



HAL
open science

Learning with Combinatorial Optimization Layers: a Probabilistic Approach

Guillaume Dalle, Léo Baty, Louis Bouvier, Axel Parmentier

► **To cite this version:**

Guillaume Dalle, Léo Baty, Louis Bouvier, Axel Parmentier. Learning with Combinatorial Optimization Layers: a Probabilistic Approach. 2022. hal-03739485v1

HAL Id: hal-03739485

<https://hal.science/hal-03739485v1>

Preprint submitted on 27 Jul 2022 (v1), last revised 3 Dec 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Learning with Combinatorial Optimization Layers: a Probabilistic Approach

Guillaume Dalle¹, Léo Baty¹, Louis Bouvier^{1,2}, Axel Parmentier^{1,*}

¹CERMICS, Ecole des Ponts, Marne-la-Vallée, France ²Groupe Renault

July 27, 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.

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

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 (2020) 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:

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

*Corresponding author: axel.parmentier@enpc.fr

The inference problem consists in predicting an output from a given input. It is solved online, and requires the knowledge of the parameters 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 the 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).

This pipeline is exactly the one considered by Vlastelica et al. (2019) 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:

$$f : \theta \mapsto \operatorname{argmax}_{v \in \mathcal{V}} \theta^\top v \tag{2}$$

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. 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

¹<https://github.com/axelparmentier/InferOpt.jl>

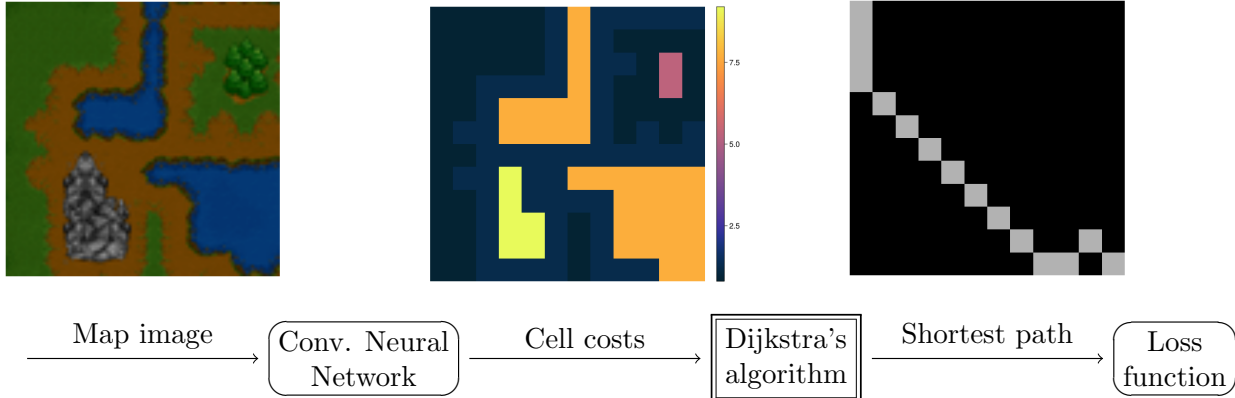


Figure 1: Pipeline for computing shortest paths on Warcraft maps – data from Vlastelica et al. (2019)

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

$$\operatorname{argmax}_{v \in \mathcal{V}} \theta^\top g(v)$$

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

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

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 on \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 mixed integer linear program (MILP) 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. Similarly, the optimum of an MILP is always reached at an integral point, or even better, 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 MILP library (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 solver 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 on 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*. Since training often relies on stochastic gradient descent (SGD), we need to be able to backpropagate loss gradients onto the weights of the ML layers. This requires each individual layer to be differentiable, so that we can compute the Jacobian of its output with respect to its input. 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 \hat{p} , such that the probability mapping $\theta \mapsto \hat{p}(\cdot|\theta)$ becomes smooth with respect to θ . If we can do that, then the expectation mapping

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

where it is understood that $V \sim \hat{p}(\cdot|\theta)$, will be just as smooth. This expectation mapping \hat{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 (\hat{p}, \hat{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 (2020), 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:

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

Our pipeline starts with an ML layer φ_w , where w stands for the vector of parameters (or 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 $y = f(\theta)$. For simplicity, we assume that f is single-valued, *i.e.* that the optimal solution is unique. Finally, the loss function \mathcal{L} is used during training to evaluate the quality of the solution. If our dataset contains N input samples $x^{(1)}, \dots, x^{(N)}$, learning involves applying SGD to the following loss minimization problem:

$$\min_w \frac{1}{N} \sum_{i=1}^N \mathcal{L} \left(\overbrace{f(\varphi_w(x^{(i)}))}^{y^{(i)}}, \dots \right). \quad (5)$$

The dots \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 target outputs $\bar{t}^{(i)}$ 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 (2021) 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. In particular, we use the pipeline of Figure 1 for learning by experience, even though the CNN encoder has tens of thousands of parameters. To the best of our knowledge, previous attempts to learn hybrid ML-CO pipelines by experience were restricted to ML layers with fewer than 100 parameters.

