of the 36th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2022, 35: 32000–32010

104 LeCun Y, Cortes C, Burges JC C. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/

105 Li Y, Bar-Shira O, Monga V, et al. Deep algorithm unrolling for biomedical imaging. arXiv:2108.06637, 2021

106 Lin J, Zhu J, Wang H, et al. Learning to branch with Tree-aware Branching Transformers. Knowledge-Based Syst, 2022, 252: 109455

107 Liu D, Fischetti M, Lodi A. Learning to search in local branching. In Proceedings of the 36th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2022, 3796–3803

108 Liu J, Chen X, Wang Z, et al. ALISTA: Analytic Weights Are As Good As Learned Weights in LISTA. In: Proceedings of the 6th International Conference on Learning Representations. New Orleans: OpenReview.net, 2018

109 Liu J, Chen X, Wang Z, et al. Towards constituting mathematical structures for learning to optimize. arXiv:2305.18577, 2023

110 Long J, Shelhamer E, Darrell T. Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. San Francisco: IEEE, 2015, 3431–3440

111 Ma Y, Li J, Cao Z, et al. Efficient neural neighborhood search for pickup and delivery problems. arXiv:2204.11399, 2022

112 Malitsky Y. Instance-Specific Algorithm Configuration. New York: Springer, 2014

113 Mandi J, Guns T. Interior point solving for LP-based prediction+optimisation. In: Proceedings of the 34th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2020, 33: 7272–7282

114 Mao X, Shen C, Yang Y B. Image restoration using very deep convolutional encoder-decoder networks with symmetric skip connections. In: Proceedings of the 30th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2016, 29

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

116 Mardani M, Sun Q, Donoho D, et al. Neural proximal gradient descent for compressive imaging. In: Proceedings of the 32nd Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2018, 31

117 McKenzie D, Fung S W, Heaton H. Faster predict-and-optimize with Davis-Yin splitting. arXiv:2301.13395, 2023

118 Meinhardt T, Moller M, Hazirbas C, et al. Learning proximal operators: Using denoising networks for regularizing inverse imaging problems. In Proceedings of the IEEE International Conference on Computer Vision. San Francisco: IEEE, 2017, 1781–1790

119 Miyato T, Kataoka T, Koyama M, et al. Spectral normalization for generative adversarial networks. In: Proceedings of the 6th International Conference on Learning Representations. New Orleans: OpenReview.net, 2018

120 Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning. Nature, 2015, 518: 529–533

121 Monga V, Li Y, Eldar YC. Algorithm unrolling: Interpretable, efficient deep learning for signal and image processing. IEEE Signal Process Mag, 2021, 38: 18–44

122 Moreau T, Bruna J. Understanding neural sparse coding with matrix factorization. In: Proceedings of the 5th International Conference on Learning Representations. New Orleans: OpenReview.net, 2017

123 Nair V, Bartunov S, Gimeno F, et al. Solving mixed integer programs using neural networks. arXiv:2012.13349, 2020

124 Oberman A M, Calder J. Lipschitz regularized deep neural networks converge and generalize. arXiv:1808.09540, 2018

125 Parsonson C W, Laterre A, Barrett T D. Reinforcement learning for branch-and-bound optimisation using retrospective trajectories. In Proceedings of the 37th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2023, 4061–4069

126 Pascanu R, Mikolov T, Bengio Y. On the difficulty of training recurrent neural networks. In: Proceedings of the 30th International Conference on Machine Learning. Ann Arbor: PMLR, 2013, 1310–1318

127 Paulus M, Krause A. Learning to dive in branch and bound. In: Proceedings of the 37th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2024, 36

128 Paulus M, Zarpellon G, Krause A, et al. Learning to cut by looking ahead: Cutting plane selection via imitation learning. In: Proceedings of the 39th International Conference on Machine Learning. Ann Arbor: PMLR, 2022, 17584–17600

129 Pramanik A, Aggarwal H K, Jacob M. Deep generalization of structured low-rank algorithms (Deep-SLR). IEEE Trans Medical Imag, 2020, 39: 4186–4197

130 Prouvost A, Dumouchelle J, Scavuzzo L, et al. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. In: Learning Meets Combinatorial Algorithms at NeurIPS 2020. New Orleans: OpenReview.net, 2020

131 Qian H, Wegman M N. L2-nonexpansive neural networks. In: Proceedings of the 6th International Conference on Learning Representations. New Orleans: OpenReview.net, 2018

132 Qu Q, Li X, Zhou Y, et al. An improved reinforcement learning algorithm for learning to branch. arXiv:2201.06213, 2022

133 Rick Chang J H, Li C L, Poczos B, et al. One network to solve them all—solving linear inverse problems using deep projection models. In Proceedings of the IEEE International Conference on Computer Vision. San Francisco: IEEE, 2017, 5888–5897

134 Rudin L I, Osher S, Fatemi E. Nonlinear total variation based noise removal algorithms. Physica D, 1992, 60: 259–268

135 Ryu E, Liu J, Wang S, et al. Plug-and-play methods provably converge with properly trained denoisers. In: Proceedings of the 36th International Conference on Machine Learning. Ann Arbor: PMLR, 2019, 5546–5557

