Learning with Combinatorial Optimization Layers: a Probabilistic Approach

Guillaume Dalle, Léo Baty, Louis Bouvier, Axel Parmentier

Learning with Combinatorial Optimization Layers: a Probabilistic Approach

Guillaume Dalle[1], Léo Baty[1], Louis Bouvier[1], and Axel Parmentier[1,*]

[1]CERMICS, Ecole des Ponts, Marne-la-Vallée, France
[*]Corresponding author: axel.parmentier@enpc.fr

December 3, 2022

Abstract

Combinatorial optimization (CO) layers in machine learning (ML) pipelines are a powerful tool to tackle data-driven decision tasks, but they come with two main challenges. First, the solution of a CO problem often behaves as a piecewise constant function of its objective parameters. Given that ML pipelines are typically trained using stochastic gradient descent, the absence of slope information is very detrimental. Second, standard ML losses do not work well in combinatorial settings. A growing body of research addresses these challenges through diverse methods. Unfortunately, the lack of well-maintained implementations slows down the adoption of CO layers.

In this paper, building upon previous works, we introduce a probabilistic perspective on CO layers, which lends itself naturally to approximate differentiation and the construction of structured losses. We recover many approaches from the literature as special cases, and we also derive new ones. Based on this unifying perspective, we present `InferOpt.jl`, an open-source Julia package that 1) allows turning any CO oracle with a linear objective into a differentiable layer, and 2) defines adequate losses to train pipelines containing such layers. Our library works with arbitrary optimization algorithms, and it is fully compatible with Julia's ML ecosystem. We demonstrate its abilities using a pathfinding problem on video game maps as guiding example, as well as three other applications from operations research.

Keywords: combinatorial optimization, machine learning, automatic differentiation, graphs, Julia programming language

Contents

1 Introduction

Machine learning (ML) and combinatorial optimization (CO) are two essential ingredients of modern industrial processes. While ML extracts meaningful information from noisy data, CO enables decision-making in high-dimensional constrained environments. But in many situations, combining both of these tools is necessary: for instance, we might want to generate predictions from data, and then use those predictions to make optimized decisions. To do that, we need *pipelines* that contain two types of *layers*: ML layers and CO layers.

Due to their many possible applications, hybrid ML-CO pipelines currently attract a lot of research interest. The recent reviews by Bengio, Lodi, and Prouvost (2021) and Kotary et al. (2021) are excellent resources on this topic. Unfortunately, relevant software implementations are scattered across paper-specific repositories, with few tests, minimal documentation and sporadic code maintenance. Not only does this make comparison and evaluation difficult for academic purposes, it also hurts practitioners wishing to experiment with such techniques on real use cases.

Let us discuss a generic hybrid ML-CO pipeline, which includes a CO oracle amid several ML layers:

$$\xrightarrow{\text{Input } x} \boxed{\text{ML layers}} \xrightarrow{\text{Objective } \theta} \boxed{\boxed{\text{CO oracle}}} \xrightarrow{\text{Solution } y} \boxed{\text{More ML layers}} \xrightarrow{\text{Output}} \tag{1}$$

The inference problem consists in predicting an output from a given input. It is solved online, and requires the knowledge of the parameters (weights) for each ML layer. On the other hand, the learning problem aims at finding parameters that lead to "good" outputs during inference. It is solved offline based on a training set that contains several inputs, possibly complemented by target outputs.

In Equation (1), we use the term *CO oracle* to emphasize that any algorithm may be used to solve the optimization problem, whether it relies on an existing solver or a handcrafted implementation. Conversely, when we talk about a *layer*, it is implied that we can compute meaningful derivatives using automatic differentiation (AD). Since it may call black box subroutines, an arbitrary CO oracle is seldom compatible with AD. And even when it is, its derivatives are zero almost everywhere, which gives us no exploitable slope information. Therefore, according to our terminology, *a CO oracle is not a layer (yet)*, and the whole point of this paper is to turn it into one.

Modern ML libraries provide a wealth of basic building blocks that allow users to assemble and train complex pipelines. We want to leverage these libraries to create hybrid ML-CO pipelines, but we face two main challenges. First, while ML layers are easy to construct, it is not obvious how to transform a CO oracle into a usable layer. Second, standard ML losses are ill-suited to our setting, because they often ignore the underlying optimization problem.

Our goal is to remove these difficulties. We introduce `InferOpt.jl`[1], a Julia package which 1) can turn any CO oracle into a layer with meaningful derivatives, and 2) provides structured loss functions that work well with the resulting layers. It contains several state-of-the-art methods that are fully compatible with Julia's AD and ML ecosystem, making CO layers as easy to use as any ML layer. To describe the available methods in a coherent manner, we leverage the unifying concept of probabilistic CO layer, hence the name of our package.

1.1 Motivating example

Let us start by giving an example of hybrid ML-CO pipeline. Suppose we want to find shortest paths on a map, but we do not have access to an exact description of the underlying terrain. Instead, all we have are images of the area, which give us a rough idea of the topography and obstacles. To solve our problem, we need a pipeline comprising two layers of very different natures. First, an image processing layer, which is typically implemented as a convolutional neural network (CNN). The CNN is tasked with translating the images into a weighted graph. Second, a CO layer performing shortest path computations on said weighted graph (*e.g.* using Dijkstra's algorithm).

[1] https://github.com/axelparmentier/InferOpt.jl

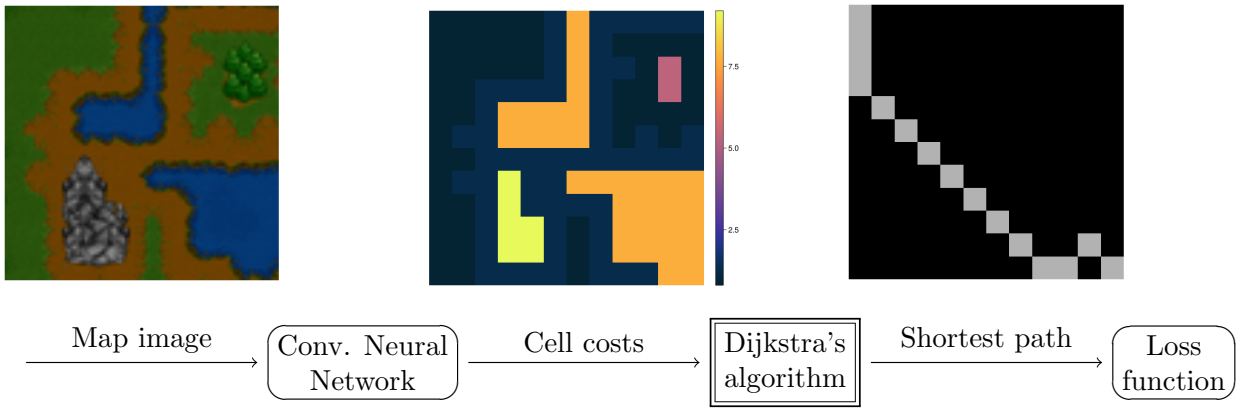


Figure 1: Pipeline for computing shortest paths on Warcraft maps – data from Vlastelica et al. (2020)

This pipeline is exactly the one considered by Vlastelica et al. (2020) and Berthet et al. (2020) for pathfinding on video game maps. We illustrate it on Figure 1, and we describe it in more detail in Section 2.3. The goal is to learn appropriate weights for the CNN, so that it feeds accurate cell costs to Dijkstra's algorithm. This is done by minimizing a loss function, such as the distance between the true optimal path and the one we predict.

1.2 Our setting

In our hybrid ML-CO pipelines, we consider CO oracles f that solve the following kind of problem:

$$f : \theta \longmapsto \underset{v \in \mathcal{V}}{\operatorname{argmax}} \, \theta^\top v \tag{2}$$

Here, the input $\theta \in \mathbb{R}^d$ is the *objective direction*. Meanwhile, $\mathcal{V} \subset \mathbb{R}^d$ (for *vertices*) denotes a finite set of feasible solutions – which may be exponentially large in d – among which the optimal solution $f(\theta)$ shall be selected. For simplicity, we assume that f is single-valued, *i.e.* that the optimal solution is unique.

The feasible set $\mathcal{V}$ and its dimension may depend on the instance. For instance, if Equation (2) is a shortest path problem, the underlying graph may change from one input to another. If we wanted to remain generic, we should therefore write $\mathcal{V}(x) \subset \mathbb{R}^{d(x)}$. To keep notations simple, we omit the dependency in x whenever it is clear from the context. Note that we could also study more general CO oracles given by

$$\underset{v \in \mathcal{V}}{\operatorname{argmax}} \, \theta^\top g(v) + h(v)$$

where g is any function from an arbitrary finite set $\mathcal{V}$ to $\mathbb{R}^d$, and h is any function from $\mathcal{V}$ to $\mathbb{R}$. As long as the objective is linear in θ, the theory we present generalizes seamlessly. However, for ease of exposition, we keep $g(v) = v$ and $h(v) = 0$. In this case, Equation (2) is equivalent to

$$\underset{v \in \operatorname{conv}(\mathcal{V})}{\operatorname{argmax}} \, \theta^\top v.$$

Indeed, when the objective is linear in v, it makes no difference to optimize over the convex hull $\operatorname{conv}(\mathcal{V})$ instead of optimizing over $\mathcal{V}$.

1.2.1 From an optimization problem to an oracle

It is important to note that the formulation $\operatorname{argmax}_{v \in \mathcal{V}} \theta^\top v$ is very generic. Any linear program (LP) or integer linear program (ILP) can be written this way, as long as its feasible set is bounded. Indeed, the

optimum of an LP is always reached at a vertex of the polytope, of which there are finitely many. Meanwhile, the optimum of an ILP is always reached at an integral point, or more precisely, at a vertex of the convex hull of the integral points. As a result, Equation (2) encompasses a variety of well-known CO problems related to graphs (paths, flows, spanning trees, coloring), resource management (knapsack, bin packing), scheduling, *etc.* See Korte and Vygen (2006) for an overview of CO and its applications.

For every one of these problems, dedicated algorithms have been developed over the years, which sometimes exploit the domain structure better than a generic ILP solver (such as Gurobi or SCIP). Thus, we have no interest in restricting the procedure used to compute an optimal solution: we want to pick the best algorithm for each application. That is why the methods discussed in this paper only need to access the CO oracle f as a black box function, without making assumptions about its implementation.

1.2.2 From an oracle to a probability distribution

When using CO oracles within ML pipelines, the first challenge we face is the *lack of useful derivatives*. Training often relies on stochastic gradient descent (SGD), so we need to be able to backpropagate loss gradients onto the weights of the ML layers. Unfortunately, since the feasible set $\mathcal{V}$ of Equation (2) is finite, the CO oracle is a piecewise constant mapping and its derivatives are zero almost everywhere. To recover useful slope information, we seek *approximate derivatives*, which is where the probabilistic approach comes into play.

To describe it, we no longer think about a CO oracle as a function returning a single element $f(\theta)$ from $\mathcal{V}$. Instead, we use it to define a probability distribution $p(\cdot|\theta)$ on $\mathcal{V}$. The naive choice would be the Dirac mass $p(v|\theta) = \delta_{f(\theta)}(v)$, but it shares the lack of differentiability of the oracle itself. Thus, our goal is to spread out the distribution p into an approximation $\widehat{p}$, such that the probability mapping $\theta \longmapsto \widehat{p}(\cdot|\theta)$ becomes smooth with respect to θ. If we can do that, then the expectation mapping

$$\widehat{f} : \theta \longmapsto \mathbb{E}_{\widehat{p}(\cdot|\theta)}[V] = \sum_{v \in \mathcal{V}} v \widehat{p}(v|\theta), \tag{3}$$

where it is understood that $V \sim \widehat{p}(\cdot|\theta)$, will be just as smooth. This expectation mapping $\widehat{f}$ is what we take to be our *probabilistic CO layer*: see Section 3 for detailed examples. In what follows, plain letters (p, f) always refer to the initial CO oracle, while letters with a hat $(\widehat{p}, \widehat{f})$ refer to the probabilistic CO layer that we wrap around it.

1.2.3 From a probability distribution to a loss function

The presence of CO oracles in ML pipelines gives rise to a second challenge: the choice of an appropriate loss function to learn the parameters. As highlighted by Bengio, Lodi, and Prouvost (2021), this choice heavily depends on the data at our disposal. They distinguish two main paradigms, which we illustrate using the pipeline of Figure 1.

If our dataset only contains the map images, then we are in a weakly supervised setting, which they call *learning by experience* (see Section 4). In that case, the loss function will evaluate the solutions computed by our pipeline using the true cell costs. On the other hand, if our dataset happens to contain precomputed targets such as the true shortest paths, then we are in a fully supervised setting, which they call *learning by imitation* (see Section 5). In that case, the loss function will compare the paths computed by our pipeline with the optimal ones, hoping to minimize the discrepancy. For both of these cases, the probabilistic perspective plays an important role in ensuring smoothness of the loss.

1.2.4 Complete pipeline

The typical pipeline we will focus on is a special case of Equation (1), which we now describe in more detail:

$$\xrightarrow{\text{Input } x} \boxed{\text{ML layer } \varphi_w} \xrightarrow{\text{Objective } \theta = \varphi_w(x)} \boxed{\boxed{\text{CO oracle } f}} \xrightarrow{\text{Solution } y = f(\theta)} \boxed{\text{Loss function } \mathcal{L}} \tag{4}$$

Our pipeline starts with an ML layer φ_w, where w stands for the vector of weights. Its role is to encode the input x into an objective direction $\theta = \varphi_w(x)$, which is why we often refer to it as the *encoder*. Then, the CO oracle defined in Equation (2) returns an optimal solution $y = f(\theta)$. Finally, the loss function $\mathcal{L}$ is used during training to evaluate the quality of the solution. If our dataset contains N input samples $x^{(1)}, ..., x^{(N)}$, training occurs by applying SGD to the following loss minimization problem:

$$\min_w \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\Big(\overbrace{f\big(\underbrace{\varphi_w(x^{(i)})}_{\theta^{(i)}}\big)}^{y^{(i)}}, \dots \Big). \tag{5}$$

The dots ... correspond to additional arguments that may be used by the loss. For instance, the loss may depend on the input $x^{(i)}$ itself, or require targets $\bar{t}^{(i)}$ for comparison (see Section 5).

Remark 1.1. *Our work focuses on individual layers and loss functions. Although we present various concrete examples, we do not give generic advice on how to build the whole pipeline for a specific application. If the use case corresponds to a "predict, then optimize" setting such as the one from Figure 1, then Elmachtoub and Grigas (2022) give a few useful pointers. If the goal is to approximate hard optimization problems with easier ones, the reader can refer to Parmentier (2021a) for a general methodology.*

1.3 Contributions

Our foremost contribution is the open-source package `InferOpt.jl`, which is written in the Julia programming language (Bezanson et al. 2017). Given a CO oracle provided as a callable object, our package wraps it into a probabilistic CO layer that is compatible with Julia's AD and ML ecosystem. This is achieved thanks to the `ChainRules.jl`[2] interface (White et al. 2022). Moreover, `InferOpt.jl` defines several structured loss functions, both for learning by experience and for learning by imitation.

On top of that, we present theoretical insights that fill some gaps in previous works. In addition to the framework of probabilistic CO layers, we propose:

- A new perturbation technique designed for CO oracles that only accept objective vectors with a certain sign (such as Dijkstra's algorithm, which fails on graphs with negative edge costs): see Section 3.3.2.
- A way to differentiate through a large subclass of probabilistic CO layers (those that rely on convex regularization) by combining the Frank-Wolfe algorithm with implicit differentiation: see Section 3.3.3.
- A probabilistic regularization of the regret for learning by experience: see Section 4.
- A generic decomposition framework for imitation losses, which subsumes most of the literature so far and suggests ways to build new loss functions: see Section 5.1.

Finally, we describe numerical experiments on our motivating example of Warcraft shortest paths, as well as three combinatorial optimization problems from operations research: the stochastic vehicle scheduling problem, the single-machine scheduling problem, and the two-stage stochastic minimum weight spanning tree problem. Here are a few highlights:

- We benchmark and show the strengths of the different learning methods proposed in the literature on hybrid ML-CO pipelines.
- We use the pipeline of Figure 1 for learning by experience on the Warcraft shortest paths problem, even though the CNN encoder has tens of thousands of parameters. To the best of our knowledge, previous attempts to learn such pipelines by experience were restricted to ML layers with fewer than 100 parameters.

[2] https://github.com/JuliaDiff/ChainRules.jl

- We tackle the stochastic vehicle scheduling problem efficiently by approximating it with its deterministic counterpart.
- We obtain a fast heuristic with state-of-the-art performance for the single machine scheduling problem, which has been a focus of the by the scheduling community for several decades.
- We show that our library enables the use of graph neural networks (GNNs) instead of generalized linear models (GLMs) for instance encoding, bringing additional performance on the two-stage spanning tree problem.

1.4 Notations

We write $\mathbf{1}$ for the vector with all components equal to 1, and e_i for the basis vector corresponding to dimension i. The notation $\mathbb{1}\{E\}$ corresponds to the indicator function of the set (or event) E. The operator $\odot$ denotes the Hadamard (componentwise) product between vectors of the same size. We use Δ^d to refer to the unit simplex of dimension d, and $\mathbb{E}_p$ to denote an expectation with respect to the distribution p. If $\mathcal{S}$ is a set, we write

$$\mathrm{conv}(\mathcal{S}) = \left\{ \sum_i p_i s_i : s_i \in \mathcal{S}, p_i \geq 0, \sum_i p_i = 1 \right\} = \{\mathbb{E}_p[S] : p \in \Delta^{\mathcal{S}}\} \tag{6}$$

for its convex hull and $\mathrm{proj}_{\mathcal{S}}$ for the orthogonal projection onto $\mathcal{S}$. If h is a real-valued function, we denote by $\nabla_a h(x)$ the gradient of h with respect to parameter a at point x and by $\partial_a h(x)$ its convex subdifferential (set of subgradients). The notation $\mathrm{dom}(h)$ stands for the domain of h, *i.e.* the set on which it takes finite values. If h is a vector-valued function, we denote by $J_a h(x)$ its Jacobian matrix.

1.5 Outline

In Section 2, we review the literature on differentiable optimization layers, before focusing on the Warcraft example. Section 3 introduces the family of probabilistic CO layers by splitting it into perturbed and regularized approaches. Then, Section 4 gives tools for learning by experience, while Section 5 discusses loss functions for learning by imitation. Practical applications of our package are presented in Section 6, before we conclude in Section 7. Proofs for our main theoretical results can be found in Appendix A.

2 Related work

2.1 Optimization layers in ML

A significant part of modern ML relies on AD: see Baydin et al. (2018) for an overview and Griewank and Walther (2008) for an in-depth treatment. In particular, AD forms the basis of the backpropagation algorithm used to train neural networks.

2.1.1 The notion of implicit layer

Standard neural architectures draw from a small collection of *explicit* layers (Goodfellow, Bengio, and Courville 2016). Whatever their connection structure (dense, convolutional, recurrent, *etc.*) and regardless of their activation function, these layers all correspond to input-output mappings that can be expressed using an analytic formula. This same formula is then used by AD to compute gradients.

On the other hand, the layers defined by `InferOpt.jl` are of the *implicit* kind, which means they can contain arbitrarily complex iterative procedures. While we focus here on optimization algorithms, those are not the only kind of implicit layers: fixed point iterations and differential equation solvers are also widely used, depending on the application at hand. See the tutorial by Kolter, Duvenaud, and Johnson (2020) for more thorough explanations.

Due to the high computational cost of unrolling iterative procedures, efficient AD of implicit layers often relies on the implicit function theorem. As long as we can specify a set of conditions satisfied by the input-output pair, this theorem equates differentiation with solving a linear system of equations. See the Python package `jaxopt`[3] for an example implementation, and its companion paper for theoretical details (Blondel, Berthet, et al. 2022). The recent Python package `theseus`[4] showcases an application of this technique to robotics and vision (Pineda et al. 2022).

2.1.2 Convex optimization layers

Among the early works on optimization layers for deep learning, the seminal OptNet paper by Amos and Kolter (2017) stands out. It describes a way to differentiate through quadratic programs (QPs) by using the Karush-Kuhn-Tucker (KKT) optimality conditions and plugging them into the implicit function theorem.

More sophisticated tools exist for disciplined conic programs, such as the Python package `cvxpylayers`[5] (Agrawal et al. 2019). The recent Julia package `DiffOpt.jl`[6] (Sharma et al. 2022) extends these ideas beyond the conic case to general convex programs. Note that both libraries only accept optimization problems formulated in a domain-specific modeling language, as opposed to arbitrary oracles.

Strong convexity makes differentiation easier because the solutions evolve smoothly as a function of the constraints and objective parameters. In particular, this means the methods listed above return exact derivatives and do not rely on approximations. Regrettably, this nice behavior falls apart as soon as we enter the combinatorial world.

2.1.3 Linear optimization layers

Let us consider an LP whose feasible set is a bounded polyhedron, also called *polytope*. It is well-known that for most objective directions, the optimal solution will be unique and located at a vertex of the polytope. Even though LPs look like continuous optimization problems, this property shows that they are fundamentally combinatorial. Indeed, a small change in the objective direction can cause the optimal solution to suddenly jump to another vertex, which results in a discontinuous mapping from objectives to solutions. In fact, this mapping is piecewise constant, which means no useful differential information can come from it: its Jacobian is undefined at the jump points and zero everywhere else.

Therefore, when differentiating LPs with respect to their objective parameters, we need to resort to approximations. Vlastelica et al. (2020) use interpolation to turn a piecewise constant mapping into a piecewise linear and continuous one. However, the dominant approximation paradigm in the literature is regularization, as formalized by Blondel, Martins, and Niculae (2020).

For instance, Wilder, Dilkina, and Tambe (2019) add a quadratic penalty to the linear objective, which allows them to reuse the QP computations of Amos and Kolter (2017). Mandi and Guns (2020) propose a log-barrier penalty, which lets them draw a connection with interior-point methods. Berthet et al. (2020) suggest perturbing the optimization problem by adding stochastic noise to the objective direction, which is a form of implicit regularization.