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{P}_p (resp. \mathbb{E}_p) to denote a probability mass function (resp. an expectation) with respect to the distribution p . If \mathcal{S} is a set, we write

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

for its convex hull and $\text{proj}_{\mathcal{S}}$ 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 $\text{dom}(h)$ 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<https://github.com/JuliaDiff/ChainRules.jl>

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. 2021). The recent Python package `theseus`⁴ showcases an application of this framework 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 generic 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.

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>

Therefore, when differentiating LPs with respect to their objective parameters, we need to resort to approximations. Vlastelica et al. (2019) 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 (2019).

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 2011), smart “predict, then optimize” (SPO+) loss (Elmachtoub and Grigas 2021) and Fenchel-Young (FY) loss (Blondel, Martins, and Niculae 2019).

2.1.4 Integer optimization layers

In theory, the methods from the previous section still work in the presence of integer variables, that is, for MILPs. To apply them, we only need to consider the polytope defined by the convex hull of integral solutions. 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 MILP 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. 2019; Berthet et al. 2020). This compatibility with arbitrary algorithms is one of the fundamental tenets of our work. The recent Python package `PyEPO`⁷ (Tang and Khalil 2022) shares this perspective, but it only implements a subset of `InferOpt.jl` since it does not address the central notion of probabilistic CO layer.

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.

⁷<https://github.com/khalil-research/PyEPO>

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.

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. 2019; Berthet et al. 2020), which was already introduced on Figure 1. The associated dataset⁸, assembled by Vlastelica et al. (2019), 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:

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

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. (2019), 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 standard 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.

⁸<https://edmond.mpd1.mpg.de/dataset.xhtml?persistentId=doi:10.17617/3.YJCQ55>

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>

```

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

```

using Graphs, GridGraphs, LinearAlgebra

function warcraft_maximizer(theta)
    g = GridGraph(-theta)
    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

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

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

```

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

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.

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 \hat{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 $\hat{p}(\cdot|\theta) \in \Delta^{\mathcal{V}}$. Second, it returns the expectation $\hat{f}(\theta) = \mathbb{E}_{\hat{p}(\cdot|\theta)}[V] \in \text{conv}(\mathcal{V})$. 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 the following conditions:

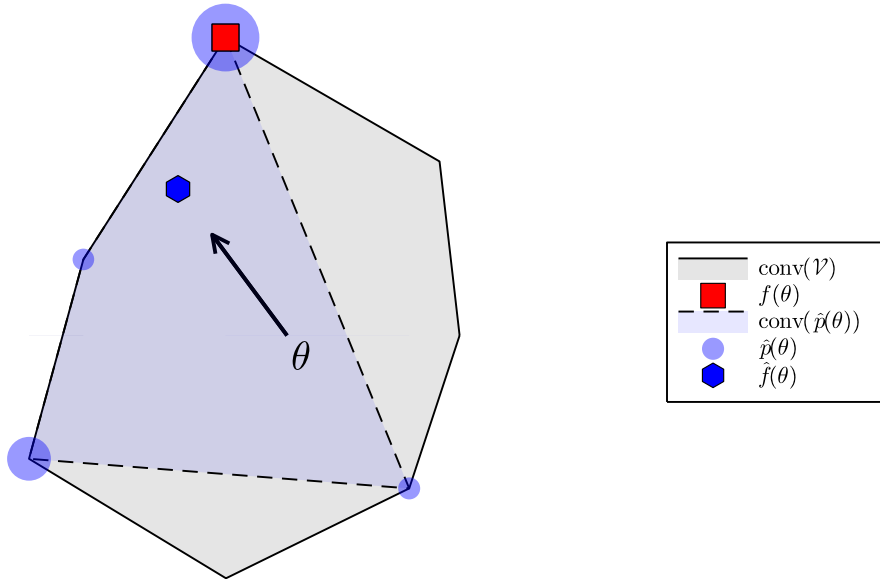


Figure 2: Effect of a probabilistic CO layer

- All computations must only require calls to the CO oracle f .
- The expectation $\mathbb{E}_{\hat{p}(\cdot|\theta)}[V]$ must be tractable (whether it is with an explicit formula, Monte-Carlo sampling, variational inference, etc.).
- The mapping $\theta \mapsto \hat{p}(\cdot|\theta)$ must be differentiable.

If these conditions are satisfied, then the Jacobian of \hat{f} is easily deduced from Equation (3):

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

Let us give an example where analytic formulas exist. If $\mathcal{V} = \{e_1, \dots, e_d\}$ is the set of basis vectors, then its convex hull $\text{conv}(\mathcal{V}) = \Delta^d$ is the unit simplex of dimension d . Given an objective direction θ , solving $\text{argmax}_{v \in \mathcal{V}} \theta^{\top} v$ yields the basis vector $f(\theta) = e_i$ where i is the index maximizing θ_i .

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 $\text{argmax}_{v \in \mathcal{V}} \theta^{\top} v$, that is, on its inner product with θ . The Boltzmann distribution is a natural candidate, leading to $\hat{p}(e_i|\theta) \propto e^{\theta^{\top} e_i} = e^{\theta_i}$. Computing the expectation reveals a well-known operation:

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

Unlike the “hardmax” function f , the softmax function \hat{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.

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

Table 1: Probabilistic CO layers and their defining features

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 2019). Instead of using the CO oracle (2), regularization solves a different problem:

$$\hat{f}_\Omega : \theta \mapsto \operatorname{argmax}_{\mu \in \operatorname{dom}(\Omega)} \theta^\top \mu - \Omega(\mu) \quad (9)$$

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

The change of notation from v to μ stresses the fact that v is an element of \mathcal{V} , while μ belongs to $\operatorname{dom}(\Omega) \subseteq \operatorname{conv}(\mathcal{V})$. 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 $\hat{f}_\Omega(\theta)$ as a convex combination of the elements of \mathcal{V} , whose weights are then interpreted as probabilities $\hat{p}_\Omega(\cdot|\theta)$. 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 $\hat{f}_\Omega(\theta) = \operatorname{softmax}(\theta)$. But the case of the unit simplex is very peculiar, because for any $\mu \in \operatorname{conv}(\mathcal{V})$, the convex decomposition of μ onto the vertices \mathcal{V} is unique. In other words, the correspondence between regularizations Ω and probability mappings \hat{p}_Ω 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. (2019). 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 $\hat{p}(\cdot|\theta)$ and $\hat{f}(\theta)$ are computed, as well as their derivatives. We also draw connections with the regularization paradigm, which ease the introduction of Fenchel-Young losses in Section 5.2.3. A hasty reader can safely skip to Section 4.

