Decision-Focused Learning to Predict Action Costs for Planning

Jayanta Mandi [a,*], Marco Foschini[a], Daniel Höller[b], Sylvie Thiébaux[c,d], Jörg Hoffmann[b,e] and Tias Guns[a]

[a]Department of Computer Science, KU Leuven, Belgium
[b]Saarland Informatics Campus, Saarland University, Germany
[c]LAAS-CNRS, Université de Toulouse, France
[d]Australian National University, Australia
[e]German Research Center for Artificial Intelligence (DFKI)

Abstract. In many automated planning applications, action costs can be hard to specify. An example is the time needed to travel through a certain road segment, which depends on many factors, such as the current weather conditions. A natural way to address this issue is to learn to predict these parameters based on input features (e.g., weather forecasts) and use the predicted action costs in automated planning afterward. Decision-Focused Learning (DFL) has been successful in learning to predict the parameters of combinatorial optimization problems in a way that optimizes solution quality rather than prediction quality. This approach yields better results than treating prediction and optimization as separate tasks. In this paper, we investigate for the first time the challenges of implementing DFL for automated planning in order to learn to predict the action costs. There are two main challenges to overcome: (1) planning systems are called during gradient descent learning, to solve planning problems with negative action costs, which are not supported in planning. We propose novel methods for gradient computation to avoid this issue. (2) DFL requires repeated planner calls during training, which can limit the scalability of the method. We experiment with different methods approximating the optimal plan as well as an easy-to-implement caching mechanism to speed up the learning process. As the first work that addresses DFL for automated planning, we demonstrate that the proposed gradient computation consistently yields significantly better plans than predictions aimed at minimizing prediction error; and that caching can temper the computation requirements.

1 Introduction

Automated planning generates plans aimed at achieving specific goals in a given environment. However, in real-world environments some information is hard to access and to specify directly in a model, e.g., in a transportation logistics planning domain [11], to find route-cost optimal solution requires access to travel time between cities. In today's world, features that are correlated with these unknown parameters are often available and must be leveraged for enhanced planning. For instance, travel time depends on various environmental and contextual factors like time-of-day, expected weather conditions (e.g., temperature, precipitation). Machine learning (ML) can facilitate the prediction of these parameters from those correlated features, which can then be used as parameters in a planning model.

These two steps, (1) predicting and (2) planning, can in principle be considered as two distinct tasks. For example, the approach by Weiss and Kaminka [25] involves generating a planning solution based on estimates provided by an external model, thus treating prediction and planning as separate tasks. If the prediction in step (1) is perfect, this would lead to optimal planning in step (2). However, ML predictions are not always fully accurate, for various reasons such as high uncertainty, limited features or ML model capacity, and the presence of noisy data [14].

Recent works in *decision-focused learning* (DFL) [16] for combinatorial optimization problems has shown that training the ML model to directly optimize the outcome of the downstream optimization problem, rather than prediction quality, leads to higher quality solutions. Training this way allows the ML model to focus on parts of the prediction that have (higher) impact on the actual solutions. This has been shown on various combinatorial optimization problems, e.g., shortest path [5], knapsack [15], TSP [21], portfolio optimization [6] or energy-cost aware scheduling [15].

Our interest lies in exploring whether DFL techniques can also be leveraged to produce plans of higher quality, when having to predict action costs for automated planning. To the best of our knowledge, this is the first paper on DFL for contextual action cost prediction.

Our starting point is the seminal 'Smart Predict-then-Optimize' (SPO) [5] work for predicting the coefficients of the linear objective function of a combinatorial optimization problem. We will show how this framework is applicable to planning by considering total plan cost as a weighted sum over the action counts of a plan. However, two more fundamental challenges present themselves and form the core of this paper: First, when using a machine learning system to predict action costs, one might get negative predictions, especially during training. However, for highly non-linear prediction problems this can even be the case when all of the training data has positive values. We hence propose and evaluate two ways of correcting negative predictions. During training, we additionally propose and evaluate an explicit penalty on negative values to guide the learning.

A second challenge is the computational cost of solving planning problems. In DFL, we need to call the planner for every training instance, so even with only 100 training instances you easily run into thousand planner calls and more. We hence investigate techniques

* Corresponding Author.

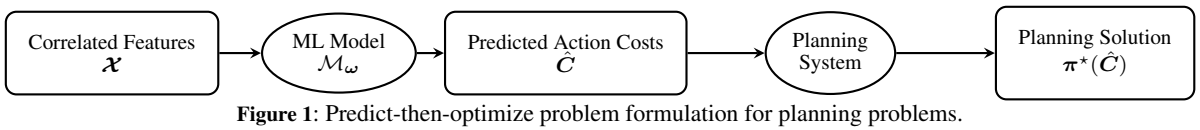


Figure 1: Predict-then-optimize problem formulation for planning problems.

from planning [13] to compute sub-optimal plans and relaxed plans. We also experiment with a solution caching approach during learning [18], as proposed in DFL for optimization problems.

We will empirically demonstrate that the proposed approach generates superior planning solutions compared to predictions aimed solely at minimizing the mean square error (MSE) of action costs. We also observe that solution caching significantly reduces training time compared to repeatedly solving the optimization problem.

2 Background

We use the STRIPS formalism [7] and define a planning problem as a tuple (P, A, s_0, g, c), where P is a set of propositions, A is a set of actions, $s_0 \subseteq P$ is the initial state, $g \subseteq P$ the goal definition, and $c : A \to \mathbb{R}_0$ is the cost function mapping each action to its (positive) costs. We identify a state $s \subseteq P$ with the set of propositions that hold in it; propositions that are not included in s are assumed to be *false*. A state s is a *goal state* if and only if $s \subseteq g$. The functions $prec$, add, and del define the precondition, add-, and delete-effects of actions. Formally, they map actions to subsets of propositions: $\{prec, add, del\} : A \to 2^P$. An action a is applicable in a state s if and only if $prec(a) \subseteq s$. When an applicable action a is applied to a state s, the resulting state s' is defined as $s' = \gamma(a, s) = (s \setminus del(a)) \cup add(a)$.

A sequence $(a_0, a_1, \ldots, a_n)$ of actions is a solution (or plan) for the problem if and only if each action a_i is applicable in the state s_i, with for $i > 0$, $s_i = \gamma(a_{i-1}, s_{i-1})$, and $s_{n+1} \subseteq g$. The cost of a solution is defined as the sum of the costs of its actions. By abuse of notation, we define the function c also on solutions. Formally, let $p = (a_0, a_1, \ldots, a_n)$ be a solution, then $c(p) = \sum_{0 \leq i \leq n} c(a_i)$. We call a solution $p^\star$ *optimal* with respect to a planning problem when there is no solution p with $c(p^\star) < c(p)$. Note that there might be multiple optimal solutions to a planning problem. In our experiments we will assume that if there exist non-unique solutions, the planner returns a single optimal solution by breaking ties in a consistent pre-specified manner.