When LP layers are located at the end of a pipeline, a clever choice of loss function can also simplify differentiation. This is illustrated by the structured support vector machine (S-SVM) loss (Nowozin and Lampert 2010), smart "predict, then optimize" (SPO+) loss (Elmachtoub and Grigas 2022) and Fenchel-Young (FY) loss (Blondel, Martins, and Niculae 2020).

2.1.4 Integer and combinatorial optimization layers

In theory, the methods from the previous section still work in the presence of integer variables, that is, for ILPs. To apply them, we only need to consider the polytope defined by the convex hull of integral solutions.

[3] https://github.com/google/jaxopt
[4] https://github.com/facebookresearch/theseus
[5] https://github.com/cvxgrp/cvxpylayers
[6] https://github.com/jump-dev/DiffOpt.jl

Alas, in the general case, there is no concise way to describe this convex hull. This is why many authors decide to differentiate through the continuous relaxation of the ILP instead (Mandi, Demirović, et al. 2020): it is an outer approximation of the integral polytope, but it can be sufficient for learning purposes. Some suggest taking advantage of techniques specific to integer programming, such as integrality cuts (Ferber et al. 2020) or a generalization of the notion of active constraint (Paulus et al. 2021).

There has also been significant progress on finding gradient approximations for combinatorial problems such as ranking (Blondel, Teboul, et al. 2020) and shortest paths (Parmentier 2021b). Yet these techniques are problem-specific, and therefore hard to generalize, which is why we leave them aside.

Instead, we want to allow implicit manipulation of the integral polytope itself, without making assumptions on its structure. To achieve that, we can only afford to invoke CO oracles as black boxes (Vlastelica et al. 2020; Berthet et al. 2020). Compatibility with arbitrary algorithms is one of the fundamental tenets of `InferOpt.jl`. The recent Python package `PyEPO`[7] (Tang and Khalil 2022) shares our generic perspective, but it only implements a subset of `InferOpt.jl` and does not address the notion of probabilistic CO layer.

This probabilistic aspect is central to our proposal, and it is mostly inspired by the works of Blondel, Martins, and Niculae (2020) and Berthet et al. (2020). Another related approach is the Implicit Maximum Likelihood Estimation of Niepert, Minervini, and Franceschi (2021), whose probabilistic formulation is slightly different and based on exponential families.

Finally, note that CO layers can be computationally heavy due to the frequent need to recompute optimal solutions at each epoch. Mulamba et al. (2021) address this issue by proposing a noise contrastive approach with solution caching, while Mandi, Bucarey, et al. (2022) build upon this method with a focus on learning to rank.

2.2 Similarities and differences with reinforcement learning

As we will see in Section 4, learning by experience can be reminiscent of reinforcement learning (RL), which also relies on a reward or cost signal given by the environment (Sutton and Barto 2018). Furthermore, the encoder layer of Equation (4) is similar to a parametric approximation of the value function, which forms the basis of deep RL approaches. This prompts us to discuss a few differences between RL and the framework we study.

The standard mathematical formulation of RL is based on Markov decision processes (MDPs), where a reward and state transition are associated with each action. Usually, the available actions are elementary decisions: pull one lever of a multi-armed bandit, cross one edge on a graph, select one move in a board game, etc. The resulting value or policy update is local: it is specific to both the current state and the action taken. The more reward information we gather, the more efficient learning becomes.

In our framework, the basic step is a call to the optimizer. But combinatorial algorithms can go beyond simple actions: they often output a structured and high-dimensional solution, which aggregates many elementary decisions. This in turn triggers a global update in our knowledge of the system, whereby the final reward is redistributed between all elementary decisions. In a way, *backpropagation through the optimizer enables efficient credit assignment*, even for sparse reward signals.

Another difference is related to the Bellman fixed point equation. In an RL setting, the Bellman equation is used explicitly to derive parameter updates. In our setting, the Bellman equation is used implicitly within optimizers such as Dijkstra's algorithm.

To conclude, while standard RL decomposes a policy into elementary decisions, the pipelines we study here are able to look directly for complex multistep solutions. Note that a similar concept of *option* exists in hierarchical RL (Barto and Mahadevan 2003): comparing both perspectives in detail would no doubt be fruitful, and we leave it for future work.

[7] https://github.com/khalil-research/PyEPO

2.3 Our guiding example: shortest paths on Warcraft maps

As a way to clarify the concepts introduced in this paper, we illustrate them on the problem of Warcraft shortest paths (Vlastelica et al. 2020; Berthet et al. 2020), which was already introduced on Figure 1. The associated dataset[8], assembled by Vlastelica et al. (2020), contains randomly-generated maps similar to those from the Warcraft II video game. Each of these maps is a red-green-blue (RGB) image of size $ks \times ks$ containing $k \times k$ square cells of side length s. Every cell has its own terrain type (grass, forest, water, earth, *etc.*) which incurs a specific cost when game characters cross it.

The goal is to find the shortest path from the top left corner of the map to the bottom right corner. At prediction time, cell costs are unknown, which means they must be approximated from the image alone. Solving the problem of Warcraft shortest paths thus requires adapting the pipeline of Equation (4) as follows:

$$\xrightarrow[x \in \mathbb{R}^{ks \times ks \times 3}]{\text{Map image}} \boxed{\text{Conv. Neural Network } \varphi_w} \xrightarrow[\theta \in \mathbb{R}^{k \times k}]{\text{Negative cell costs}} \boxed{\begin{array}{c}\text{Dijkstra's} \\ \text{algorithm} \\ \operatorname{argmax}_{v \in \mathcal{P}_k} \theta^\top v\end{array}} \xrightarrow[y \in \mathcal{P}_k]{\text{Shortest path}} \boxed{\text{Loss function } \mathcal{L}} \tag{7}$$

where we have defined the set of feasible paths

$$\mathcal{P}_k = \{v \in \{0,1\}^{k \times k} : \ v \text{ represents a path from } (1,1) \text{ to } (k,k)\}.$$

Training proceeds based on the map images and (possibly) the true shortest paths or cell costs provided in the dataset.

As suggested by Vlastelica et al. (2020), we design the CNN based on the first few layers of a ResNet18 (He et al. 2016). We also append a negative softplus activation, in order to make sure that all outputs are negative. The sign constraint is there to ensure that Dijkstra's algorithm will terminate, but it is not obvious why we require negative costs instead of positive ones. The reason behind this sign switch is that Dijkstra's algorithm is a minimization oracle, whereas by convention `InferOpt.jl` works with maximization oracles.

We now show how to implement this pipeline in Julia. Throughout the paper, in addition to `InferOpt.jl`, the following packages are used: `Flux.jl`[9] (Innes et al. 2018; Innes 2018), `Graphs.jl`[10] (Fairbanks et al. 2021), `GridGraphs.jl`[11], `Metalhead.jl`[12] and `Zygote.jl`[13] (Innes 2019), along with the base libraries `LinearAlgebra` and `Statistics`. Code sample 1 creates the CNN encoder layer. Meanwhile, Code sample 2 shows how to define the Dijkstra oracle and the true cost function.

Finally, Code sample 3 demonstrates the full prediction and optimization pipeline. It assumes that we have already parsed the Warcraft dataset into three vectors:

- `images`, whose elements are three-dimensional arrays representing map images;
- `cells`, whose elements are floating-point matrices representing true cell costs;
- `paths`, whose elements are binary matrices representing true shortest paths.

These functions do not rely on `InferOpt.jl`, but they will be used throughout the paper inside the differentiable wrappers provided by the package. Note that some snippets shown here have been shortened for clarity. Please refer to the documentation of `InferOpt.jl` and satellite packages for actual runnable examples.

[8] https://edmond.mpdl.mpg.de/dataset.xhtml?persistentId=doi:10.17617/3.YJCQ5S
[9] https://github.com/FluxML/Flux.jl
[10] https://github.com/JuliaGraphs/Graphs.jl
[11] https://github.com/gdalle/GridGraphs.jl
[12] https://github.com/FluxML/Metalhead.jl
[13] https://github.com/FluxML/Zygote.jl

```julia
using Flux, Metalhead, Statistics

resnet18 = ResNet(
    18; pretrain=false, nclasses=1
)

warcraft_encoder = Chain(
    resnet18.layers[1][1:4],
    AdaptiveMaxPool((12, 12)),
    x -> mean(x; dims=3),
    x -> dropdims(x; dims=(3, 4)),
    x -> -softplus.(x)
)
```

Code sample 1: CNN encoder for Warcraft

```julia
using Graphs, GridGraphs, LinearAlgebra
using GridGraphs: QUEEN_DIRECTIONS

function warcraft_maximizer(theta)
    g = GridGraph(
        -theta;
        directions=QUEEN_DIRECTIONS
    )
    path = grid_dijkstra(g, 1, nv(g))
    y = path_to_matrix(g, path)
    return y
end

function warcraft_cost(y; theta_ref)
    return dot(y, theta_ref)
end
```

Code sample 2: Dijkstra optimizer for Warcraft

```julia
x, theta_ref, y_ref = images[1], cells[1], paths[1]
theta = warcraft_encoder(x)
y = warcraft_maximizer(theta)
c = warcraft_cost(y; theta_ref=theta_ref)
```

Code sample 3: Full pipeline for Warcraft shortest paths

3 Probabilistic CO layers

In this section, we focus on the CO oracle f defined in Equation (2), which is piecewise constant. By adopting a probabilistic point of view, we construct several smooth approximations $\widehat{f}$, which can be computed and differentiated based solely on calls to f.

3.1 The expectation of a differentiable probability distribution

As announced in Section 1.2.2, a probabilistic CO layer works in two steps. First, it constructs a probability distribution $\widehat{p}(\cdot|\theta) \in \Delta^{\mathcal{V}}$. Second, it returns the expectation $\widehat{f}(\theta) = \mathbb{E}_{\widehat{p}(\cdot|\theta)}[V] \in \mathrm{conv}(\mathcal{V})$. Figure 2 illustrates this behavior on a two-dimensional polytope, with a maximization problem defined by the vector θ (black arrow). While the CO oracle outputs a single optimal vertex (red square), the probabilistic CO layer defines a distribution on all the vertices (light blue circles). Its output (dark blue hexagon) is a convex combination of the vertices with nonzero weights, which belongs to the convex hull of $\mathcal{V}$ (gray surface).

For such a layer to be useful in our setting, we impose several conditions. First, all computations must only require calls to the CO oracle f. Second, the expectation $\mathbb{E}_{\widehat{p}(\cdot|\theta)}[V]$ must be tractable (whether it is with an explicit formula, Monte-Carlo sampling, variational inference, *etc.*). Third, the mapping $\theta \longmapsto \widehat{p}(\cdot|\theta)$ must be differentiable. If the last condition is satisfied, then the Jacobian of $\widehat{f}$ is easily deduced from Equation (3):

$$J_\theta \widehat{f}(\theta) = J_\theta \mathbb{E}_{\widehat{p}(\cdot|\theta)}[V] = \sum_{v \in \mathcal{V}} v \nabla_\theta \widehat{p}(v|\theta)^\top \tag{8}$$

Let us give an example where analytic formulas exist. If $\mathcal{V} = \{\mathrm{e}_1, ..., \mathrm{e}_d\}$ is the set of basis vectors, then its convex hull $\mathrm{conv}(\mathcal{V}) = \Delta^d$ is the unit simplex of dimension d. Given an objective direction θ, solving $\mathrm{argmax}_{v \in \mathcal{V}} \theta^\top v$ yields the basis vector $f(\theta) = \mathrm{e}_i$ where i is the index maximizing θ_i.

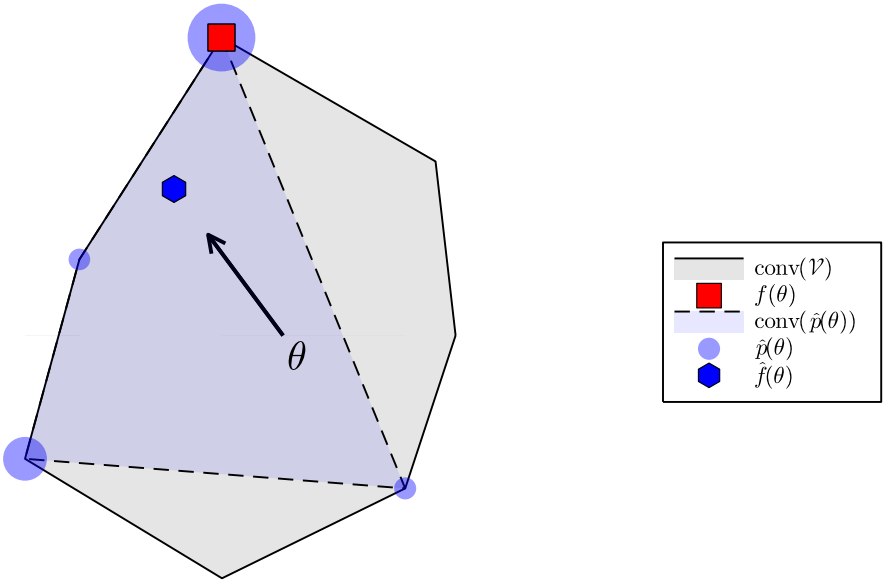


Figure 2: Effect of a probabilistic CO layer

We want a probability distribution that evolves smoothly with θ, so we need to spread out the naive Dirac mass $\delta_{f(\theta)}(\cdot)$ by putting weight on several vertices instead of just one. Let us assign to each vertex a probability that depends on its level of optimality in the optimization problem $\operatorname{argmax}_{v \in \mathcal{V}} \theta^\top v$, that is, on its inner product with θ. The Boltzmann distribution is a natural candidate, leading to $\widehat{p}(\mathrm{e}_i|\theta) \propto e^{\theta^\top \mathrm{e}_i} = e^{\theta_i}$. Computing the expectation reveals a well-known operation:

$$\widehat{f}(\theta) = \mathbb{E}_{\widehat{p}(\cdot|\theta)}[V] = \sum_{i=1}^{d} \frac{e^{\theta_i}}{\sum_{j=1}^{d} e^{\theta_j}} \mathrm{e}_i = \mathrm{softmax}(\theta)$$

Unlike the "hardmax" function f, the softmax function $\widehat{f}$ is differentiable, which justifies its frequent use as an activation function in classification tasks.

While the Boltzmann distribution can be used for specific sets $\mathcal{V}$, in the general case, it has an intractable normalizing constant. This means we would need Markov chain Monte-Carlo (MCMC) methods to compute derivatives, which undermines the simplicity we are looking for. Fortunately, Sections 3.3.1 and 3.3.2 present other probability distributions which can be easily approximated through sampling.

3.2 Regularization as another way to define a distribution

Although this probabilistic point of view was recently put forward by Berthet et al. (2020), the most popular paradigm in the literature remains regularization (Blondel, Martins, and Niculae 2020). Instead of using the CO oracle (2), regularization solves a different problem:

$$\widehat{f}_\Omega : \theta \longmapsto \operatorname*{argmax}_{\mu \in \mathrm{dom}(\Omega)} \theta^\top \mu - \Omega(\mu) \tag{9}$$

where $\Omega : \mathbb{R}^d \to \mathbb{R}$ is a smooth and convex function that penalizes the output μ. Usually, Ω is chosen to enforce $\Omega(\mu) = +\infty$ whenever $\mu \notin \mathrm{conv}(\mathcal{V})$, which means $\mathrm{dom}(\Omega) \subseteq \mathrm{conv}(\mathcal{V})$. Remember that since $\mathcal{V}$ is finite, $\mathrm{conv}(\mathcal{V})$ is a polytope whose vertices form a subset of $\mathcal{V}$.

The change of notation from v (vertex) to μ (moment) stresses the fact that v is an element of $\mathcal{V}$, while μ belongs to $\text{dom}(\Omega) \subseteq \text{conv}(\mathcal{V})$. By Equation (6), any feasible μ is the expectation of some distribution over $\mathcal{V}$, hence our choice of letter. This means we can write $\widehat{f}_\Omega(\theta)$ as a convex combination of the elements of $\mathcal{V}$, whose weights are then interpreted as probabilities $\widehat{p}_\Omega(\cdot|\theta)$. In other words, the two perspectives are not opposed: regularization is just another way to define a probability distribution.

For instance, if we go back to the concrete example from Section 3.1 and select $\Omega(\mu) = \sum_i \mu_i \log \mu_i$ (the negative Shannon entropy), we find once again that $\widehat{f}_\Omega(\theta) = \text{softmax}(\theta)$. But the case of the unit simplex is very peculiar, because for any $\mu \in \text{conv}(\mathcal{V})$, the convex decomposition of μ onto the vertices $\mathcal{V}$ is unique. In other words, the correspondence between regularizations Ω and probability mappings $\widehat{p}_\Omega$ is one-to-one.

This does not hold for arbitrary polytopes. As a result, we need to be more specific in how we choose the convex decomposition. In particular, we need the weights to be differentiable, in order to compute the Jacobian with Equation (8). Section 3.3.3 describes one possible approach, which relies on the Frank-Wolfe algorithm and implicit differentiation.

Conversely, the probability distributions given in Sections 3.3.1 and 3.3.2 also give rise to an implicit regularization, which can be expressed using Fenchel conjugates. This point of view is especially useful when we want to combine these layers with Fenchel-Young losses (Section 5.2.3). In essence, we claim that *probabilistic CO layers and regularization are two sides of the same coin.*

3.3 Collection of probabilistic CO layers

Our package implements various flavors of probabilistic CO layers, which are summed up in Table 1. Code sample 4 displays the operations they all support.

Remark 3.1. *We also implement an* `Interpolation` *layer, corresponding to the piecewise linear interpolation of Vlastelica et al. (2020). However, to the best of our knowledge it cannot be cast as a probabilistic CO layer, so it does not support as many operations, and we only mention it for benchmarking purposes.*

We now present each row of Table 1 in more detail. The main goal of Sections 3.3.1, 3.3.2 and 3.3.3 is to explain how $\widehat{p}(\cdot|\theta)$ and $\widehat{f}(\theta)$ are computed, as well as their derivatives. We also draw connections with the regularization paradigm, which will ease the introduction of Fenchel-Young losses in Section 5.2.3. A hasty reader can safely skip to Section 4.

Layer	Notations	Probability $\widehat{p}(\cdot\|\theta)$	Regularization
`PerturbedAdditive`	$\widehat{p}_\varepsilon^+$, $\widehat{f}_\varepsilon^+$	Explicit: $f(\theta + \varepsilon Z)$	Implicit: Fenchel conjugate
`PerturbedMultiplicative`	$\widehat{p}_\varepsilon^\odot$, $\widehat{f}_\varepsilon^\odot$	Explicit: $f(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2})$	Implicit: Fenchel conjugate
`RegularizedGeneric`	$\widehat{p}_\Omega^{\text{FW}}$, $\widehat{f}_\Omega^{\text{FW}}$	Implicit: Frank-Wolfe weights	Explicit: function Ω

Table 1: Probabilistic CO layers and their defining features

3.3.1 Additive perturbation

A natural way to define a distribution on $\mathcal{V}$ is to solve (2) with a stochastic perturbation of the objective direction θ. Berthet et al. (2020) suggest the following additive perturbation mechanism:

$$\widehat{f}_\varepsilon^+(\theta) = \mathbb{E}\left[\underset{v \in \mathcal{V}}{\text{argmax}} (\theta + \varepsilon Z)^\top v\right] = \mathbb{E}\left[f(\theta + \varepsilon Z)\right] \tag{10}$$

where $\varepsilon > 0$ controls the amplitude of the perturbation, $Z \sim \mathcal{N}(0, I)$ is a standard Gaussian vector and the expectation is taken with respect to Z unless otherwise specified. Choosing ε is a trade-off between

```julia
using InferOpt, Zygote

p = compute_probability_distribution(layer, theta)
rand(p)
compute_expectation(p)

y = layer(theta)  # equal to the expectation of p
Zygote.jacobian(layer, theta)
```

Code sample 4: Supported operations for a probabilistic CO layer

smoothness (large ε) and accuracy of the approximation (small ε). The associated probability distribution on $\mathcal{V}$ can be described explicitly:

$$\widehat{f}_\varepsilon^+(\theta) = \sum_{v \in \mathcal{V}} v \widehat{p}_\varepsilon^+(v|\theta) \qquad \text{with} \qquad \widehat{p}_\varepsilon^+(v|\theta) = \mathbb{P}\left(f(\theta + \varepsilon Z) = v\right). \tag{11}$$

Meanwhile, Proposition 3.1 allows us to compute differentials. Although the expectations cannot be expressed in closed form, they can be estimated using M Monte-Carlo samples $Z_1, ..., Z_M \sim \mathcal{N}(0, I)$. Increasing M yields smoother approximations but makes the complexity grow linearly.

Proposition 3.1 (Differentiating through an additive perturbation (Berthet et al. 2020))**.** *We have:*

$$\nabla_\theta \widehat{p}_\varepsilon^+(v|\theta) = \frac{1}{\varepsilon}\mathbb{E}\left[\mathbb{1}\{f(\theta + \varepsilon Z) = v\} Z\right]$$
$$J_\theta \widehat{f}_\varepsilon^+(\theta) = \frac{1}{\varepsilon}\mathbb{E}\left[f(\theta + \varepsilon Z) Z^\top\right]$$

Proof. See Appendix A.1.1. □

In order to recover the regularization associated with p_ε^+, we leverage convex conjugation. Let F_ε^+ be the function defined by

$$F_\varepsilon^+(\theta) = \mathbb{E}\left[\max_{v \in \mathcal{V}} (\theta + \varepsilon Z)^\top v\right]$$

and let $\Omega_\varepsilon^+ = (F_\varepsilon^+)^*$ denote its Fenchel conjugate.

Proposition 3.2 (Regularization associated with an additive perturbation (Berthet et al. 2020))**.** *The function Ω_ε^+ is convex, it satisfies $\mathrm{dom}(\Omega_\varepsilon^+) \subset \mathrm{conv}(\mathcal{V})$ and*

$$\widehat{f}_\varepsilon^+(\theta) = \operatorname*{argmax}_{\mu \in \mathrm{conv}(\mathcal{V})} \theta^\top \mu - \Omega_\varepsilon^+(\mu) = \widehat{f}_{\Omega_\varepsilon^+}(\theta).$$

Proof. See Appendix A.1.2. □

Code sample 5 shows how this translates into `InferOpt.jl` syntax.