```

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

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:

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

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

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

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 $Z_1, \dots, Z_M \sim \mathcal{N}(0, I)$. 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:*

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

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 $\operatorname{dom}(\Omega_\varepsilon^+) \subset \operatorname{conv}(\mathcal{V})$ and*

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

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:

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

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:

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

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:*

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

Proof. See Appendix A.2.1. □

As far as regularization is concerned, we need a slight tweak compared to the additive case. Let F_ε^\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 Ω_ε^\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 Ω_ε^\odot . 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 $\hat{f}_\Omega(\theta)$ 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 $\hat{f}_\Omega(\theta) \in \text{conv}(\mathcal{V})$: it also defines a sparse probability distribution $\hat{p}_\Omega^{\text{FW}}(\cdot|\theta)$ over the vertices \mathcal{V} such that

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

This distribution is called sparse because most of the weights are actually zero. Note that $\hat{p}_\Omega^{\text{FW}}(\cdot|\theta)$ 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. (2021, Appendix C), there exists a function $g(p, \theta)$ defined on $\Delta^\mathcal{V} \times \mathbb{R}^d$ such that $\hat{p}_\Omega^{\text{FW}}(\cdot|\theta)$ is a fixed point of its projected gradient operator $p \mapsto \text{proj}_{\Delta^\mathcal{V}}(p - \nabla_p g(p, \theta))$. 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 $\nabla_\theta \hat{p}_\Omega(v|\theta)$ that we use to compute a Jacobian for $\hat{f}_\Omega(\theta)$. Again, by sparsity, this sum only has a few non-zero terms, which makes it tractable:

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

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:

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

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 2021).

Remark 3.2. Blondel, Martins, and Niculae (2019) also suggest distribution regularization, whereby $\Omega(\mu)$ is defined through a generalized entropy $H(p)$ on $\Delta^\mathcal{V}$:

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

Distribution regularization can only be computed explicitly for certain entropies H (Shannon entropy, Gini index) and certain polytopes $\text{conv}(\mathcal{V})$ (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>

```

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

```

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

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

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

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 $c : \mathcal{V} \rightarrow \mathbb{R}$, 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:

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

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 \hat{f} , thus minimizing $c(\hat{f}(\theta))$. 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 $\mu \in \text{conv}(\mathcal{V})$.

