Noname manuscript No.
(will be inserted by the editor)

PyEPO: A PyTorch-based End-to-End Predict-then-Optimize Library for Linear and Integer Programming

Bo Tang · Elias B. Khalil

Received: date / Accepted: date

Abstract In deterministic optimization, it is typically assumed that all parameters of the problem are fixed and known. In practice, however, some parameters may be a priori unknown but can be estimated from historical data. A typical predict-then-optimize approach separates predictions and optimization into two stages. Recently, end-to-end predict-then-optimize has become an attractive alternative. In this work, we present the *PyEPO* package, a *PyTorch*-based end-to-end predict-then-optimize library in Python. To the best of our knowledge, *PyEPO* (pronounced like *pineapple* with a silent "n") is the first such generic tool for linear and integer programming with predicted objective function coefficients. It provides two base algorithms: the first is based on the convex surrogate loss function from the seminal work of Elmachtoub and Grigas [14], and the second is based on the differentiable black-box solver approach of Pogančić et al. [33]. *PyEPO* provides a simple interface for the definition of new optimization problems, the implementation of state-of-the-art predict-then-optimize training algorithms, the use of custom neural network architectures, and the comparison of end-to-end approaches with the two-stage approach. *PyEPO* enables us to conduct a comprehensive set of experiments comparing a number of end-to-end and two-stage approaches along axes such as prediction accuracy, decision quality, and running time on problems such as Shortest Path, Multiple Knapsack, and the Traveling Salesperson Problem. We discuss some empirical insights from these experiments which

Bo Tang
Department of Mechanical and Industrial Engineering, University of Toronto
5 King's College Rd, Toronto, ON M5S 3G8
E-mail: botang@mie.utoronto.ca

Elias B. Khalil
Department of Mechanical and Industrial Engineering, University of Toronto
5 King's College Rd, Toronto, ON M5S 3G8
E-mail: khalil@mie.utoronto.ca

could guide future research. *PyEPO* and its documentation are available at `https://github.com/khalil-research/PyEPO`.

Keywords Data-driven optimization · Mixed integer programming · Machine learning

Mathematics Subject Classification (2020) 90-04, 90C11, 62J05

1 Introduction

Predictive modeling is ubiquitous in real-world decision-making. For instance, in many applications, the objective function coefficients of the optimization problem, such as travel time in a routing problem, customer demand in a delivery problem, and assets return in portfolio optimization, are unknown at the time of decision making. In this work, we are interested in the commonly used paradigm of prediction followed by optimization in the context of linear programs or integer linear programs, two widely applicable modeling frameworks. Here, it is assumed that a set of features describe an instance of the optimization problem. A regression model maps the features to the (unknown) objective function coefficients. A deterministic optimization problem is then solved to obtain a solution. Due to its wide applicability and simplicity compared to other frameworks for optimization under uncertain parameters, the predict-then-optimize paradigm has received increasing attention in recent years.

One natural idea is to proceed in two stages, first training an accurate predictive model, then solving the downstream optimization problem using predicted coefficients. While a perfect prediction would yield an optimal decision, learning a model without errors is impracticable. Bengio [5], Ford et al. [17], and Elmachtoub and Grigas [14] reported that training a predictive model based on prediction error leads to worse decisions than directly considering decision error. Thus, the state-of-art alternative is to integrate optimization into prediction, taking into account the impact on the decision, the so-called end-to-end learning framework.

End-to-end predict-then-optimize requires embedding an optimization solver into the model training loop. Classical solution approaches for linear and integer linear models, including graph algorithms, linear programming, integer programming, constraint programming, etc., are well established and efficient in practice. In addition, commercial solvers such as *Gurobi* [19] and *CPLEX* [9] are highly-optimized and enable users to easily turn business or academic problems into optimization models without a deep understanding of theory and algorithms. However, embedding a solver for end-to-end learning requires additional computation and integration (for example, gradient calculation) that current software does not provide.

On the other hand, the field of machine learning has witnessed tremendous growth in recent decades. In particular, breakthroughs in deep learning have led to remarkable improvements in several complex tasks. As a result, neural networks now pervade disparate applications spanning computer vision, natural language, and planning, among others. Python-based machine learning frameworks such as *Scikit-Learn* [32], *TensorFlow* [1], *PyTorch* [31], *MXNet* [8], etc., have been developed and extensively used for research and production needs. Although deep learning has proven highly effective in regression and classification, it lacks the ability to handle constrained optimization such as integer linear programming.

Since Amos and Kolter [4] first introduced a neural network layer for mathematical optimization, there have been some prominent attempts to bridge the gap between optimization solvers and the deep learning framework. The critical component is typically a differentiable block for optimization tasks. With a differentiable optimizer, neural network packages enable the computation of gradients for optimization operations and then update predictive model parameters based on a loss function that depends on decision quality.

While research code implementing a number of predict-then-optimize training algorithms have been made available for particular classes of optimization problems and/or predictive models [12, 34, 15, 24, 14, 33, 24, 11, 3, 2], there is a dire need for a generic end-to-end learning framework, especially for linear and integer programming. In this paper, we propose the open-source software package *PyEPO* which aims to customize and train end-to-end predict-then-optimize for linear and integer programming. Our contributions are as follows:

1. We implement `SPO+` ("Smart Predict-then-Optimize+") loss [14], and `DBB` (differentiable black-box) solver [33], which are two typical end-to-end methods for linear and integer programming.
2. We build *PyEPO* based on *PyTorch*. As one of the most popular deep learning frameworks, *PyTorch* makes it easy to use and integrate any deep neural network.
3. We provide interfaces to the Python-based optimization modeling frameworks *GurobiPy* and *Pyomo*. Such high-level modeling languages allow non-specialists to formulate optimization models with *PyEPO*.
4. We enable parallel computing for the forward pass and backward pass in *PyEPO*. Optimizations in training are carried out in parallel, allowing users to harness multiple processors to reduce training time.
5. We present new benchmark datasets for end-to-end predict-then-optimize, allowing us to compare the performance of different approaches.
6. We conduct and analyze a comprehensive set of experiments for end-to-end predict-then-optimize. We compare the performance of different methods, hyperparameters, and neural network architectures on a number of datasets. We show the competitiveness of end-to-end learning, the surprising effect of hyperparameter tuning, and potential benefits and issues when using deep neural networks. A number of empirical findings are reported to support new research directions within this topic.

2 Related work

In early work on the topic, Bengio [5] introduced a differentiable portfolio optimizer and suggested that direct optimization with financial criteria has better performance in neural networks compared to the mean squared error of predicted values. Kao et al. [22] trained a linear regressor with a convex combination of prediction error and decision error, but they only considered unconstrained quadratic programming. More recently, the interest has been in constrained optimization problems which represent much of the real-world applications. With the success of deep learning, predict-then-optimize research has adopted gradient-based methods. A comparison of methodologies is presented in Table 1 and Table 2. One notable piece of work that does not employ gradients is that of the predict-then-optimize decision tree of Elmachtoub et al. [13].

2.1 Gradients of optimal solutions via the KKT Conditions

Gradient-based end-to-end learning requires well-defined, useful first-order derivatives of an optimal solution with respect to the cost vector. The KKT conditions become an attractive option, because they make the optimization problem with hard constraints differentiable.

Amos and Kolter [4] proposed `OptNet`, which derives gradients of constrained quadratic programs from the KKT conditions. Based on `OptNet`, Donti et al. [12] investigated a general end-to-end framework, `DQP`, for learning with constrained quadratic programming, which improved decision-making over two-stage models. Although linear programming is a special case of quadratic programming, `DQP` has no ability to tackle linear objective functions because the gradient is zero almost everywhere and undefined otherwise.

Subsequently, Wilder et al. [34] extended `DQP` into linear programming by adding a small quadratic term to make it second-order differentiable, resulting in `QPTL`. Wilder et al. [34] also discussed the relaxation and rounding for the approximation of binary problems. Further, Ferber et al. [15] followed up on `QPTL` with `MIPaaL`, a cutting-plane approach to support (mixed) integer programming. With the cutting-plane method, `MIPaaL` generates (potentially exponentially many) valid cuts to convert a discrete problem into an equivalent continuous problem, which is theoretically sound for combinatorial optimization but extremely time-consuming. In addition, Mandi and Guns [24] introduced `IntOpt` based on the interior-point method, which computes gradients for linear programming with log-barrier term instead of the quadratic term of the `QPTL`. Except for `MIPaaL` [15], end-to-end learning approaches for (mixed) integer programming use the linear relaxation during training but evaluate with optimal integer solutions at test time.

Besides `DQP` and its extension, Agrawal et al. [2] introduced the "differentiable convex optimization layers" method and package, `CvxpyLayers`. Compared to `DQP`, `CvxpyLayers` is applicable to a wider range of convex optimization problems. The central idea in `CvxpyLayers` is to canonicalize disciplined convex programming as conic programming, and then use implicit differentiation [3] based on the primal-dual form and KKT conditions.

However, these KKT-based methods require a solver for either quadratic or conic programming. For linear programming, the solver efficiency of the above implementations is not comparable to commercial MILP solvers such as *Gurobi* [19] and *CPLEX* [9]. Furthermore, they do not naturally support discrete optimization due to non-convexity.

Method	In PyEPO	w/ Constr	w/ Unk Constr	Disc Var	Lin Obj	Quad Obj
`DQP` [12] [code]	✗	✓	✓	✗	✗	✓
`QPTL` [34] [code]	✗	✓	✗	✗	✓	✓
`MIPaaL` [15]	✗	✓	✗	✓	✓	✗
`IntOpt` [24] [code]	✗	✓	✗	✗	✓	✗
`CvxpyLayers` [2] [code]	✗	✓	✓	✗	✓	✓
`SPO+` [14] [code]	✓	✓	✗	✓	✓	✗
`SPO+ Rel` [25] [code]	✓	✓	✗	✓	✓	✗
`SPO+ WS` [25] [code]	✗	✓	✗	✓	✓	✗
`DBB` [33] [code]	✓	✓	✗	✓	✓	✗
`PEYL` [6] [code]	✗	✓	✗	✓	✓	✗

Table 1: Methodology Comparison

This is a comparison diagram for different methodologies.
The first set of methods uses the KKT conditions, and the second part is based on differentiable approximations.
"In PyEPO" denotes whether the method is available in *PyEPO*.
"w/ Constr" denotes whether the optimization problem includes constraints.
"w/ Unk Constr" denotes whether unknown parameters occur in constraints.
"Disc Var" denotes whether the method supports integer variables.
"Lin Obj" denotes whether the method supports a linear objective function.
"Quad Obj" denotes whether the method supports a quadratic objective function.

2.2 Differentiable approximations as an alternative to KKT

Since the KKT conditions may not be ideal for linear programming, researchers have also explored designing gradient approximations. For example, Elmachtoub and Grigas [14] proposed regret (SPO loss in their paper) to measure decision error. Since the regret loss suffers from non-convexity and discontinuity of linear programming, they then developed a method `SPO+`, in which a convex and sub-differentiable loss guaranteed that a useful subgradient could be computed and used to guide training. Same as previous approaches, `SPO+` solves an optimization problem in each forward pass. In contrast to the KKT-based methods, `SPO+` is limited to the linear objective function. Because optimization is the computational bottleneck of `SPO+`, Mandi et al. [25] utilized `SPO+` on combinatorial problems by applying relaxation as well as warm starting. As Mandi et al. [25] reported, the usage of relaxation reduced the solving time at the cost of performance.