3.3.2 Multiplicative perturbation

Since the Gaussian distribution puts mass on all of $\mathbb{R}^d$, it can happen that some components of $\theta + \varepsilon Z$ switch their sign with respect to θ. This may cause problems whenever the CO oracle for f has sign-dependent behavior. For instance, Dijkstra's algorithm for shortest paths requires all the edges of a graph to have a positive cost. In those cases, we need a sign-preserving kind of perturbation. Changing the distribution of Z to make it positive almost surely is not the right answer because it would bias the pipeline, leading

to $\mathbb{E}[\theta + \varepsilon Z] > \theta$ for the componentwise order. So instead of being additive, the perturbation becomes multiplicative:

$$\widehat{f}_\varepsilon^{\odot}(\theta) = \mathbb{E}\left[\operatorname*{argmax}_{v \in \mathcal{V}} \left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right)^\top v\right] = \mathbb{E}\left[f\left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right)\right] \tag{12}$$

Here, $\odot$ denotes the Hadamard product, and the exponential is taken componentwise. Since $\mathbb{E}[e^{\varepsilon Z}] = e^{\varepsilon^2 \mathbf{1}/2} \neq 1$, we add a correction term in the exponent to remove any bias: $\mathbb{E}[\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}] = \theta$. As before, the associated probability distribution is easy to describe:

$$\widehat{f}_\varepsilon^{\odot}(\theta) = \sum_{v \in \mathcal{V}} v \widehat{p}_\varepsilon^{\odot}(v|\theta) \qquad \text{with} \qquad \widehat{p}_\varepsilon^{\odot}(v|\theta) = \mathbb{P}\left(f\left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right) = v\right). \tag{13}$$

And Proposition 3.3 provides differentiation formulas that are very similar to the additive case.

Proposition 3.3 (Differentiating through a multiplicative perturbation)**.** *We have:*

$$\nabla_\theta \widehat{p}_\varepsilon^{\odot}(v|\theta) = \frac{1}{\varepsilon\theta} \odot \mathbb{E}\left[\mathbb{1}\left\{f\left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right) = v\right\} Z\right]$$
$$J_\theta \widehat{f}_\varepsilon^{\odot}(\theta) = \frac{1}{\varepsilon\theta} \odot \mathbb{E}\left[f\left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right) Z^\top\right]$$

Proof. See Appendix A.2.1. □

As far as regularization is concerned, we need a slight tweak compared to the additive case. Let $F_\varepsilon^{\odot}$ be the function defined by

$$F_\varepsilon^{\odot}(\theta) = \mathbb{E}\left[\max_{v \in \mathcal{V}} \left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right)^\top v\right]$$

and let $\Omega_\varepsilon^{\odot} = (F_\varepsilon^{\odot})^*$ denote its Fenchel conjugate. We define

$$\widehat{f}_\varepsilon^{\odot\text{scaled}}(\theta) = \mathbb{E}\left[e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2} \odot f\left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right)\right]$$

Proposition 3.4 (Regularization associated with a multiplicative perturbation)**.** *The function $\Omega_\varepsilon^{\odot}$ is convex and satisfies*

$$\widehat{f}_\varepsilon^{\odot\text{scaled}}(\theta) = \operatorname*{argmax}_{\mu \in \operatorname{dom}(\Omega_\varepsilon^{\odot})} \theta^\top \mu - \Omega_\varepsilon^{\odot}(\mu) = \widehat{f}_{\Omega_\varepsilon^{\odot}}(\theta).$$

Unlike in the additive case, it is not $\widehat{f}_\varepsilon^{\odot}$ itself that can be viewed as the product of regularization with $\Omega_\varepsilon^{\odot}$, but $\widehat{f}_\varepsilon^{\odot\text{scaled}}$. Furthermore, this time we have $\operatorname{dom}(\Omega_\varepsilon^{\odot}) \not\subseteq \operatorname{conv}(\mathcal{V})$.

Proof. See Appendix A.2.2. □

Code sample 5 shows how this translates into `InferOpt.jl` syntax.

3.3.3 Generic regularization

We now switch our focus to the case of an explicit regularization Ω. Provided the regularization is convex and smooth, approximate computation of $\widehat{f}_\Omega(\theta)$ is made possible by the Frank-Wolfe algorithm (Frank and Wolfe 1956). This algorithm is interesting for two reasons. First, it only requires access to the CO oracle f and the gradient of Ω. Second, its output is expressed as a convex combination of only a few polytope vertices (Jaggi 2013). In other words, the Frank-Wolfe algorithm does not just return a single point $\widehat{f}_\Omega(\theta) \in \operatorname{conv}(\mathcal{V})$: it also defines a sparse probability distribution $\widehat{p}_\Omega^{\text{FW}}(\cdot|\theta)$ over the vertices $\mathcal{V}$ such that

$$\widehat{f}_\Omega(\theta) = \sum_{v \in \mathcal{V}} v \widehat{p}_\Omega^{\text{FW}}(v|\theta).$$

```julia
using InferOpt

perturbed_add = PerturbedAdditive(
    warcraft_maximizer;
    epsilon=0.5, nb_samples=10
)

perturbed_mult = PerturbedMultiplicative(
    warcraft_maximizer;
    epsilon=0.5, nb_samples=10
)
```

Code sample 5: Probabilistic CO layers defined by perturbation

```julia
using InferOpt

regularized = RegularizedGeneric(
    warcraft_maximizer;
    omega=y -> 0.5 * sum(y .^ 2),
    omega_grad=y -> y
)
```

Code sample 6: Probabilistic CO layer defined by regularization

This distribution is called sparse because most of the weights are actually zero. Note that $\widehat{p}_\Omega^{\mathrm{FW}}(\cdot|\theta)$ is not uniquely specified by the regularization Ω, but instead depends on the precise implementation of the Frank-Wolfe algorithm (initialization, step size, convergence criterion, *etc.*). In particular, the number of atoms in the distribution is upper-bounded by the number of Frank-Wolfe iterations.

As pointed out by Blondel, Berthet, et al. (2022, Appendix C), there exists a function $g(p, \theta)$ defined on $\Delta^{\mathcal{V}} \times \mathbb{R}^d$ such that $\widehat{p}_\Omega^{\mathrm{FW}}(\cdot|\theta)$ is a fixed point of its projected gradient operator $p \longmapsto \mathrm{proj}_{\Delta^{\mathcal{V}}}(p - \nabla_p g(p, \theta))$. Since the orthogonal projection onto the simplex $\Delta^{\mathcal{V}}$ is itself differentiable (Martins and Astudillo 2016), we can apply the implicit function theorem to this fixed point equation. Doing so yields gradients $\nabla_\theta \widehat{p}_\Omega(v|\theta)$ that we use to compute a Jacobian for $\widehat{f}_\Omega(\theta)$. Again, by sparsity, this sum only has a few non-zero terms, which makes it tractable:

$$J_\theta \widehat{f}_\Omega(\theta) = \sum_{v \in \mathcal{V}} v \nabla_\theta \widehat{p}_\Omega^{\mathrm{FW}}(v|\theta)^\top. \tag{14}$$

Among all the possible functions Ω, the quadratic penalty $\Omega(\mu) = \frac{1}{2}\|\mu\|^2$ is particularly interesting. It gives rise to the SparseMAP method (Niculae et al. 2018), whose name comes from the sparsity of the Euclidean projection onto a polytope:

$$\widehat{f}_\Omega(\theta) = \operatorname*{argmax}_{\mu \in \mathrm{conv}(\mathcal{V})} \left\{ \theta^\top \mu - \frac{1}{2}\|\mu\|^2 \right\} = \operatorname*{argmin}_{\mu \in \mathrm{conv}(\mathcal{V})} \|\mu - \theta\|^2.$$

This is the one we used for the example of Code sample 6. Our implementation relies on the recent package `FrankWolfe.jl`[14] (Besançon, Carderera, and Pokutta 2022).

Remark 3.2. *Blondel, Martins, and Niculae (2020) also suggest distribution regularization, whereby $\Omega(\mu)$ is defined through a generalized entropy $H(p)$ on $\Delta^{\mathcal{V}}$:*

$$\Omega(\mu) = -\max_{p \in \Delta^{\mathcal{V}}} H(p) \quad s.t. \quad \mathbb{E}_p[V] = \mu.$$

Distribution regularization can only be computed explicitly for certain entropies H (Shannon entropy, Gini index) and certain polytopes $\mathrm{conv}(\mathcal{V})$ (unit simplex, permutahedron, spanning trees, etc.). In each case, a custom combinatorial algorithm is required. Since we aim for a generic approach, we only consider mean regularization, which is defined directly on the expectation μ.

[14] https://github.com/ZIB-IOL/FrankWolfe.jl

3.4 The case of inexact CO oracles

In our discussion so far, an implicit assumption was that the CO oracle f returns an exact solution to Equation (2). For most polynomial problems (as well as some NP-hard problems which are tractable in practice), this is perfectly reasonable. But in some cases, exact solutions are too expensive to compute. Then, our CO oracle may only be able to return an inexact solution, for instance because branch & bound has to be interrupted before the whole tree can be explored. What kind of impact does this have on the precision of the computed Jacobian?

Let us denote by f a hypothetical exact oracle, and by g an inexact oracle.

Proposition 3.5 (Jacobian precision for inexact oracles – perturbed case)**.** *Suppose we use g instead of f with additive (resp. multiplicative) perturbation. Then the error on the Jacobian of the probabilistic CO layer satisfies:*

$$\left\| J_\theta \widehat{g}_\varepsilon^{+}(\theta) - J_\theta \widehat{f}_\varepsilon^{+}(\theta) \right\|^2 \leq \frac{\sqrt{d}}{\varepsilon} \|g - f\|_\infty$$
$$\left\| J_\theta \widehat{g}_\varepsilon^{\odot}(\theta) - J_\theta \widehat{f}_\varepsilon^{\odot}(\theta) \right\|^2 \leq \frac{\sqrt{d}}{\varepsilon \min_i |\theta_i|} \|g - f\|_\infty.$$

While requiring the inexact oracle g to be uniformly close to f is quite restrictive, this result does provide heuristic justification for the use of inexact oracles in practice.

Proof. See Appendix A.3. □

4 Learning by experience

Now that we have seen several ways to construct probabilistic CO layers, we turn to the definition of an appropriate loss function. Let us start with learning by experience, which takes place when we only have access to input samples without target outputs. In that case, Equation (5) simplifies as

$$\min_w \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\Big(f\big(\varphi_w(x^{(i)})\big)\Big). \tag{15}$$

As we will see below, the *regret*, which is the natural choice of loss, does not yield interesting gradients. That is why we propose a family of *smooth regret surrogates* derived from our probabilistic CO layers, and explain how to differentiate them. While similar losses have been hinted at in previous works, to the best of our knowledge, our general point of view is new.

To make notations lighter, we restrict ourselves to a single input x. Furthermore, we write losses as functions of θ instead of w. Indeed, our losses do not just rely on $y = f(\theta)$: they use f as an ingredient internally. In practice, we leave it to AD to exploit the relation $\theta = \varphi_w(x)$ in order to compute gradients with respect to w.

4.1 Minimizing a smooth regret surrogate

When we learn by experience, the problem statement usually includes a cost function $c : \mathcal{V} \to \mathbb{R}$, and we want our pipeline to generate solutions that are as cheap as possible. Internally, this cost function may use parameters that are unknown to us at prediction time: typically, it may assess the quality of our solution using the true objective direction $\bar{\theta}$. It may be useful to think about c as the feedback provided by an outside evaluator, rather than a function we implement ourselves.

The natural loss to minimize is the cost incurred by our prediction pipeline, also called regret:

$$\mathcal{R}(\theta) = c(f(\theta)). \tag{16}$$

```julia
using InferOpt

regret_pert = Pushforward(
    perturbed_add, warcraft_cost
)
regret_reg = Pushforward(
    regularized, warcraft_cost
)
```

Code sample 7: Expected regrets associated with probabilistic CO layers

```julia
using Zygote

R = regret(theta)
Zygote.gradient(regret, theta)
```

Code sample 8: Supported operations for an expected regret

This function relies on the CO oracle f, which is piecewise constant. Our spontaneous impulse would be to replace the CO oracle f with a probabilistic CO layer $\widehat{f}$, thus minimizing $c(\widehat{f}(\theta))$. Unfortunately, the cost function c is not necessarily smooth either. To make matters worse, c may only be defined on vertices $v \in \mathcal{V}$, and not on general convex combinations $\mu \in \mathrm{conv}(\mathcal{V})$.

The solution we propose relies on the *pushforward measure* (also called image measure) of $\widehat{p}(\cdot|\theta)$ with respect to the function c. Recall that a probabilistic CO layer is defined by $\widehat{f}(\theta) = \mathbb{E}_{\widehat{p}(\cdot|\theta)}[V]$. To compose it with an arbitrary cost, instead of applying c outside the expectation, we apply it inside the expectation. In other words, we first push the measure $\widehat{p}(\cdot|\theta)$ forward through the function c, before taking the expectation. This gives rise to the notion of *expected regret*:

$$\mathcal{R}_{\widehat{p}}(\theta) = \mathbb{E}_{\widehat{p}(\cdot|\theta)}[c(V)] \tag{17}$$

By integration, this loss is just as smooth as the probability mapping $\theta \longmapsto \widehat{p}(\cdot|\theta)$, which means we can compute its gradient easily. We therefore suggest using the expected regret $\mathcal{R}_{\widehat{p}}$ that stems from the probabilistic CO layers $\widehat{p}^{+}_{\varepsilon}$, $\widehat{p}^{\odot}_{\varepsilon}$, and $\widehat{p}^{\mathrm{FW}}_{\Omega}$ defined in Section 3.

Note that if c is linear and defined on all of $\mathrm{conv}(\mathcal{V})$, then $\mathbb{E}_{\widehat{p}(\cdot|\theta)}[c(V)] = c\left(\mathbb{E}_{\widehat{p}(\cdot|\theta)}[V]\right)$ and the two quantities coincide. Furthermore, if c is convex, then $\mathbb{E}_{\widehat{p}(\cdot|\theta)}[c(V)] \geq c\left(\mathbb{E}_{\widehat{p}(\cdot|\theta)}[V]\right)$ by Jensen's inequality, which means the expected regret is an upper bound.

Code sample 7 demonstrates how to define an expected regret from probabilistic CO layers, while Code sample 8 shows that we can compute and differentiate it automatically. Finally, Code sample 9 displays a complete program for learning by experience. The rest of this section explains how to compute derivatives of the expected regret $\mathcal{R}_{\widehat{p}}$ and can be skipped without danger.

```julia
using Flux, InferOpt

gradient_optimizer = Adam()
parameters = Flux.params(warcraft_encoder)
data = images

function pipeline_loss(x)
    theta = warcraft_encoder(x)
    return regret(theta)
end

for epoch in 1:1000
    train!(pipeline_loss, parameters, data, gradient_optimizer)
end
```

Code sample 9: Learning with an expected regret

Remark 4.1. *Since the learning problem is non-convex, we may also try to minimize the (non-smooth) regret $\mathcal{R}$ using global optimization algorithms such as DIRECT (Jones, Perttunen, and Stuckman 1993). Perhaps surprisingly, this has been shown to yield good results when φ_w is a generalized linear model and the dimension of the weights w is not too large, i.e., no greater that 100 (Parmentier 2021a). When the ML layer φ_w is a large neural network, we cannot use this approach anymore.*

4.2 Derivatives of the regret for learning by experience

When $\widehat{p}$ comes from a random perturbation, we can formulate the gradient of the expected regret as an expectation too, and approximate it with Monte-Carlo samples.

Proposition 4.1 (Gradient of the expected regret, perturbation setting)**.** *We have:*

$$\nabla_\theta \mathcal{R}_{\widehat{p}_\varepsilon^+}(\theta) = \frac{1}{\varepsilon}\mathbb{E}\left[(c \circ f)(\theta + \varepsilon Z)Z\right]$$
$$\nabla_\theta \mathcal{R}_{\widehat{p}_\varepsilon^\odot}(\theta) = \frac{1}{\varepsilon\theta} \odot \mathbb{E}\left[(c \circ f)\left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right) Z\right].$$

Proof. For the additive perturbation, it is a consequence of Proposition 3.1. For the multiplicative perturbation, it is a consequence of Proposition 3.3. □

These gradients obey a simple logic: the more the perturbation Z increases the cost of a solution, the more positive weight it gets, and vice versa. To see it, we remember that $\mathbb{E}[Z] = 0$ for a standard Gaussian, and rewrite the regret gradients as follows:

$$\nabla_\theta \mathcal{R}_{\widehat{p}_\varepsilon^+}(\theta) = \frac{1}{\varepsilon}\mathbb{E}\left[(c \circ f)(\theta + \varepsilon Z)Z - (c \circ f)(\theta)Z\right]$$
$$\nabla_\theta \mathcal{R}_{\widehat{p}_\varepsilon^\odot}(\theta) = \frac{1}{\varepsilon\theta} \odot \mathbb{E}\left[(c \circ f)\left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right) Z - (c \circ f)(\theta)Z\right].$$

On the other hand, when $\widehat{p}$ is derived from an explicit regularization Ω, the expected regret is amenable to implicit differentiation of the Frank-Wolfe algorithm. Once more, the sparsity property makes exact computation tractable by reducing the number of terms in the sum:

$$\nabla_\theta \mathcal{R}_{\widehat{p}_\Omega^{\mathrm{FW}}}(\theta) = \sum_{v \in \mathcal{V}} c(v) \nabla_\theta \widehat{p}_\Omega^{\mathrm{FW}}(v|\theta)$$

Remark 4.2. *Although the previous discussion focuses on a scalar-valued cost, it actually applies to any pushforward function c, even with vector values. The formulas for the generic Jacobian are given below:*

$$J_\theta \mathbb{E}_{\widehat{p}_\varepsilon^+(\cdot|\theta)}[c(V)] = \frac{1}{\varepsilon}\mathbb{E}\left[(c \circ f)(\theta + \varepsilon Z)Z^\top\right]$$
$$J_\theta \mathbb{E}_{\widehat{p}_\varepsilon^\odot(\cdot|\theta)}[c(V)] = \frac{1}{\varepsilon\theta} \odot \mathbb{E}\left[(c \circ f)\left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right) Z^\top\right]$$
$$J_\theta \mathbb{E}_{\widehat{p}_\Omega^{\mathrm{FW}}(\cdot|\theta)}[c(V)] = \sum_{v \in \mathcal{V}} c(v) \nabla_\theta \widehat{p}_\Omega^{\mathrm{FW}}(v|\theta)^\top$$

At the moment, `InferOpt.jl` only handles the case where c is a fully-defined function without free parameters. In the near future, we will add support for the case where c is itself an ML layer with learnable weights.

5 Learning by imitation

We now move on to learning by imitation, where additional information is used to guide the training procedure. For each input sample $x^{(i)}$, we assume we have access to a *target* $\bar{t}^{(i)}$. In that case, Equation (5)

simplifies as

$$\min_w \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\Big(f\big(\varphi_w(x^{(i)})\big), \bar{t}^{(i)}\Big), \tag{18}$$

and we can see that the loss takes the target as an additional argument. In this section, we introduce imitation losses that are well-suited to hybrid ML-CO pipelines, and explain how to compute their gradients. As in Section 4, we only consider a single input x, and we write losses as $\mathcal{L}(\theta, \bar{t})$.

5.1 A loss that takes the optimization layer into account

There are two main kinds of target. The first one is a good quality solution $\bar{t} = \bar{y}$. The second one is the true objective direction $\bar{\theta}$, from which we can also deduce $\bar{y} = f(\bar{\theta})$, so that $\bar{t} = (\bar{\theta}, \bar{y})$. When learning by imitation, it is tempting to focus only on reproducing the targets, but this would be misguided. To explain why, we revisit the pipeline of Equation (4).

Remember that we may have access to the true objective direction $\bar{\theta}$ during training, but at prediction time, the CO oracle f is applied to the encoder output $\theta = \varphi_w(x)$ instead. Minimizing a naive square loss like $\|\varphi_w(x) - \bar{\theta}\|^2$ completely neglects the asymmetric impacts of the prediction errors on θ: for example, overestimating or underestimating θ may have very different consequences on the quality of the downstream solution. That is why, according to Elmachtoub and Grigas (2022), we need a loss function that takes the optimization step into account. The same holds true when we have access to a precomputed solution $\bar{y}$. Berthet et al. (2020) present experiments showing that the naive square loss $\|\widehat{f}(\varphi_w(x)) - \bar{y}\|^2$ performs poorly compared with more refined approaches. Our own numerical findings (Section 6) support their conclusion.

To sum up, we want a loss that does not neglect the optimization step. Let y temporarily denote the output of our pipeline. When surveying the literature, we realized that most flavors of imitation learning use losses that combine the same components:

$$\mathcal{L}^{\text{aux}}_{\ell,\Omega}(\theta, \bar{t}, y) = \underbrace{\ell(y, \bar{t})}_{\text{base loss}} + \underbrace{\theta^\top (y - \bar{y})}_{\substack{\text{gap between} \\ y \text{ and } \bar{y} \text{ for the} \\ \text{CO problem (2)}}} - \underbrace{\big(\Omega(y) - \Omega(\bar{y})\big)}_{\text{regularization term}} \tag{19}$$

Here is another way to write it:

$$\mathcal{L}^{\text{aux}}_{\ell,\Omega}(\theta, \bar{t}, y) = \underbrace{\ell(y, \bar{t})}_{\text{base loss}} + \underbrace{\big(\theta^\top y - \Omega(y)\big) - \big(\theta^\top \bar{y} - \Omega(\bar{y})\big)}_{\substack{\text{gap between } y \text{ and } \bar{y} \\ \text{for the regularized CO problem (9)}}}$$

The base loss $\ell(y, \bar{t})$ is similar in spirit to the cost function $c(y)$ from Section 4. But it is the gap term that truly makes it possible for the optimization problem to play a role in the loss. Indeed, minimizing the gap encourages the (regularized) CO problem to output a solution y that is close to the target $\bar{y}$.