The solution we propose relies on the *pushforward measure* (also called image measure) of $\hat{p}(\cdot|\theta)$ with respect to the function c . Recall that a probabilistic CO layer is defined by $\hat{f}(\theta) = \mathbb{E}_{\hat{p}(\cdot|\theta)}[V]$. 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*:

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

By integration, this loss is just as smooth as the probability mapping $\theta \mapsto \widehat{p}(\cdot|\theta)$, 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 $\widehat{p}_\varepsilon^+$, $\widehat{p}_\varepsilon^\circ$, and $\widehat{p}_\Omega^{\text{FW}}$ defined in Section 3.

Note that if c is linear and defined on all of $\text{conv}(\mathcal{V})$, then $\mathbb{E}_{\widehat{p}(\cdot|\theta)}[c(V)] = c(\mathbb{E}_{\widehat{p}(\cdot|\theta)}[V])$ and the two quantities coincide. Furthermore, if c is convex, then $\mathbb{E}_{\widehat{p}(\cdot|\theta)}[c(V)] \geq c(\mathbb{E}_{\widehat{p}(\cdot|\theta)}[V])$ 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.

```
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

```
using Zygote

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

Code sample 8: Supported operations for an expected regret

```
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., non greater than 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 \hat{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:*

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

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. \square

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:

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

On the other hand, when \hat{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:

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

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:*

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

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

$$\min_w \frac{1}{N} \sum_{i=1}^N \mathcal{L} \left(f(\varphi_w(x^{(i)})), \bar{t}^{(i)} \right), \quad (18)$$

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 (2021), 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 $\|\hat{f}(\varphi_w(x)) - \bar{y}\|^2$ 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:

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

Here is another way to write it:

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

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 :

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

The following result justifies why this is an interesting loss.

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

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

Proof. As a pointwise maximum of affine functions, $\theta \mapsto \mathcal{L}(\theta, \bar{t})$ is convex. Its subgradient is obtained using Danskin's theorem (Danskin 1967). \square

The idea is that solving $\operatorname{argmax}_{y \in \text{dom}(\Omega)} \mathcal{L}^{\text{aux}}(\theta, \bar{t}, y)$ 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.

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}^{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

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.*

```
using InferOpt

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

Code sample 10: Example imitation losses

```
using Zygote

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

Code sample 11: Supported operations for an imitation loss

```
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 2011, 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:

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

The subgradient formula (21) becomes

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

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 (2021) define the SPO+ loss function as follows:

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

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:

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

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 2019). Starting from a target solution \bar{y} and a regularization Ω , a loss is constructed as follows:

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

This time, the loss and subgradient require access to \hat{f}_Ω :

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

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 $\Omega_\varepsilon^\circ(y)$ 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:

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

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.

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 $\mathcal{L}(\theta, \bar{t})$ 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

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

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 first version of the paper, we only present one application of our `InferOpt.jl` package: Warcraft shortest paths. Experiments on other concrete problems will be added to the final version.

6.1 Shortest paths on Warcraft maps

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. (2019). It contains 200 samples, which we split into 80 training samples, 100 validation samples (for hyperparameter tuning) and 20 test samples (for performance evaluation). The train, test and validation sets are the same for each learning setting. 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 2014), and `GridGraphs.jl` to compute shortest paths. Our code will soon be made available in the `WarcraftShortestPaths.jl`¹⁵ repository (which is still private at the time of writing). For each learning setting, we tune a subset of the hyperparameters stated in Table 3. The experiments are conducted on a MacBook Pro with 2,3 GHz Intel Core i9, 8 cores and 16 Go 2667 MHz DDR4 RAM.

¹⁵<https://github.com/LouisBouvier/WarcraftShortestPaths.jl>

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.
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 of 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.

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.

7 Conclusion

In this paper, we present a Julia package which allows easy insertion of CO layers into ML pipelines. Our probabilistic perspective provides a coherent framework for several approximate differentiation techniques, some of which are new. We also construct a natural loss for learning by experience, which is successfully applied to a hybrid ML-CO pipeline with tens of thousands of parameters. Finally, we propose a generic loss decomposition for learning by imitation.