On the other hand, Pogančić et al. [33] computed a subgradient from continuous interpolation of linear objective functions, an approach they referred to as the "differentiable black-box solver", `DBB`. The interpolation approximation was non-convex but also avoided vanishing gradients. Compared to `SPO+`, `DBB` requires an extra optimization problem for the backward pass, and the loss of `DBB` is flexible (the Hamming distance in their paper). In addition, Berthet et al. [6] applied stochastic perturbations via adding random noise to the cost vector so that a nonzero expected derivative of linear programming can be obtained. As the key algorithms of *PyEPO*, `SPO+` and `DBB` are further discussed in Section 3.

2.3 Software for Predict-then-Optimize

Research code. Table 1 lists the codebases that will follow here along with their links. Amos and Kolter [4] developed a *PyTorch*-based solver `qpth` for `OptNet`, which was used to solve a quadratic program and compute its derivatives efficiently. The solver was based on an efficient primal-dual interior-point method [27] and can solve batches of quadratic programs on a GPU. Using this solver, Donti et al. [12] provided an open-source repository to reproduce the `DQP` experiments; the repository was specifically designed for the inventory, power scheduling, and battery storage problems. Wilder et al. [34] provided code for `QPTL` for budget allocation, bipartite matching, and diverse recommendation. `MIPaaL` [15] relied on `qpth` and used *CPLEX* to generate cutting planes, but there is no available open-source code. Mandi and Guns [24] released `IntOpt` code for knapsack, shortest path and power scheduling. Berthet et al. [6] contributed the *TensorFlow*-based `PEYL` implementation, which provided universal functions for end-to-end predict-then-optimize but required users to create additional helper methods for tensors operations. Other approaches, including `SPO+` [14, 25], `DBB` [33], have open-source code. Elmachtoub and Grigas [14] provided an implentation of `SPO+` in Julia, which contained the shortest path and portfolio optimization, while Mandi et al. [25] implemented `SPO+` with Python the knapsack and power scheduling. The solvers of the shortest path, traveling salesperson, ranking, perfect matching, and graph matching are available for the `DBB` [33] library.

Except `PEYL`, the above contributions provided solutions to specific optimization problems, and `PEYL` was not wrapped up as a generic library. In conclusion, they were confined to research-grade code for purposes of reproducibility.

Software packages. `CvxpyLayers` [2] is the first generic end-to-end predict-then-optimize learning framework. In contrast to the above codes, it requires modeling with a domain-specific language *CVXPY*, which is embedded into a differentiable layer in a straightforward way. The emergence of cvxpylayers provides a more powerful tool for academia and industry. However, the solver of `CvxpyLayers` cannot compete with commercial solvers on efficiency, especially for linear and integer linear programming. Since end-to-end training requires repeated optimization in each iteration, the inefficiency of the solver becomes a bottleneck. In addition, the nature of `CvxpyLayers` means that it cannot support training with integer variables, which limits applicability to many real-world decision-making problems.

Method	Computation per Gradient
`DQP` [12]	GPU-based primal-dual interior-point method for quadratic programming
`QPTL` [34]	GPU-based primal-dual interior-point method for quadratic programming
`MIPaaL` [15]	Cutting-plane method + GPU-based primal-dual interior-point method for quadratic programming
`IntOpt` [24]	GPU-based primal-dual interior-point method for quadratic programming
`CvxpyLayers` [2]	GPU-based primal-dual interior-point method for conic programming
`SPO+` [14]	Linear/integer programming
`SPO+ Rel` [25]	Linear programming
`SPO+ WS` [25]	Integer programming with warm starting
`DBB` [33]	Two Linear/integer programming solves
`PEYL` [6]	Monte-Carlo method, multiple linear/integer programming solves with random noise

Table 2: Computational cost per gradient calculation for different methodologies.

3 Preliminaries

3.1 Definitions and Notation

For the sake of convenience, we define the following linear programming problem without loss of generality, where the decision variables are $\boldsymbol{w} \in \mathbb{R}^d$ and all $w_i \geq 0$, the cost coefficients are $\boldsymbol{c} \in \mathbb{R}^d$, the constraint coefficients are $\boldsymbol{A} \in \mathbb{R}^{k \times d}$, and the right-hand sides of the constraints are $\boldsymbol{b} \in \mathbb{R}^k$:

$$\begin{aligned} \min_{\boldsymbol{w}} \quad & \boldsymbol{c}^T \boldsymbol{w} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{w} \leq \boldsymbol{b} \\ & \boldsymbol{w} \geq \boldsymbol{0} \end{aligned} \tag{1}$$

When some variables w_i are restricted to be integers, we obtain a (mixed) integer program:

$$\begin{aligned} \min_{\boldsymbol{w}} \quad & \boldsymbol{c}^T \boldsymbol{w} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{w} \leq \boldsymbol{b} \\ & \boldsymbol{w} \geq \boldsymbol{0} \\ & w_i \in \mathbb{Z} \quad \forall i \in D^{'}, \quad D^{'} \subseteq \{1, 2, ..., d\} \end{aligned} \tag{2}$$

For both linear and integer programming, let S be the feasible region, $z^*(\boldsymbol{c})$ be the optimal objective value with respect to cost vector $\boldsymbol{c}$, and $\boldsymbol{w}^*(\boldsymbol{c}) \in W^*(\boldsymbol{c})$ be a particular optimal solution derived from some solver. We define the optimal solution set $W^*(\boldsymbol{c})$ because there may be multiple optima.

As mentioned before, some coefficients are unknown and must be predicted before optimizing. Here we assume that only the cost coefficients of the objective function $\boldsymbol{c}$ are unknown but they correlate with a feature vector $\boldsymbol{x} \in \mathbb{R}^p$. Let $\hat{\boldsymbol{c}}$ be a prediction of the cost coefficient vector $\boldsymbol{c}$. Given a training dataset $\mathcal{D} = \{(\boldsymbol{x}_1, \boldsymbol{c}_1), (\boldsymbol{x}_2, \boldsymbol{c}_2), ..., (\boldsymbol{x}_n, \boldsymbol{c}_n)\}$, one can train a machine learning predictor $g(\cdot)$ to minimize a loss function $l(\cdot)$, where $\boldsymbol{\theta}$ is the vector of predictor parameters and $\hat{\boldsymbol{c}} = g(\boldsymbol{x}; \boldsymbol{\theta})$ is the predicted cost vector.

3.2 The Two-Stage Method

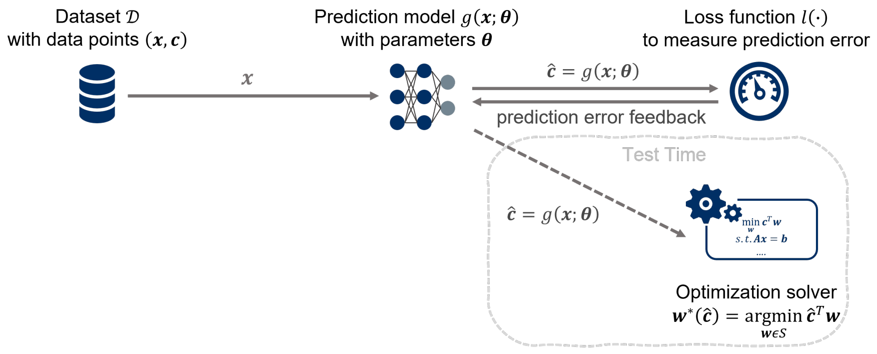

Dataset $\mathcal{D}$
with data points $(\boldsymbol{x}, \boldsymbol{c})$
Prediction model $g(\boldsymbol{x}; \boldsymbol{\theta})$
with parameters $\boldsymbol{\theta}$
Loss function $l(\cdot)$
to measure prediction error
x
$\hat{\boldsymbol{c}} = g(\boldsymbol{x}; \boldsymbol{\theta})$
prediction error feedback
Test Time
$\hat{\boldsymbol{c}} = g(\boldsymbol{x}; \boldsymbol{\theta})$
$\min_{\boldsymbol{w}} \boldsymbol{c}^T \boldsymbol{w}$
$s.t. \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$
....
Optimization solver
$\boldsymbol{w}^*(\hat{\boldsymbol{c}}) = \underset{\boldsymbol{w} \in S}{\text{argmin}}\, \hat{\boldsymbol{c}}^T \boldsymbol{w}$

Fig. 1: Illustration of the two-stage predict-then-optimize framework: A labeled dataset $\mathcal{D}$ of $(\boldsymbol{x}, \boldsymbol{c})$ pairs is used to fit a machine learning predictor that minimizes prediction error. At test time (grey box), the predictor is used to estimate the parameters of an optimization problem, which is then tackled with an optimization solver. The two stages are thus completely separate.

As Figure 1 shows, the two-stage approach trains a predictor $g(\cdot)$ by minimizing a loss function w.r.t. the true cost vector $\boldsymbol{c}$ such as mean squared error (MSE), $l_{\text{MSE}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \frac{1}{n}\|\hat{\boldsymbol{c}} - \boldsymbol{c}\|^2$. Following training, and given an instance with feature vector $\boldsymbol{x}$, the predictor outputs a cost vector $\hat{\boldsymbol{c}} = g(\boldsymbol{x}; \boldsymbol{\theta})$, which is then used for solving the optimization problem. The advantage of the two-stage approach is the utilization of existing machine learning methods. It decomposes the predict-then-optimize problem into traditional regression then optimization.

3.3 Gradient-based End-to-end Predict-then-Optimize

The main drawback of the two-stage approach is that the decision error is not taken into account in training. In contrast, the end-to-end predict-then-optimize method in Figure 2 attempts to minimize the decision error. Consistent with deep learning terminology, we will use the term "backward pass" to

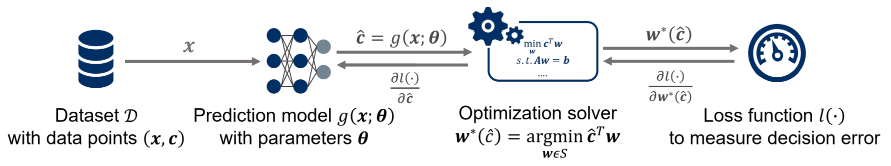

x
$\hat{c} = g(x;\theta)$
$\frac{\partial l(\cdot)}{\partial \hat{c}}$
$\min_w c^T w$
$s.t. Aw = b$
....
$w^*(\hat{c})$
$\frac{\partial l(\cdot)}{\partial w^*(\hat{c})}$
Dataset $\mathcal{D}$ with data points (x, c)
Prediction model $g(x;\theta)$ with parameters θ
Optimization solver $w^*(\hat{c}) = \operatorname{argmin}_{w \in S} \hat{c}^T w$
Loss function $l(\cdot)$ to measure decision error

Fig. 2: Illustration of the end-to-end predict-then-optimize framework: A labeled dataset $\mathcal{D}$ of $(\boldsymbol{x}, \boldsymbol{c})$ pairs is used to fit a machine learning predictor that directly minimizes decision error. The critical component is an optimization solver which is embedded into a differentiable predictor (e.g., a neural network). At test time, this approach is similar to the two-stage approach from Figure 1; only the predictor training is different.

refer to the gradient computation via the backpropagation algorithm. In order to incorporate optimization into the prediction, we can derive the derivative of the optimization task and then apply the gradient descent algorithm, Algorithm 1, to update the parameters of the predictor.