Putting these components together yields a linear function of θ, and we can remove the dependency in y by maximizing over y:

$$\mathcal{L}^{\text{gen}}_{\ell,\Omega}(\theta, \bar{t}) = \max_{y \in \text{dom}(\Omega)} \mathcal{L}^{\text{aux}}_{\ell,\Omega}(\theta, \bar{t}, y) = \max_{y \in \text{dom}(\Omega)} \big[\ell(y, \bar{t}) + \theta^\top (y - \bar{y}) - (\Omega(y) - \Omega(\bar{y}))\big]. \tag{20}$$

The following result justifies why this is an interesting loss.

Proposition 5.1 (Properties of the generic loss for learning by imitation)**.** *The function $\mathcal{L}^{gen}_{\ell,\Omega}(\theta, \bar{t})$ is convex with respect to θ, and a subgradient is given by*

$$\Big(\operatorname*{argmax}_{y \in \text{dom}(\Omega)} \mathcal{L}^{aux}_{\ell,\Omega}(\theta, \bar{t}, y)\Big) - \bar{y} \quad \in \quad \partial_\theta \mathcal{L}^{gen}_{\ell,\Omega}(\theta, \bar{t}). \tag{21}$$

Proof. As a pointwise maximum of affine functions, $\theta \longmapsto \mathcal{L}^{\text{gen}}_{\ell,\Omega}(\theta, \bar{t})$ is convex. Its subgradient is obtained using Danskin's theorem (Danskin 1967). □

Method	Notation	Target	Base loss	Regul.	Loss formula
S-SVM	$\mathcal{L}_{\ell}^{\text{S-SVM}}$	$\bar{y}$	$\ell(y, \bar{y})$	No	$\max_{y} \ell(y, \bar{y}) + \theta^{\top}(y - \bar{y})$
SPO+	$\mathcal{L}^{\text{SPO+}}$	$(\bar{\theta}, \bar{y})$	$\bar{\theta}^{\top}(\bar{y} - y)$	No	$\max_{y} \bar{\theta}^{\top}(\bar{y} - y) + 2\theta^{\top}(y - \bar{y})$
FY	$\mathcal{L}_{\Omega}^{\text{FY}}$	$\bar{y}$	0	Yes	$\max_{y} \theta^{\top}(y - \bar{y}) - (\Omega(y) - \Omega(\bar{y}))$
Generic	$\mathcal{L}_{\ell,\Omega}^{\text{gen}}$	$\bar{t}$	$\ell(y, \bar{t})$	Yes	$\max_{y} \ell(y, \bar{t}) + \theta^{\top}(y - \bar{y}) - (\Omega(y) - \Omega(\bar{y}))$

Table 2: A common decomposition for loss functions in imitation learning

```julia
using InferOpt

fyl_pert = FenchelYoungLoss(perturbed_add)
fyl_reg = FenchelYoungLoss(regularized)
spol = SPOPlusLoss(warcraft_maximizer)
```

Code sample 10: Example imitation losses

```julia
using Zygote

L = loss(theta, y_ref)
Zygote.gradient(loss, theta, y_ref)
```

Code sample 11: Supported operations for an imitation loss

The idea is that solving $\text{argmax}_{y \in \text{dom}(\Omega)} \mathcal{L}_{\ell,\Omega}^{\text{aux}}(\theta, \bar{t}, y)$ should not be much harder than the regularized CO problem (9). Therefore, using such a loss function dispenses us from differentiating through the probabilistic CO layer: most of the time, we only need to compute the layer output in order to obtain a loss subgradient for free.

5.2 Collection of losses for learning by imitation

Several prominent loss functions from the literature are special cases of our decomposition (20): we gather them in Table 2. Code sample 10 clarifies their construction, while Code sample 11 displays supported operations. Finally, the entire program necessary for learning by imitation is shown on Code sample 12.

In Sections 5.2.1, 5.2.2, 5.2.3 and 5.2.4, we go over these special cases to explain how to compute each loss and its subgradient using Equation (21). They can be skipped without danger.

Remark 5.1. *While the S-SVM and SPO+ losses do not fall within the framework of probabilistic CO layers (due to the absence of regularization), we still include them for benchmarking purposes.*

```julia
using Flux, InferOpt

gradient_optimizer = ADAM()
parameters = Flux.params(warcraft_encoder)
data = zip(images, paths)

function pipeline_loss(x, y)
    theta = warcraft_encoder(x)
    return loss(theta, y)
end

for epoch in 1:1000
    train!(pipeline_loss, parameters, data, gradient_optimizer)
end
```

Code sample 12: Learning with an imitation loss

5.2.1 Structured support vector machines

The structured support vector machine (S-SVM) was among the first methods introduced for learning in structured spaces (Nowozin and Lampert 2010, Chapter 6). Given a target solution $\bar{y}$ and an underlying distance function $\ell(y, \bar{y})$ on $\mathcal{V}$, the S-SVM loss is computed as follows:

$$\mathcal{L}_\ell^{\text{S-SVM}}(\theta, \bar{y}) = \max_{y \in \mathcal{V}} \{\ell(y, \bar{y}) + \theta^\top (y - \bar{y})\}. \tag{22}$$

The subgradient formula (21) becomes

$$\operatorname*{argmax}_{y \in \mathcal{V}} \{\ell(y, \bar{y}) + \theta^\top (y - \bar{y})\} - \bar{y} \quad \in \quad \partial_\theta \mathcal{L}_\ell^{\text{S-SVM}}.$$

Note that due to the presence of ℓ, computing a subgradient requires an auxiliary solver that is different from the linear oracle f. This is why we do not illustrate the S-SVM with a code sample. In `InferOpt.jl`, we only implement this auxiliary solver for the unit simplex, in the case where ℓ is the Hamming distance. However, we also provide a generic layer where the user can plug in the relevant auxiliary solver.

5.2.2 Smart "predict, then optimize"

The smart "predict, then optimize" (SPO) paradigm is applicable when the true objective direction $\bar{\theta}$ is known (remember that in this case, we have $\bar{y} = f(\bar{\theta})$). Elmachtoub and Grigas (2022) define the SPO+ loss function as follows:

$$\begin{aligned} \mathcal{L}^{\text{SPO}^+}(\theta, \bar{\theta}) &= (2\theta - \bar{\theta})^\top f(2\theta - \bar{\theta}) + (\bar{\theta} - 2\theta)^\top \bar{y} \\ &= \max_{y \in \mathcal{V}} \left\{ \bar{\theta}^\top (\bar{y} - y) + 2\theta^\top (y - \bar{y}) \right\}. \end{aligned} \tag{23}$$

It can be seen as a special case of S-SVM. But this time, computing the loss and its subgradient only requires calling f twice:

$$2f(2\theta - \bar{\theta}) - 2\bar{y} \quad \in \quad \partial_\theta \mathcal{L}^{\text{SPO}^+}(\theta, \bar{\theta}).$$

5.2.3 Fenchel-Young losses

The framework of Fenchel-Young losses is built on the theory of convex conjugates, in particular the Fenchel-Young inequality (Blondel, Martins, and Niculae 2020). Starting from a target solution $\bar{y}$ and a regularization Ω, a loss is constructed as follows:

$$\begin{aligned} \mathcal{L}_\Omega^{\text{FY}}(\theta, \bar{y}) &= \Omega^*(\theta) + \Omega(\bar{y}) - \theta^\top \bar{y} \\ &= \max_{y \in \text{conv}(\mathcal{V})} \left(\theta^\top y - \Omega(y)\right) - \left(\theta^\top \bar{y} - \Omega(\bar{y})\right) \end{aligned} \tag{24}$$

This time, the loss and subgradient require access to $\widehat{f}_\Omega$:

$$\widehat{f}_\Omega(\theta) - \bar{y} \quad \in \quad \partial_\theta \mathcal{L}_\Omega^{\text{FY}}(\theta, \bar{y}).$$

As can be inferred from the expression above, there are deep connections between Fenchel-Young losses and the regularization paradigm of Section 3.2. In particular, it is also possible to use implicit regularization by perturbation (Berthet et al. 2020). The fact that we cannot compute $\Omega_\varepsilon^+(y)$ or $\Omega_\varepsilon^\odot(y)$ is not a real obstacle: since those terms do not depend on θ, we can just drop them from the loss during training. We end up with the following estimators for the loss and its subgradient:

$$\begin{aligned} \mathcal{L}_{\Omega_\varepsilon^+}^{\text{FY}}(\theta, \bar{y}) &= F_\varepsilon^+(\theta) - \theta^\top \bar{y} & \widehat{f}_\varepsilon^+(\theta) - \bar{y} &\in \partial_\theta \mathcal{L}_{\Omega_\varepsilon^+}^{\text{FY}}(\theta, \bar{y}) \\ \mathcal{L}_{\Omega_\varepsilon^\odot}^{\text{FY}}(\theta, \bar{y}) &= F_\varepsilon^\odot(\theta) - \theta^\top \bar{y} & \widehat{f}_\varepsilon^{\odot\text{scaled}}(\theta) - \bar{y} &\in \partial_\theta \mathcal{L}_{\Omega_\varepsilon^\odot}^{\text{FY}}(\theta, \bar{y}) \end{aligned}$$

5.2.4 Generic imitation loss

Of course, it is tempting to fill in the blanks of Table 2 by combining every single term of the loss decomposition (20). To the best of our knowledge, this has not been done before in the literature, but there is no theoretical obstacle.

If we use this generic loss together with regularization, then it is interesting to remark that $\mathcal{L}^{\text{gen}}_{\ell,\Omega}(\theta, \bar{t})$ acts as a convex upper bound on the base loss $\ell(\widehat{f}_\Omega(\theta), \bar{t})$. Indeed, since $\bar{y}$ is a worse solution than $\widehat{f}_\Omega(\theta)$ for (9), we have

$$\begin{aligned}\ell(\widehat{f}_\Omega(\theta), \bar{t}) &\leq \ell(\widehat{f}_\Omega(\theta), \bar{t}) + \left[\theta^\top \widehat{f}_\Omega(\theta) - \Omega(\widehat{f}_\Omega(\theta))\right] - \left[\theta^\top \bar{y} - \Omega(\bar{y})\right] \\ &\leq \max_{y \in \text{conv}(\mathcal{V})} \left(\ell(y, \bar{t}) + \theta^\top (y - \bar{y}) - (\Omega(y) - \Omega(\bar{y}))\right) = \mathcal{L}^{\text{gen}}_{\ell,\Omega}(\theta, \bar{t}).\end{aligned}$$

Therefore, our generic loss can be seen as a crossover between the Fenchel-Young loss and a problem-specific base loss. It is not yet implemented in `InferOpt.jl`, and we leave its thorough testing for future work.

6 Applications

In this section, we define specific ML-CO pipelines, in both learning settings (by experience and by imitation), to address four problems: our shortest path problem on Warcraft maps, a stochastic vehicle scheduling problem, a single-machine scheduling problem, and a two-stage stochastic minimum weight spanning tree problem. The first one stems from the ML community. The other three are classics from the field of operations research. They illustrate the idea of approximating hard optimization problems with simpler ones using ML-CO pipelines.

6.1 Shortest paths on Warcraft maps

First we come back to our guiding example of Section 2.3. Our aim is to illustrate the various learning settings introduced in this paper, and to evaluate their relative performance. We do so with two kinds of shortest path (SP) oracles. The first one uses Dijkstra's algorithm. The second one uses the Ford-Bellman algorithm with a bounded number of iterations.

6.1.1 Experimental setting

In every experiment presented here, we only consider a sub-dataset, made up of 1% of the original Warcraft dataset from Vlastelica et al. (2020). It contains 200 samples, or maps, which we split into 80 training samples, 100 validation samples (for hyperparameter tuning) and 20 test samples (for performance evaluation). For each learning setting, the train, test and validation sets remain the same, and we individually tune a subset of the hyperparameters stated in Table 3. Our motivation for reducing the dataset is to show that we can still obtain convincing results with a limited amount of computation.

We use the `Metalhead.jl` package to build a truncated ResNet18 CNN, `Flux.jl` to train our pipelines with the Adam optimizer (Kingma and Ba 2015), and `GridGraphs.jl` to compute shortest paths. Our code is available in the `WarcraftShortestPaths.jl`[15] repository. The experiments are conducted on a MacBook Pro with 2,3 GHz Intel Core i9, 8 cores and 16 Go 2667 MHz DDR4 RAM.

To obtain a precise learning setting, we need to define:

1. The combinatorial problem we need to solve, along with an appropriate oracle.
2. The probabilistic CO layer used to wrap said oracle.
3. The data we have at our disposal to train our pipeline.

[15] https://github.com/LouisBouvier/WarcraftShortestPaths.jl

Hyperparameter	Description
`epsilon`	Scale of the noise for perturbation.
`nb_samples`	Number of noise samples M for perturbation.
`batch_size`	Size of the batches to compute gradients.
`lr_start`	Starting learning rate.

Table 3: Hyperparameters for learning Warcraft shortest paths.

4. The loss function we want to minimize.

No matter the setting, the data always contains a list of RGB map images. When we learn by experience, we also have access to a black box cost function, which evaluates paths based on the true cell costs (see the beginning of Section 4). On the other hand, when we learn by imitation, we add targets to the maps (as defined in Section 5). The target in our case always includes the optimal path, with or without the true cell costs.

All those ingredients are detailed in Table 4 for each learning setting we consider. The names given in the first column are reused in the legends of Figure 3. Most of the probabilistic CO layers considered in this paper do not prevent the objective vector θ from changing its sign, and the same goes for the losses As a result, we need oracles able to accommodate negative cell costs. That is why we use the Ford-Bellman algorithm, while limiting the number of iterations to the number of nodes in the grid graph (to ensure termination even with negative cycles). Our multiplicative perturbation is the only approach that preserves non-negative costs. It enables us to apply Dijkstra's algorithm, which has a smaller time and space complexity.

Seeting name	**CO problem (CO oracle)**	**Probabilistic CO layer**	**Exp./Imit. Target**	**Loss**
Cost perturbed multiplicative noise	SP with non-negative costs (Dijkstra)	Multiplicative perturbation	Experience No target	Perturbed cost
Cost perturbed additive noise	SP on an extended acyclic graph (Ford-Bellman)	Additive perturbation	Experience No target	Perturbed cost
Cost regularized half square norm	SP on an extended acyclic graph (Ford-Bellman)	Half square norm	Experience No target	Regularized cost
SPO+	SP on an extended acyclic graph (Ford-Bellman)	No regularization	Imitation Cost and path	SPO+ loss
MSE perturbed multiplicative noise	SP with non-negative costs (Dijkstra)	Multiplicative perturbation	Imitation Path	Mean squared error
MSE regularized half square norm	SP on an extended acyclic graph (Ford-Bellman)	Half square norm	Imitation Path	Mean squared error
Fenchel-Young perturbed multiplicative noise	SP with non-negative costs (Dijkstra)	Multiplicative perturbation	Imitation Path	Fenchel-Young
Fenchel-Young perturbed additive noise	SP on an extended acyclic graph (Ford-Bellman)	Additive perturbation	Imitation Path	Fenchel-Young
Fenchel-Young regularized half square norm	SP on an extended acyclic graph (Ford-Bellman)	Half square norm	Imitation Path	Fenchel-Young

Table 4: Learning settings for Warcraft shortest paths.

6.1.2 Results

In Figure 3, we show the average train (Figure 3a) and test (Figure 3b) optimality gaps, computed using the true cell costs. We compare all the settings detailed in Table 4, with the exception of MSE base loss + additive noise (we could not get satisfactory results using only 80 training samples). To quantify training effort, instead of counting epochs (*i.e.* passes through the dataset), we use the number of optimizer calls, because these calls are the truly time-consuming part. This aims at comparing learning settings which involve different amounts of computation per gradient step. For instance, using SPO+, we need 2 optimizer

calls to compute the loss gradient for one sample. On the other hand, we need M optimizer calls if we choose Fenchel-Young perturbed additive noise.

When learning by imitation, SPO+ reaches almost zero average gap both on train and test sets after very few optimizer calls, even though we only kept 1% of the initial dataset. This impressive result can be understood since, with SPO+, we have access to the true cell costs during training, and we leverage the problem structure within the loss.

Assuming we only have access to target paths, we obtain better results with Fenchel-Young losses than with MSE losses. The train and test average gaps are lower than 5% with the former, and we observe good generalization performance. This may be explained by the use of the optimization problem in the Fenchel-Young loss definition. On the contrary, in the MSE setting, although we have access to target paths, we only seek to imitate them without truly accounting for solution cost.

Perhaps surprisingly, we also manage to learn by experience with our small sub-dataset. Indeed, using the techniques introduced in Section 4, we reach 7% average test gaps in the cost perturbed multiplicative noise setting, which is better than learning by imitation with an MSE loss. To the best of our knowledge, it is the first time that learning by experience (as defined in Section 4) is combined with CNNs.

6.2 Approximating hard optimization problems

Solving hard optimization problems is one of the main applications of hybrid CO-ML pipelines. The principle is to build a pipeline which approximates a hard combinatorial problem, for which we do not have an efficient algorithm. The approximation relies on a similar but easier problem, for which an efficient algorithm exists. This "easy" algorithm is then used as a CO layer. In order to obtain the best possible approximation (in terms of closeness to the original problem), this CO layer is complemented with encoding and decoding layers, whose weights can be learned.

In the remaining of this section, we use `InferOpt.jl` in this framework, and apply it to three different problems. In Section 6.3, we focus on the *stochastic vehicle scheduling problem* with similar experiments as in Parmentier (2021a) for the learning by imitation, and Parmentier (2021b) for the learning by experience. Then, in Section 6.4, we study the *single machine scheduling problem* and compare our results to those in Parmentier and T'Kindt (2021). Finally, in Section 6.5, we look at the *two-stage stochastic minimum weight spanning tree problem*, and demonstrate the use of a GNN in combination with `InferOpt.jl`. Source code for these three applications can be found in their respective satellite packages.

6.3 Stochastic vehicle scheduling problem

We use `InferOpt.jl` to solve the *stochastic vehicle scheduling problem* (StoVSP), by learning a transformation that approximates its instances as instances of the easier to solve *vehicle scheduling problem* (VSP). Source code can be found in the satellite package `StochasticVehicleScheduling.jl`[16].

6.3.1 Problem formulation

Vehicle scheduling problem The (deterministic) VSP consists in assigning vehicles to cover a set of scheduled tasks, while minimizing the total cost. Let $\bar{V}$ be the set of tasks. Each task $v \in \bar{V}$ has a scheduled beginning time t_v^b, and a scheduled end time t_v^e, such that $t_v^e > t_v^b$. We denote $t_{(u,v)}^{tr}$ the travel time from task u to task v. A task v can be scheduled after another task u only if we can reach it in time, before it starts:

$$t_v^b \geq t_u^e + t_{(u,v)}^{tr} \tag{25}$$

An instance of VSP can be modeled with a directed graph $D = (V, A)$, with $V = \bar{V} \cup \{o, d\}$, and o, d origin and destination dummy nodes. For all task $v \in V$, (o, v) and (v, d) are arcs in A. Additionally, there is an arc between two tasks u and v only if (25) is satisfied. The resulting graph D is acyclic.

[16] https://github.com/BatyLeo/StochasticVehicleScheduling.jl

A solution of the VSP problem is a set of $o-d$ paths partitioning D, such that all tasks are covered exactly once. The objective is to minimize the sum of path edge costs. This can be formulated as an LP (see Appendix B.1.1), and can be solved either using a flow algorithm, or using a general purpose LP solver.

Stochastic Vehicle Scheduling In the stochastic VSP, we consider the same setting as the deterministic version, to which we add the following. Once the scheduling decision is set, we observe random delays, which propagate along vehicle paths. The objective is to minimize the sum of vehicle costs and expected total delay costs.

We consider a finite set of scenarios $s \in S$. For each task $v \in \bar{V}$, we denote γ_v^s the intrinsic delay of v in scenario s, and d_v^s its total delay. We also denote $\delta_{u,v}^s$ the slack between tasks u and v. These quantities follow the delay propagation equation when u and v are consecutively operated by the same vehicle:

$$d_v^s = \gamma_v^s + \max(d_u^s - \delta_{u,v}^s, 0) \tag{26}$$

This leads to a much more difficult problem to solve. In Appendix B.1.2, we provide a compact MILP formulation, which enables to easily solve optimally instances with up to 25 tasks using commercial MILP solvers.

6.3.2 Datasets

To generate our instance datasets, we use a generator similar to the one used by Parmentier (2021b). More details are given in Appendix B.1.3. We consider 3 training/validation datasets used to train our models. Each dataset contains 100 instances, and is divided into 50 training instances and 50 validation instances. The first dataset contains only instances with 25 tasks and 10 scenarios, the second one only instances with 50 tasks and 50 scenarios, and the last one only instances with 100 tasks and 50 scenarios. The 25 tasks dataset contains small instances for which we can easily compute optimal solutions to use as target solutions during the learning. For the two other datasets, we instead used a local search heuristic to compute "good" (non necessarily optimal) solutions of each instance. For testing purposes, we consider several additional test datasets we only use for evaluating the final performances and generalization abilities of our learned models. These test datasets contain larger instances with up to 1000 tasks. Datasets content is summarized in Table 5.

Dataset type	**Number of tasks**							
	25	**50**	**100**	**200**	**300**	**500**	**750**	**1000**
Training	50	50	50	0	0	0	0	0
Validation	50	50	50	0	0	0	0	0
Test	50	50	50	50	50	50	50	50

Table 5: Summary of the number of instances in each used dataset

6.3.3 InferOpt pipeline

We use the deterministic VSP as a CO layer in the following pipeline to solve pipeline to solve the stochastic VSP. The goal is to train a model able to generalize well enough to instances it has not seen before, especially bigger instances, for which we cannot compute a good solution in reasonable time. For this, we consider the pipeline presented in Figure 4.