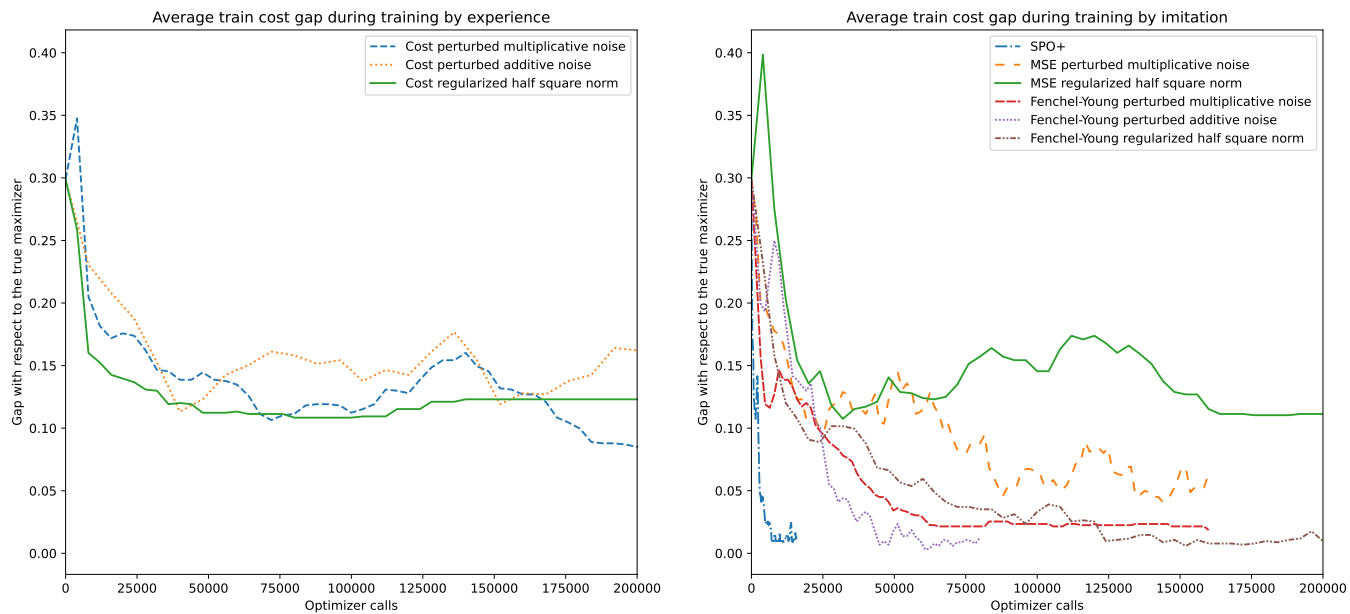
Numerical experiments using `InferOpt.jl` to solve hard combinatorial problems (such as stochastic vehicle scheduling and two-stage minimum spanning trees) will be included in the final version of the paper.

Acknowledgements

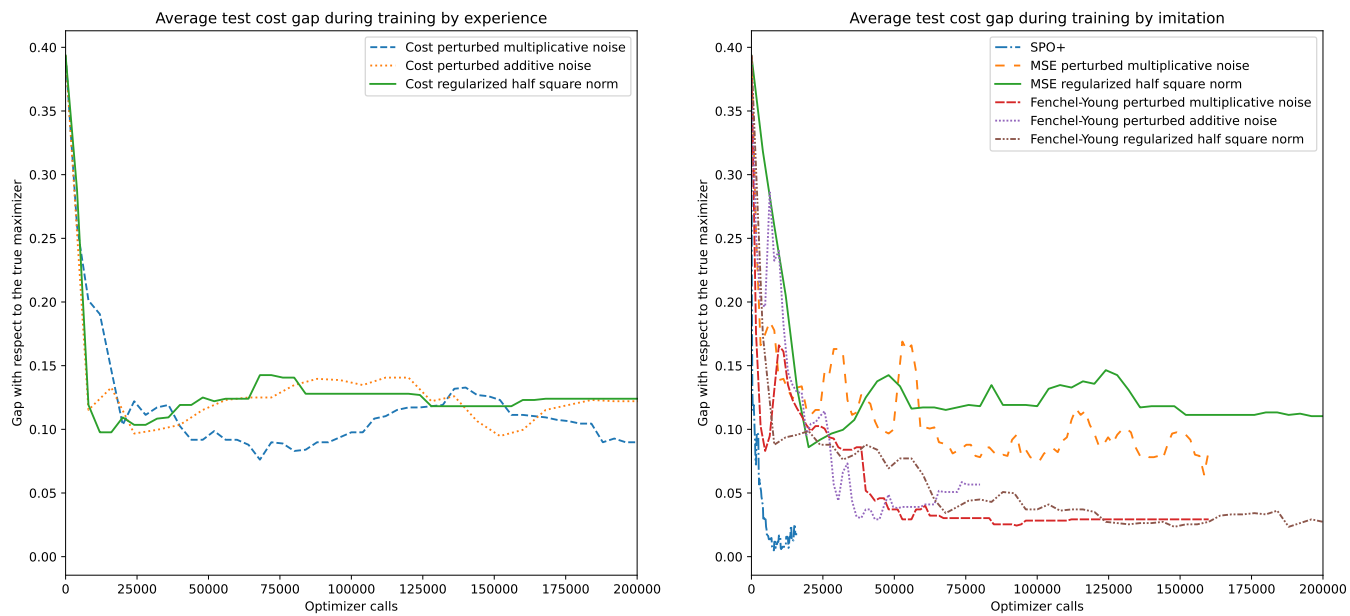
The authors want to thank Mathieu Besançon for his support with various Julia packages, as well as his expert proofreading. We also thank Éloïse Berthier, Alexandre Forel, Pierre Marion and Clément Mantoux (in alphabetical order) for their time and advice.