Algorithm 1 End-to-end Gradient Descent

```
Require: coefficient matrix $\boldsymbol{A}$, right-hand side $\boldsymbol{b}$, data $\mathcal{D}$
 1: Initialize predictor parameters $\boldsymbol{\theta}$ for predictor $g(\boldsymbol{x};\boldsymbol{\theta})$
 2: for epochs do
 3:     for each batch of training data $(\boldsymbol{x}, \boldsymbol{c})$ do
 4:         Sample batch of the cost vectors $\boldsymbol{c}$ with the corresponding features $\boldsymbol{x}$
 5:         Predict cost using predictor $\hat{\boldsymbol{c}} := g(\boldsymbol{x};\boldsymbol{\theta})$
 6:         Forward pass to compute optimal solution $\boldsymbol{w}^*(\hat{\boldsymbol{c}}) := \operatorname{argmin}_{\boldsymbol{w} \in S} \hat{\boldsymbol{c}}^T \boldsymbol{w}$
 7:         Forward pass to compute decision loss $l(\hat{\boldsymbol{c}}, \boldsymbol{c})$
 8:         Backward pass from loss $l(\hat{\boldsymbol{c}}, \boldsymbol{c})$ to update parameters $\boldsymbol{\theta}$ with gradient
 9:     end for
10: end for
```

For an appropriately defined loss function, i.e., one that penalizes decision error, the chain rule can be used to calculate the following gradient of the loss w.r.t. predictor parameters:

$$\begin{aligned} &\frac{\partial l(\hat{\boldsymbol{c}}, \boldsymbol{c})}{\partial \boldsymbol{\theta}} = \frac{\partial l(\hat{\boldsymbol{c}}, \boldsymbol{c})}{\partial \hat{\boldsymbol{c}}} \frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}} = \frac{\partial l(\hat{\boldsymbol{c}}, \boldsymbol{c})}{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})} \frac{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}{\partial \hat{\boldsymbol{c}}} \frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}} \\ &\text{Note: } \frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}} = \frac{\partial g(\boldsymbol{x};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \end{aligned} \tag{3}$$

The last term $\frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}}$ is the gradient of the predictions w.r.t. the model parameters, which is trivial to calculate in modern deep learning frameworks. The challenging part is to compute $\frac{\partial l(\hat{\boldsymbol{c}}, \boldsymbol{c})}{\partial \hat{\boldsymbol{c}}}$ or $\frac{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}{\partial \hat{\boldsymbol{c}}}$. Because the optimal solution $\boldsymbol{w}^*(\boldsymbol{c})$ for linear and integer programming is a piecewise constant function from cost vector $\boldsymbol{c}$ to solution vector $\boldsymbol{w}^*$, the predictor parameters cannot be updated with gradient descent. Thus, `SPO+` and `DBB` construct approximate gradients: `SPO+` derives $\frac{\partial l(\hat{\boldsymbol{c}}, \boldsymbol{c})}{\partial \hat{\boldsymbol{c}}}$ and `DBB` computes $\frac{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}{\partial \hat{\boldsymbol{c}}}$.

3.3.1 Decision loss

To measure the error in decision-making, the notion of regret (also called SPO Loss [14]) has been proposed and is defined as the difference in objective value between an optimal solution (using the true but unknown cost vector) and one obtained using the predicted cost vector:

$$l_{\text{Regret}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \boldsymbol{c}^T \boldsymbol{w}^*(\hat{\boldsymbol{c}}) - z^*(\boldsymbol{c}). \tag{4}$$

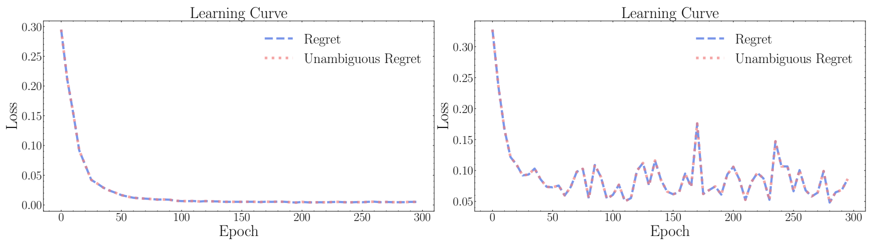


Fig. 3: As shown for the the learning curves of the training of `SPO+` and `DBB` on the shortest path, regret and unambiguous regret in the various tasks overlap almost exactly.

Given a cost vector $\hat{c}$, there may be multiple optimal solutions to $\min_{\boldsymbol{w} \in S} \hat{\boldsymbol{c}}^T \boldsymbol{w}$. Therefore, Elmachtoub and Grigas [14] devised the "unambiguous" regret (also called unambiguous SPO Loss): $l_{\text{URegret}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \max_{\boldsymbol{w} \in W^*(\boldsymbol{c})} \boldsymbol{w}^T \boldsymbol{c} - z^*(\boldsymbol{c})$. This loss considers the worst case among all optimal solutions w.r.t. the predicted cost vector. *PyEPO* provides an evaluation module (Section 4.3) that includes both the regret and the unambiguous regret. However, as Figure 3 shows, the regret and the unambiguous regret are almost the same in all training procedures. Therefore, although the unambiguous regret is more theoretically rigorous, it is not necessary to consider it in practice.

3.4 Methodologies

3.4.1 Smart Predict-then-Optimize

To make the decision error differentiable, Elmachtoub and Grigas [14] proposed `SPO+`, a convex upper bound on the regret:

$$l_{SPO+}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \min_{\boldsymbol{w} \in S} \{(2\hat{\boldsymbol{c}} - \boldsymbol{c})^T \boldsymbol{w}\} + 2\hat{\boldsymbol{c}}^T \boldsymbol{w}^*(\boldsymbol{c}) - z^*(\boldsymbol{c}). \tag{5}$$

One proposed subgradient for this loss writes as follows:

$$2(\boldsymbol{w}^*(\boldsymbol{c}) - \boldsymbol{w}^*(2\hat{\boldsymbol{c}} - \boldsymbol{c})) \in \frac{\partial l_{\text{SPO+}}(\hat{\boldsymbol{c}}, \boldsymbol{c})}{\partial \hat{\boldsymbol{c}}} \tag{6}$$

Thus, we can use Algorithm 1 to directly minimize $l_{\text{SPO+}}(\hat{\boldsymbol{c}}, \boldsymbol{c})$ with gradient descent. This algorithm with `SPO+` requires solving $\min_{\boldsymbol{w} \in S}(2\hat{\boldsymbol{c}} - \boldsymbol{c})^T \boldsymbol{w}$ for each training iteration.

To accelerate the `SPO+` training, Mandi et al. [25] employed relaxations (`SPO+ Rel`) and warm starting (`SPO+ WS`) to speed-up the optimization. The idea of `SPO+ Rel` is to use the continuous relaxation of the integer program during training. This simplification greatly reduces the training time at the expense of model performance. Compared to `SPO+`, the improvement of `SPO+ Rel` on training efficiency is not negligible. For example, linear programming can be solved in polynomial time while integer programming is worst-case exponential. In Section 6, we will further discuss this performance-efficiency tradeoff. For `SPO+ WS`, Mandi et al. [25] suggested using the previous solution as a starting point for integer programming, which potentially improves the efficiency by narrowing down the search space.

3.4.2 Differentiable Black-Box Solver

`DBB` was developed by Pogančić et al. [33] to estimate gradients from interpolation, replacing the zero gradient in $\frac{\partial \boldsymbol{w}^*(\boldsymbol{c})}{\partial \boldsymbol{c}}$. Thus, Pogančić et al. [33] add a slight perturbation with hyperparameter λ, and then utilize finite differences to obtain a zero-order estimate of the gradient. The substitute of $\boldsymbol{w}^*(\boldsymbol{c})$ becomes piecewise affine. Therefore, when computing $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$, a useful nonzero gradient is obtained at the cost of faithfulness. The forward pass and backward pass are shown in Algorithm 2 and Algorithm 3. The hyperparameter $\lambda \geq 0$ controls the interpolation degree. However, compared to `SPO+`, the approximation function of `DBB` is non-convex, so the convergence to a global optimum is compromised, even when the predictor is convex in its parameters.

Algorithm 2 DBB Forward Pass

```
Require: $\hat{\boldsymbol{c}}$
1: Solve $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$
2: Save $\hat{\boldsymbol{c}}$ and $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$ for backward pass
3: return $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$
```

Algorithm 3 DBB Backward Pass

```
Require: $\frac{\partial l(\hat{\boldsymbol{c}}, \boldsymbol{c})}{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}$, $\lambda$
1: Load $\hat{\boldsymbol{c}}$ and $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$ from forward pass
2: $\boldsymbol{c}' := \hat{\boldsymbol{c}} + \lambda \frac{\partial l(\hat{\boldsymbol{c}}, \boldsymbol{c})}{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}$
3: Solve $\boldsymbol{w}^*(\boldsymbol{c}')$
4: return $\frac{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}{\partial \hat{\boldsymbol{c}}} := \frac{1}{\lambda}(\boldsymbol{w}^*(\boldsymbol{c}') - \boldsymbol{w}^*(\hat{\boldsymbol{c}}))$
```

Similar to `SPO+`, `DBB` requires solving the optimization problem in each training iteration. Thus, utilizing a relaxation/rounding approach may also work for `DBB`. However, Pogančić et al. [33] did not consider this option. Given the potential efficiency gains that a continuous relaxation can bring, we also conducted experiments for `DBB Rel` in section 6.

4 Implementation and Modeling

The core module of *PyEPO* is an "autograd" function which is inherited from *PyTorch* [30]. These functions implement a forward pass that yields optimal solutions to the optimization problem and a backward pass to obtain non-zero gradients such that the prediction model can learn from the decision error or its surrogates. Thus, our implementation extends *PyTorch*, which facilitates the deployment of end-to-end predict-then-optimize tasks using any neural network that can be implemented in *PyTorch*.

We choose *GurobiPy* [19] and *Pyomo* [21] to build optimization models. Both *GurobiPy* and *Pyomo* are algebraic modeling languages (AMLs) written in Python. *GurobiPy* is a *Gurobi* Python interface, which combines the expressiveness of a modeling language with the flexibility of a programming language. As an official interface of *Gurobi*, *GurobiPy* has a simple algebraic syntax and natively supports all features of *Gurobi*. Considering that users may not have a *Gurobi* license, we have additionally designed a *Pyomo* interface as an alternative. As an open-source optimization modeling language, *Pyomo* supports a variety of solvers, including *Gurobi* and *GLPK*. Both of the above provide a natural way to express mathematical programming models. Users without specialized optimization knowledge can easily build and maintain optimization models through high-level algebraic representations. Besides *GurobiPy* and *Pyomo*, *PyEPO* also allows users to construct optimization models from scratch using any algorithm and solver. As a result, fast and flexible model customization for research and production is possible in *PyEPO*.

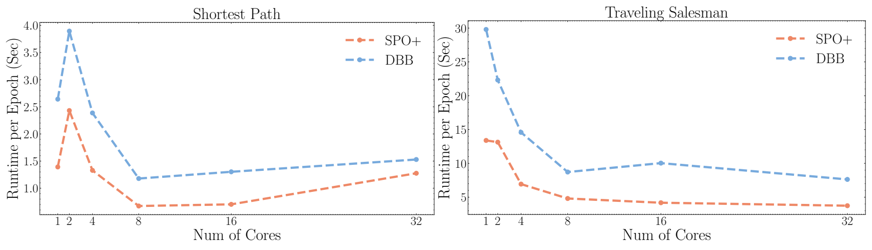


Fig. 4: Parallel efficiency: Although there is additional overhead in creating a new process, parallel computing of `SPO+` and `DBB` with an appropriate number of processors can reduce the training time effectively.

In addition, *PyEPO* supports parallel computing. For `SPO+` and `DBB`, the computational cost is a major challenge that cannot be ignored, particularly for integer programs. Both require solving an optimization problem per instance to obtain the gradient.