136 Ryu E, Yin W. Large-scale convex optimization: Algorithms & Analyses via Monotone Operators. Cambridge: Cambridge Univ Press, 2022

137 Samuel N, Diskin T, Wiesel A. Learning to detect. IEEE Trans Signal Process, 2019, 67: 2554–2564

138 Scarlett J, Heckel R, Rodrigues M R, et al. Theoretical perspectives on deep learning methods in inverse problems. IEEE J Sel Area Inform Theo, 2022, 3: 433–453
139 Scavuzzo L, Chen F, Chtelat D, et al. Learning to branch with tree mdps. In: Proceedings of the 36th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2022, 35: 18514–18526
140 Schnoor E, Behboodi A, Rauhut H. Generalization error bounds for iterative recovery algorithms unfolded as neural networks. Inform Infer: A J IMA, 2023, 12: 2267–2299
141 Shen Y, Sun Y, Eberhard A, et al. Learning primal heuristics for mixed integer programs. In: Proceedings of the 2021 International Joint Conference on Neural Networks. San Francisco: IEEE, 2021, 1–8
142 Silver D, Huang A, Maddison C J, et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016, 529: 484-489
143 Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556, 2014
144 Snoek J, Larochelle H, Adams R P. Practical bayesian optimization of machine learning algorithms. In: Proceedings of the 26th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2012, 25
145 Solomon O, Cohen R, Zhang Y, et al. Deep unfolded robust PCA with application to clutter suppression in ultrasound. IEEE Trans Medical Imag, 2019, 39: 1051–1063
146 Song J, Yue Y, Dilkina B. A general large neighborhood search framework for solving integer linear programs. In: Proceedings of the 34th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2020, 33: 20012–20023
147 Song W, Liu Y, Cao Z, et al. Instance-specific algorithm configuration via unsupervised deep graph clustering. Eng Appl Artificial Intell, 2023, 125: 106740
148 Sonnerat N, Wang P, Ktena I, et al. Learning a large neighborhood search algorithm for mixed integer programs. arXiv:2107.10201, 2021
149 Sreehari S, Venkatakrishnan S V, Wohlberg B, et al. Plug-and-play priors for bright field electron tomography and sparse interpolation. IEEE Trans Comput Imag, 2016, 2: 408–423
150 Sreter H, Giryes R. Learned convolutional sparse coding. In: Proceedings of the 2018 IEEE International Conference on Acoustics, Speech and Signal Processing. San Francisco: IEEE, 2018, 219-2195
151 Sutton R S, McAllester D, Singh S, et al. Policy gradient methods for reinforcement learning with function approximation. In: Proceedings of the 13th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 1999, 12
152 Takabe S, Wadayama T. Theoretical interpretation of learned step size in deep-unfolded gradient descent. arXiv:2001.05142, 2020
153 Takabe S, Wadayama T, Eldar Y C. Complex trainable ista for linear and nonlinear inverse problems. In: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing. San Francisco: IEEE, 2020, 5020–5024
154 Tang Y, Agrawal S, Faenza Y. Reinforcement learning for integer programming: Learning to cut. In: Proceedings of the 37th International Conference on Machine Learning. Ann Arbor: PMLR, 2020, 9367–9376
155 Teerapittayanon S, McDanel B, Kung H T. Branchynet: Fast inference via early exiting from deep neural networks. In: Proceedings of the 23rd International Conference on Pattern Recognition. San Francisco: IEEE, 2016, 2464–2469
156 Terris M, Repetti A, Pesquet J C, et al. Enhanced convergent PnP algorithms for image restoration. In: Proceedings of the IEEE International Conference on Image Processing. San Francisco: IEEE, 2021, 1684–1688
157 Turner M, Koch T, Serrano F, et al. Adaptive cut selection in mixed-integer linear programming. Open J Math Optim, 2023, 4: 1–28
158 Ulyanov D, Vedaldi A, Lempitsky V. Deep image prior. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. San Francisco: IEEE, 2018, 9446–9454
159 Valentin R, Ferrari C, Scheurer J, et al. Instance-wise algorithm configuration with graph neural networks. arXiv:2202.04910, 2022
160 Venkatakrishnan S V, Bouman C A, Wohlberg B. Plug-and-play priors for model based reconstruction. In: Proceedings of the IEEE Global Conference on Signal and Information Processing. San Francisco: IEEE, 2013, 945–948
161 Vu B C. A splitting algorithm for dual monotone inclusions involving cocoercive operators. Adv Comput Math, 2013, 38: 667–681
162 Wadayama T, Takabe S. Deep learning-aided trainable projected gradient decoding for LDPC codes. In: Proceedings of the IEEE International Symposium on Information Theory. San Francisco: IEEE, 2019, 2444–2448
163 Wang Z, Li X, Wang J, et al. Learning cut selection for mixed-integer linear programming via hierarchical sequence model. In: Proceedings of the 11th International Conference on Learning Representations. New Orleans: OpenReview.net, 2022
164 Wang Z, Liu D, Chang S, et al. D3: Deep dual-domain based fast restoration of JPEG-compressed images. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. San Francisco: IEEE, 2016,