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://papers.nips.cc/paper/2019/hash/9ce3c52fc54362e22053399d3181c638-Abstract.html> (cit. on p. 6).
- Amos, B. and Z. Kolter (July 17, 2017). “OptNet: Differentiable Optimization as a Layer in Neural Networks”. 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 pp. 6, 7).
- Barto, A. G. and S. Mahadevan (Jan. 1, 2003). “Recent Advances in Hierarchical Reinforcement Learning”. In: *Discrete Event Dynamic Systems* 13.1, pp. 41–77. ISSN: 1573-7594. DOI: [10.1023/A:1022140919877](https://doi.org/10.1023/A:1022140919877). URL: <https://doi.org/10.1023/A:1022140919877> (cit. on p. 8).
- 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. 6).
- Bengio, Y., A. Lodi, and A. Prouvost (Aug. 8, 2020). “Machine Learning for Combinatorial Optimization: A Methodological Tour d’horizon”. In: *European Journal of Operational Research*. ISSN: 0377-2217. DOI: [10.1016/j.ejor.2020.07.063](https://doi.org/10.1016/j.ejor.2020.07.063). URL: <http://www.sciencedirect.com/science/article/pii/S0377221720306895> (cit. on pp. 1, 4).



(a) Average train gap



(b) Average test gap

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

- 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*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., pp. 9508–9519. URL: <https://proceedings.neurips.cc/paper/2020/file/6bb56208f672af0dd65451f869fedfd9-Paper.pdf> (cit. on pp. 2, 7, 8, 11, 12, 18, 20, 27–29, 31).
- Besaçon, M., A. Carderera, and S. Pokutta (Oct. 5, 2021). *FrankWolfe.Jl: A High-Performance and Flexible Toolbox for Frank-Wolfe Algorithms and Conditional Gradients*. DOI: 10.48550/arXiv.2104.06675. arXiv: 2104.06675 [math]. URL: <http://arxiv.org/abs/2104.06675> (cit. on p. 14).
- Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah (Jan. 1, 2017). “Julia: A Fresh Approach to Numerical Computing”. In: *SIAM Review* 59.1, pp. 65–98. ISSN: 0036-1445. DOI: 10.1137/141000671. URL: <https://epubs.siam.org/doi/10.1137/141000671> (cit. on p. 5).
- Blondel, M., Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert (Oct. 7, 2021). “Efficient and Modular Implicit Differentiation”. arXiv: 2105.15183 [cs, math, stat]. URL: <http://arxiv.org/abs/2105.15183> (cit. on pp. 6, 14).
- Blondel, M., A. F. T. Martins, and V. Niculae (Jan. 8, 2019). “Learning with Fenchel-Young Losses”. In: URL: <https://arxiv.org/abs/1901.02324v2> (cit. on pp. 7, 11, 14, 20).
- Blondel, M., O. Teboul, Q. Berthet, and J. Djolonga (Nov. 21, 2020). “Fast Differentiable Sorting and Ranking”. 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. 7).
- 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: 978-3-642-46092-0. DOI: 10.1007/978-3-642-46092-0. URL: <http://link.springer.com/10.1007/978-3-642-46092-0> (cit. on p. 18).
- Elmachtoub, A. N. and P. Grigas (Mar. 12, 2021). “Smart "Predict, Then Optimize"”. In: *Management Science*. ISSN: 0025-1909. DOI: 10/gnmp22. URL: <http://pubsonline.informs.org/doi/abs/10.1287/mnsc.2020.3922> (cit. on pp. 4, 7, 18, 20).
- Fairbanks, J., M. Besaçon, 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. 8).
- Ferber, A., B. Wilder, B. Dilkina, and M. Tambe (Apr. 3, 2020). “MIPaaL: Mixed Integer Program as a Layer”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.02 (02), pp. 1504–1511. ISSN: 2374-3468. DOI: 10/gn3fxx. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/5509> (cit. on p. 7).
- Frank, M. and P. Wolfe (1956). “An Algorithm for Quadratic Programming”. In: *Naval Research Logistics Quarterly* 3.1-2, pp. 95–110. ISSN: 1931-9193. DOI: 10.1002/nav.3800030109. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800030109> (cit. on p. 14).
- Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. MIT Press. URL: <http://www.deeplearningbook.org> (cit. on p. 6).
- Griewank, A. and A. Walther (Nov. 6, 2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. SIAM. 448 pp. ISBN: 978-0-89871-659-7. Google Books: [qMLUIsgCwvUC](https://books.google.com/books?id=qMLUIsgCwvUC) (cit. on p. 6).
- He, K., X. Zhang, S. Ren, and J. Sun (June 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), pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on p. 8).
- Innes, M. (Mar. 9, 2019). “Don’t Unroll Adjoint: Differentiating SSA-Form Programs”. arXiv: 1810.07951 [cs]. URL: <http://arxiv.org/abs/1810.07951> (cit. on p. 8).