Figure 4 shows the average running time per epoch for a mini-batch gradient descent algorithm with a batch size of 32 as a function of the number of cores. The decrease in running time per epoch is sublinear in the number of cores. This may be explained by the overhead associated with starting up additional cores, which might dominate computation cost. For example, in Figure 4, for the shortest path, the easily solvable polynomial problem, the running time actually increases when the number of cores exceeds 8, while the more complicated $\mathcal{NP}$-complete problem, TSP, continues to benefit slightly from additional cores. Overall, we believe this feature is crucial for large-scale predict-then-optimize tasks.

4.1 Optimization Model

The first step in using *PyEPO* is to create an optimization model that inherits from the **optModel** class. Since *PyEPO* tackles predict-then-optimize with unknown cost coefficients, it is first necessary to instantiate an optimization model, **optModel**, with fixed constraints and variable costs. Such an optimization model would accept different cost vectors and be able to find the corresponding optimal solutions with identical constraints. The construction of **optModel** is separated from the autograd functions, **SPOPlus** and **blackboxOpt**. Then, it would be passed as an argument into the above functions.

4.1.1 Optimization Model from Scratch

In *PyEPO*, the **optModel** works as a black-box, which means that we do not specifically require a certain algorithm or a certain solver. This design is intended to give the users more freedom to customize their tasks. To build an **optModel** from scratch, users need to override abstract methods **_getModel** to build a model and get its variables, **setObj** to set the objective function with a given cost vector, and **solve** to find an optimal solution. In addition, **optModel** provides an attribute **modelSense** to indicate whether the problem is one of minimization or maximization. The following shortest path example uses the Python library *NetworkX* [20] and its built-in Dijkstra's algorithm:

```python
import networkx as nx
from pyepo import EPO
from pyepo.model.opt import optModel

class myShortestPathModel(optModel):

    def __init__(self):
        self.modelSense = EPO.MINIMIZE
        self.grid = (5,5) # graph size
        self.arcs = self._getArcs() # list of arcs
        super().__init__()

    def _getArcs(self):
        """
        A helper method to get list of arcs for grid network
        """
        arcs = []
        h, w = self.grid
        for i in range(h):
            # edges on rows
            for j in range(w - 1):
                v = i * w + j
                arcs.append((v, v + 1))
            # edges in columns
            if i == h - 1:
                continue
            for j in range(w):
                v = i * w + j
                arcs.append((v, v + w))
        return arcs

    def _getModel(self):
        """
        A method to build model
        """
        # build graph as model
        model = nx.Graph()
        # add arcs as variables
        model.add_edges_from(self.arcs, cost=0)
        var = model.edges
        return model, var

    def setObj(self, c):
        """
        A method to set objective function
        """
        for i, e in enumerate(self.arcs):
            self._model.edges[e]["cost"] = c[i]

    def solve(self):
        """
        A method to solve model
        """
        # dijkstra
        s = 0 # source node
        t = self.grid[0] * self.grid[1] - 1 # target node
        path = nx.shortest_path(self._model, weight="cost",
                                source=s, target=t)
        # convert path into active edges
        edges = []
        u = 0
        for v in path[1:]:
            edges.append((u,v))
            u = v
        # init sol & obj
        sol = [0] * self.num_cost
        obj = 0
        # convert active edges into solution and obj
        for i, e in enumerate(self.arcs):
            if e in edges:
                # active edge
                sol[i] = 1
                # cost of active edge
                obj += self._model.edges[e]["cost"]
        return sol, obj

model = myShortestPathModel()
```

4.1.2 Optimization Model with Gurobi

On the other hand, we provide **optGrbModel** to create an optimization model with *GurobiPy*. Unlike **optModel**, **optGrbModel** is more lightweight but less flexible for users. Let us use the following optimization model 7 as an example, where c_i is an unknown cost coefficient:

$$\begin{aligned}
&\max_{x} \sum_{i=0}^{4} c_i x_i \\
s.t.\quad & 3x_0 + 4x_1 + 3x_2 + 6x_3 + 4x_4 \leq 12 \\
& 4x_0 + 5x_1 + 2x_2 + 3x_3 + 5x_4 \leq 10 \\
& 5x_0 + 4x_1 + 6x_2 + 2x_3 + 3x_4 \leq 15 \\
& \forall x_i \in \{0, 1\}
\end{aligned} \tag{7}$$

Inheriting **optGrbModel** is the convenient way to use *Gurobi* with *PyEPO*. The only implementation required is to override **_getModel** and return a *Gurobi* model and the corresponding decision variables. In addition, there is no need to assign a value to the attribute **modelSense** in **optGrbModel** manually. An example for Equation 7 is as follows:

```
import gurobipy as gp
from gurobipy import GRB
from pyepo.model.grb import optGrbModel

class myModel(optGrbModel):
    def _getModel(self):
        # ceate a model
        m = gp.Model()
        # varibles
        x = m.addVars(5, name="x", vtype=GRB.BINARY)
        # sense (must be minimize)
        m.modelSense = GRB.MAXIMIZE
        # constraints
        m.addConstr(3*x[0]+4*x[1]+3*x[2]+6*x[3]+4*x[4]<=12)
        m.addConstr(4*x[0]+5*x[1]+2*x[2]+3*x[3]+5*x[4]<=10)
        m.addConstr(5*x[0]+4*x[1]+6*x[2]+2*x[3]+3*x[4]<=15)
        return m, x

optmodel = myModel()
```

4.1.3 Optimization Model with Pyomo

Similarly, **optOmoModel** allows modeling mathematical programming with *Pyomo*. In contrast to **optGrbModel**, **optOmoModel** requires an explicit object attribute **modelSense**. Since *Pyomo* supports multiple solvers, instantiating an **optOmoModel** requires a parameter **solver** to specify the solver. The following is the implementation of problem 7:

```python
from pyomo import environ as pe
from pyepo import EPO
from pyepo.model.omo import optOmoModel

class myModel(optOmoModel):
    def _getModel(self):
        # sense
        self.modelSense = EPO.MAXIMIZE
        # ceate a model
        m = pe.ConcreteModel()
        # varibles
        x = pe.Var([0,1,2,3,4], domain=pe.Binary)
        m.x = x
        # constraints
        m.cons = pe.ConstraintList()
        m.cons.add(3*x[0]+4*x[1]+3*x[2]+6*x[3]+4*x[4]<=12)
        m.cons.add(4*x[0]+5*x[1]+2*x[2]+3*x[3]+5*x[4]<=10)
        m.cons.add(5*x[0]+4*x[1]+6*x[2]+2*x[3]+3*x[4]<=15)
        return m, x

optmodel = myModel(solver="glpk")
```

4.2 Autograd Functions

Training neural networks with modern deep learning libraries such as *TensorFlow* [1] or *PyTorch* [31] requires gradient calculations for backpropagation. For this purpose, the numerical technique of automatic differentiation [30] is used. For example, *PyTorch* provides autograd functions, so that users are allowed to utilize or create functions that automatically compute partial derivatives.

Autograd functions are the core modules of *PyEPO* that solve and backpropagate the optimization problems with predicted costs. These functions can be integrated with different neural network architectures to achieve end-to-end predict-then-optimize for various tasks. In *PyEPO*, autograd functions include **SPOPlus** [14] and **blackboxOpt** [33].

4.2.1 Function SPOPlus

The function **SPOPlus** calculates SPO+ loss, which measures the decision error of an optimization. This optimization is represented as an instance of **optModel** and passed into the **SPOPlus** as an argument. As shown below, **SPOPlus** also requires **processes** to specify the number of processes.

```python
from pyepo.func import SPOPlus
# init SPO+ Pytorch function
spo = SPOPlus(optmodel, processes=8)
```

The parameters for the forward pass of **SPOPlus** are as follows:

- **pred_cost**: a batch of predicted cost vectors, one vector per instance;
- **true_cost**: a batch of true cost vectors, one vector per instance;
- **true_sol**: a batch of true optimal solutions, one vector per instance;
- **true_obj**: a batch of true optimal objective values, one value per instance.

The following code block is the **SPOPlus** forward pass:

```
# calculate SPO+ loss
loss = spo.apply(pred_cost, true_cost, true_sol, true_obj)
```

4.2.2 Function blackboxOpt

SPOPlus directly obtains a loss while **blackboxOpt** provides a solution. Thus, **blackboxOpt** makes it possible to use various loss functions. Compared to **SPOPlus**, **blackboxOpt** requires an additional parameter **lambd**, which is the hyperparameter λ for the differentiable black-box solver. According to Pogančić et al. [33], the values of λ should be between 10 and 20.

```
from pyepo.func import blackboxOpt
# init DBB solver
dbb = blackboxOpt(optmodel, lambd=10, processes=8)
```

Since **blackboxOpt** works as a solver, there is only one parameter **pred_cost** for the forward pass. As in the code below, the output is the optimal solution for the given predicted cost:

```
# solve
pred_sol = dbb.apply(pred_cost)
```

4.3 Metrics

PyEPO provides evaluation functions to measure model performance, in particular the two metrics mentioned in Section 3.3.1: regret and unambiguous regret. Both of them require the following parameters:

- **predmodel**: a regression neural network for cost prediction
- **optmodel**: a *PyEPO* optimization model
- **dataloader**: a *PyEPO* data loader

Assume that we have trained a regression neural network **predmodel** for an optimization task **optmodel**, and then we want to evaluate its performance on some data set:

```
from pyepo.metric import regret, unambRegret
regret = regret(predmodel, optmodel, dataloader)
uregret = unambRegret(predmodel, optmodel, dataloader)
```

5 Benchmark Datasets

In this section, we describe our new datasets designed for the task of end-to-end predict-then-optimize. Overall, we generate datasets in a similar way to Elmachtoub and Grigas [14]. The synthetic dataset $\mathcal{D}$ includes features $\boldsymbol{x}$ and cost coefficients $\boldsymbol{c}$: $\mathcal{D} = \{(\boldsymbol{x}_1, \boldsymbol{c}_1), (\boldsymbol{x}_2, \boldsymbol{c}_2), ..., (\boldsymbol{x}_n, \boldsymbol{c}_n)\}$. The feature vector $\boldsymbol{x}_i \in \mathbb{R}^p$ follows a standard multivariate Gaussian distribution $\mathcal{N}(0, \boldsymbol{I}_p)$ and the corresponding cost vector $\boldsymbol{c}_i \in \mathbb{R}^d$ comes from a (possibly nonlinear) polynomial function of $\boldsymbol{x}_i$ with additional random noise. $\epsilon_{ij} \sim \boldsymbol{U}(1-\bar{\epsilon}, 1+\bar{\epsilon})$ is the multiplicative noise term for c_{ij}, the j^{th} element of cost $\boldsymbol{c}_i$.

Our dataset includes three of the most classical optimization problems: the shortest path problem, the multi-dimensional knapsack problem, and the traveling salesperson problem. *PyEPO* provides functions to generate these data with the adjustable data size n, number of features p, cost vector dimension d, polynomial degree deg, and noise half-width $\bar{\epsilon}$.

5.1 Shortest Path

We consider a $h \times w$ grid network and the goal is to find the shortest path [29] from northwest to southeast. We generate a random matrix $\mathcal{B} \in \mathbb{R}^{d \times p}$, where $\mathcal{B}_{ij}$ follows Bernoulli distribution with probability 0.5. Then, the cost vector $\boldsymbol{c}_i$ is almost the same as in [14], and is generated from

$$\left[\frac{1}{3.5^{deg}\sqrt{p}}((\mathcal{B}\boldsymbol{x}_i)_j + 3)^{deg} + 1\right] \cdot \epsilon_{ij}. \tag{8}$$

The following code generates data for the shortest path on the grid network:

```python
from pyepo.data.shortestpath import genData
x, c = genData(n, p, grid=(h,w), deg=deg, noise_width=e)
```

5.2 Multi-Dimensional Knapsack

The multi-dimensional knapsack problem [26] is one of the most well-known integer programming models. It maximizes the value of selected items under multiple resource constraints. Due to its computational complexity, solving this problem can be challenging especially with the increase in the number of constraints (or resources, or knapsacks).