2.1 From Planning to Learning

Predict-then-Optimize problem. In domains like travelling or delivery services, the costs of actions are hard to specify at design time, because they depend on the current situation, e.g., regarding weather or traffic. However, one can estimate these costs using contextual features that are correlated with the costs. In this case, predicting the costs using ML methods is a natural choice. When the ground truth action costs are unknown, we employ a trained ML model $\mathcal{M}_{\boldsymbol{\omega}}$ to predict the action costs from features $\boldsymbol{\mathcal{X}}$. The trainable parameters, denoted as $\boldsymbol{\omega}$, are estimated using a set of past observations, used as a training dataset for the ML model. To obtain a feasible plan in this setting, the action costs are first predicted using ML, followed by the generation of a plan optimized with respect to the predicted costs. This pipeline is commonly referred to as the *Predict-then-Optimize* problem formulation in the literature [5, 15]. We present a schematic diagram illustrating Predict-then-Optimize in the context of planning problems in Figure 1.

Vector representation. State-of-the-art ML architectures, including neural networks, represent the data in matrix and vector form. As we will be using neural networks as the predictive model, we will introduce a vector based notation of the action costs and the solution. Consider the left side of Figure 2. It shows the illustration of a simple planning problem and an optimal solution as defined before.

We create a vector representation of a plan by storing the number of times each action occurs in this plan. Since this discards the orderings of the actions in the plan, more than one plan might map to the same vector. We refer to this as the action count vector by $\boldsymbol{\pi}$. More formally, let $m = |A|$ be the number of possible actions a_i in the model, and $\overline{A} = (a_0, a_1, \ldots, a_{m-1})$ a *sequence* containing the actions of A in an arbitrary but fixed ordering. Given some plan $p = (p_0, \ldots, p_n)$, we define the action count vector $\boldsymbol{\pi} = (o_0, \ldots, o_{m-1})$ with $o_i = \sum_{j=0}^{n} \mathbb{1}(a_i = p_j)$.

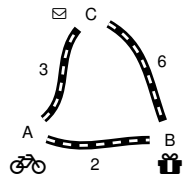
We need a similar vector representation for action costs, which we denote by $\boldsymbol{C}$. We define $\boldsymbol{C} = (c(a_0), c(a_1), \ldots, c(a_{m-1}))$, where a_i is the i^{th} element of $\overline{A}$. Hereafter, we will use $\boldsymbol{\pi}^\star(\boldsymbol{C})$ to denote the action count vector of an optimal plan with respect to $\boldsymbol{C}$. Observe that both vectors $\boldsymbol{C}$ and $\boldsymbol{\pi}^\star(\boldsymbol{C})$ have the same length. An example of $\boldsymbol{C}$, $\boldsymbol{\pi}^\star(\boldsymbol{C})$ and $\overline{A}$ is given on the right of Figure 2. With this notation, we can represent the training dataset as $\{(\boldsymbol{\mathcal{X}}_\kappa, \boldsymbol{C}_\kappa)\}_{\kappa=1}^{N}$.

Regret. In the predict-then-optimize problem, we need to distinguish between the ground truth action costs that we want to learn and the predicted action costs. We will denote them as $\boldsymbol{C}$ and $\hat{\boldsymbol{C}}$, respectively. Let $\boldsymbol{\pi}^\star(\boldsymbol{C})$ and $\boldsymbol{\pi}^\star(\hat{\boldsymbol{C}})$ be optimal action count vectors with respect to $\boldsymbol{C}$ and $\hat{\boldsymbol{C}}$ respectively. Using vector notation, the cost of executing $\boldsymbol{\pi}^\star(\boldsymbol{C})$ can be expressed as $\boldsymbol{C}^\top \boldsymbol{\pi}^\star(\boldsymbol{C})$. Importantly, when a plan that was created using the predicted costs is actually executed in practice, the actual costs, $\boldsymbol{C}$, is revealed, and the efficacy of the plan is evaluated with respect to $\boldsymbol{C}$. Hence the *real* cost of executing $\boldsymbol{\pi}^\star(\hat{\boldsymbol{C}})$ is $\boldsymbol{C}^\top \boldsymbol{\pi}^\star(\hat{\boldsymbol{C}})$, e.g. the real cost times the action count vector. The quality of a predicted cost in a predict-then-optimize problem is evaluated based on *regret*. Regret measures the difference between the realized cost of the solution, made using the predicted cost and the true optimal cost, which is obviously *not* known a priori. It can be expressed in the following form:

$$\mathbf{regret}(\hat{\boldsymbol{C}}, \boldsymbol{C}) = \boldsymbol{C}^\top \boldsymbol{\pi}^\star(\hat{\boldsymbol{C}}) - \boldsymbol{C}^\top \boldsymbol{\pi}^\star(\boldsymbol{C}) \tag{1}$$

2.2 Decision-Focused Learning

In a predict-then-optimize setup, the final goal of predicting the cost is to make a planning solution with zero or low regret. The motivation of DFL is to directly train an ML model to predict $\hat{\boldsymbol{C}}$ in a manner that minimizes regret. We are particularly interested in gradient descent training, a widely utilized method for training neural networks. In gradient descent training, the neural network is trained by computing the gradient of the loss function. Modern neural network frameworks like TensorFlow [1] and PyTorch [20] compute this gradient automatically by representing the set of all neural network layers as a *computational graph* [2]. However, in DFL, as the final loss is the regret; this would require computing the derivative the regret and hence of plan $\boldsymbol{\pi}^\star(\hat{\boldsymbol{C}})$ with respect to $\hat{\boldsymbol{C}}$. Firstly, one cannot rely on automatic differentiation to compute this derivative as the planning problem is solved outside the neural network computational graph. Moreover, the planning process is not a differentiable



p^* = (*drive-A-B*, *pickup-pack-B*, *drive-B-A*, *drive-A-C*, *drop-pack-C*, *pickup-letter-C*, *drive-C-A*, *drive-A-B*, *drop-letter-B*)