2764–2772

165 Wei K, Aviles-Rivero A, Liang J, et al. Tuning-free plug-and-play proximal algorithm for inverse imaging problems. In: Proceedings of the 37th International Conference on Machine Learning. Ann Arbor: PMLR, 2020, 10158–10169

166 Weng T W, Zhang H, Chen P Y, et al. Evaluating the robustness of neural networks: An extreme value theory approach. In: Proceedings of the 7th International Conference on Learning Representations. New Orleans: OpenReview.net, 2018

167 Wilder B, Dilkina B, Tambe M. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In Proceedings of the 33rd AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2019, 1658–1665

168 Wolpert D H, Macready W G. No free lunch theorems for optimization. IEEE Trans Evol Comput, 1997, 1: 67–82

169 Wolsey L A. Integer Programming. New York: John Wiley & Sons, 2020

170 Wöllmer M, Kaiser M, Eyben F, et al. LSTM-modeling of continuous emotions in an audiovisual affect recognition framework. Image Vision Comput, 2013, 31: 153–163

171 Wu K, Guo Y, Li Z, et al. Sparse coding with gated learned ISTA. In: Proceedings of the 7th International Conference on Learning Representations. New Orleans: OpenReview.net, 2019

172 Wu L, Cui P, Pei J, et al. Graph neural networks: Foundation, frontiers and applications. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. New York: Association for Computing Machinery, 2022, 4840–4841

173 Wu Y, Song W, Cao Z, et al. Learning large neighborhood search policy for integer programming. In: Proceedings of the 35th Conference on Neural Information Processing Systems. Adv Neural Informn Process Syst, 2021, 34: 30075-30087

174 Xie X, Wu J, Liu G, et al. Differentiable linearized ADMM. In: Proceedings of the 36th International Conference on Machine Learning. Ann Arbor: PMLR, 2019, 6902–6911

175 Xu L, Hutter F, Hoos H H, et al. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In: Proceedings of the RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence. RCRA, 2011, 16–30

176 Yang C, Gu Y, Chen B, et al. Learning proximal operator methods for nonconvex sparse recovery with theoretical guarantee. IEEE Trans Signal Process, 2020, 68: 5244–5259

177 Yang L, Shami A. On hyperparameter optimization of machine learning algorithms: Theory and practice. Neurocomputing, 2020, 415: 295–316

178 Yilmaz K, Yorke-Smith N. A study of learning search approximation in mixed integer branch and bound: Node selection in scip. Artificial Intell, 2021, 2: 150–178

179 Yuan X, Liu Y, Suo J, et al. Plug-and-play algorithms for large-scale snapshot compressive imaging. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. San Francisco: IEEE, 2020, 1447–1457

180 Zarka J, Thiry L, Angles T, et al. Deep Network Classification by Scattering and Homotopy Dictionary Learning. In: Proceedings of the 8th International Conference on Learning Representations. New Orleans: OpenReview.net, 2020

181 Zarpellon G, Jo J, Lodi A, et al. Parameterizing branch-and-bound search trees to learn branching policies. In Proceedings of the 35th AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2021, 3931–3939

182 Zhang J, Ghanem B. ISTA-Net: Interpretable optimization-inspired deep network for image compressive sensing. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. San Francisco: IEEE, 2018, 1828–1837

183 Zhang K, Li Y, Zuo W, et al. Plug-and-play image restoration with deep denoiser prior. IEEE Trans Pattern Anal Mach Intell, 2021, 44: 6360–6376

184 Zhang K, Zuo W, Chen Y, et al. Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising. IEEE Trans Image Process, 2017, 26: 3142–3155

185 Zhang K, Zuo W, Gu S, et al. Learning deep CNN denoiser prior for image restoration. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. San Francisco: IEEE, 2017, 3929–3938

186 Zhang K, Zuo W, Zhang L. FFDNet: Toward a fast and flexible solution for CNN-based image denoising. IEEE Trans Image Process, 2018, 27: 4608–4622

187 Zhang M, Yin W, Wang M, et al. Mindopt tuner: Boost the performance of numerical software by automatic parameter tuning. arXiv:2307.08085, 2023

188 Zhang T, Banitalebi-Dehkordi A, Zhang Y. Deep reinforcement learning for exact combinatorial optimization: Learning to branch. In: Proceedings of the 26th International Conference on Pattern Recognition. San Francisco: IEEE, 2022, 3105–3111

189 Zhang X, Lu Y, Liu J, et al. Dynamically unfolding recurrent restorer: A moving endpoint control method for image restoration. In: Proceedings of the 6th International Conference on Learning Representations. New Orleans:

OpenReview.net, 2018

190 Zhao B, Li F F. Online detection of unusual events in videos via dynamic sparse coding. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. San Francisco: IEEE, 2011, 3313–3320

191 Zou Y, Zhou Y, Chen X, et al. Proximal gradient-based unfolding for massive random access in IoT networks. arXiv:2212.01839, 2022