Because we assume that the uncertain coefficients exist only in the objective function, the weights of items are fixed throughout the data. We use k to denote the number of resources; the number of items is same as the dimension of the cost vector d. The weights $\mathcal{W} \in \mathbb{R}^{k \times m}$ are sampled from 3 to 8 with a precision of 1 decimal place. With the same $\mathcal{B} \in \mathbb{R}^{d \times p}$ as in Section 5.1, cost c_{ij} is calculated according to (8).

To generate k-dimensional knapsack data, a user simply executes the following:

```python
from pyepo.data.knapsack import genData
W, x, c = genData(n, p, num_item=d, dim=k, deg=deg, noise_width=e)
```

5.3 Traveling Salesperson

As one of the most famous combinatorial optimization problems, the traveling salesperson problem (TSP) aims to find the shortest possible tour that visits every node exactly once. Here, we introduce the symmetric TSP with the number of nodes to be visited v.

PyEPO generates costs from a distance matrix. The distance is the sum of two parts: one comes from Euclidean distance, the other derived from feature encoding. For Euclidean distance, we create coordinates from the mixture of Gaussian distribution $\mathcal{N}(0, I)$ and uniform distribution $\boldsymbol{U}(-2, 2)$. For feature encoding, the polynomial kernel function is $\frac{1}{3^{deg-1}\sqrt{p}}((\mathcal{B}\boldsymbol{x_i})_j + 3)^{deg} \cdot \epsilon_{ij}$, where the elements of $\mathcal{B}$ come from the multiplication of Bernoulli $\boldsymbol{B}(0.5)$ and uniform $\boldsymbol{U}(-2, 2)$.

An example of a TSP data generation is as follow:

```python
from pyepo.data.tsp import genData
x, c = genData(n, p, num_node=v, deg=deg, noise_width=e)
```

6 Empirical Evaluation

In this section, we present experimental results for the benchmark datasets of Section 5. In these experiments, We examine the training time and the normalized regret on a test set with a sample size of $n_{\text{test}} = 1000$. Recall that the regret was defined in (4). We define the normalized regret by

$$\frac{\sum_{i=1}^{n_{\text{test}}} l_{\text{Regret}}(\hat{\boldsymbol{c}}_i, \boldsymbol{c}_i)}{\sum_{i=1}^{n_{\text{test}}} |z^*(\boldsymbol{c}_i)|}.$$

As Table 3 shows, the methods we compare include the two-stage approach with different predictors and `SPO+`/`DBB` with a linear model or neural networks. We mainly focus on the linear model (i.e., a neural network without hidden layers), but deep neural network architectures are also explored in Section 6.8.

Unlike `SPO+`, `DBB` allows the use of arbitrary loss functions and the flexibility in the loss could be useful for different problems. In the original paper, Pogančić et al. [33] used the Hamming distance between the true optimum and the predicted solution. However, in our experiments, compared to the regret, `DBB` using the Hamming distance is only sensible for the shortest path problem but leads to much worse decisions in knapsack and TSP. For the sake of consistency, we only use regret (4) as the loss for `DBB`.

All the numerical experiments were conducted in Python v3.7.9 with Intel E5-2683 v4 Broadwell CPU processors and 8GB memory. Specifically, we used *PyTorch* [31] v1.10.0 for training end-to-end models, and *Scikit-Learn* [32]

Method	**Description**
2-stage LR	Two-stage method where the predictor is a linear regression
2-stage RF	Two-stage method where the predictor is a random forest with default parameters
2-stage Auto	Two-stage method where the predictor is *Auto-Sklearn* [16] with 10 minutes time limit and uses MSE as metric
`SPO+`	Linear model with SPO+ loss [14]
`DBB`	Linear model with DBB optimizer [33]
`SPO+ Rel`	Linear model with SPO+ loss [14], using linear relaxation for training
`DBB Rel`	Linear model with DBB optimizer [33], using linear relaxation for training
`SPO+ L1`	Linear model with SPO+ loss [14], using l_1 regularization for cost
`SPO+ L2`	Linear model with SPO+ loss [14], using l_2 regularization for cost
`DBB L1`	Linear model with DBB optimizer [33], using l_1 regularization for cost
`DBB L2`	Linear model with DBB optimizer [33], using l_2 regularization for cost
`SPO+` $h_1 \times ... \times h_L$	Fully connected neural network with L hidden layers of width h_1, ..., h_L and SPO+ loss [14]
`DBB` $h_1 \times ... \times h_L$	Fully connected neural network with L hidden layers of width h_1, ..., h_L and DBB optimizer [33]

Table 3: Methods compared in the experiments.

v0.24.2 and *Auto-Sklearn* [16] v0.14.6 as predictor of the two-stage method. *Gurobi* [19] v9.1.2 was the optimization solver in the background.

6.1 Performance Comparison between Different Methods

We compare the performance between two-stage methods, `SPO+`, and `DBB` with varying training data size $n \in \{100, 1000, 5000\}$, polynomial degree $deg \in \{1, 2, 4, 6\}$, and noise half-width $\bar{\epsilon} \in \{0.0, 0.5\}$. We use a linear model without any regularization for the cost vector. We then conduct small-scale experiments on the validation set to select gradient descent hyperparameters, namely the batch size, learning rate, and momentum for shortest path, knapsack, and TSP in `SPO+` and `DBB`. The hyperparameter tuning uses a limited random search in the space of hyperparameter configurations. Thus, there is no guarantee of the best performance in the results. We repeated all experiments 10 times, each with a different $\boldsymbol{x}$, $\mathcal{B}$, and ϵ to generate 10 different training/validation/test datasets. We use boxplots to summarize the statistical outcomes.

Problem	**Parameters**	**Feature Size**	**Cost Dimension**
Shortest Path	Height of the grid is 5 Width of the grid is 5	5	40
Knapsack	Dimension of resource is 2 Number of items is 32 Capacity is 20	5	32
Traveling Saleman	Number of nodes is 20	10	190

Table 4: Problem Parameters for Performance Comparison

We generate synthetic datasets with parameters in Table 4, so the dimensions of the cost vectors d are 40, 32, and 190, respectively. For the TSP, we use the Dantzig–Fulkerson–Johnson (DFJ) formulation [10] because it is faster to solve than alternative formulations.

Figures 5, 6, and 7 summarize the performance comparison for the shortest path problem, 2D knapsack problem, and traveling salesperson problem.

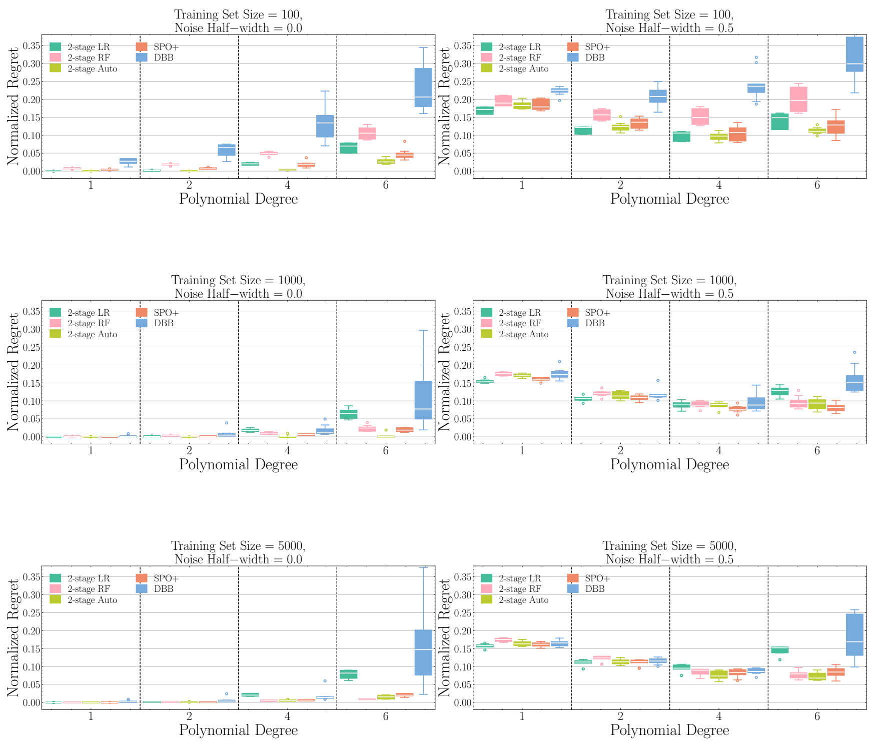


Fig. 5: Normalized regret for the shortest path problem on the test set: The size of the grid network is 5×5. The methods in the experiment include two-stage approaches with linear regression, random forest and *Auto-Sklearn* and end-to-end learning such as `SPO+` and `DBB`. The normalized regret is visualized under different sample sizes, noise half-width, and polynomial degrees. For the normalized regret, lower is better.

These figures should be read as follows: the left column is for noise-free costs (easier), while the right column includes noise. Each row of figures is for a training set size in increasing order. Within each figure and from left to right, the degree of the polynomial that generates the costs from the feature vector increases. Within each such polynomial degree, the different methods' box-plots are shown, summarizing the test set normalized regret results for the 10 different experiments; lower is better.

Two-stage linear regression (2-stage LR) performs well at lower polynomial degrees but loses its advantage at higher polynomial degrees. The two-stage random forest (2-stage RF) is robust at high polynomial degrees but requires a large amount of training data. With 5000 data samples, random forest achieves the best performance in many cases. The two-stage method with

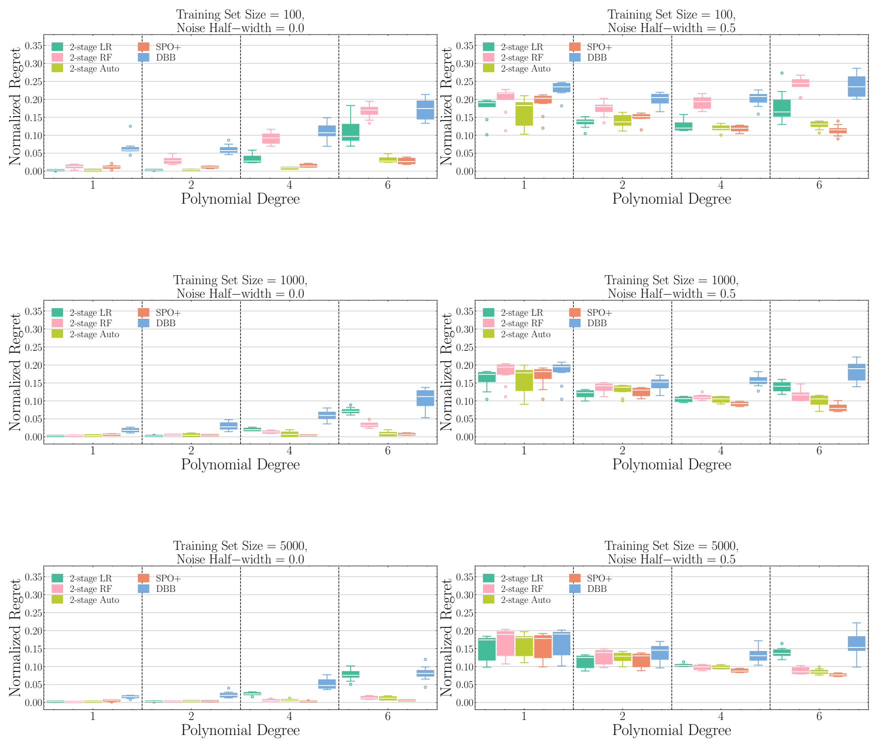


Fig. 6: Normalized regret for the 2D knapsack problem on the test set: There are 32 items, and the capacity of the two resources is 20. The methods in the experiment include two-stage approaches with linear regression, random forest and *Auto-Sklearn* and end-to-end learning such as `SPO+` and `DBB`. The normalized regret is visualized under different sample sizes, noise half-width, and polynomial degrees. For the normalized regret, lower is better.

automated hyperparameter tuning using the *Auto-Sklearn* tool [16] (which will be discussed further in the next section) is attractive: despite tuning for lower prediction error (not decision error), *Auto-Sklearn* effectively reduces the regret so that it usually performs better than two-stage linear regression and random forest. However, with the increase of input and output dimension, *Auto-Sklearn* fails to be competitive for the TSP (Figure 7).