C	$\boldsymbol{\pi}$	$\overline{A}$
2	2	*drive-A-B*
3	1	*drive-A-C*
2	1	*drive-B-A*
6	0	*drive-B-C*
3	1	*drive-C-A*
6	0	*drive-C-B*
1	0	*pickup-pack-A*
1	1	*pickup-pack-B*
1	0	*pickup-pack-C*
1	0	*pickup-letter-A*
1	0	*pickup-letter-B*
1	1	*pickup-letter-C*
1	0	*drop-pack-A*
1	0	*drop-pack-B*
1	1	*drop-pack-C*
1	0	*drop-letter-A*
1	1	*drop-letter-B*
1	0	*drop-letter-C*

Figure 2: Top left: Illustration of a planning problem. The bike needs to deliver the letter to position B and the package to C. It cannot carry both at the same time. The road segments come with different costs, *pickup* and *drop* actions cost 1. An optimal solution $p^\star$ is given below. The right-hand side of the picture successively shows the action costs ($\boldsymbol{C}$), the action count vector ($\boldsymbol{\pi}^\star(\boldsymbol{C})$), and $\overline{A}$.

operation, since slight changes in action costs either do not affect the solution or change the solution abruptly, just like in combinatorial optimization [16]. So, the derivative of the planning solution is either zero or undefined.

To obtain gradients useful for DFL for different classes of optimization problems, numerous techniques have been proposed. For an extensive analysis of existing DFL techniques, we refer readers to the survey by Mandi et al. [16]. In this work, we focus on the seminal and robust 'Smart Predict then Optimize' (SPO) technique [5], which has demonstrated success in implementing DFL across various applications, including e.g. power systems [4] or antenna design [3].

Smart Predict-then-Optimize (SPO). SPO is a DFL approach which proposes a convex upper-bound of the regret. This upper-bound can be expressed in the following form, as shown below:

$$SPO_+ = \zeta(\boldsymbol{C} - 2\hat{\boldsymbol{C}}) + 2\hat{\boldsymbol{C}}^\top \boldsymbol{\pi}^\star(\boldsymbol{C}) - \boldsymbol{C}^\top \boldsymbol{\pi}^\star(\boldsymbol{C}) \tag{2}$$

where $\zeta(\boldsymbol{C}) \doteq \max_{\boldsymbol{\pi}}\{\boldsymbol{C}^\top \boldsymbol{\pi}\}$. However, to minimize the SPO_+ loss in gradient-based training, a difficulty arises because it does not have a gradient. It is easy to verify that $-\boldsymbol{\pi}^\star(\boldsymbol{C})$ is a subgradient of $\zeta(-\boldsymbol{C})$, allowing to write the following subgradient of SPO_+ loss

$$\nabla_{SPO_+} = 2(\boldsymbol{\pi}^\star(\boldsymbol{C}) - \boldsymbol{\pi}^\star(2\hat{\boldsymbol{C}} - \boldsymbol{C})) \tag{3}$$

This subgradient is used for gradient-based training in DFL.

3 DFL in the Context of Planning

A classical planning problem is essentially a compact definition of a huge graph akin to a finite automaton. The objective is to find a path in this graph from a given initial state to a goal state without explicitly building the graph. In practice, we further want not only to generate *some* plan, but one minimizing the costs of the contained actions. All existing techniques in classical planning assume that actions costs are non-negative (negative action costs constitute a very different form of problem as, in that setting, an action sequence may become cheaper when continued). This assumption presents a challenge for our DFL setting, not only because the ML model may predict costs of some actions to be negative; but also during training, there is the aspect of having to solve the planning problem with negative action costs.

3.1 Regret Evaluation in the Presence of Negative Action Costs

We highlight that while the ground truth action costs are positive, the predicted cost, returned by the ML model, might turn negative. The explanation for why this could occur is provided in Appendix A3 in the full paper's preprint version [17]. One could use a Relu activation layer to enforce the predicted costs $\hat{\boldsymbol{C}}$ to be non-negative. This can be formulated by using an element-wise $\mathbf{max}$ operator.

$$relu(\hat{\boldsymbol{C}}) = \mathbf{max}(\hat{\boldsymbol{C}}, \mathbf{0}) \tag{4}$$

We can naively use *relu*, by feeding the planning system with action costs after setting the negative ones to zero. We refer to it as *thresholding*. However, this approach may yield a subpar plan by turning all negative predictions into zeros, losing the relative ordering of negative action costs.

Next, we propose an improved method for transforming all action costs into positive values before feeding them to the planning system. Our idea is to add a scalar value to each element of the cost vector, if any element in it is negative. We implement this by adding the absolute value of the smallest action cost to the cost vector. For a given $\hat{\boldsymbol{C}}$, it can be computed as follows:

$$\underline{c} = \left| \min(0, \min(\hat{\boldsymbol{C}})) \right| \tag{5}$$

where $\min(\hat{\boldsymbol{C}})$ is the minimum value in the cost vector $\hat{\boldsymbol{C}}$. Eq. (5) ensures that if all the elements in $\hat{\boldsymbol{C}}$ are positive, the value of $\underline{c}$ is 0. The action count vector obtained after this transformation, can be expressed in the following form:

$$\boldsymbol{\pi}^\star_{min+}(\hat{\boldsymbol{C}}) = \boldsymbol{\pi}^\star(\hat{\boldsymbol{C}} + \underline{\mathrm{c}}) \tag{6}$$

where $\underline{\mathrm{c}}$ is defined in Eq. (5). We refer to this approach as *add-min*. We will evaluate which among these two approaches would be suitable for evaluating regret in the presence of negative action costs.

3.2 Training in the Presence of Negative Action Costs

The second challenge is associated with *training* an ML model in the DFL paradigm. As mentioned before, DFL involves computing the planning solution with the predicted action costs during the training of the ML model. As the action costs might turn negative, it also requires finding a planning solution over negative action costs. We emphasise that while training with the SPO method, $(2\hat{\boldsymbol{C}} - \boldsymbol{C})$ can turn negative, even if we ensure that both $\hat{\boldsymbol{C}}$ and $\boldsymbol{C}$ are positive.

During evaluation, as we mentioned earlier, our aim is to create a planning solution with negative action costs. However, this objective differs during training. In the DFL paradigm, the primary focus of solving the planning problem during training is to produce a useful gradient for training. This concept is reflected in Equation 3; where the gradient does not include a planning solution for the predicted action cost $\hat{\boldsymbol{C}}$; rather, it considers a solution for $(2\hat{\boldsymbol{C}} - \boldsymbol{C})$, as it yields a suitable gradient for training.