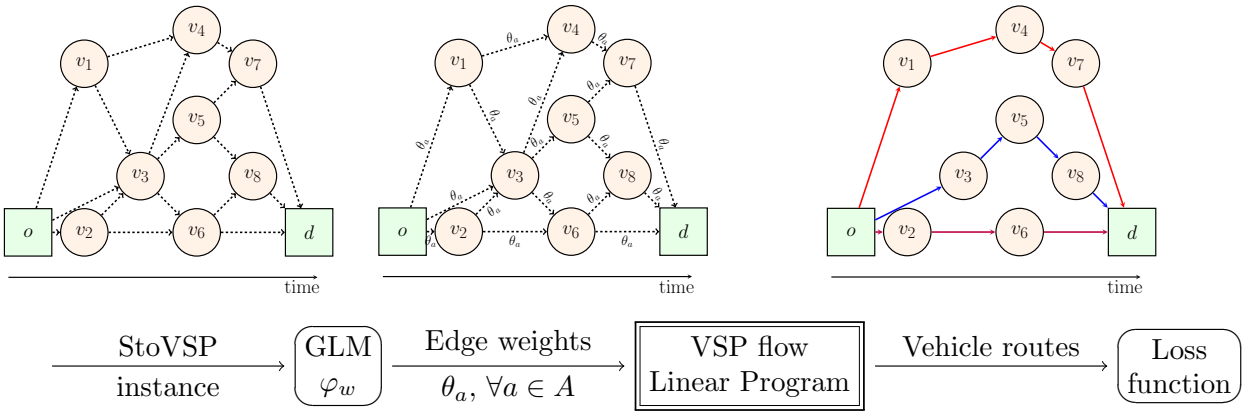


Figure 4: Pipeline for the stochastic VSP

An instance x of StoVSP is encoded by computing a vector $\phi(x, a)$ of 20 features on each arc $a \in A$. These features contain edge length, slack quantiles and cumulative distribution information. For more details, refer to Appendix B.1.3. Instance x is then given as input to a GLM φ_w with learnable weights w, such that $\varphi_w(x) = \theta$, with $\theta_a = w^\top \phi(x, a)$. Then, we use the predicted θ as the objective arc weights of the VSP flow linear program to compute the solution paths:

$$
\begin{aligned}
\max_{y} \quad & \sum_{a \in A} \theta_a y_a \\
\text{s.t.} \quad & \sum_{a \in \delta^-(v)} y_a = \sum_{a \in \delta^+(v)} y_a, && \forall v \in \bar{V} \quad \text{(flow polytope)} \\
& \sum_{a \in \delta^-(v)} y_a = 1, && \forall v \in \bar{V} \quad \text{(task covering)} \\
& y_a \geq 0, && \forall a \in A
\end{aligned}
$$

Training this pipeline means finding parameter vector w that minimizes the chosen loss.

6.3.4 Experimental setting

For each of the three datasets considered, we train two models: we learn the first one by imitation thanks to a Fenchel-Young loss, and the second one by experience thanks to the expected regret. In both cases, we use a `PerturbedAdditive` regularization with parameters ε and `nb_samples`. We use the `Flux.jl` Julia library and the Adam optimizer for training. Hyperparameters used for each model/dataset pair can be found in Table 6.

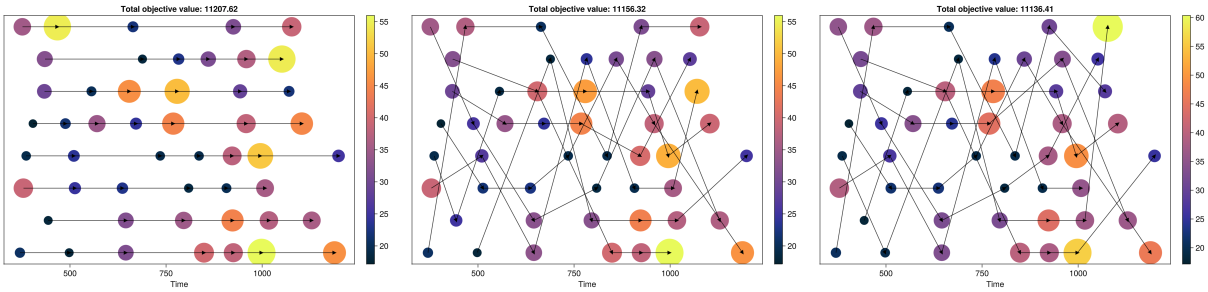
Hyperparameters	**Learn by imitation models**			**Learn by experience models**		
	25 tasks	50 tasks	100 tasks	25 tasks	50 tasks	100 tasks
ε	0.1	0.1	0.1	50	100	300
`nb_samples`	20	20	20	20	20	20
training epochs	50	50	50	200	200	200
batch size	1	1	1	1	1	1

Table 6: Hyperparameter for Stochastic Vehicle Scheduling experiments

Datasets features are normalized during training by dividing each feature by its standard deviation in the train dataset. At the end of training, the obtained w is renormalized accordingly. We optimize hyperparameters using metric values on validation datasets. The experiments are conducted on a 2.6 GHz Intel Core i9-11950H, 16 cores, with 64 Go RAM. We faced one practical difficulty during the training: the calibration of hyperparameter ε when learning by experience. Indeed, training performance is quite sensible to its value, which lead to a lot of hyperparameter tuning on the validation set. Detailed training plots can be found in Appendix B.1.4.

6.3.5 Results

Finally, we can look at the performance of the learned pipelines on all the different datasets, and compare them with each others. In Figure 5, we observe the resulting solution of three different algorithms on the same input instance. Interestingly, the model learned by imitation outputs a different solution than the model learned by experience.



(a) Local search heuristic (b) Model learned by imitation (c) Model learned by experience

Figure 5: Comparison of predictions of three different models on the same 50-task instance. Each circle represents a task, with color value and size corresponding to its total delay (intrinsic + propagated) in the predicted solution. Tasks are ordered respect to start time along the x axis. Each arrow path represents a vehicle path in the predicted solution.

In Table 7, we compare the average execution time between the local search heuristic and one of our learned pipelines. We see that, once the model is learned offline, online prediction of a solution through the pipeline is very fast.

Test dataset					
25 tasks		50 tasks		100 tasks	
heuristic	learned pipeline	heuristic	learned pipeline	heuristic	learned pipeline
$0.12s$	$0.004s$	$1.47s$	$0.01s$	$5.28s$	$0.043s$

Table 7: Learning by imitation: comparison of average execution time on an instance between the local search heuristic (10000 iterations) a learned pipeline

Learning by imitation We train three models in the learning by imitation setting, one for each training dataset. Final results on test data are summarized in Tables 8 and 9. First, we observe the average and maximum cost gaps on test datasets with instances of the same size as in training datasets, between predicted solutions and labeled solutions. We recall that labeled solutions are optimal for 25 tasks, and heuristic (obtained from local search) for 50 and 100 tasks. That is why gaps should stay positive when predicting on the 25 tasks test dataset, but can be negative when predicting on the two other datasets.

We can see that all three models obtain very good gap values, with less than 1% on 25 tasks instances on average, and average negative gaps on 50 and 100 tasks instances, which means better solutions on average than the local search heuristic used as labels for the learning.

Train dataset	**Test dataset**					
	25 tasks		50 tasks		100 tasks	
	mean	max	mean	max	mean	max
25 tasks	0.68%	9.46%	−0.41%	4.26%	−1.02%	2.4%
50 tasks	0.49%	3.01%	−0.46%	2.34%	−1.6%	0.62%
100 tasks	0.62%	3.36%	−0.14%	9.9%	−1.2%	0.11%

Table 8: Learning by imitation: cost gap

Then, we measure the performance of our models on test datasets with larger instances than training ones. We do not necessarily have label solutions available for these big instances, but can evaluate the *average cost per task*. This metric is very useful to assess how well a model generalizes. Indeed, since all instances share the same map size regardless of the number of tasks, the average cost per task should decrease when the number of tasks increases, because optimal paths contain more tasks. It is the case for the model trained on 25 tasks instances, but not for the other two, which deteriorate on instances larger than 300 tasks. This is probably due to the fact that 50 and 100 tasks datasets are not labeled with optimal solutions, but only with solutions from the local search heuristic. Indeed, the 100-task model performs even worse than the 50-task model, because the local search heuristic performance decreases when the number of tasks increases.

Train dataset	**Test dataset** (number of tasks in each instance)							
	25	50	100	200	300	500	750	1000
25 tasks	274.72	225.29	207.14	194.46	186.68	182.56	178.57	177.3
50 tasks	274.27	225.23	205.97	195.78	193.12	194.48	196.99	199.38
100 tasks	274.61	225.87	206.8	197.97	195.53	207.02	219.34	227.14

Table 9: Learning by imitation: average cost per task

Learning by experience In the learning by experience setting, we also train three models on the same three training datasets, and summarize results in Tables 10 and 11. First, we observe that average cost gaps and maximum cost gaps are lower on all models, respect to the corresponding models learned by imitation.

Train dataset	**Test dataset**					
	25 tasks		50 tasks		100 tasks	
	mean	max	mean	max	mean	max
25 tasks	0.45%	4.2%	−0.77%	0.63%	−2.11%	−0.14%
50 tasks	0.43%	3.04%	−0.78%	0.74%	−2.06%	−0.22%
100 tasks	0.43%	3.28%	−0.83%	0.97%	−2.06%	−0.29%

Table 10: Learning by experience: cost gap

Finally, unlike the learning by imitation setting, we observe decreasing average cost per task on all three models for instances up to 1000 tasks. This means models learned by experience generalize much better, because they are not biased trying to imitate non-optimal solutions.

Train dataset	Test dataset (number of tasks in each instance)							
	25	50	100	200	300	500	750	1000
25 tasks	274.19	224.55	204.9	191.86	184.71	181.29	178.0	177.02
50 tasks	274.12	224.51	205.0	191.85	184.3	180.48	176.96	176.0
100 tasks	274.13	224.41	205.0	191.85	184.63	181.08	177.81	176.74

Table 11: Learning by experience: average cost per task

In the stochastic VSP, learning by experience seems more performant than learning by imitation, the only downside being the practical difficulty of the training. Indeed, results are more sensible to hyperparameters and the model needs to be trained for more epochs because the loss convergence is slower.

6.4 Single-machine scheduling

We solve a single-machine scheduling problem with release times and sum of completion times. Given an instance of this NP-hard problem, we use an ML predictor to define the input of a ranking problem, which is an LP on the permutahedron and can be solved instantaneously. Its optimal solution (permutation) can then be decoded into an approximate solution of the hard problem. This follows the recent work of Parmentier and T'Kindt (2021), re-coded in Julia in the satellite package `SingleMachineScheduling.jl`[17].

6.4.1 Problem formulation

We need to schedule n jobs on a single machine. Each job $j \in [n]$ has an associated processing time p_j, and release date r_j. Before r_j, the job j cannot start. The machine can process only one job at a time. Preemption is not allowed, which means whenever a job has started, it must be completed, and cannot be paused to be finished later. Our aim is to find a schedule (permutation) $s = (j_1, ..., j_n)$ of $[n]$, such that the sum of completion times $\sum_j C_j(s)$ is minimal. In this sum, $C_j(s)$ is the completion time of job j in the schedule s. More precisely, given the schedule s, we have $C_{j_1}(s) = r_{j_1} + p_{j_1}$, and for $k > 1$, we have $C_{j_k}(s) = \max\left(r_{j_k}, C_{j_{k-1}}(s)\right) + p_{j_k}$.

6.4.2 Existing algorithms for the hard problem

Two kinds of algorithms are considered, both to benchmark our learning pipelines, and to provide solutions for the learning dataset. The first kind includes several heuristics: the RDI/APRTF heuristic (Chand, Traub, and Uzsoy 1996), referred to as `RDIA`, the RBS heuristic with beam width $w = 2$ (Della Croce, Salassa, and T'kindt 2014), referred to as `RBS`, and the matheuristic (Della Croce, Salassa, and T'kindt 2014), referred to as `MATH`. The second kind is an exact algorithm, used to solve instances with up to $n = 110$ jobs, the branch-and-memorize algorithm (Shang, T'Kindt, and Della Croce 2021).

6.4.3 Datasets

We consider two types of datasets generated randomly:

- A train dataset made of 420 instances with $n \in \{50, 70, 90, 110\}$ jobs, and corresponding solutions derived by means of the branch-and-memorize algorithm (Shang, T'Kindt, and Della Croce 2021). Each value of n is equally represented.
- A test dataset made of 1820 instances with $n \in \{50 + k \times 100, k \in [25]\}$ jobs, and the associated best solutions derived by the algorithms noted above. Each value of n is equally represented.

[17] https://github.com/axelparmentier/SingleMachineScheduling.jl

Probabilistic CO layer	ε	nb_samples
Additive perturbation	0.01	100
Multiplicative perturbation	0.2	100
Half square norm	0.2	

Table 12: Hyperparameters for scheduling experiments.

6.4.4 InferOpt pipeline

In the pipeline (27), an instance of the hard problem x is passed through an ML layer involving the GLM φ_w, leading to the parameter $\theta = \varphi_w(x)$. We then sort the components of θ using a ranking algorithm. It leads to a permutation s. This permutation is then passed through a decoder. The output of the decoder y is the candidate solution to the hard problem.

$$\underset{\text{instance } x}{\text{Hard problem}} \longrightarrow \boxed{\text{ML layer } \varphi_w} \xrightarrow[\theta = \varphi_w(x)]{\text{Parameter}} \boxed{\boxed{\text{Ranking } f}} \xrightarrow[s = f(\theta)]{\text{Permutation}} \boxed{\text{Decoder } \psi} \xrightarrow[y = \psi(s)]{\text{Solution}} \tag{27}$$

Parmentier and T'Kindt (2021) consider two possibilities for the decoder ψ. First, we can use the permutation $s = f(\theta)$ as the candidate solution of the hard problem. We call this algorithm *pure machine learning heuristic*, referred to as PMLH. It corresponds to $\psi = \text{id}$. Second, in the *improved machine learning heuristic*, referred to as IMLH, we combine a local search and the RDI procedure in the decoder ψ. We consider PMLH in the following.

6.4.5 Experimental setting

A fast ML-CO pipeline. We focus on a learning by imitation setting, with $\psi = \text{id}$. Since our train dataset is made of small instances (with up to 110 jobs), and our test dataset has large instances (with up to 2550 jobs), we want to see if our model generalizes well. Since ranking is quasi-instantaneous, and the algorithms defined in Section 6.4.2 require much more computational effort, small average gaps on the test set would lead to an ML-CO algorithm that saves a lot of computation time on large-scale instances.

Learning process. In the learning phase, we use the permutation $s = f(\theta)$ as candidate solution to be compared with the associated target solution in the dataset. We use three types of probabilistic CO layers and associated Fenchel-Young loss functions: additive perturbation, multiplicative perturbation, and half square norm regularization. Our hyperparameters are detailed in Table 12. The value of ε is tuned with a grid search on a validation dataset for each probabilistic CO layer, selecting the one leading to the smallest average gap. We use a batch size of 1 sample and learn through 300 epochs in each case. Experiments are done on a computing cluster with processor 32 x Intel Xeon E5-2667, 3.30GHz (hyperthreading), and 192 Go of RAM.

6.4.6 Results

In Table 13, we compare the average train and test gaps given by each of the three probabilistic CO layers detailed in Table 12, as well as the GLM-based pipeline from Parmentier and T'Kindt (2021). In each case, we derive solutions in the following way: given an instance x, we follow the pipeline (27), where the weights of the GLM w are fixed thanks to the learning phase based on the specific probabilistic CO layer (or by Parmentier and T'Kindt (2021)), and where the decoder is $\psi = \text{id}$. We recall that on the train set, gaps are computed with respect to the optimal solutions. On the test set, we consider the best solutions found by the algorithms detailed in Section 6.4.2. Overall, we manage to obtain low test gaps. We even get better solutions than the heuristics with our ML-CO pipelines on the test set.

	Train set gap (%)	Test set gap (%)
Additive perturbation (GLM)	1.68	-0.26
Multiplicative perturbation (GLM)	3.02	0.34
Half square norm regularization (GLM)	1.52	0.59
Parmentier and T'Kindt (2021) model (GLM)	2.32	1.93

Table 13: Train and test average gaps in % for pipelines with decoder $\psi = \mathrm{id}$.

In Figure 6, we show the histograms of train and test gaps given by the pipeline trained with the additive perturbation, which gives the best average results in Table 13. It emphasizes a good generalization performance, even trained on small instances. Since ranking is very fast, it highlights the practical interest of this kind of ML-CO algorithm. For further studies in a learning by experience setting, as well as additional details on the decoder part, we refer to Parmentier (2021a) and Parmentier and T'Kindt (2021).

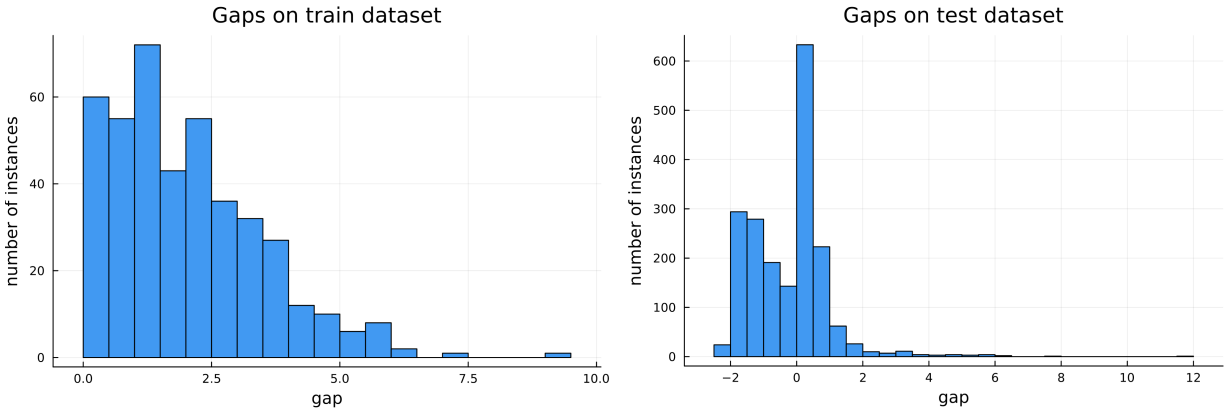


Figure 6: Histograms of train and test gaps in % given by the pipeline trained with the additive perturbation CO layer and decoder $\psi = \mathrm{id}$.

6.5 Two-stage stochastic minimum weight spanning tree

Let us now focus on the *two-stage minimum weight spanning tree problem*. This last subsection has two main purposes: first, having an application on a two stage stochastic optimization problem where we use the single stage problem as a CO layer, and second, we show how to use `InferOpt.jl` with a pipeline containing a graph neural network (GNN), and compare it to a basic generalized linear model (GLM). Source code for this application can be found in the corresponding satellite package `TwoStageSpanningTree.jl`[18].

6.5.1 Problem formulation

Let $G = (V, E)$ be an undirected graph, and S be a (finite) set of scenarios. The goal is to build a spanning tree on G over two stages at minimum cost, the second-stage building costs being unknown when first-stage decisions are taken. For each edge $e \in E$ and scenario and $s \in S$, we denote by c_e in $\mathbb{R}$ the scenario-independent first stage cost of building e, and by d_{es} the scenario-dependent second stage cost. The

[18] https://github.com/axelparmentier/MinimumWeightTwoStageSpanningTree.jl

two-stage spanning tree problem can be formulated as the following ILP:

$$\min_{y,z} \sum_{e\in E} c_e y_e + \frac{1}{|S|}\sum_{e\in E}\sum_{s\in S} d_{es} z_{es}, \tag{28a}$$

$$\text{s.t.} \sum_{e\in E} y_e + z_{es} = |V| - 1, \qquad \forall s \in S, \tag{28b}$$

$$\sum_{e\in E(Y)} y_e + z_{es} \leq |Y| - 1, \qquad \forall Y, \emptyset \subsetneq Y \subsetneq V, \forall s \in S, \tag{28c}$$

$$y_e \in \{0,1\}, \qquad \forall e \in E, \tag{28d}$$

$$z_{es} \in \{0,1\}, \qquad \forall e \in E, \forall s \in S, \tag{28e}$$

where y_e is a binary variable indicating if e is selected in the first-stage solution, and z_{es} is a binary variable indicating if e is selected in the second-stage solution for scenario s. The two constraints (28b) and (28c) jointly ensures that $y_e + z_{es}$ belongs to the spanning tree polytope.

Formulation (28) contains an exponential number of constraints. Several mathematical programming approaches enable to solve it. Instances with up to 50 vertices can be solved to optimality using a cut-generation approach or a Benders decomposition. A Lagrangian relaxation and a Lagrangian heuristic can provide solutions within a 2% optimality gap for instances with up to 10000 vertices in approximately one hour. A detailed description of this heuristic can be found in Appendix B.2.1.

6.5.2 Datasets

We build three datasets, one for training, one for validation, and one for testing. Each dataset contains 600 grid graphs instances whose widths range from 10 to 60 (that is, with 100 to 3600 vertices), and from 5 to 20 scenarios. For all instances, first stage costs are drawn uniformly between 0 and 20. For the second stage costs, depending on the instances, they are drawn uniformly between 0 and n, with n varying between 10 and 30. Each instance is labeled with a solution given by the Lagrangian heuristic.

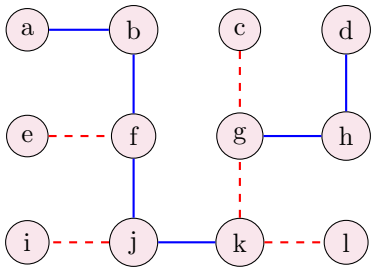


Figure 7: Example of a two stage spanning tree on a 3×4 grid graph. Blue edges correspond to first-stage forest, and red dashed edges correspond to second-stage forest.

6.5.3 InferOpt pipeline

The learning pipeline is presented in Figure 8.