`SPO+` shows its advantage: it performs best, or at least relatively well, in all cases. `SPO+` is comparable to linear regression under low polynomial degree and depends less on the sample size than random forest. At high polynomial degrees, `SPO+` outperforms *Auto-Sklearn*, which exposes the limitations of the two-stage approach. Although the results of `DBB` are not ideal, Pogančić et al. [33] demonstrated in their experiments that `DBB` can operate on complex image

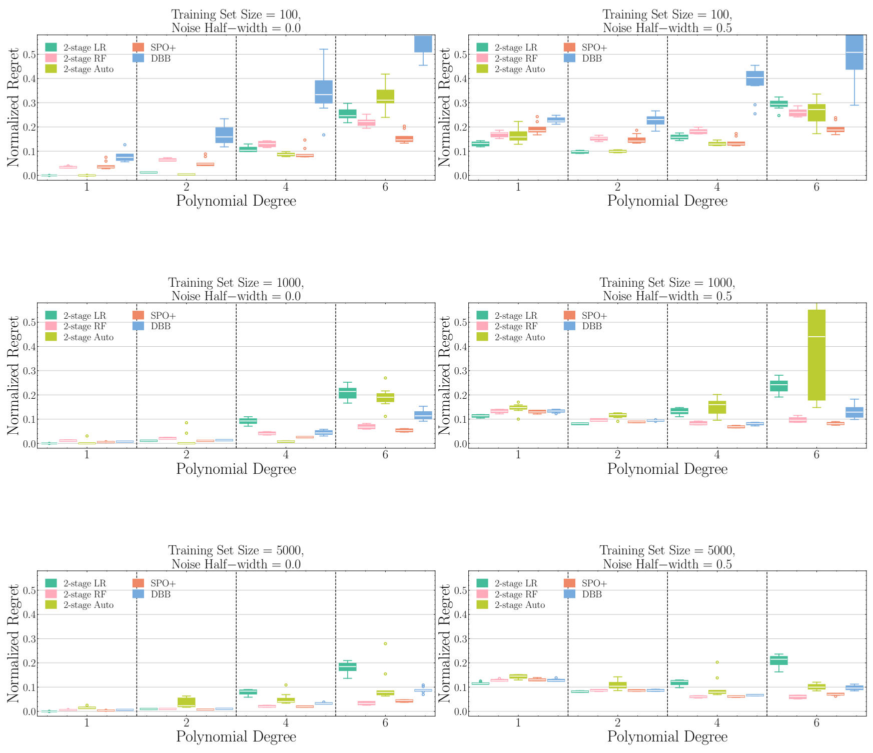


Fig. 7: Normalized regret for the TSP problem on the test set: There are 20 nodes to visit. The methods in the experiment include two-stage approaches with linear regression, random forest and *Auto-Sklearn* and end-to-end learning such as `SPO+` and `DBB`. The normalized regret is visualized under different sample sizes, noise half-width, and polynomial degrees. For the normalized regret, lower is better.

inputs using convolutional neural networks for shortest path and TSP, showing the ability to extract features from complex raw data; future additions to our package will include such experiments.

Finding #1

`SPO+` can robustly achieve relatively good decisions under different scenarios.

6.2 Two-stage Method with Automated Hyperparameter Tuning

This method leverages the sophisticated *Auto-Sklearn* [16] tool that uses bayesian optimization methods for automated hyperameter tuning of *Scikit-Learn* regression models. The metric of "2-stage Auto" is the mean squared error of the predicted costs, which does not reduce decision error directly. Because of the limitation of multioutput regression in *Auto-Sklearn* v0.14.6, the choices of the predictor in 2-stage Auto only include five models: k-nearest neighbor (KNN), decision tree, random forest, extra-trees, and Gaussian process. Even with these limitations, *Auto-Sklearn* can achieve a low regret. Although the training of 2-stage Auto is time-consuming, it is still a competitive method.

When we examine the selection of the 2-stage Auto predictor, the results have some similarities. Compared to the other models, decision trees are never the ideal choice. A fine-tuned Gaussian process has impressive performance for datasets with small samples, no noise, or high polynomial degree, while KNN and random forest can also be the best models at times.

Finding #2

Even with successful model selection and hyperparameter tuning, the two-stage method performs worse than `SPO+` in terms of decision quality, which substantiates the value of the end-to-end approach.

6.3 Exact Method and Relaxation

Training `SPO+` and `DBB` with a linear relaxation instead of solving the integer program improves computational efficiency. However, the use of a "weaker" solver theoretically undermines model performance. Therefore, an important question arises about the tradeoff when using the linear relaxation in training. To this end, we compare the performance of end-to-end approaches with their relaxation using 2D knapsack and TSP as examples. We use the same instances, model, and hyperparameters as before.

There are several integer programming formulations for the TSP. Besides DFJ, we also implemented the Miller-Tucker-Zemlin (MTZ) formulation [28] and the Gavish-Graves (GG) formulation [18]. Although all of them have the same integer solution, they differ in terms of their linear relaxations. Since DFJ requires column generation to handle the exponential subtour constraints, its linear relaxation is hard to implement. The GG formulation is shown to have a tighter linear relaxation than MTZ. Thus, we use DFJ for exact `SPO+` and `DBB`, and MTZ and GG for the relaxation to investigate the effect of solution quality on regret.

According to Figure 9, using a linear relaxation significantly reduces the running time. Note that DFJ is more efficient than GG, so it is reasonable that relaxing the GG formulation sometimes takes more time than the exact DFJ. As shown in Figure 8, the impact on `SPO+` performance of knapsack is almost

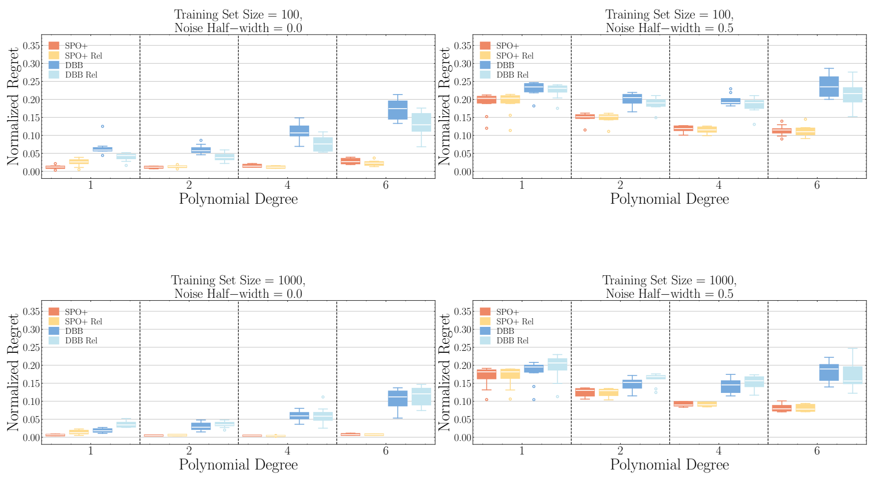


Fig. 8: Normalized regret for the 2D knapsack problem on the test set: There are 32 items, and the capacity of the two resources is 20. The methods in the experiment include `SPO+`, `SPO+ Rel`, `DBB`, and `DBB Rel`. Then, we visualize the normalized regret under different sample sizes, noise half-width, and polynomial degrees to investigate the impact of the relaxation method. For the normalized regret, lower is better.

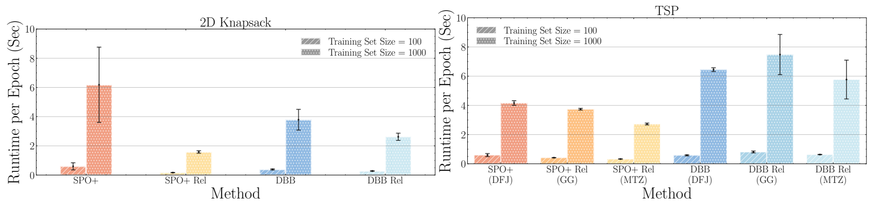


Fig. 9: Average training time per epoch for relaxation methods with standard deviation error bars: We visualized the mean training time for the Knapsack and TSP problem under different sample sizes. Lower is better.

negligible. Interestingly, `DBB Rel` performs better than `DBB` on small data, perhaps because the linear relaxation acts as a regularization to avoid overfitting on small data. For TSP, Figure 10 demonstrates that a tighter bound does reduce the regret, and `DBB Rel` shows advantages over `DBB`. Overall, using a relaxation achieves fairly good performance with superior computational efficiency. Moreover, formulations with tighter linear relaxation lead to better performance.

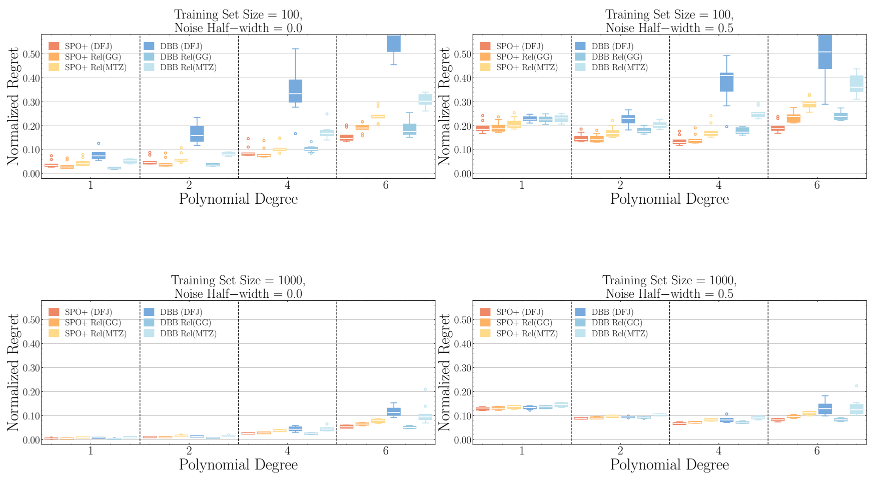


Fig. 10: Normalized regret for the TSP problem on the test set: There are 20 nodes to visit. The methods in the experiment include `SPO+` with DFJ, `SPO+ Rel` with GG, `SPO+ Rel` with MTZ, `DBB` with DFJ, `DBB Rel` with GG, and `DBB Rel` with MTZ. Then, we visualize the normalized regret under different sample sizes, noise half-width, and polynomial degrees to investigate the impact of the relaxation method. For the normalized regret, lower is better.

Finding #3

`SPO+` with relaxation has excellent potential to improve computation efficiency at a slight degradation in performance. A tighter relaxation is always a better choice.