- Innes, M., E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah (Nov. 10, 2018). “Fashionable Modelling with Flux”. arXiv: [1811.01457](https://arxiv.org/abs/1811.01457) [cs]. URL: <http://arxiv.org/abs/1811.01457> (cit. on p. 8).
- Innes, M. (May 3, 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](https://doi.org/10.21105/joss.00602). URL: <https://joss.theoj.org/papers/10.21105/joss.00602> (cit. on p. 8).
- Jaggi, M. (Feb. 13, 2013). “Revisiting Frank-Wolfe: Projection-Free Sparse Convex Optimization”. 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. 14).
- Jones, D. R., C. D. Perttunen, and B. E. Stuckman (Oct. 1, 1993). “Lipschitzian Optimization without the Lipschitz Constant”. In: *Journal of Optimization Theory and Applications* 79.1, pp. 157–181. ISSN: 1573-2878. DOI: [10.1007/BF00941892](https://doi.org/10.1007/BF00941892). URL: <https://doi.org/10.1007/BF00941892> (cit. on p. 16).
- Kingma, D. P. and J. Ba (Dec. 22, 2014). “Adam: A Method for Stochastic Optimization”. arXiv: [1412.6980](https://arxiv.org/abs/1412.6980) [cs]. URL: <http://arxiv.org/abs/1412.6980> (cit. on p. 21).
- Kolter, Z., D. Duvenaud, and M. Johnson (2020). *Deep Implicit Layers - Neural ODEs, Deep Equilibrium Models, and Beyond*. URL: <http://implicit-layers-tutorial.org/> (cit. on p. 6).
- Korte, B. and J. Vygen (2006). *Combinatorial Optimization: Theory and Algorithms*. 3rd ed. Algorithms and Combinatorics. Berlin Heidelberg: Springer-Verlag. ISBN: 978-3-540-29297-5. URL: <https://www.springer.com/us/book/9783540292975> (cit. on p. 3).
- Kotary, J., F. Fioretto, P. V. Hentenryck, and B. Wilder (Aug. 9, 2021). “End-to-End Constrained Optimization Learning: A Survey”. In: *Twenty-Ninth International Joint Conference on Artificial Intelligence*. Vol. 5, pp. 4475–4482. DOI: [10.24963/ijcai.2021/610](https://doi.org/10.24963/ijcai.2021/610). URL: <https://www.ijcai.org/proceedings/2021/610> (cit. on p. 1).
- Mandi, J., E. Demirović, P. J. Stuckey, and T. Guns (Apr. 3, 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. DOI: [10/gn3fwv](https://doi.org/10/gn3fwv). URL: <https://ojs.aaai.org/index.php/AAAI/article/view/5521> (cit. on p. 7).
- Mandi, J. and T. Guns (2020). “Interior Point Solving for LP-based Prediction+optimisation”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., pp. 7272–7282. URL: <https://proceedings.neurips.cc/paper/2020/file/51311013e51adebc3c34d2cc591fefee-Paper.pdf> (cit. on p. 7).
- Martins, A. F. T. and R. F. Astudillo (Feb. 8, 2016). “From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification”. arXiv: [1602.02068](https://arxiv.org/abs/1602.02068) [cs, stat]. URL: <http://arxiv.org/abs/1602.02068> (cit. on p. 14).
- Niculae, V., A. Martins, M. Blondel, and C. Cardie (July 3, 2018). “SparseMAP: Differentiable Sparse Structured Inference”. 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. 14).
- Nowozin, S. and C. H. Lampert (May 23, 2011). “Structured Learning and Prediction in Computer Vision”. In: *Foundations and Trends® in Computer Graphics and Vision* 6.3–4, pp. 185–365. ISSN: 1572-2740, 1572-2759. DOI: [10.1561/06000000033](https://doi.org/10.1561/06000000033). URL: <https://www.nowpublishers.com/article/Details/CGV-033> (cit. on pp. 7, 19).
- Parmentier, A. (July 2021a). “Learning Structured Approximations of Operations Research Problems”. URL: <https://hal.archives-ouvertes.fr/hal-03281894> (cit. on pp. 4, 16).
- (Apr. 16, 2021b). “Learning to Approximate Industrial Problems by Operations Research Classic Problems”. In: *Operations Research*. ISSN: 0030-364X. DOI: [10/gj3t75](https://doi.org/10/gj3t75). URL: <http://pubsonline.informs.org/doi/abs/10.1287/opre.2020.2094> (cit. on p. 7).