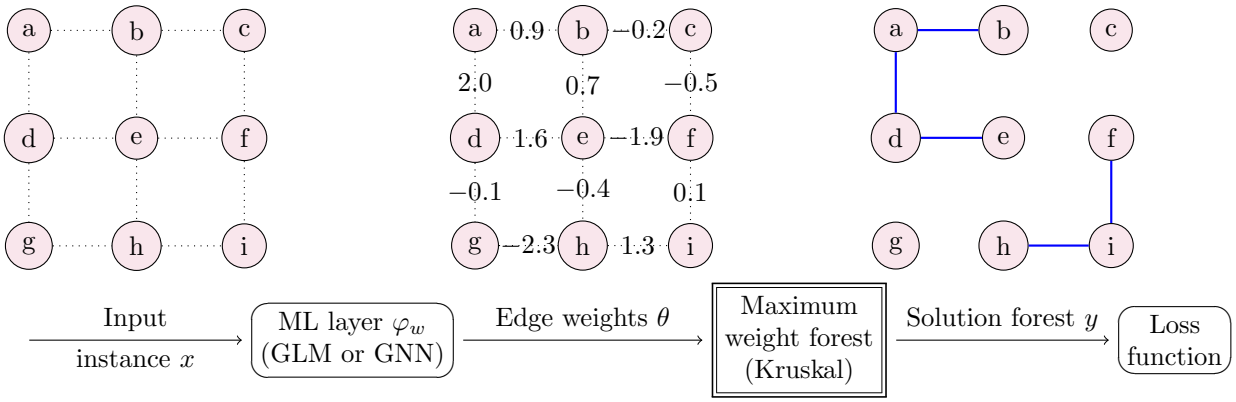


Figure 8: Two-stage minimum spanning tree pipeline

Input Each edge e of an instance x is encoded by a feature vector $\phi(x, e)$. In the following experiments, we test two different set of features. The first one is called *Basic features*, and only contains information about first-stage costs and quantiles of second-stage costs. The second feature set is called *Advanced features*, and contains multiple other statistics. See Appendix B.2.2 for a detailed description of the features.

ML predictor The feature matrix is given as input to an ML layer φ_w with learnable parameters w, which predicts edge weights θ_e. We tested and compared two different predictors: first a GLM such that $\theta_e = w^\top \phi(x, e)$, and then a GNN, in particular the `GraphConv` architecture (Morris et al. 2019). 1

Combinatorial layer We use the predicted edge weights θ as the objective of a maximum weight forest problem layer. A maximum weight forest can be efficiently found using Kruskal's algorithm.

Decoder As a post-processing, the output forest of the training pipeline is interpreted as a first stage solution of instance x. For each scenario, we can complete it into a minimum cost spanning tree containing this forest, using Kruskal's algorithm.

6.5.4 Experimental setting

We train each model by imitation using a `FenchelYoungLoss` with `PerturbedAdditive` regularization. Hypeparameters used for each model can be found in table 14.

	ε	`nb_samples`	learning rate	batch size	training epochs
GLM	1	20	0.01	1	20
GNN	1	20	0.001	1	20

Table 14: Hypeparameters for two-stage spanning tree experiments

We use the `Flux.jl` Julia library and the Adam optimizer for training. We construct our GNN with the `GraphNeuralNetworks.jl`[19] Julia library (Lucibello and contributors 2021). The GNN architecture contains 3 `GraphConv` layers with hidden size 50.

The GLM and GNN models are both trained on each of the two feature sets. Features are normalized by dividing each feature by its train dataset standard deviation. We optimize hyperparameters and perform model selection by using the average optimality gap values on the validation dataset. The experiments are

[19] https://github.com/CarloLucibello/GraphNeuralNetworks.jl

conducted on a 2.6 GHz Intel Core i9-11950H, 16 cores, with 64 Go RAM. Epochs with best gaps on the validation dataset are selected for each experiment.

6.5.5 Results

The CO layer takes the same CPU time as one iteration of the Lagrangian relaxation, and the feature computation takes about the same time as 2 iterations. Therefore, resulting pipelines are roughly 15000 times faster than the Lagrangian heuristic.

We evaluate the performance of the trained pipelines on the test dataset. For this we compute the average, minimum, and maximum gaps respect to the lower bound and upper bounds respectively given by the Lagrangian relaxation and the Lagrangian heuristic. Results are gathered in Table 15 below.

	Optimality gap (%)			**Lagrangian heuristic gap (%)**		
	Average	**Min**	**Max**	**Average**	**Min**	**Max**
GLM on basic features	4.77	1.08	12.33	2.91	0.71	8.81
GNN on basic features	3.06	0.09	12.09	1.23	-0.24	7.07
GLM on advanced features	2.21	0.09	7.66	0.41	-0.73	3.12
GNN on advanced features	2.11	0.09	7.91	0.31	-0.73	2.86

Table 15: Test gaps in % for models trained with the basic features

On both feature sets we observe low gaps for both GLM and GNN, with the GNN performing better. We also see that the GLM trained on advanced features performs better that the GNN on basic features, which means that feature engineering is more effective than replacing the linear model with a more complex neural network. Advanced features combined with the GNN performs best. As can be seen in section 6.3 on the stochastic VSP, we could maybe further improve performance if we learn by experience instead of learning by imitation. Indeed, the labels computed by the Lagrangian heuristic are not necessarily optimal and probably introduce bias in the learning.

7 Conclusion

In this paper, we introduce `InferOpt.jl`, a Julia package that seamlessly turns CO algorithms into layers of ML pipelines. The package is compatible with most approximate differentiation techniques and structured losses from the literature, but also includes novel ones. Its theoretical foundation is a probabilistic point of view on CO layers, which unifies existing and new methods. Our probabilistic framework notably provides natural "learning by experience" counterparts to the more common "learning by imitation" approaches. `InferOpt.jl` also unlocks the use of large neural networks in hybrid pipelines, even when learned by experience. We demonstrate it with a hybrid ML-CO pipeline containing a CNN with tens of thousands of parameters, which to the best of our knowledge, has never been achieved.

Using `InferOpt.jl`, we benchmark pipeline architectures and learning algorithms. In general, we observe that hybrid pipelines can yield fast algorithms with state-of-the-art performance on well-studied applications such as the single machine scheduling problem. Depending on the learning setting, we may require more or less information in the training set. Unsurprisingly, the more information we have in the training set, the better our predicted solutions become. This is notably clear on the Warcraft shortest path problem. However, on very hard combinatorial tasks where no algorithm scales on larges instances, learning by experience can outperform learning by imitation, as can be seen on the stochastic vehicle scheduling problem.

When we learn by imitating precomputed solutions, Fenchel-Young losses seem to outperform alternatives. If we use a CO algorithm that is sensitive to the sign of the input, the multiplicative perturbation seems the

best approach. Concerning the pipeline architecture, the most important aspect seems to be the choice of the combinatorial optimization layer and of the features. Using GNNs instead of GLMs for encoding can be beneficial, as we remarked on the two-stage spanning tree problem.

We also highlight some perspectives. In this paper, we learn the objective function of CO problems based on the notion of probabilistic CO layers. We do not consider the case when the desired flexibility concerns the constraints, as studied by Paulus et al. (2021) for instance, even though this perspective would be equally natural. Furthermore, we make the assumption that the CO oracle derives optimal solutions within our layer. We briefly mention the case of inexact CO oracles, but additional research would be necessary in this direction. Finally, a large part of the ML literature is dedicated to reinforcement learning. The connection between this field and structured learning deserves further exploration.

Acknowledgements

The authors want to thank Mathieu Besançon for his support with various Julia packages, as well as his expert proofreading. We also thank Éloïse Berthier, Alexandre Forel, Jérôme Malick, Clément Mantoux, and Pierre Marion for their time and advice.

References

Agrawal, A., B. Amos, S. Barratt, S. Boyd, S. Diamond, and J. Z. Kolter (2019). "Differentiable Convex Optimization Layers". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2019/hash/9ce3c52fc54362e22053399d3181c638-Abstract.html (cit. on p. 8).

Amos, B. and J. Z. Kolter (2017). "OptNet: Differentiable Optimization as a Layer in Neural Networks". en. In: *Proceedings of the 34th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, pp. 136–145. URL: https://proceedings.mlr.press/v70/amos17a.html (cit. on p. 8).

Barto, A. G. and S. Mahadevan (2003). "Recent Advances in Hierarchical Reinforcement Learning". en. In: *Discrete Event Dynamic Systems* 13.1, pp. 41–77. ISSN: 1573-7594. DOI: 10.1023/A:1022140919877. URL: https://doi.org/10.1023/A:1022140919877 (cit. on p. 9).

Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind (2018). "Automatic Differentiation in Machine Learning: A Survey". In: *Journal of Machine Learning Research* 18.153, pp. 1–43. ISSN: 1533-7928. URL: http://jmlr.org/papers/v18/17-468.html (cit. on p. 7).

Bengio, Y., A. Lodi, and A. Prouvost (2021). "Machine Learning for Combinatorial Optimization: A Methodological Tour d'horizon". en. In: *European Journal of Operational Research* 290.2, pp. 405–421. ISSN: 03772217. DOI: 10.1016/j.ejor.2020.07.063. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377221720306895 (cit. on pp. 3, 5).

Berthet, Q., M. Blondel, O. Teboul, M. Cuturi, J.-P. Vert, and F. Bach (2020). "Learning with Differentiable Perturbed Optimizers". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 9508–9519. URL: https://proceedings.neurips.cc/paper/2020/hash/6bb56208f672af0dd65451f869fedfd9-Abstract.html (cit. on pp. 4, 8–10, 12–14, 20, 22, 40, 41, 43).

Besançon, M., A. Carderera, and S. Pokutta (2022). "FrankWolfe.jl: A High-Performance and Flexible Toolbox for Frank–Wolfe Algorithms and Conditional Gradients". en. In: *INFORMS Journal on Computing*, ijoc.2022.1191. ISSN: 1091-9856, 1526-5528. DOI: 10.1287/ijoc.2022.1191. URL: http://pubsonline.informs.org/doi/10.1287/ijoc.2022.1191 (cit. on p. 16).

Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah (2017). "Julia: A Fresh Approach to Numerical Computing". en. In: *SIAM Review* 59.1, pp. 65–98. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/141000671. URL: https://epubs.siam.org/doi/10.1137/141000671 (cit. on p. 6).

Blondel, M., Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert (2022). *Efficient and Modular Implicit Differentiation*. arXiv: 2105.15183 [cs, math, stat]. URL: http://arxiv.org/abs/2105.15183 (cit. on pp. 8, 16).

Blondel, M., A. F. T. Martins, and V. Niculae (2020). "Learning with Fenchel-Young Losses". In: *Journal of Machine Learning Research* 21.35, pp. 1–69. ISSN: 1533-7928. URL: http://jmlr.org/papers/v21/19-021.html (cit. on pp. 8, 9, 12, 16, 22).

Blondel, M., O. Teboul, Q. Berthet, and J. Djolonga (2020). "Fast Differentiable Sorting and Ranking". en. In: *Proceedings of the 37th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, pp. 950–959. URL: https://proceedings.mlr.press/v119/blondel20a.html (cit. on p. 9).

Chand, S., R. Traub, and R. Uzsoy (1996). "An Iterative Heuristic for the Single Machine Dynamic Total Completion Time Scheduling Problem". en. In: *Computers & Operations Research* 23.7, pp. 641–651. ISSN: 03050548. DOI: 10.1016/0305-0548(95)00071-2. URL: https://linkinghub.elsevier.com/retrieve/pii/0305054895000712 (cit. on p. 30).

Danskin, J. M. (1967). *The Theory of Max-Min and Its Application to Weapons Allocation Problems*. Red. by M. Beckmann, R. Henn, A. Jaeger, W. Krelle, H. P. Künzi, K. Wenke, and P. Wolfe. Vol. 5. Ökonometrie Und Unternehmensforschung / Econometrics and Operations Research. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-46094-4 978-3-642-46092-0. DOI: 10.1007/978-3-642-46092-0. URL: http://link.springer.com/10.1007/978-3-642-46092-0 (cit. on p. 20).

Della Croce, F., F. Salassa, and V. T'kindt (2014). "A Hybrid Heuristic Approach for Single Machine Scheduling with Release Times". en. In: *Computers & Operations Research* 45, pp. 7–11. ISSN: 03050548. DOI: 10.1016/j.cor.2013.11.016. URL: https://linkinghub.elsevier.com/retrieve/pii/S0305054813003390 (cit. on p. 30).

Elmachtoub, A. N. and P. Grigas (2022). "Smart "Predict, Then Optimize"". en. In: *Management Science* 68.1, pp. 9–26. ISSN: 0025-1909, 1526-5501. DOI: 10.1287/mnsc.2020.3922. URL: http://pubsonline.informs.org/doi/10.1287/mnsc.2020.3922 (cit. on pp. 6, 8, 20, 22).

Fairbanks, J., M. Besançon, S. Simon, J. Hoffiman, N. Eubank, and S. Karpinski (2021). *JuliaGraphs/Graphs.jl: An Optimized Graphs Package for the Julia Programming Language*. URL: https://github.com/JuliaGraphs/Graphs.jl/ (cit. on p. 10).

Ferber, A., B. Wilder, B. Dilkina, and M. Tambe (2020). "MIPaaL: Mixed Integer Program as a Layer". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.02, pp. 1504–1511. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v34i02.5509. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5509 (cit. on p. 9).

Frank, M. and P. Wolfe (1956). "An Algorithm for Quadratic Programming". en. In: *Naval Research Logistics Quarterly* 3.1-2, pp. 95–110. ISSN: 00281441, 19319193. DOI: 10.1002/nav.3800030109. URL: https://onlinelibrary.wiley.com/doi/10.1002/nav.3800030109 (cit. on p. 15).

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. eng. Cambridge, Massachusetts: The MIT Press. ISBN: 978-0-262-33737-3. URL: https://www.deeplearningbook.org/ (cit. on p. 7).

Griewank, A. and A. Walther (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics. 438 pp. ISBN: 978-0-89871-659-7 (cit. on p. 7).

He, K., X. Zhang, S. Ren, and J. Sun (2016). "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Las Vegas, NV, USA: IEEE, pp. 770–778. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.90. URL: http://ieeexplore.ieee.org/document/7780459/ (cit. on p. 10).

Innes, M. (2019). *Don't Unroll Adjoint: Differentiating SSA-Form Programs*. arXiv: 1810.07951 [cs]. URL: http://arxiv.org/abs/1810.07951 (cit. on p. 10).

Innes, M., E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah (2018). *Fashionable Modelling with Flux*. arXiv: 1811.01457 [cs]. URL: http://arxiv.org/abs/1811.01457 (cit. on p. 10).

Innes, M. (2018). "Flux: Elegant Machine Learning with Julia". In: *Journal of Open Source Software* 3.25, p. 602. ISSN: 2475-9066. DOI: 10.21105/joss.00602. URL: http://joss.theoj.org/papers/10.21105/joss.00602 (cit. on p. 10).

Jaggi, M. (2013). "Revisiting Frank-Wolfe: Projection-Free Sparse Convex Optimization". en. In: *Proceedings of the 30th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, pp. 427–435. URL: https://proceedings.mlr.press/v28/jaggi13.html (cit. on p. 15).

Jones, D. R., C. D. Perttunen, and B. E. Stuckman (1993). "Lipschitzian Optimization without the Lipschitz Constant". en. In: *Journal of Optimization Theory and Applications* 79.1, pp. 157–181. ISSN: 0022-3239, 1573-2878. DOI: 10.1007/BF00941892. URL: http://link.springer.com/10.1007/BF00941892 (cit. on p. 19).

Kingma, D. P. and J. Ba (2015). "Adam: A Method for Stochastic Optimization". In: *ICLR (Poster)*. URL: http://arxiv.org/abs/1412.6980 (cit. on p. 23).

Kolter, J. Z., D. Duvenaud, and M. Johnson (2020). *Deep Implicit Layers - Neural ODEs, Deep Equilibirum Models, and Beyond*. en. URL: http://implicit-layers-tutorial.org/ (cit. on p. 7).

Korte, B. and J. Vygen (2006). *Combinatorial Optimization: Theory and Algorithms*. eng. 3rd edition. Algorithms and Combinatorics 21. Berlin: Springer. ISBN: 978-3-540-29297-5 (cit. on p. 5).

Kotary, J., F. Fioretto, P. Van Hentenryck, and B. Wilder (2021). "End-to-End Constrained Optimization Learning: A Survey". en. In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*. Thirtieth International Joint Conference on Artificial Intelligence {IJCAI-21}. Montreal, Canada: International Joint Conferences on Artificial Intelligence Organization, pp. 4475–4482. ISBN: 978-0-9992411-9-6. DOI: 10.24963/ijcai.2021/610. URL: https://www.ijcai.org/proceedings/2021/610 (cit. on p. 3).

Lucibello, C. and o. contributors (2021). *GraphNeuralNetworks.jl: A Geometric Deep Learning Library for the Julia Programming Language*. URL: https://github.com/CarloLucibello/GraphNeuralNetworks.jl (cit. on p. 34).

Mandi, J., V. Bucarey, M. M. K. Tchomba, and T. Guns (2022). "Decision-Focused Learning: Through the Lens of Learning to Rank". en. In: *Proceedings of the 39th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, pp. 14935–14947. URL: https://proceedings.mlr.press/v162/mandi22a.html (cit. on p. 9).

Mandi, J., E. Demirović, P. J. Stuckey, and T. Guns (2020). "Smart Predict-and-Optimize for Hard Combinatorial Optimization Problems". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.02, pp. 1603–1610. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v34i02.5521. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5521 (cit. on p. 9).

Mandi, J. and T. Guns (2020). "Interior Point Solving for LP-based Prediction+optimisation". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 7272–7282. URL: https://proceedings.neurips.cc/paper/2020/hash/51311013e51adebc3c34d2cc591fefee-Abstract.html (cit. on p. 8).

Martins, A. F. T. and R. Astudillo (2016). "From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification". en. In: *Proceedings of The 33rd International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, pp. 1614–1623. URL: https://proceedings.mlr.press/v48/martins16.html (cit. on p. 16).

Morris, C., M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe (2019). "Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33, pp. 4602–4609. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v33i01.33014602. URL: https://aaai.org/ojs/index.php/AAAI/article/view/4384 (cit. on p. 34).

Mulamba, M., J. Mandi, M. Diligenti, M. Lombardi, V. Bucarey, and T. Guns (2021). "Contrastive Losses and Solution Caching for Predict-and-Optimize". en. In: Twenty-Ninth International Joint Conference on Artificial Intelligence. Vol. 3, pp. 2833–2840. DOI: 10.24963/ijcai.2021/390. URL: https://www.ijcai.org/proceedings/2021/390 (cit. on p. 9).

Niculae, V., A. F. T. Martins, M. Blondel, and C. Cardie (2018). "SparseMAP: Differentiable Sparse Structured Inference". en. In: *Proceedings of the 35th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, pp. 3799–3808. URL: https://proceedings.mlr.press/v80/niculae18a.html (cit. on p. 16).

Niepert, M., P. Minervini, and L. Franceschi (2021). "Implicit MLE: Backpropagating Through Discrete Exponential Family Distributions". In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., pp. 14567–14579. URL: https://proceedings.neurips.cc/paper/2021/hash/7a430339c10c642c4b2251756fd1b484-Abstract.html (cit. on p. 9).

Nowozin, S. and C. H. Lampert (2010). "Structured Learning and Prediction in Computer Vision". en. In: *Foundations and Trends® in Computer Graphics and Vision* 6.3-4, pp. 185–365. ISSN: 1572-2740, 1572-2759. DOI: 10.1561/0600000033. URL: https://www.nowpublishers.com/article/Details/CGV-033 (cit. on pp. 8, 22).

Parmentier, A. (2019). "Algorithms for Non-Linear and Stochastic Resource Constrained Shortest Path". en. In: *Mathematical Methods of Operations Research* 89.2, pp. 281–317. ISSN: 1432-2994, 1432-5217. DOI: 10.1007/s00186-018-0649-x. URL: http://link.springer.com/10.1007/s00186-018-0649-x (cit. on p. 45).

— (2021a). *Learning Structured Approximations of Operations Research Problems*. arXiv: 2107.04323 [cs]. URL: http://arxiv.org/abs/2107.04323 (cit. on pp. 6, 19, 25, 32).

— (2021b). "Learning to Approximate Industrial Problems by Operations Research Classic Problems". en. In: *Operations Research* 70.1, pp. 606–623. ISSN: 0030-364X, 1526-5463. DOI: 10.1287/opre.2020.2094. URL: http://pubsonline.informs.org/doi/10.1287/opre.2020.2094 (cit. on pp. 9, 25, 26, 45).

Parmentier, A. and V. T'Kindt (2021). *Learning to Solve the Single Machine Scheduling Problem with Release Times and Sum of Completion Times*. arXiv: 2101.01082 [cs, math]. URL: http://arxiv.org/abs/2101.01082 (cit. on pp. 25, 30–32).

Paulus, A., M. Rolinek, V. Musil, B. Amos, and G. Martius (2021). "CombOptNet: Fit the Right NP-Hard Problem by Learning Integer Programming Constraints". en. In: *Proceedings of the 38th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, pp. 8443–8453. URL: https://proceedings.mlr.press/v139/paulus21a.html (cit. on pp. 9, 36).

Pineda, L., T. Fan, M. Monge, S. Venkataraman, P. Sodhi, R. Chen, J. Ortiz, D. DeTone, A. Wang, S. Anderson, J. Dong, B. Amos, and M. Mukadam (2022). *Theseus: A Library for Differentiable Nonlinear Optimization*. arXiv: 2207.09442 [cs, math]. URL: http://arxiv.org/abs/2207.09442 (cit. on p. 8).

Shang, L., V. T'Kindt, and F. Della Croce (2021). "Branch & Memorize Exact Algorithms for Sequencing Problems: Efficient Embedding of Memorization into Search Trees". en. In: *Computers & Operations Research* 128, p. 105171. ISSN: 03050548. DOI: 10.1016/j.cor.2020.105171. URL: https://linkinghub.elsevier.com/retrieve/pii/S0305054820302884 (cit. on p. 30).

Sharma, A., M. Besançon, J. D. Garcia, and B. Legat (2022). *Flexible Differentiable Optimization via Model Transformations*. arXiv: 2206.06135 [cs, math]. URL: http://arxiv.org/abs/2206.06135 (cit. on p. 8).

Sutton, R. S. and A. G. Barto (2018). *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press. 526 pp. ISBN: 978-0-262-03924-6 (cit. on p. 9).

Tang, B. and E. B. Khalil (2022). *PyEPO: A PyTorch-based End-to-End Predict-then-Optimize Library for Linear and Integer Programming*. arXiv: 2206.14234 [cs, math]. URL: http://arxiv.org/abs/2206.14234 (cit. on p. 9).

Vlastelica, M., A. Paulus, V. Musil, G. Martius, and M. Rolinek (2020). "Differentiation of Blackbox Combinatorial Solvers". en. In: International Conference on Learning Representations. URL: https://openreview.net/forum?id=BkevoJSYPB (cit. on pp. 4, 8–10, 13, 23).