It is obvious that the thresholding or add-min approach introduced for evaluation of regret can also be used while training. Computing the SPO subgradient using thresholding would result in the following:

$$\nabla^{relu}_{SPO_+} = 2\big(\boldsymbol{\pi}^\star(\boldsymbol{C}) - \boldsymbol{\pi}^\star_{relu}(2\hat{\boldsymbol{C}} - \boldsymbol{C})\big) \tag{7}$$

On the other hand, computing the SPO subgradient through the add-min approach would yield the following:

$$\nabla^{min^+}_{SPO_+} = 2\big(\boldsymbol{\pi}^\star(\boldsymbol{C}) - \boldsymbol{\pi}^\star_{min+}(2\hat{\boldsymbol{C}} - \boldsymbol{C})\big) \tag{8}$$

Note that we do not have to change $\boldsymbol{\pi}^\star(\boldsymbol{C})$ as it does not have any negative element in it.

3.3 Explicit Training Penalty

We highlight that when training the ML model using Eq. (7) or Eq. (8), the conversion of negative action costs to non-negative ones occurs outside the gradient computational graph. Consequently, the ML model does not receive feedback indicating the necessity of such corrective measures before computing the regret. This limitation motivates us to explore alternative gradient computation techniques that not only make the ML model aware of the need for such corrective actions but also has no impact when there are no negative predictions.

We propose to add a penalty function in the loss function if any element of the vector $(2\hat{\boldsymbol{C}} - \boldsymbol{C})$ is negative.

$$SPO_{+\mathcal{P}} = SPO_{+} + \lambda\, \mathbf{1}^\top relu(\boldsymbol{C} - 2\hat{\boldsymbol{C}}) \tag{9}$$

where, λ signifies the weight assigned to the penalty function, $\mathbf{1}$ denotes a vector of ones with the same dimension as $\boldsymbol{C}$. So, $\mathbf{1}^\top relu(\boldsymbol{C} - 2\hat{\boldsymbol{C}})$ is the sum of all non-zero elements in $\boldsymbol{C} - 2\hat{\boldsymbol{C}}$. In this formulation, $\mathbf{1}^\top relu(\boldsymbol{C} - 2\hat{\boldsymbol{C}})$ will be zero only if $2\hat{c}(a_i) < c(a_i)$ for all actions a_i. In Eq. (9), the second term can be viewed as a regularizer that penalizes predicting $2\hat{\boldsymbol{C}} < \boldsymbol{C}$, as for such predictions we have to make the transformation of the cost vector before feeding it to the planning system. To train with the $SPO_{+\mathcal{P}}$ loss, we can use the subgradient $\nabla^{relu}_{SPO_+}$ (7) or $\nabla^{min^+}_{SPO_+}$ (8) for the SPO_+ part; we will denote the respective loss functions as $SPO_{+\mathcal{P}}^{relu}$ and $SPO_{+\mathcal{P}}^{min^+}$.

3.4 From Loss to Gradient Computation

The subgradient of $SPO_{+\mathcal{P}}$ in Eq. (9) can be expressed in the following form:

$$\nabla_{SPO_{+\mathcal{P}}} = \nabla_{SPO_+} - 2\,\lambda\, \boldsymbol{I}_{<0}(2\hat{\boldsymbol{C}} - \boldsymbol{C}) \tag{10}$$

We use an indicator function $\boldsymbol{I}_{<0}$, which outputs a vector with elements equal to 1 for actions a_i if $2\hat{c}(a_i) < c(a_i)$. For instance, if we use $\nabla^{min^+}_{SPO_+}$ as the SPO_+ subgradient, $\nabla^{min^+}_{SPO_{+\mathcal{P}}}$ takes the following form:

$$\begin{aligned}
\nabla^{min^+}_{SPO_{+\mathcal{P}}} &= 2\big(\boldsymbol{\pi}^\star(\boldsymbol{C}) - \boldsymbol{\pi}^\star_{min^+}(2\hat{\boldsymbol{C}} - \boldsymbol{C})\big) - 2\,\lambda\, \boldsymbol{I}_{<0}(2\hat{\boldsymbol{C}} - \boldsymbol{C}) \\
&= 2\Big(\boldsymbol{\pi}^\star(\boldsymbol{C}) - \big(\boldsymbol{\pi}^\star_{min^+}(2\hat{\boldsymbol{C}} - \boldsymbol{C}) + \lambda\, \boldsymbol{I}_{<0}(2\hat{\boldsymbol{C}} - \boldsymbol{C})\big)\Big) \\
&= 2\Big(\boldsymbol{\pi}^\star(\boldsymbol{C}) - \check{\boldsymbol{\pi}}^\star(2\hat{\boldsymbol{C}} - \boldsymbol{C})\Big)
\end{aligned} \tag{11}$$

where $\check{\boldsymbol{\pi}}^\star(2\hat{\boldsymbol{C}} - \boldsymbol{C})$ is defined as follows:

$$\check{\boldsymbol{\pi}}^\star(2\hat{\boldsymbol{C}} - \boldsymbol{C}) = \boldsymbol{\pi}^\star_{min^+}(2\hat{\boldsymbol{C}} - \boldsymbol{C}) + \lambda\, \boldsymbol{I}_{<0}(2\hat{\boldsymbol{C}} - \boldsymbol{C}) \tag{12}$$

Using the SPO methodology, we can use Eq. (11) as a subgradient to minimize $SPO_{+\mathcal{P}}^{min^+}$.

The vector $\check{\boldsymbol{\pi}}^\star(2\hat{\boldsymbol{C}} - \boldsymbol{C})$ increments the count of any action a_i where $2\hat{c}(a_i) < c(a_i)$ by 1 after obtaining a solution with add-min. In other words, these actions are executed once more. In this way it penalizes for the need to correct the predicted cost by selecting any action a_i for which $2\hat{c}(a_i) < c(a_i)$. We highlight that there might be no solution to the original planning problem corresponding to the vector representation $\check{\boldsymbol{\pi}}^\star(\hat{\boldsymbol{C}})$. Since we added actions apart from the solution returned by the planning system, there might not even be an executable permutation of the actions represented in the vector.

Intuitive interpretation of the subgradient. The motivation behind introducing the subgradient formulation (11) is that we can associate an intuitive interpretation to it. The intuition behind the subgradient (11) is that the action count vector $\check{\boldsymbol{\pi}}^\star(2\hat{\boldsymbol{C}} - \boldsymbol{C})$ increases the count of action a_i if $2\hat{c}(a_i) < c(a_i)$; which makes the corresponding elements in the subgradient vector $2(\boldsymbol{\pi}^\star(\boldsymbol{C}) - \check{\boldsymbol{\pi}}^\star(2\hat{\boldsymbol{C}} - \boldsymbol{C}))$ negative. As the ML model is updated using the opposite of the subgradient, the corresponding action costs are increased in the next iteration. So, in this way we incentivize the model to avoid predicting $2\hat{c}(a_i) < c(a_i)$.