6.4 Solution Regularization

As proposed in [14], the mean absolute error $l_{\text{MAE}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \frac{1}{n}\sum_i^n \|\hat{\boldsymbol{c}}_i - \boldsymbol{c}_i\|_1$ or mean squared error $l_{\text{MSE}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \frac{1}{2n}\sum_i^n \|\hat{\boldsymbol{c}}_i - \boldsymbol{c}_i\|_2^2$ of the predicted cost vector w.r.t. true cost vector can be added to the decision loss as l_1 or l_2 regularizers. When using regularization, we set either the l_1 regularization parameter $\phi_1 = 0.001$ and the l_2 regularization parameter $\phi_2 = 0.001$, which could be tuned further to improve performance and convergence. For the experiments, we still use the same instances, model, and hyperparameters as before, while the noise half-width $\bar{\epsilon}$ is fixed at 0.5.

Based on Fig 11, regularization could reduce the regret, but its impact on `SPO+` is insignificant. For `DBB`, adding regularization is helpful to improve decision-making and reduce the variance of the model.

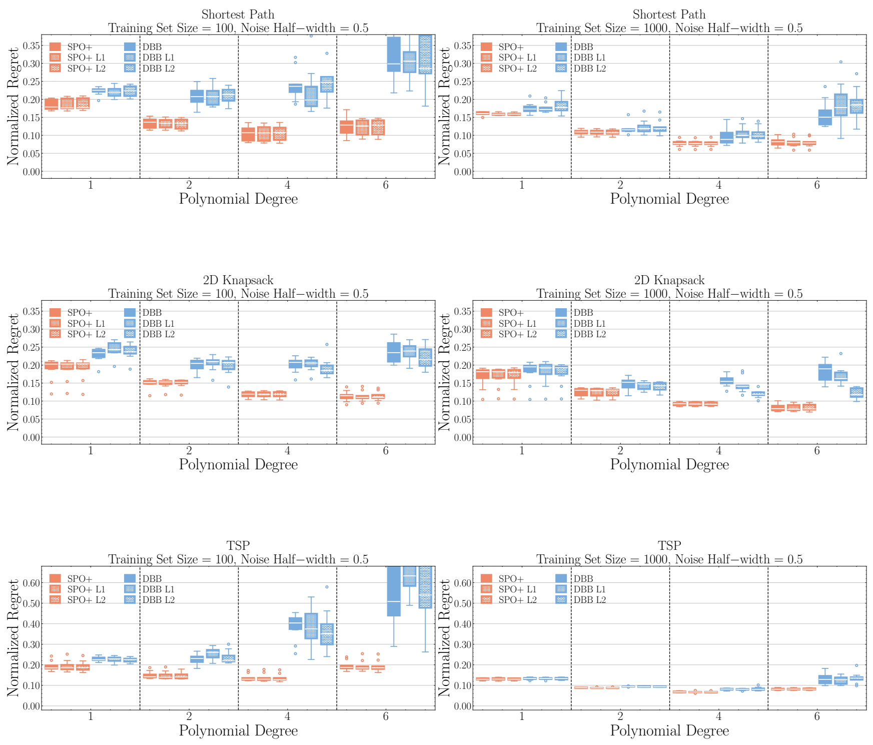


Fig. 11: Normalized regret on the test set: In these experiments, we compare the performance on `SPO+` and `DBB` w/o regularization. The normalized regret under different problems, sample sizes, and polynomial degrees is shown, and lower is better.

Finding #4

Solution regularization can help reduce regret for `DBB`, but only slightly so for `SPO+`.

6.5 Hyperparameter Sensitivity

The hyperparameters of an end-to-end model have to be set appropriately, a process referred to as hyperparameter tuning. These hyperparameters include batch size, learning rate, gradient descent optimizer, l_1 regularization parameter ϕ_1, and l_2 regularization parameter ϕ_2. `DBB` requires the additional smoothness parameter λ to control the interpolation degree. With a training sample size of 1000, a noise half-width of 0.5, and a polynomial degree of 4, we use *Weights & Biases* [7] to perform random search with 50 trials, investigating hyperparameter sensitivity for both `SPO+` and `DBB` in shortest path, 2D knapsack, and TSP. The configuration space is the cross-product of the values in Table 5.

Hyperparameter	**Values**
Batch Size	$32, 64, 128$
Learning Rate	$10^{-3}, 5 \times 10^{-3}, 10^{-2}, 5 \times 10^{-2}, 10^{-1}, 5 \times 10^{-1}$
l_1 regularization parameter ϕ_1	$0, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}$
l_2 regularization parameter ϕ_2	$0, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}$
DBB Smooth Parameter λ	$10, 12, 16, 18, 20$
Optimizer	SGD, Adam [23]

Table 5: Hyperparameters space: The choices among discrete values are defined for each hyperparameter.

Weights & Biases measures the effect of hyperparameters on the regret loss by importance and correlation. Correlation is the linear correlation between the hyperparameter and the regret loss (higher is better). The importance comes from feature importance values of a random forest that is trained to take a hyperparameter configuration as input and predict its regret.

Based on Figures 12, 13, and 14, the optimal hyperparameter configuration for `SPO+` and `DBB` is problem-specific. Overall, the learning rate has a significant impact on regret, and the result from the Adam optimizer is more stable. Whether it is `SPO+` or `DBB`, the regularization of predicted values correlates positively with model performance.

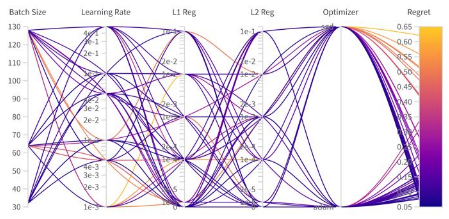
Finding #5

Learning rate is critical and Adam is better than SGD.

6.6 Trade-offs between MSE and Regret

The above experiments have focused only on regret. In addition to decision error, prediction error should also be of concern in many predict-then-optimize tasks. For example, investors seek not only the optimal portfolio but also wonder about the forecasted return of each security. Figure 15 examines the prediction-decision trade-off on shortest path with 100 and 1000 training samples, a 0.5 noise half-width, and polynomial degree 4. We calculate the average MSE and regret for 10 repeated random experiments. Apart from this, mean training time is also annotated and circle sizes are proportional to it. Finally, we remove the circle of `DBB` because they are at the right top corner and far away from others.

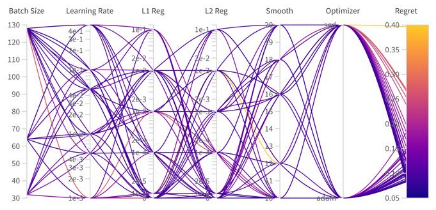
Figure 15 demonstrates that `SPO+` yields improved decisions at the cost of higher prediction errors. `SPO+` can reach a low decision error, especially



Random Search for SPO+

Hyperparameter	Importance	Correlation
Batch Size	0.080	-0.396
Learning Rate	0.561	0.242
L1 Reg	0.026	0.029
L2 Reg	0.040	-0.175
Optimizer Adam	0.166	0.401
Optimizer SGD	0.018	-0.401

Effect of SPO+ Hyperparameters



Random Search for DBB

Hyperparameter	Importance	Correlation
Batch Size	0.026	-0.053
Learning Rate	0.522	0.131
L1 Reg	0.040	0.005
L2 Reg	0.178	0.124
Smooth	0.022	0.060
Optimizer Adam	0.139	0.350
Optimizer SGD	0.015	-0.350

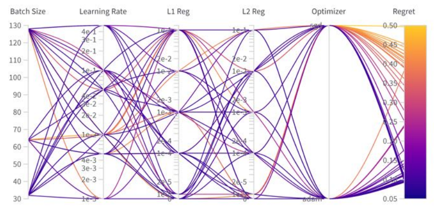
Effect of DBB Hyperparameters

Fig. 12: Hyperparameter Tuning for the Shortest Path: The figure summarizes the validation performance of a wide range of hyperparameter configurations. Each spline is a hyperparameter combination with the values denoted on the appropriate axis. The rightmost axis of the figure shows the regret (lower is better). For the table, the correlations are the linear relationship between a hyperparameters and the regret.

in large training samples, while the MSE is about 5 times larger than two-stage methods. Further examination reveals that the higher prediction error of `SPO+` comes mainly from multiplicative shifts in the predicted cost values, which does not alter the optima of an optimization problem with a linear objective function. In addition, training with *Auto-Sklearn*, which comes from automated algorithm selection and hyperparameter tuning, is time-consuming but provides both high-quality prediction error and decision error. However, compared to `SPO+`, even the competitive 2-stage Auto model does not have an advantage in decision-making with 1000 training data samples.

Finding #6

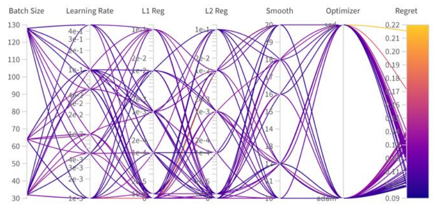
Generally, `SPO+` can achieve good decisions, but its predictions may be inaccurate. If one is seeking a balanced tradeoff between decision quality and prediction accuracy, a well-tuned two-stage approach may be preferable.



Random Search for SPO+

Hyperparameter	Importance	Correlation
Batch Size	0.104	-0.322
Learning Rate	0.091	0.308
L1 Reg	0.010	-0.041
L2 Reg	0.016	0.180
Optimizer Adam	0.103	0.614
Optimizer SGD	0.233	-0.614

Effect of SPO+ Hyperparameters



Random Search for DBB

Hyperparameter	Importance	Correlation
Batch Size	0.071	-0.027
Learning Rate	0.284	0.183
L1 Reg	0.060	-0.028
L2 Reg	0.157	0.230
Smooth	0.126	-0.114
Optimizer Adam	0.041	0.271
Optimizer SGD	0.033	-0.271

Effect of DBB Hyperparameters

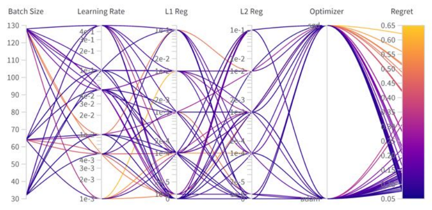
Fig. 13: Hyperparameter Tuning for the 2D Knapsack: The figure summarizes the validation performance of a wide range of hyperparameter configurations. Each spline is a hyperparameter combination with the values denoted on the appropriate axis. The rightmost axis of the figure shows the regret (lower is better). For the table, the correlations are the linear relationship between a hyperparameters and the regret.

6.7 Training Scalability with Increasing Problem Size

For the end-to-end predict-then-optimize, a crucial concern is scalability with increasing problem size, whether in the number of decision variables or the number of constraints. On the one hand, both `SPO+` and `DBB` rely on solving optimization problems iteratively for training, so the scale of the optimization problem has a greater impact on the end-to-end approach than the two-stage method. On the other hand, as the optimization problem becomes complex, the decision error of the predict-then-optimize task also increases. Therefore, we investigate the change in graph size on the shortest path problem with 1000 training sample size and 0.5 noise half-width.

For the shortest path problem, the growth in the graph (grid) size leads to an increase in decision variables. As Fig 16 shows, the increase in the number of decision variables significantly increases the training time for end-to-end learning, both for `SPO+` and `DBB`. Thus, for an optimization problem that is hard to solve, end-to-end methods could hit a computational bottleneck. Moreover, there is a only a small increase in the regret, which validates the robustness of end-to-end approach for large-scale optimization.