- Paulus, A., M. Rolínek, V. Musil, B. Amos, and G. Martius (May 5, 2021). “CombOptNet: Fit the Right NP-Hard Problem by Learning Integer Programming Constraints”. In: URL: <https://arxiv.org/abs/2105.02343v1> (cit. on p. 7).
- 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 (July 19, 2022). *Theseus: A Library for Differentiable Nonlinear Optimization*. DOI: [10.48550/arXiv.2207.09442](https://doi.org/10.48550/arXiv.2207.09442). arXiv: [2207.09442](https://arxiv.org/abs/2207.09442) [cs, math]. URL: <http://arxiv.org/abs/2207.09442> (cit. on p. 6).
- Sharma, A., M. Besançon, J. D. Garcia, and B. Legat (June 10, 2022). *Flexible Differentiable Optimization via Model Transformations*. DOI: [10.48550/arXiv.2206.06135](https://doi.org/10.48550/arXiv.2206.06135). arXiv: [2206.06135](https://arxiv.org/abs/2206.06135) [cs, math]. URL: <http://arxiv.org/abs/2206.06135> (cit. on p. 6).
- Sutton, R. S. and A. G. Barto (Nov. 13, 2018). *Reinforcement Learning: An Introduction*. MIT Press. 549 pp. ISBN: 978-0-262-35270-3. Google Books: [uWV0DwAAQBAJ](https://books.google.com/books?id=uWV0DwAAQBAJ) (cit. on p. 7).
- Tang, B. and E. B. Khalil (June 28, 2022). *PyEPO: A PyTorch-based End-to-End Predict-then-Optimize Library for Linear and Integer Programming*. DOI: [10.48550/arXiv.2206.14234](https://doi.org/10.48550/arXiv.2206.14234). arXiv: [2206.14234](https://arxiv.org/abs/2206.14234) [cs, math]. URL: <http://arxiv.org/abs/2206.14234> (cit. on p. 7).
- Vlastelica, M., A. Paulus, V. Musil, G. Martius, and M. Rolinek (Sept. 25, 2019). “Differentiation of Blackbox Combinatorial Solvers”. In: International Conference on Learning Representations. URL: <https://openreview.net/forum?id=BkevoJSYPB> (cit. on pp. 2, 3, 7, 8, 11, 21).
- White, F. C., M. Abbott, M. Zgubic, J. Revels, A. Arslan, S. Axen, S. Schaub, N. Robinson, Y. Ma, G. Dhingra, willtebutt, N. Heim, D. Widmann, A. D. W. Rosemberg, N. Schmitz, C. Rackauckas, R. Heintzmann, frankschae, C. Lucibello, K. Fischer, A. Robson, J. Ling, mattBrzezinski, A. Zhabinski, D. Wennberg, M. Besançon, P. Vertechi, S. Gowda, and A. Fitzgibbon (June 9, 2022). *JuliaDiff/ChainRules.Jl: V1.35.2*. Zenodo. DOI: [10.5281/zenodo.6626962](https://doi.org/10.5281/zenodo.6626962). URL: <https://zenodo.org/record/6626962> (cit. on p. 5).
- Wilder, B., B. Dilkina, and M. Tambe (July 17, 2019). “Melding the Data-Decisions Pipeline: Decision-Focused Learning for Combinatorial Optimization”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (01), pp. 1658–1665. ISSN: 2374-3468. DOI: [10/ghkjd9](https://doi.org/10/ghkjd9). URL: <https://ojs.aaai.org/index.php/AAAI/article/view/3982> (cit. on p. 7).

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 $\theta \mapsto \max_{v \in \mathcal{V}} \theta^\top v$, 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:

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

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

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

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

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

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

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

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

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

□

A.1.2 Regularization

Proof of Proposition 3.2.

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

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

We consider $\mu \notin \text{conv}(\mathcal{V})$. 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 $t > 0$,

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

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

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

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

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

Hence, we can conclude:

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

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

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:

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

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

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

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

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

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

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

We arrive at a simple variant of the previous expression:

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

□

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 don't have $\text{dom}(\Omega_{\varepsilon}^{\odot}) \subset \text{conv}(\mathcal{V})$ (unlike in the additive case). Indeed, even when $\mu \notin \text{conv}(\mathcal{V})$, 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

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

Danskin's theorem helps us compute the gradient of F_{ε}^{\odot} :

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

We could prove in a way similar to Berthet et al. (2020, Proposition 2.2) that Ω_ε^\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

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

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

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

□