4 Scaling up DFL for Planning Problems

As reported by Mandi et al. [16], DFL comes with substantial computational costs. This is due to the fact that DFL requires solving the planning problem with the predicted (action) costs while training the underlying ML model. This means that we need to solve a planning problem repeatedly, which is computationally expensive. This computational burden poses a significant challenge in applying DFL to real-world planning problems, often resulting in long training times. In this section, we present some strategies to tackle this crucial issue.

4.1 Use of Planning Techniques to Expedite Training

As DFL involves repeatedly solving the planning problem during training, one strategy to expedite training is to use planning techniques without optimality guarantees or even solutions to relaxed planning problems (as usually done when computing planning heuristics). The advantage of this is that it is easier and faster to solve. Although such solutions may not be identical to the optimal ones, they can still provide a useful gradient (11). Note that the gradient computation in DFL is computed across a batch of training instances. In such cases, the exact optimal solution with the predicted action costs might not be necessary to determine the direction of the gradient update. A non-optimal solution, *reasonably close to the true solution*, often provides a good gradient direction and suffices for gradient computation.

For integer linear problems (ILPs), Mandi et al. [15] observed that solving their linear relaxation is sufficient for obtaining informative DFL gradients. In planning, we have several options towards approximating the optimal solution: we can either use planning algorithms that are bounded optimal, those without optimally guarantee, or even use solutions to relaxed planning problems. This leads us to the following settings, where both plan quality and computational effort decrease:

- opt – Use an optimal planning system to get an optimal solution.
- $bound_n$ – Use an algorithm that guarantees a solution not worse than n times the optimal plan.
- $no\text{-}bound$ – Use a planning system without optimality guarantees.
- h – Return a solution to a relaxation of the planning problem as usually done to compute heuristics in planning.

In our experiments, we use an A^* search and the admissible LM-Cut heuristic [12] for optimal planning (opt). For bounded optimal planning ($bound_n$), we combined LM-Cut with a weighted A^* search. In the latter setting, the heuristic value is multiplied with a factor, which practically leads to finding solutions more quickly, but comes at the cost of losing the guarantee of finding an optimal solution. However, solutions are guaranteed to be bounded optimal.

For planning without optimality guarantees ($no\text{-}bound$), using a non-admissible heuristic is usually the better option to find plans

more quickly. In our experiments, we combine a Greedy Best First Search (GBFS) with the h^{FF} heuristic [13]. h^{FF} internally finds a solution to the so-called delete-relaxed (DR) planning problem, which ignores the delete effects in the original planning problem. This simplifies the problem and makes it possible to find a solution in polynomial time (while finding the *optimal* solution is still NP-hard). The heuristic estimate is then the costs of the solution to the DR problem.

For the last option (h), we need to choose a heuristic that computes not only a heuristic value, but also a relaxed plan, because we need one to compute the gradient as discussed above. Since h^{FF} internally computes a DR solution, it is well suited for our setting and we can use the DR solution as well as the heuristic estimate computed by the h^{FF} heuristic for our learning process.

4.2 Use of Solution Caching to Expedite Training

As shown by Mulamba et al. [18], an alternative approach to tackle the scalability of DFL is to replace solving an optimization problem with a *cache* lookup strategy, where the cache is a set of feasible solutions and acts as an inner approximation of the convex-hull of feasible solutions. How this cache is formed is crucial to the success of this approach.

Mulamba et al. [18] propose to keep all the solutions in the training data in the cache. Moreover, as the predicted action costs may deviate significantly from true action costs, particularly in early training stages, their solutions may be different from the solutions in the training instances. To address this, they solve the problem for a percentage, $p\%$, of predicted action costs and include the corresponding solutions in the cache as well. Hence, this approach reduces the computational burden by a margin of $p\%$. They report that keeping p as low as 5% is often sufficient for DFL training. We will implement implement this approach by caching action count vectors and investigate whether such a solution caching approach would speed up training without compromising the quality of decisions.

5 Experimental Evaluation

In this section, we first describe our benchmark set and the system setup. We come to the results afterwards. The code and data have been made publicly available [1].

5.1 Experimental Setup

5.1.1 Benchmark Set

For our experiments we need domains with meaningful action costs that have impact on solution quality (otherwise we will not be able to measure the impact of our methods). Further, to have a wide range of solving techniques available we want to stay in the standard classical planning (i.e., non-temporal) setting. We use a problem generator to generate problems of different sizes. In the Rovers domain, meeting these requirements required some adjustments. Next, we detail the domains, their source, and (if necessary) modifications we made.

Shortest path. This domain models a $n \times n$ grid environment an agent needs to navigate through. Each node is connected to its top and right nodes. The objective is to find a path starting from the bottom left cell to the top right cell with minimal costs. This domain is particularly interesting for our experiments, because it is a widely used benchmark in DFL [5, 16, 24]. In these works, the problem is solved using an LP solver. We include this to have a direct comparison to existing DFL methods.

Transport. In this domain we use the standard domain and generator [23] from the generator repository[2]. Each transport problem instance revolves around the task of delivering p number of packages using t number of trucks. We consider a $n \times n$ grid for the transport problem, within which both pickup and delivery operations occur. We denote each transport problem instance as n-p-t, signifying that the grid is of $n \times n$ dimension, with p representing the number of packages and t indicating the available truck count.

Rovers. This domain describes the task of a fleet of Mars rovers, each equipped with a (probably different) set of sensors. They need to navigate and gather data (e.g. rock samples or pictures). The data then needs to be send to the lander. Our domain is based on the one from the 2006 International Planning Competition. However, the domains from the different competition tracks did not directly fit our needs: *MetricTime* contains durative actions, *Propositional* and *QualitativePreferences* do not include action costs. We created a model based on the *MetricSimplePreferences* track and made the preferences normal goals. To get integer costs, we multiplied the included action costs by 10 and rounded them afterwards to integers.

For our domains, we generated two groups of problem instances: small-sized instances that can be solved within 0.25 seconds, and large-sized instances that take 0.5–1 seconds to solve. For more details, please refer to the Appendix section in the full paper's preprint version [17].