White, F. C., M. Abbott, M. Zgubic, J. Revels, S. Axen, A. Arslan, S. Schaub, N. Robinson, Y. Ma, G. Dhingra, W. Tebbutt, N. Heim, D. Widmann, A. D. W. Rosemberg, N. Schmitz, C. Rackauckas, R.

Heintzmann, Frankschae, A. Noack, C. Lucibello, K. Fischer, A. Robson, Cossio, J. Ling, MattBrzezinski, R. Finnegan, A. Zhabinski, D. Wennberg, M. Besançon, and P. Vertechi (2022). *JuliaDiff/ChainRules.jl: V1.44.7*. Version v1.44.7. Zenodo. DOI: 10.5281/ZENODO.4754896. URL: https://zenodo.org/record/4754896 (cit. on p. 6).

Wilder, B., B. Dilkina, and M. Tambe (2019). "Melding the Data-Decisions Pipeline: Decision-Focused Learning for Combinatorial Optimization". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33, pp. 1658–1665. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v33i01.33011658. URL: https://aaai.org/ojs/index.php/AAAI/article/view/3982 (cit. on p. 8).

A Proofs

In this Section, the dominated convergence theorem is used implicitly to justify any differentiation under the integral sign. When dealing with polyhedral functions such as $\theta \longmapsto \max_{v\in\mathcal{V}} \theta^\top v$, we often write ∇_θ for simplicity even though they are only subdifferentiable, because the set of non-differentiability has measure zero. We denote the standard Gaussian density by ν.

A.1 Additive perturbation

These proofs are already given by Berthet et al. (2020), but we include them for comparison purposes.

A.1.1 Derivatives

Proof of Proposition 3.1.

Proof. The following change of variable is a diffeomorphism:

$$u = \theta + \varepsilon z \quad \iff \quad \frac{u-\theta}{\varepsilon} = z.$$

We apply it to the definition of $\widehat{p}_\varepsilon^+(v|\theta)$.

$$\begin{aligned}\widehat{p}_\varepsilon^+(v|\theta) &= \int_{\mathbb{R}^d} \mathbb{1}\{f(\theta+\varepsilon z) = v\}\ \nu(z)\ \mathrm{d}z \\ &= \int_{\mathbb{R}^d} \mathbb{1}\{f(u) = v\}\ \nu\left(\frac{u-\theta}{\varepsilon}\right)\frac{\mathrm{d}u}{\varepsilon^d}.\end{aligned}$$

We now differentiate with respect to θ before applying the reverse change of variable:

$$\begin{aligned}\nabla_\theta \widehat{p}_\varepsilon^+(v|\theta) &= \int_{\mathbb{R}^d} \mathbb{1}\{f(u) = v\}\ \left(\frac{-1}{\varepsilon}\nabla\nu\left(\frac{u-\theta}{\varepsilon}\right)\right)\frac{\mathrm{d}u}{\varepsilon^d} \\ &= \frac{-1}{\varepsilon}\int_{\mathbb{R}^d} \mathbb{1}\{f(\theta+\varepsilon z) = v\}\ \nabla\nu(z)\ \mathrm{d}z \\ &= \frac{1}{\varepsilon}\int_{\mathbb{R}^d} \mathbb{1}\{f(\theta+\varepsilon z) = v\}\ z\nu(z)\ \mathrm{d}z.\end{aligned}$$

The last equality holds because the standard Gaussian density satisfies $\nabla\nu(z) = -z\nu(z)$. From there, we deduce the Jacobian of $\widehat{f}_\varepsilon^+(\theta)$:

$$\begin{aligned}J_\theta \widehat{f}_\varepsilon^+(\theta) &= \sum_{v\in\mathcal{V}} v\nabla_\theta \widehat{p}_\varepsilon^+(\theta, y)^\top \\ &= \frac{1}{\varepsilon}\int_{\mathbb{R}^d} \underbrace{\left(\sum_{v\in\mathcal{V}} v\ \mathbb{1}\{f(\theta+\varepsilon z) = v\}\right)}_{f(\theta+\varepsilon z)}\ z^\top \nu(z)\ \mathrm{d}z\end{aligned}$$

We arrive at the following simple expression, which was already given by Berthet et al. (2020):

$$J_\theta \widehat{f}_\varepsilon^+(\theta) = \frac{1}{\varepsilon}\mathbb{E}\left[f(\theta + \varepsilon Z)Z^\top\right]$$

□

A.1.2 Regularization

Proof of Proposition 3.2.

Proof. Because $\Omega_\varepsilon^+ = (F_\varepsilon^+)^*$ is a Fenchel conjugate, it is automatically convex. Furthermore,

$$\begin{aligned}\Omega_\varepsilon^+(\mu) &= \sup_{\theta\in\mathbb{R}^d}\left\{\theta^\top\mu - F_\varepsilon^+(\theta)\right\}\\ &= \sup_{\theta\in\mathbb{R}^d}\left\{\theta^\top\mu - \mathbb{E}\left[\max_{v\in\mathcal{V}}(\theta+\varepsilon Z)^\top v\right]\right\}.\end{aligned}$$

We consider $\mu \notin \mathrm{conv}(\mathcal{V})$. By convex separation, there exists $\tilde{\theta}\in\mathbb{R}^d$ and $\alpha > 0$ such that $\tilde{\theta}^\top\mu \geq \alpha + \tilde{\theta}^\top v$ for all $v\in\mathcal{V}$. This implies that, for all $t > 0$,

$$\begin{aligned}\Omega_\varepsilon^+(\mu) &\geq t\tilde{\theta}^\top\mu - \mathbb{E}\left[\max_{v\in\mathcal{V}}\left(t\tilde{\theta}+\varepsilon Z\right)^\top v\right]\\ &\geq t\tilde{\theta}^\top\mu - t\mathbb{E}\left[\max_{v\in\mathcal{V}}\tilde{\theta}^\top v\right] - \varepsilon\mathbb{E}\left[\max_{v\in\mathcal{V}}Z^\top v\right]\\ &\geq t\alpha - \varepsilon\mathbb{E}\left[\max_{v\in\mathcal{V}}Z^\top v\right] \xrightarrow[t\to+\infty]{} +\infty\end{aligned}$$

We have shown that $\mu\notin\mathrm{dom}(\Omega_\varepsilon^+)$, and therefore $\mathrm{dom}(\Omega_\varepsilon^+)\subset\mathrm{conv}(\mathcal{V})$. We define

$$f_Z(\theta, v) = (\theta+\varepsilon Z)^\top v \qquad \text{so that} \qquad F_\varepsilon^+(\theta) = \mathbb{E}\left[\max_{v\in\mathcal{V}} f_Z(\theta, v)\right].$$

Danskin's theorem helps us compute the gradient of F_ε^+:

$$\begin{aligned}\nabla_\theta F_\varepsilon^+(\theta) &= \mathbb{E}\left[\nabla_\theta\left(\max_{v\in\mathcal{V}} f_Z(\theta, v)\right)\right]\\ &= \mathbb{E}\left[\nabla_1 f_Z\left(\theta, \operatorname*{argmax}_{v\in\mathcal{V}} f_Z(\theta, v)\right)\right]\\ &= \mathbb{E}\left[\operatorname*{argmax}_{v\in\mathcal{V}} f_Z(\theta, v)\right]\\ &= \mathbb{E}\left[\operatorname*{argmax}_{v\in\mathcal{V}}(\theta+\varepsilon Z)^\top v\right] = \widehat{f}_\varepsilon^+(\theta).\end{aligned}$$

As shown by Berthet et al. (2020, Proposition 2.2), the function Ω_ε^+ is a Legendre type function, which means that

$$\nabla_\theta F_\varepsilon^+ = \nabla_\theta(\Omega_\varepsilon^+)^* = (\nabla_\theta\Omega_\varepsilon^+)^{-1}.$$

From this, we deduce

$$\begin{aligned}\nabla_\theta F_\varepsilon^+(\theta) &= \operatorname*{argmax}_{\mu\in\mathbb{R}^d}\left\{\theta^\top\mu - \Omega_\varepsilon^+(\mu)\right\}\\ &= \operatorname*{argmax}_{\mu\in\mathrm{dom}(\Omega_\varepsilon^+)}\left\{\theta^\top\mu - \Omega_\varepsilon^+(\mu)\right\} = \widehat{f}_{\Omega_\varepsilon^+}(\theta).\end{aligned}$$

Hence, we can conclude:

$$\widehat{f}_\varepsilon^+(\theta) = \nabla_\theta F_\varepsilon^+(\theta) = \underset{\mu \in \mathrm{dom}(\Omega_\varepsilon^+)}{\mathrm{argmax}} \{\theta^\top \mu - \Omega_\varepsilon^+(\mu)\} = \widehat{f}_{\Omega_\varepsilon^+}(\theta).$$

To recover the formula given in Proposition 3.2, we simply remember that $\mathrm{dom}(\Omega_\varepsilon^+) \subseteq \mathrm{conv}(\mathcal{V})$. □

A.2 Multiplicative perturbation

A.2.1 Derivatives

Proof of Proposition 3.3.

Proof. Suppose $\theta \in \mathbb{R}^d$ only has positive components. Then the following change of variable is a diffeomorphism:

$$u = \theta \odot e^{\varepsilon z - \varepsilon^2 \mathbf{1}/2} \quad \Longleftrightarrow \quad \frac{\log(u) - \log(\theta)}{\varepsilon} + \frac{\varepsilon \mathbf{1}}{2} = z$$

We apply it to the definition of $\widehat{p}_\varepsilon^{\odot}(v|\theta)$.

$$\begin{aligned}
\widehat{p}_\varepsilon^{\odot}(\theta, y) &= \int_{\mathbb{R}^d} \mathbb{1}\Big\{ f\left(\theta \odot e^{\varepsilon z - \varepsilon^2 \mathbf{1}/2}\right) = v \Big\} \, \nu(z) \, \mathrm{d}z \\
&= \int_{(0,+\infty)^d} \mathbb{1}\{f(u) = v\} \, \nu\left(\frac{\log(u) - \log(\theta)}{\varepsilon} + \frac{\varepsilon \mathbf{1}}{2} \right) \frac{\mathrm{d}u}{\varepsilon^d \prod_i u_i}.
\end{aligned}$$

We now differentiate with respect to θ before applying the reverse change of variable:

$$\begin{aligned}
\nabla_\theta \widehat{p}_\varepsilon^{\odot}(\theta, y) &= \int_{(0,+\infty)^d} \mathbb{1}\{f(u) = v\} \, \left(\frac{-1}{\varepsilon \theta} \odot \nabla \nu \left(\frac{\log(u) - \log(\theta)}{\varepsilon} + \frac{\varepsilon \mathbf{1}}{2} \right) \right) \frac{\mathrm{d}u}{\varepsilon^d \prod_i u_i} \\
&= \frac{-1}{\varepsilon \theta} \odot \int_{\mathbb{R}^d} \mathbb{1}\Big\{ f\left(\theta \odot e^{\varepsilon z - \varepsilon^2 \mathbf{1}/2}\right) = v \Big\} \, \nabla \nu(z) \, \mathrm{d}z \\
&= \frac{1}{\varepsilon \theta} \odot \int_{\mathbb{R}^d} \mathbb{1}\Big\{ f\left(\theta \odot e^{\varepsilon z - \varepsilon^2 \mathbf{1}/2}\right) = v \Big\} \, z \nu(z) \, \mathrm{d}z.
\end{aligned}$$

From there, we deduce the Jacobian of $\widehat{f}_\varepsilon^{\odot}(\theta)$:

$$\begin{aligned}
J_\theta \widehat{f}_\varepsilon^{\odot}(\theta) &= \sum_{v \in \mathcal{V}} v \nabla_\theta \widehat{p}_\varepsilon^{\odot}(\theta, y)^\top \\
&= \frac{1}{\varepsilon \theta} \odot \int_{\mathbb{R}^d} \underbrace{\left(\sum_{v \in \mathcal{V}} v \, \mathbb{1}\Big\{ f\left(\theta \odot e^{\varepsilon z - \varepsilon^2 \mathbf{1}/2}\right) = v \Big\} \right)}_{f\left(\theta \odot e^{\varepsilon z - \varepsilon^2 \mathbf{1}/2}\right)} z^\top \nu(z) \, \mathrm{d}z
\end{aligned}$$

We arrive at a simple variant of the previous expression:

$$J_\theta \widehat{f}_\varepsilon^{\odot}(\theta) = \frac{1}{\varepsilon \theta} \odot \mathbb{E}\left[f\left(\theta \odot e^{\varepsilon z - \varepsilon^2 \mathbf{1}/2}\right) Z^\top \right]$$

□

A.2.2 Regularization

Proof of Proposition 3.4.

Proof. Because $\Omega_\varepsilon^\odot = (F_\varepsilon^\odot)^*$ is a Fenchel conjugate, it is automatically convex. Furthermore,

$$\begin{aligned}\Omega_\varepsilon^\odot(\mu) &= \sup_{\theta\in\mathbb{R}^d} \left\{\theta^\top \mu - F_\varepsilon^\odot(\theta)\right\} \\ &= \sup_{\theta\in\mathbb{R}^d} \left\{\theta^\top \mu - \mathbb{E}\left[\max_{v\in\mathcal{V}} \left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right)^\top v\right]\right\} \\ &= \sup_{\theta\in\mathbb{R}^d} \left\{\theta^\top \mu - \mathbb{E}\left[\max_{v\in\mathcal{V}} \theta^\top \left(v \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right)\right]\right\}.\end{aligned}$$

This last expression shows why we do not have $\mathrm{dom}(\Omega_\varepsilon^\odot) \subset \mathrm{conv}(\mathcal{V})$ (unlike in the additive case). Indeed, even when $\mu \notin \mathrm{conv}(\mathcal{V})$, the multiplicative scaling of v might allow it to compensate the inner product $\theta^\top \mu$ and stop $\Omega_\varepsilon^\odot(\mu)$ from going to $+\infty$. We define

$$f_Z(\theta, v) = \left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right)^\top v \qquad \text{so that} \qquad F_\varepsilon^\odot(\theta) = \mathbb{E}\left[\max_{v\in\mathcal{V}} f_Z(\theta, v)\right].$$

Danskin's theorem helps us compute the gradient of $F_\varepsilon^\odot$:

$$\begin{aligned}\nabla_\theta F_\varepsilon^\odot(\theta) &= \mathbb{E}\left[\nabla_\theta \left(\max_{v\in\mathcal{V}} f_Z(\theta, v)\right)\right] \\ &= \mathbb{E}\left[\nabla_1 f_Z\left(\theta, \operatorname*{argmax}_{v\in\mathcal{V}} f_Z(\theta, v)\right)\right] \\ &= \mathbb{E}\left[e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2} \odot \operatorname*{argmax}_{v\in\mathcal{V}} f_Z(\theta, v)\right] \\ &= \mathbb{E}\left[e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2} \odot \operatorname*{argmax}_{v\in\mathcal{V}} \left(\theta \odot e^{\varepsilon Z - \varepsilon^2 \mathbf{1}/2}\right)^\top v\right] \\ &= \widehat{f}_\varepsilon^{\odot\mathrm{scaled}}(\theta) \neq \widehat{f}_\varepsilon^\odot(\theta)\end{aligned}$$

We could prove in a way similar to Berthet et al. (2020, Proposition 2.2) that $\Omega_\varepsilon^\odot$ is a Legendre type function, which means that

$$\nabla_\theta F_\varepsilon^\odot = \nabla_\theta (\Omega_\varepsilon^\odot)^* = (\nabla_\theta \Omega_\varepsilon^\odot)^{-1}.$$

From this, we deduce

$$\begin{aligned}\nabla_\theta F_\varepsilon^\odot(\theta) &= \operatorname*{argmax}_{\mu\in\mathbb{R}^d} \left\{\theta^\top \mu - \Omega_\varepsilon^\odot(\mu)\right\} \\ &= \operatorname*{argmax}_{\mu\in\mathrm{dom}(\Omega_\varepsilon^\odot)} \left\{\theta^\top \mu - \Omega_\varepsilon^\odot(\mu)\right\} = \widehat{f}_{\Omega_\varepsilon^+}(\theta).\end{aligned}$$

This time we cannot replace $\mathrm{dom}(\Omega_\varepsilon^\odot)$ by $\mathrm{conv}(\mathcal{V})$, but we still obtain a similar conclusion:

$$\widehat{f}_\varepsilon^{\odot\mathrm{scaled}}(\theta) = \nabla_\theta F_\varepsilon^\odot(\theta) = \operatorname*{argmax}_{\mu\in\mathrm{dom}(\Omega_\varepsilon^\odot)} \left\{\theta^\top \mu - \Omega_\varepsilon^\odot(\mu)\right\} = \widehat{f}_{\Omega_\varepsilon^\odot}(\theta).$$

□

A.3 Inexact oracles

Proof of Proposition 3.5.

Proof. We start with additive perturbation. By Proposition 3.1, we have:

$$\begin{aligned}J_\theta \widehat{g}_\varepsilon^+(\theta) - J_\theta \widehat{f}_\varepsilon^+(\theta) &= \frac{1}{\varepsilon}\mathbb{E}\left[g(\theta + \varepsilon Z) Z^\top\right] - \frac{1}{\varepsilon}\mathbb{E}\left[f(\theta + \varepsilon Z) Z^\top\right] \\ &= \frac{1}{\varepsilon}\mathbb{E}\left[\left(g(\theta + \varepsilon Z) - f(\theta + \varepsilon Z)\right) Z^\top\right].\end{aligned}$$

We bound the spectral norm of the error using Jensen's inequality:

$$\begin{aligned}\left\|J_\theta \widehat{g}_\varepsilon^+(\theta) - J_\theta \widehat{f}_\varepsilon^+(\theta)\right\|^2 &\leq \frac{1}{\varepsilon^2}\mathbb{E}\left\|(g(\theta+\varepsilon Z) - f(\theta+\varepsilon Z))\, Z^\top\right\|^2 \\ &= \frac{1}{\varepsilon^2}\mathbb{E}\left[\|g(\theta+\varepsilon Z) - f(\theta+\varepsilon Z)\|^2 \|Z\|^2\right] \\ &\leq \frac{1}{\varepsilon^2}\|g-f\|_\infty^2 \mathbb{E}\left[\|Z\|^2\right] \\ &= \frac{d}{\varepsilon^2}\|g-f\|_\infty^2.\end{aligned}$$

We now move on to multiplicative perturbation. For multiplicative perturbation, following the same proof starting from Proposition 3.3 yields a similar inequality:

$$\left\|J_\theta \widehat{g}_\varepsilon^\odot(\theta) - J_\theta \widehat{f}_\varepsilon^\odot(\theta)\right\|^2 \leq \frac{d}{\varepsilon^2 \min_i |\theta_i|^2}\|g-f\|_\infty^2.$$

□

B More details on applications

B.1 Stochastic vehicle scheduling problem

B.1.1 Linear program formulation of (VSP):

The deterministic Vehicle Scheduling Problem (VSP), can be formulated as the following linear program:

$$\begin{aligned}\min \sum_{a\in A} \theta_a y_a & \\ s.t. \sum_{a\in\delta^-(v)} y_a = \sum_{a\in\delta^+(v)} y_a, & \quad \forall v \in V\backslash\{o,d\} \\ \sum_{a\in\delta^-(v)} y_a = 1, & \quad \forall v \in V\backslash\{o,d\} \\ y_a \in \{0,1\}, & \quad \forall a \in A\end{aligned} \tag{29}$$

Since the constraint matrix is totally unimodular, the constraint polytope gives a perfect formulation, therefore binary constraints can be relaxed, and this problem can be solved using either a flow algorithm or a linear programming solver.

B.1.2 Two MILP formulations for the StoVSP

MILP formulation We denote respectively c_{vehicle} and c_{delay} the cost of one vehicle, and the cost of one minute of delay. The StoVSP problem can be formulated as the following MIP with quadratic constraints:

$$\min_{d,y} c_{\text{delay}} \frac{1}{|S|} \sum_{s \in S} \sum_{v \in V \setminus \{o,d\}} d_v^s + c_{\text{vehicle}} \sum_{a \in \delta^+(o)} y_a \tag{30a}$$

$$\text{s.t.} \sum_{a \in \delta^-(v)} y_a = \sum_{a \in \delta^+(v)} y_a \qquad \forall v \in V \setminus \{o,d\} \tag{30b}$$

$$\sum_{a \in \delta^-(v)} y_a = 1 \qquad \forall v \in V \setminus \{o,d\} \tag{30c}$$

$$d_v^s \geq \gamma_v^s + \sum_{\substack{a \in \delta^-(v) \\ a=(u,v)}} (d_u^s - s_{u,v}^s) y_a \qquad \forall v \in V \setminus \{o,d\}, \forall s \in S \tag{30d}$$

$$d_v^s \geq \gamma_v^s \qquad \forall v \in V \setminus \{o,d\}, \forall s \in S \tag{30e}$$

$$y_a \in \{0,1\} \qquad \forall a \in A \tag{30f}$$

These quadratic delay constraints (30b) can be linearized using McCormick inequalities, and we can then solve the resulting MILP with industrial solvers. This approach does not scale well on instances with large number of tasks and scenarios.

Column generation formulation Another possible formulation for StoVSP is the column generation approach. Let $\mathcal{P}$ the set of feasible vehicle routes. Let us define the cost of a vehicle route $P \in \mathcal{P}$ by $c_P^s = c_{\text{vehicle}} + c_{\text{delay}} \times \sum_{v \in P} d_v^s$. We introduce decision variables $y_P \in \{0,1\}$ that equals 1 if the vehicle route P is chosen. We can then formulate StoVSP as follows:

$$\begin{aligned} &\min \frac{1}{|S|} \sum_{s \in S} \sum_{P \in \mathcal{P}} c_P^s y_P \\ &\text{s.t.} \sum_{p \ni v} y_P = 1 \qquad \forall v \in V \setminus \{o,d\} \quad (\lambda_v \in \mathbb{R}) \\ &\quad y_P \in \{0,1\} \qquad \forall p \in \mathcal{P} \end{aligned} \tag{31}$$

Since there is an exponential number of variables, this formulation can be solved using a column generation algorithm. We denote λ_v the dual variables of (31). The associated sub-problem is a constrained shortest path problem :

$$\min_{P \in \mathcal{P}} (c_P - \sum_{v \in P} \lambda_v) \tag{32}$$

The sub-problem (32) can be solved using a stochastic constrained shortest path algorithm (see Parmentier 2019 for in-depth details about these algorithms, and `ConstrainedShortestPaths.jl`[20] for a Julia implementation).