5.1.2 Generation of Training Data

While we adopt the planning problems from planning benchmark domains, we synthetically generate the action costs. Such synthetic data generation processes are common in DFL literature. We follow the synthetic data generation process exactly as done by Elmachtoub and Grigas [5]. We generate a set of pairs of features and action costs $\{(\boldsymbol{\mathcal{X}}_\kappa, \boldsymbol{C}_\kappa)\}_{\kappa=1}^{N}$ for training and evaluation. The dimension of $\boldsymbol{C}_\kappa$ is equal to the number of actions, $|A|$, which is specific to the planning problem. The dimension of $\boldsymbol{\mathcal{X}}_\kappa$ is 5. and each $\boldsymbol{\mathcal{X}}_\kappa$ is sampled from a multivariate Gaussian distribution with zero mean and unit variance, i.e., $\boldsymbol{\mathcal{X}}_\kappa \sim \mathbf{N}(0, I_5)$ (I_5 is a 5×5 identity matrix). To set up a mapping from $\boldsymbol{\mathcal{X}}_\kappa$ to $\boldsymbol{C}_\kappa$, first, a matrix $B \in \mathbb{R}^{|A| \times 5}$ is constructed, and then $\boldsymbol{C}_\kappa$ is generated according to the following formula:

$$c^\kappa(a_i) = \left[\left(\frac{1}{\sqrt{p}} (B \boldsymbol{\mathcal{X}}_\kappa) + 3 \right)^{Deg} + 1 \right] \xi_\kappa^i \tag{13}$$

where $c^\kappa(a_i)$ is the the cost of action i in instance κ, the parameter *Deg* parameter signifies the extent of *model misspecification*, and ξ_κ^i is a multiplicative noise term sampled randomly from the uniform distribution. Note that the action costs generated in this manner are always *positive* if *Deg* is a even number. Furthermore, since the action costs are random numbers sampled from a continuous distribution, it is highly improbable that two feasible plans will have exactly identical execution costs. Therefore, in this scenario, we do *not* encounter the phenomenon of multiple non-unique solutions.

Elmachtoub and Grigas [5] use a *linear model* to predict the cost vector from features. The higher the value of *Deg*, the more the true relation between the features and action costs deviates from the linear model and the larger the errors of the linear predictive model. Such model misspecification is a common phenomenon in ML, because

[1] https://github.com/ML-KULeuven/DFLPredict-Action-Costs-for-Planning

[2] https://github.com/AI-Planning/pddl-generators

Table 1: Evaluation of learning based on optimal plans (opt) for small-size problem instances. We report **percentage regret.**

	Shortest Path		Transport				Rovers		
	SP-5	SP-10	5-1-1 (a)	5-1-1 (b)	5-2-1 (a)	5-2-1 (b)	Rovers1	Rovers2	Rovers3
MSE	9.37 ± 0.17	12.55 ± 0.11	9.38 ± 0.08	7.43 ± 0.18	7.74 ± 0.07	8.59 ± 0.04	4.18 ± 0.01	4.68 ± 0.01	1.45 ± 0.01
${SPO_+}^{relu}$	27.82 ± 5.21	39.76 ± 2.29	16.27 ± 0.98	10.49 ± 1.81	10.99 ± 0.84	14.15 ± 2.34	7.16 ± 0.92	12.1 ± 0.57	2.21 ± 0.23
${SPO_+}^{min^+}$	8.13 ± 0.16	9.63 ± 0.09	8.9 ± 0.2	7.67 ± 0.42	7.72 ± 0.44	9.19 ± 0.36	5.35 ± 0.32	4.95 ± 0.24	$\mathbf{1.2 \pm 0.03}$
${SPO_+}_{\mathcal{P}}^{relu}$	$\mathbf{8.07 \pm 0.09}$	$\mathbf{9.16 \pm 0.03}$	8.0 ± 0.11	6.03 ± 0.18	5.33 ± 0.15	$\mathbf{6.75 \pm 0.13}$	$\mathbf{3.94 \pm 0.12}$	4.18 ± 0.1	1.22 ± 0.01
${SPO_+}_{\mathcal{P}}^{min^+}$	8.12 ± 0.14	9.41 ± 0.17	$\mathbf{7.93 \pm 0.06}$	$\mathbf{5.97 \pm 0.15}$	$\mathbf{5.07 \pm 0.07}$	6.86 ± 0.05	4.02 ± 0.1	$\mathbf{4.13 \pm 0.09}$	1.21 ± 0.04

the in practise the data generation process is not observable. In our experiments, we will report result with *Deg* being 4.

5.1.3 Planning and Learning Setup

Similarly to Elmachtoub and Grigas [5], we will also use a linear model to predict the action costs from features. We use PyTorch [20] to implement the linear predictive model and train it by minibatch stochastic gradient descent [8, 22]. The gradient is backpropagated for training the model using PyTorch's automatic differentiation. As a planning tool, we use the Fast Downward (FD) planning system [10] and run the algorithms described in Section 4.1. For small-size planning problems, we generated 400, 100 and 400 training, validation and test instances. For large-size planning problems, these values are 200, 25 and 25 respectively.

5.2 Results

In this section, we will present key insights from our empirical evaluation. After training the ML model, we report *percentage regret* on the test data, which is computed as follows:

$$\frac{1}{N_{test}} \sum_{\kappa=1}^{N_{test}} \frac{\boldsymbol{C}_\kappa^\top \boldsymbol{\pi}^\star(\hat{\boldsymbol{C}}_\kappa) - \boldsymbol{C}_\kappa^\top \boldsymbol{\pi}^\star(\boldsymbol{C}_\kappa)}{\boldsymbol{C}_\kappa^\top \boldsymbol{\pi}^\star(\boldsymbol{C}_\kappa)}. \tag{14}$$

For each set of experiments, we run 5 experiments each time with different seeds and report average and standard deviation of percentage regret in the tables. We confirmed that all the models converge within 20 epochs. We report results after the 20$^{\text{th}}$ epoch.

Table 2: Evaluation on Shortest path problem instances trained using LP solver with and without *relu*. We report **percentage regret.**

	Without Relu	With Relu
	SP-5	
MSE	$\mathbf{9.37 \pm 0.17}$	13.8 ± 0.61
SPO_+	$\mathbf{8.26 \pm 0.15}$	10.89 ± 0.42
	SP-10	
MSE	$\mathbf{12.55 \pm 0.11}$	15.79 ± 0.17
SPO_+	$\mathbf{9.44 \pm 0.13}$	12.89 ± 0.57

Table 3: Comparison between *add-min* and *thresholding* regret of models trained using LP solver for the shortest path solver. We report their **deviations from the regret evaluated using an LP solver**.

	Thresholding Difference	Add-min Difference
	SP-5	
MSE	0.38 ± 0.32	$\mathbf{0.0 \pm 0.01}$
SPO_+	10.55 ± 1.18	$\mathbf{0.0 \pm 0.01}$
	SP-10	
MSE	-0.74 ± 0.17	$\mathbf{0.02 \pm 0.04}$
SPO_+	14.68 ± 1.65	$\mathbf{0.01 \pm 0.02}$

5.2.1 Evaluating Regret for Planning Problems

RQ1: Does training with a *relu* activation layer impact regret? One can enforce the predictions to be non-negative by adding *relu* as a final activation layer. However, when the predictive model does not fully represent the data generation process, imposing non-negativity constraint using *relu* may distort the predictions resulting in higher prediction as well as decision errors. To investigate whether using *relu* affects the regret, we consider the Shortest path problem, which is a widely used benchmark in DFL.

As this is a shortest path problem over a directed *acyclic* graph, negative action costs cannot lead to loops and degenerate behaviour. Hence, we can obtain the true *optimal* solution even in the presence of negative action costs using an LP solver. In this experiment, for both training and evaluation, we use Gurobi LP solver [9]. We observe in Table 2 that for both MSE and SPO_+, the regret increases as we use *relu* activation layer. From this we conclude, we are better-off without the *relu* activation layer.

RQ2: How to evaluate regret given that planning system does not allow negative costs? As we will not be using *relu* activation layer in the final layer, the predictions generated by the ML model can turn negative, even though the groundtruth action costs are positive. As action costs with negative values, are not supported by a planner; we will be using *thresholding* (4) or *add-min* (6) to solve the planning problem with negative predicted action costs. We again consider the shortest path problem. This time we again use the LP solver for training. However, during evaluation, we compute regret using both the LP solver and a planner, allowing to compare the true regret with the regret obtained by a planner. We want to find out which method, *thresholding* or *add-min*, gives a regret measure closest to the LP regret. We see in Table 3 that *add-min* regret demonstrates greater fidelity to true LP regret. Note that *thresholding* regret shows significant deviations, particularly evident SPO_+ . Hence in our latter experiments, **we will use *add-min* regret** to evaluate the regret of predicted action costs. With the evaluation protocol set, we now focus on DFL learning methods.

5.2.2 Training With and Without Explicit Penalty

RQ3: How do the proposed SPO subgradients perform? After comparing with related work on the *Shortest path* domain, we evaluate our methods on the *Transport* and *Rovers* domain known from the planning literature. So, in this case, we use the FD planner for DFL training as well as evaluation. We seek to answer whether adding the explicit training penalty results in lower regret. As DFL training requires repeatedly solving a planning problem for every training instance, we restrict ourselves to planning problems that are fast to solve. We consider small-size planning problems, which can be solved quite fast (within 0.25 seconds). In an earlier stage, we experimented with different integer λ values and found that $\lambda = 1$ resulted in the lowest regret. A higher λ increases the influence of the penalty in the final loss (9), reducing the impact of SPO+ loss.

We report the result in Table 1. ${SPO_+}^{relu}$ loss performs very poorly, as its regret is much higher than MSE. This is due to the

Table 4: Evaluation of models trained with different planning techniques without optimality guarantees with $SPO_+{}^{min^+}_{\mathcal{P}}$ for large-size problem instances. We report **percentage regret** and **training time of 20 epochs in seconds**. We highlight those which have lower regret than MSE.

Problem		MSE	opt A^* with LM-Cut	$bound_n$ $WA^*(2)$ with LM-Cut	$no\text{-}bound$ GBFS with h^{FF}	h h^{FF} del. relaxed plan
Transport Problem						
5-3-1	Regret	5.84 ± 0.26	$\mathbf{4.19 \pm 0.4}$	6.06 ± 0.8	9.2 ± 0.9	8.06 ± 0.41
	Training Time	350	4800	1700	700	250
5-2-2	Regret	14.15 ± 0.0	$\mathbf{11.4 \pm 0.89}$	$\mathbf{12.43 \pm 0.73}$	$\mathbf{13.12 \pm 1.23}$	$\mathbf{13.1 \pm 1.01}$
	Training Time	350	9050	5050	800	200
10-1-1	Regret	12.99 ± 0.17	$\mathbf{12.16 \pm 0.8}$	$\mathbf{12.55 \pm 1.34}$	16.86 ± 1.39	15.65 ± 0.89
	Training Time	100	3650	3550	700	100
Rovers Problem						
Rovers4	Regret	2.69 ± 0.05	$\mathbf{2.3 \pm 0.15}$	2.78 ± 0.15	3.66 ± 0.49	4.97 ± 0.27
	Training Time	250	9300	1550	700	200
Rovers5	Regret	2.92 ± 0.09	$\mathbf{2.91 \pm 0.25}$	3.8 ± 0.73	5.36 ± 0.44	5.76 ± 0.21
	Training Time	300	10300	850	700	200

Table 5: Evaluation of $SPO_+{}^{min^+}_{\mathcal{P}}$ trained with optimal plans (opt) and caching $p = 10\%$ and 20% for large-size problem instances. We report **percentage regret** and **training time of 20 epochs in seconds**. We highlight those which have lower regret than MSE.

	MSE		opt		Caching($p = 10\%$)		Caching($p = 20\%$)	
Problem	Regret	Time	Regret	Time	Regret	Training Time	Regret	Time
Transport Problem								
5-3-1	5.84 ± 0.26	350	$\mathbf{4.19 \pm 0.4}$	4800	5.85 ± 0.75	800	$\mathbf{4.7 \pm 0.52}$	1050
5-2-2	14.15 ± 0.0	350	$\mathbf{11.4 \pm 0.89}$	9050	$\mathbf{11.03 \pm 1.57}$	900	$\mathbf{11.07 \pm 1.31}$	1550
10-1-1	12.99 ± 0.17	100	$\mathbf{12.16 \pm 0.8}$	3650	14.5 ± 1.28	450	$\mathbf{12.07 \pm 1.1}$	800
Rovers Problem								
Rovers4	2.69 ± 0.05	250	$\mathbf{2.3 \pm 0.15}$	9300	2.72 ± 0.34	1050	$\mathbf{2.29 \pm 0.22}$	2000
Rovers5	2.95 ± 0.0	300	$\mathbf{2.81 \pm 0.05}$	10300	3.55 ± 0.18	1250	$\mathbf{2.92 \pm 0.38}$	2300

fact that turning negative costs to zero without considering their values causes loss of information. $SPO_+{}^{min^+}$ performs much better. However, even in some cases its regret is higher than MSE. On the other hand, $SPO_+{}^{relu}_{\mathcal{P}}$ and $SPO_+{}^{min^+}_{\mathcal{P}}$, which add explicit training penalty if $2\hat{c}(a_i) < c(a_i)$ for a action a_i, are able to improve $SPO_+{}^{relu}$ and $SPO_+{}^{min^+}$. It is interesting to note that the difference between $relu$ and min^+ is insignificant after adding explicit training penalty. This experiment suggests both $SPO_+{}^{relu}$ and $SPO_+{}^{min^+}$ are effective DFL approaches for predicting action costs in planning problems.