Solving (31) using column generation gives good bounds on instances with up to 50 tasks. A Branch-and-price could be used to try to find the optimal solution.

B.1.3 Data generation

Instance generator To generate our dataset, we use a similar approach as by Parmentier 2021b. We define a squared map of 50 minutes width, divided in 25 squared (10×10) districts. Each task v has a (uniformly drawn) start point, a start time t_v^b, end point, and end time t_v^e.

[20] https://github.com/BatyLeo/ConstrainedShortestPaths.jl

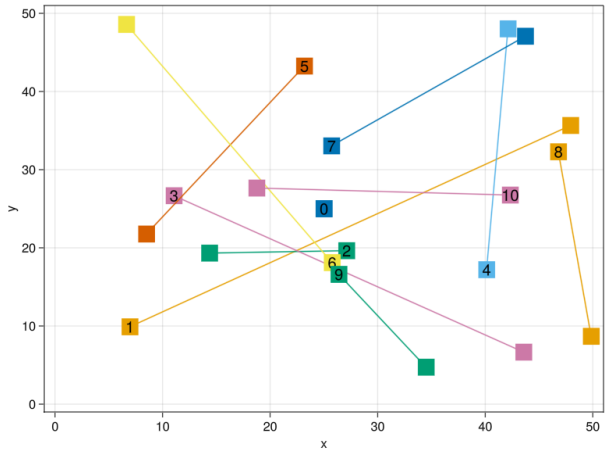


Figure 9: Example of a map with 10 tasks. Task 0 represents the depot, in the middle of the map. Square with task indices are starting points, and tasks without are end points.

We draw each scenario $s \in S$ separately and independently. For each district d and hour h of the day, we draw $\beta_{d,h} \sim \log\mathcal{N}(\mu_{d,h}, \sigma_{d,h})$, with $(\mu_{d,h}, \sigma_{d,h}) \sim \mathcal{U}([1,3]) \times \mathcal{U}([0.4, 0.6])$. For each district d and hour h of the day, $\zeta^{dis}_{d,h}$ models the congestion in the district at this time, and is computed as follows:

$$\begin{cases} \forall d,\ \zeta^{dis}_{d,0} = \beta_{d,0} \\ \forall d, h,\ \zeta^{dis}_{d,h+1} = \frac{1}{2}\zeta^{dis}_{d,h} + \beta_{d,h} \end{cases} \tag{33}$$

For each hour h of the day, ζ^{inter}_{h} models the congestion on roads between districts, and is computed similarly. With $I \sim \log\mathcal{N}(\mu = 0.02, \sigma = 0.05)$:

$$\begin{cases} \zeta^{inter}_{0} = I \\ \zeta^{inter}_{h+1} = (\zeta^{inter}_{h} + 0.1)I \end{cases} \tag{34}$$

Let v be a task corresponding to a trip between district d_1 and d_2. We compute the perturbed start and end times are:

$$\begin{cases} \xi^{b}_{v} = t^{b}_{v} + \beta_{v} \\ \xi^{e}_{v} = \xi^{b}_{v} + t^{e}_{v} - t^{b}_{v} + \zeta^{dis}_{d_1,h(\xi_1)} + \zeta^{inter}_{h(\xi_2)} + \zeta^{dis}_{d_2,h(\xi_3)} \end{cases} \tag{35}$$

with $\xi_1 = \xi^{b}_{v}$, $\xi_2 = \xi_1 + \zeta^{dis}_{d_1,h(\xi_1)}$, and $\xi_3 = \xi_2 + t^{e}_{v} - t^{b}_{v} + \xi^{inter}_{h(\xi_2)}$. And the perturbed travel times along arcs $a = (u, v)$:

$$\xi^{tr}_{a} = \xi^{b}_{v} + t^{tr}_{a} + \zeta^{dis}_{d_1,h(\xi_1)} + \zeta^{inter}_{h(\xi_2)} + \zeta^{dis}_{d_2,h(\xi_3)} \tag{36}$$

with $\xi_1 = \xi^{e}_{u}$, $\xi_2 = \xi_1 + \zeta^{dis}_{d_1,h(\xi_1)}$, and $\xi_3 = \xi_2 + t^{tr}_{a} + \xi^{inter}_{h(\xi_2)}$.

Encoding features For an instance, we compute a matrix of 20 features for every arc of the corresponding graph. Let $a = (u, v) \in A$. The first feature is the length of a in minutes, i.e. the deterministic travel time between u and v. The second feature is equal to c_{vehicle} if a is connected to the source (i;.e if $u = o$), else 0. Then, we add the 9 deciles of $\xi^{b}_{v} - (\xi^{e}_{u} + \xi^{tr}_{a})$ which represents the slack between u and v. Finally, we add the values of the cumulative probability distribution of the slack, evaluated in -100, -50, -20, -10, 0, 50, 200, and 500.

Label solutions We label each instance with a local search heuristic initialized with the solution of the deterministic problem. Its implementation can be found on the GitHub repository[21].

[21] https://github.com/BatyLeo/StochasticVehicleScheduling.jl

B.1.4 Training plots

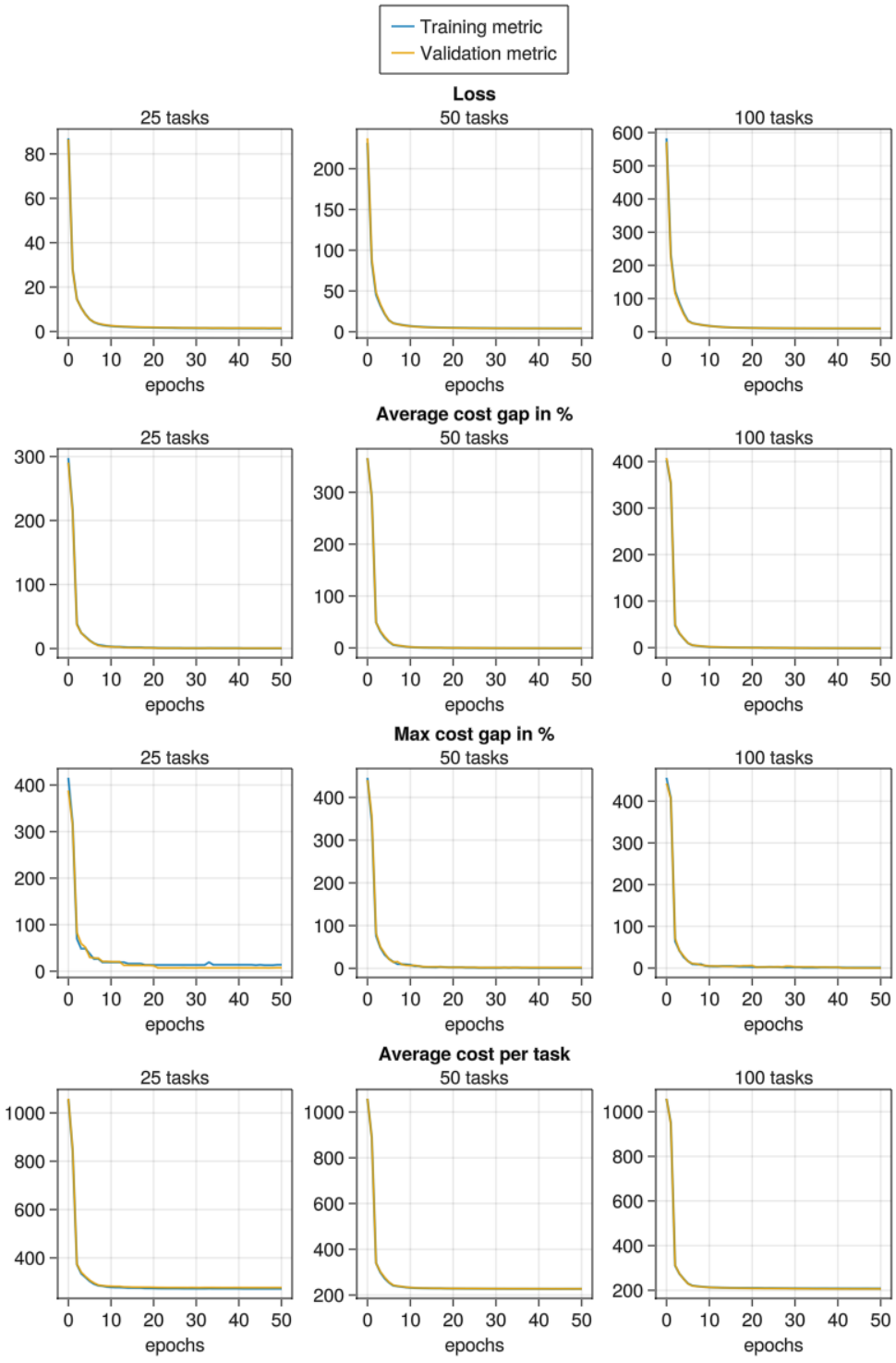


Figure 10: Learning by imitation metric evolution during learning

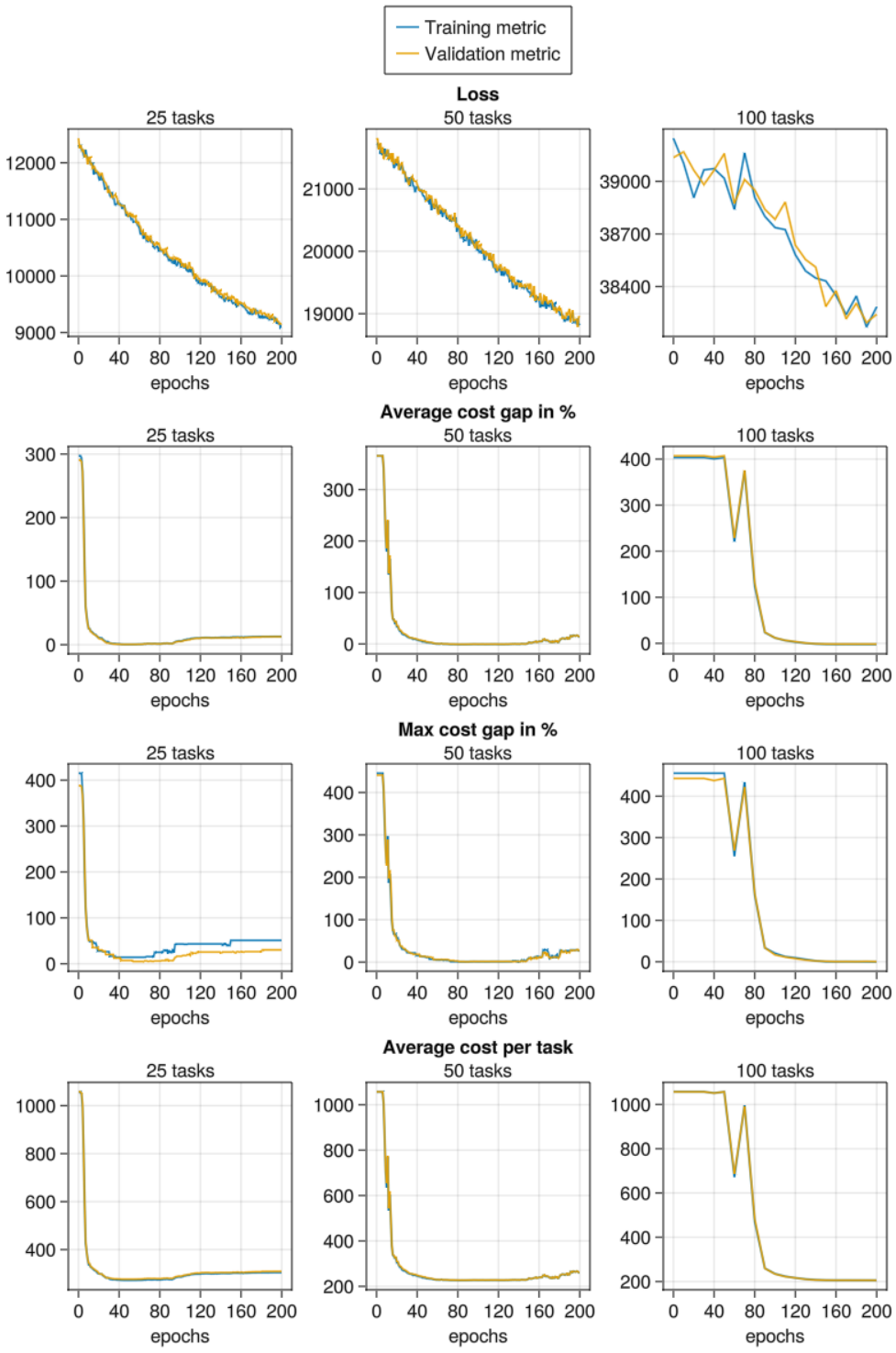


Figure 11: Learning by experience metric evolution during learning

B.2 Two-stage minimum weight spanning tree

This subsection contains additional material about the two-stage stochastic spanning tree problem.

B.2.1 Lagrangian heuristic

We present the heuristic based on Lagrangian relaxation used to label our datasets.

Lagrangian relaxation An option to compute a lower bound on the optimal value of the *two-stage minimum weight spanning tree* is to apply a Lagrangian relaxation. For this, we duplicate complicating variables y for each scenario and introduce associated θ_{es} dual variable for each new constraint:

$$\begin{array}{lll} \min & \displaystyle\sum_{e\in E} c_e y_e + \frac{1}{|S|}\sum_{e\in E}\sum_{s\in S} d_{es} z_{es} & \\ \text{s.t.} & \displaystyle\sum_{e\in E} y_e + z_{es} = |V| - 1 & \forall s \in S \\ & \displaystyle\sum_{e\in E(Y)} y_e + z_{es} \leq |Y| - 1, & \forall Y, \emptyset \subsetneq Y \subsetneq E, \forall s \in S \\ & y_{es} = y_e, & \forall e \in E, \forall s \in S \ (\theta_{es} \text{ dual variables}) \\ & y, z \in \{0,1\} & \end{array} \tag{37}$$

Lagrangian dual problem can be formulated as follows:

$$\max_{\theta} G(\theta) = \begin{cases} \min\limits_{y} & \displaystyle\sum_{e\in E}(c_e + \frac{1}{|S|}\sum_{s\in S}\theta_{es})y_e \\ \text{s.t.} & 0 \leq y \leq M \end{cases} + \frac{1}{|S|}\sum_{s\in S} \begin{cases} \min\limits_{y_s, z_s} & \displaystyle\sum_{e\in E} d_{es} z_{es} - \theta_{es} y_{es} \\ \text{s.t.} & y_s + z_s \in \text{spanning tree polytope} \end{cases}, \tag{38}$$

where M is a large constant. In theory M is not needed, we can take $M = +\infty$, but taking a finite M leads to more informative gradients. G can be maximized by gradient ascent:

$$(\nabla_\theta G(\theta))_{es} = \frac{1}{|S|}(y_e - y_{es}) \tag{39}$$

Lagrangian heuristic Once we find a good θ with the Lagrangian relaxation, we can retrieve the corresponding y_{es} solution, and use it as input of the Lagrangian heuristic 1. The general idea is to rebuild a forest from the edges that are selected in the first stage for most scenarios.

Algorithm 1: Lagrangian heuristic

```
Data: y_es ∈ {0,1}, ∀e ∈ E, ∀s ∈ S
forall e ∈ E do
    if (1/|S|) Σ_{s∈S} y_es > 0.5 then
        w_e ← 1;
    else
        w_e ← −1;
    end
end
Find maximum weight forest for weights w;
Result: y_e ∈ {0,1}, ∀e ∈ E
```

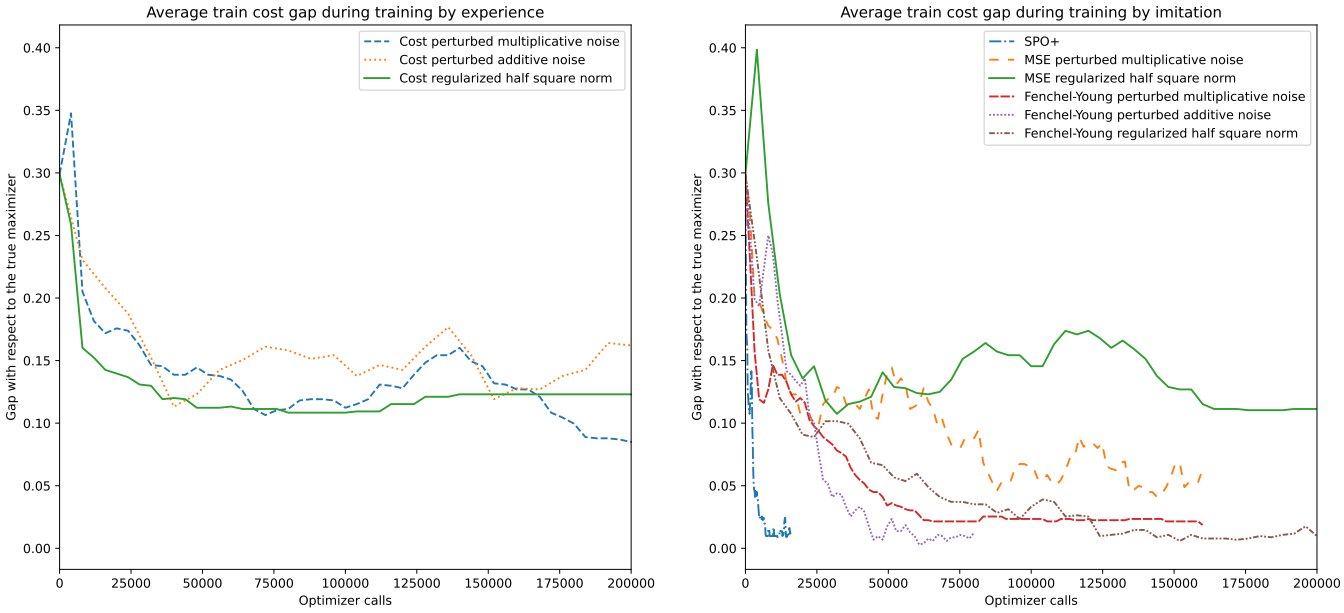
In the numerical experiments, we use the Lagrangian heuristic to build the training set. Practically, we stop the Lagrangian relaxation after 50.000 iterations or if the gap between the lower bound provided by the $G(\theta)$ and the upper bound obtained from the heuristic is non-grater than 0.1%.

B.2.2 Instance features

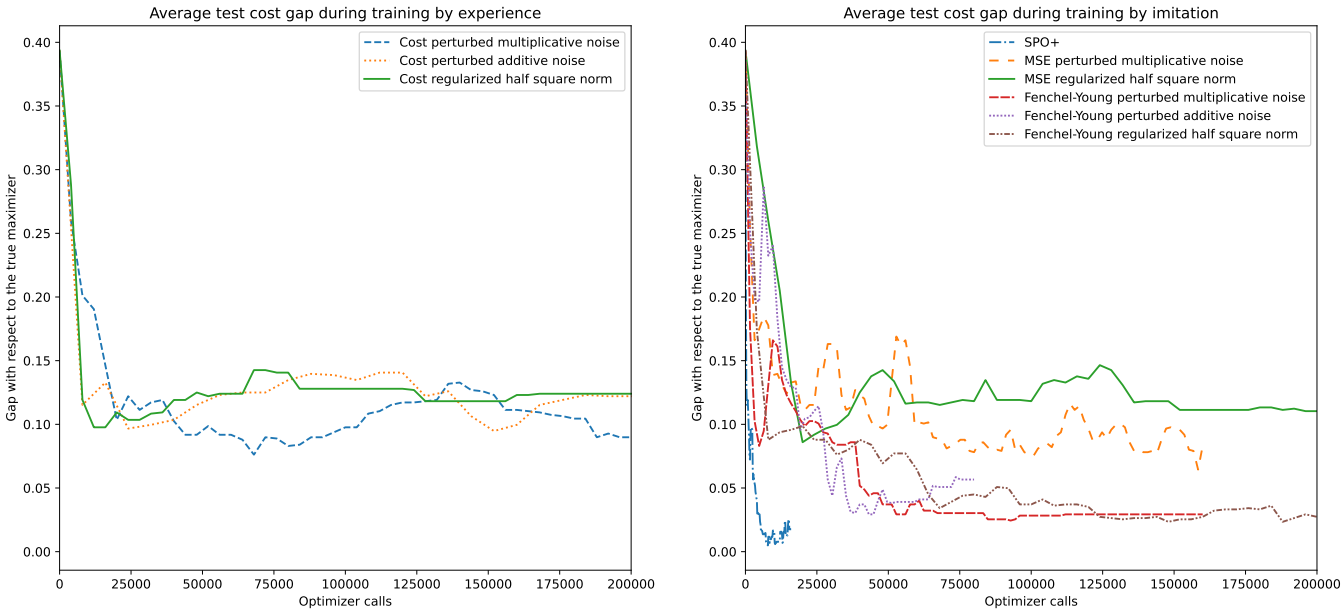
We detail here all the features used for training.

Basic feature set Basic features only contain twelve features. Let e be an edge. The first one is the first stage cost c_e, and the other 11 are the quantiles of the second stage costs $(d_{es})_{s \in S}$ for values $\{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$. All quantiles in the advanced feature set use the same values.

Advanced feature set Advanced features contain 79 features in total. Let e be an edge. The first 12 features are the same as the basic features. To these, we add quantiles of best stage costs ($i.e$ for each scenario s the value $\min(c_e, d_{es})$), quantiles of neighbors first stage costs, and quantiles of neighbors second stage costs. Then, we compute several single stage minimum spanning trees with Kruskal's algorithm: one with first stage costs, one for each scenario with corresponding second stage costs, and one for each scenario with corresponding best stage costs $\min(c_e, d_{es})$. We use these spanning trees to compute new useful features: a binary features that tells if e is in the first stage spanning tree, and quantile features on its presence in second stage trees. Finally, we use best stage spanning trees to compute quantiles for the cost of e in the best stage costs spanning tree: one set of quantiles for first edge costs, and another set for second stage costs.



(a) Average train gap



(b) Average test gap

Figure 3: Train and test optimality gaps along training in the Warcraft application