5.2.3 Optimal Planning Versus Non-Optimal Planning

RQ4: Can we use non-optimal planning for DFL training? As DFL requires solving the planning problem repeatedly while training, which creates a considerable computational burden when challenging planning problems are considered. Hence we seek to answer whether we can utilize non-optimal planning algorithms in DFL.

To investigate this, we consider larger problem instances (solving such instance requires between 0.5 and 1.5 seconds). We train each time with $SPO_+{}^{min^+}_{\mathcal{P}}$ loss; but non-optimal planning and plans for DR planning problems. In Table 4, we observe that DFL training with $no\text{-}bound$ and h results in considerably higher regret. The regret of $bound_n$ is higher than opt, but mostly lower than $no\text{-}bound$ and h. However, in most cases its regret is higher than MSE regret, which is not desirable.

5.2.4 Optimal Planning versus Solution Caching

RQ5: Can we use solution caching to speed up training? Next we investigate whether solution caching, as implemented by Mulamba et al. [18] in the context of DFL for optimization, is effective for planning problems too. We initialize the cache with all the solutions present in the training data. We experiment with $p = 10\%$ and 20%. We can see in Table 5, the training time of caching faster compared to opt due to they solve the planning is solved for only $p\%$ of instances using the predicted action costs. While $p = 10\%$ does not consistently outperform MSE regret, $p = 20\%$ produces regret lower than MSE for all instances. This indicates for large planning instances, use of solution caching with $p = 20\%$ could prove to be a useful approach.

6 Conclusion

In this work, we investigated for the first time how we can use techniques from DFL in the planning domain. More specifically for the case of predicting action costs from correlated features and historic data, we showed how the SPO technique, which places no assumptions on the solver used, can also be used for planning problems. Other DFL techniques which work with a black-box solver [21, 19] are now equally applicable.

We proposed an implementation of DFL which accounts for the fact that planners do not support negative action costs. Our findings suggest that imposing non-negativity through relu leads to an increase in regret, both for model trained with MSE and DFL loss. Moreover, training with an explicit penalty for correcting negative action costs before solving the planning problem yields significant improvements. Indeed, our DFL approach always leads to lower regret than when training to minimize MSE of predicted action costs. While using sub-optimal plans did not consistently lead to lower-than-MSE regret, a moderate amount of caching was able to reduce computation cost significantly.

Future work includes reducing computational costs further, as well as DFL for state-dependent action cost prediction or other action components; for which SPO and related techniques is insufficient.

Acknowledgements

This research received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101070149, project Tuples. Jayanta Mandi is supported by the Research Foundation Flanders (FWO) project G0G3220N.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283. USENIX Association, 2016.

[2] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

[3] Z. Chai, K.-K. Wong, K.-F. Tong, Y. Chen, and Y. Zhang. Port selection for fluid antenna systems. *IEEE Communications Letters*, 26(5):1180–1184, 2022.

[4] X. Chen, Y. Yang, Y. Liu, and L. Wu. Feature-driven economic improvement for network-constrained unit commitment: A closed-loop predict-and-optimize framework. *IEEE Transactions on Power Systems*, 37(4): 3104–3118, 2021.

[5] A. N. Elmachtoub and P. Grigas. Smart "predict, then optimize". *Management Science*, 68(1):9–26, 2022.

[6] A. M. Ferber, B. Wilder, B. Dilkina, and M. Tambe. MIPaaL: Mixed Integer Program as a layer. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pages 1504–1511. AAAI Press, 2020.

[7] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4): 189–208, 1971.

[8] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[9] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

[10] M. Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246, 2006.

[11] M. Helmert. On the complexity of planning in transportation domains. In *Proceedings of the 6th European Conference on Planning (ECP)*, pages 120–126. AAAI, 2014.

[12] M. Helmert and C. Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI press, 2009.

[13] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302, 2001.

[14] E. Hüllermeier and W. Waegeman. Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods. *Machine Learning*, 110:457–506, 2021.

[15] J. Mandi, E. Demirovic, P. J. Stuckey, and T. Guns. Smart predict-and-optimize for hard combinatorial optimization problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pages 1603–1610. AAAI Press, 2020.

[16] J. Mandi, J. Kotary, S. Berden, M. Mulamba, V. Bucarey, T. Guns, and F. Fioretto. Decision-focused learning: Foundations, state of the art, benchmark and future opportunities, 2023.

[17] J. Mandi, M. Foschini, D. Holler, S. Thiebaux, J. Hoffmann, and T. Guns. Decision-focused learning to predict action costs for planning, 2024. URL https://arxiv.org/abs/2408.06876.

[18] M. Mulamba, J. Mandi, M. Diligenti, M. Lombardi, V. Bucarey, and T. Guns. Contrastive losses and solution caching for predict-and-optimize. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2833–2840. IJCAI organization, 2021.

[19] M. Niepert, P. Minervini, and L. Franceschi. Implicit mle: Backpropagating through discrete exponential family distributions. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 14567–14579. Curran Associates, Inc., 2021.

[20] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[21] M. V. Pogancic, A. Paulus, V. Musil, G. Martius, and M. Rolínek. Differentiation of blackbox combinatorial solvers. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*. OpenReview, 2020.

[22] H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[23] J. Seipp, Á. Torralba, and J. Hoffmann. PDDL generators. https://doi.org/10.5281/zenodo.6382173, 2022.

[24] B. Tang and E. B. Khalil. PyEPO: A PyTorch-based end-to-end predict-then-optimize library for linear and integer programming. *arXiv preprint arXiv:2206.14234*, 2022.

[25] E. Weiss and G. A. Kaminka. Planning with multiple action-cost estimates. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 427–437. AAAI Press, 2023.