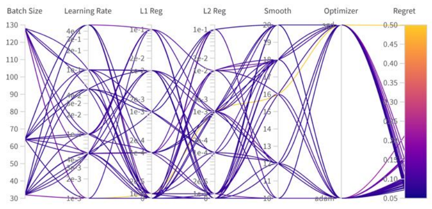
We choose knapsack datasets with 1000 training samples and 0.5 noise half-width under various polynomial degrees, and then study the problem with



Random Search for SPO+

Hyperparameter	Importance	Correlation
Batch Size	0.223	-0.475
Learning Rate	0.211	0.311
L1 Reg	0.040	-0.090
L2 Reg	0.023	-0.012
Optimizer Adam	0.180	0.483
Optimizer SGD	0.082	0.483

Effect of SPO+ Hyperparameters



Random Search for DBB

Hyperparameter	Importance	Correlation
Batch Size	0.135	-0.340
Learning Rate	0.288	0.060
L1 Reg	0.050	0.016
L2 Reg	0.127	0.018
Smooth	0.086	0.158
Optimizer Adam	0.080	0.068
Optimizer SGD	0.078	-0.068

Effect of DBB Hyperparameters

Fig. 14: Hyperparameter Tuning for the TSP: The figure summarizes the validation performance of a wide range of hyperparameter configurations. Each spline is a hyperparameter combination with the values denoted on the appropriate axis. The rightmost axis of the figure shows the regret (lower is better). For the table, the correlations are the linear relationship between a hyperparameters and the regret.

increasing number of resources (knapsacks). The number of items is 32 and the capacity for each resource is 20. As before, each experiment was performed 10 times with different randomly sampled datasets. Finally, we visualize the average normalized regret on the test set with standard deviation under 1D, 2D and 3D knapsack in Figure 17, which shows the regret loss grows slightly as the number of constraints increases.

Finding #7

As the optimization model grows larger in terms of variables and/or constraints, the decision quality of both two-stage and end-to-end predict-then-optimize approaches degrades slightly.

6.8 Impact of Neural Network Architecture

While the previous experiments were based on linear regression models, i.e., neural networks without hidden layers, we investigate here the effect of deeper neural network architectures. We examine learning curves to assess the impact of deeper/wider neural networks on the training process. Since different net-

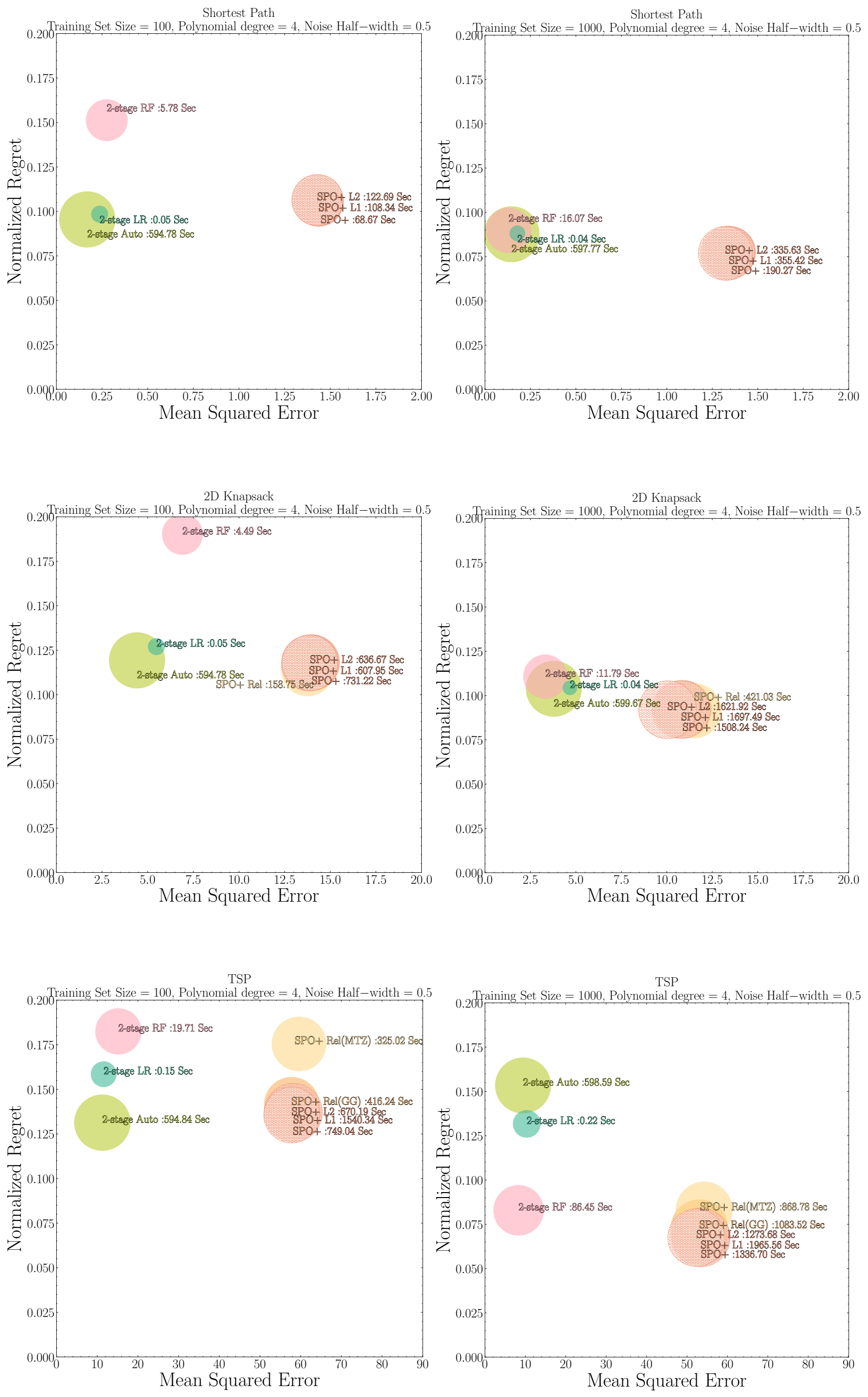


Fig. 15: MSE v.s. Regret: The result covers different two-stage methods, `SPO+` and its variants, including relaxation and regularization. `DBB` is omitted because it is far away from others. The size of the circles is proportional to the logarithm of the training time, so the smaller is better.

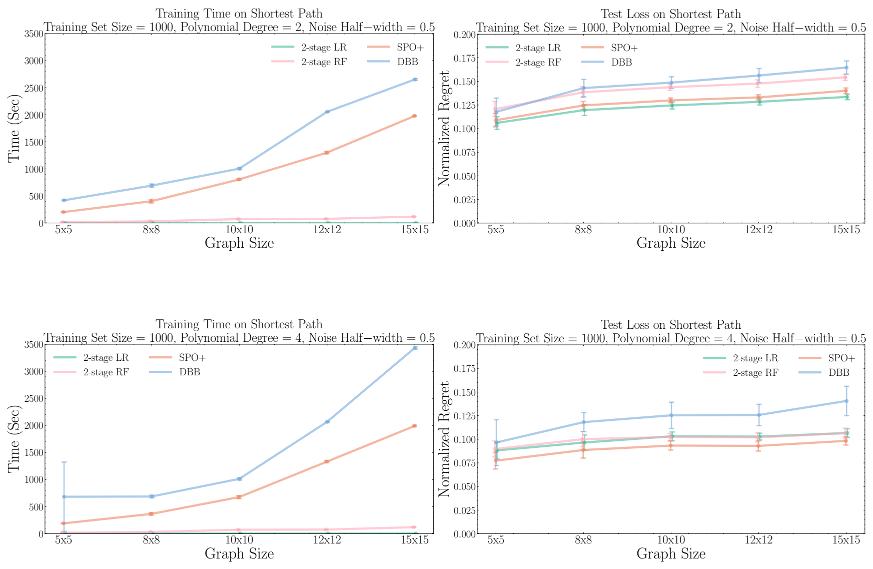


Fig. 16: Normalized regret on the test set increases with the graph size: With the increasing of the graph size of the shortest path problem, the normalized regret and training time on two-stage with linear regression, two-stage with random forest, `SPO+`, and `DBB` rises. Lower is the better.

work architectures have different numbers of floating point operations, we use process time instead of epochs as the horizontal axis. We select a 2D knapsack dataset as an example, with a training sample size of 1000, a noise half-width of 0.5, and a polynomial degree of 4.

As Figure 18 shows, training with the linear model is relatively stable for our dataset, and a slight addition of nonlinearity can help the model converge faster. Furthermore, `DBB` seems to be more sensitive to neural network architectures. Inappropriate network architecture can have more negative effects for `DBB`, even causing divergence at times.

Finding #8

For end-to-end learning, deeper neural network models can yield better decisions ultimately, but also run the risk of training instability due to the non-convex nature of the neural network training problem.

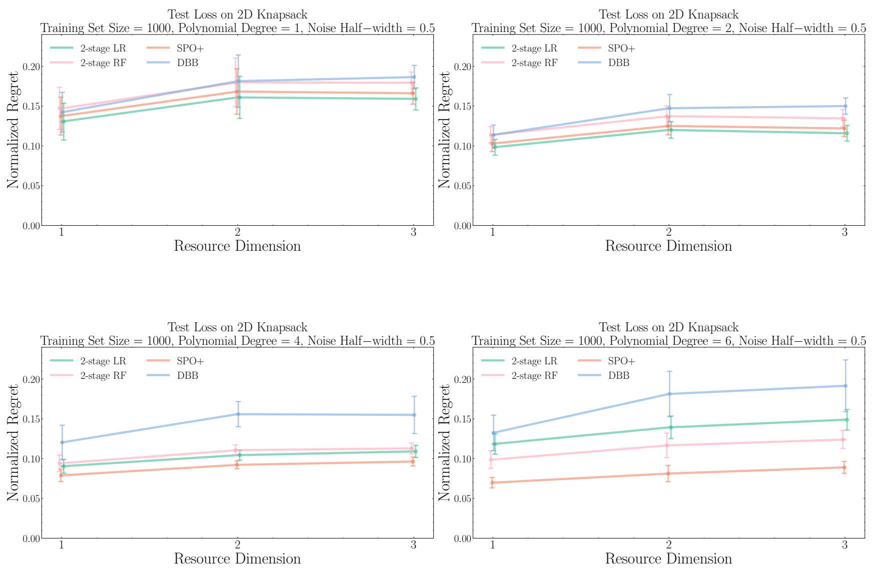


Fig. 17: Normalized regret on the test set increases with the Knapsack dimension: With the increasing of the dimension of the knapsack problem from 1 to 3, the normalized regret on two-stage with linear regression, two-stage with random forest, `SPO+`, and `DBB` rises. Lower is the better.

7 Conclusion

Because of the lack of easy-to-use generic tools, the potential power of the end-to-end predict-then-optimize has been underestimated or even overlooked in various applications. Our *PyEPO* package aims to alleviate barriers between the theory and practice of the end-to-end approach.

PyEPO, the *PyTorch*-based end-to-end predict-then-optimize tool, is specifically designed for linear objective functions, including linear programming and (mixed) integer programming. The tool is extended from the automatic differentiation function of *PyTorch*, one of the most widespread open-source machine learning frameworks. Hence, with *PyTorch*, *PyEPO* allows to leverage of numerous state-of-art deep learning models and techniques as they have been implemented in *PyTorch*.

PyEPO allows users to build optimization problems as black boxes, additionally providing user interfaces to *GurobiPy* and *Pyomo*. With *GurobiPy* and *Pyomo*, *PyEPO* provides broad compatibility with both commercial and open-source solvers and supports high-level modeling languages.

As part of the *PyEPO* framework, we generate three synthetic benchmark datasets, including shortest paths, knapsack, and traveling salesperson, with varying computational complexities. Comprehensive experiments and analysis

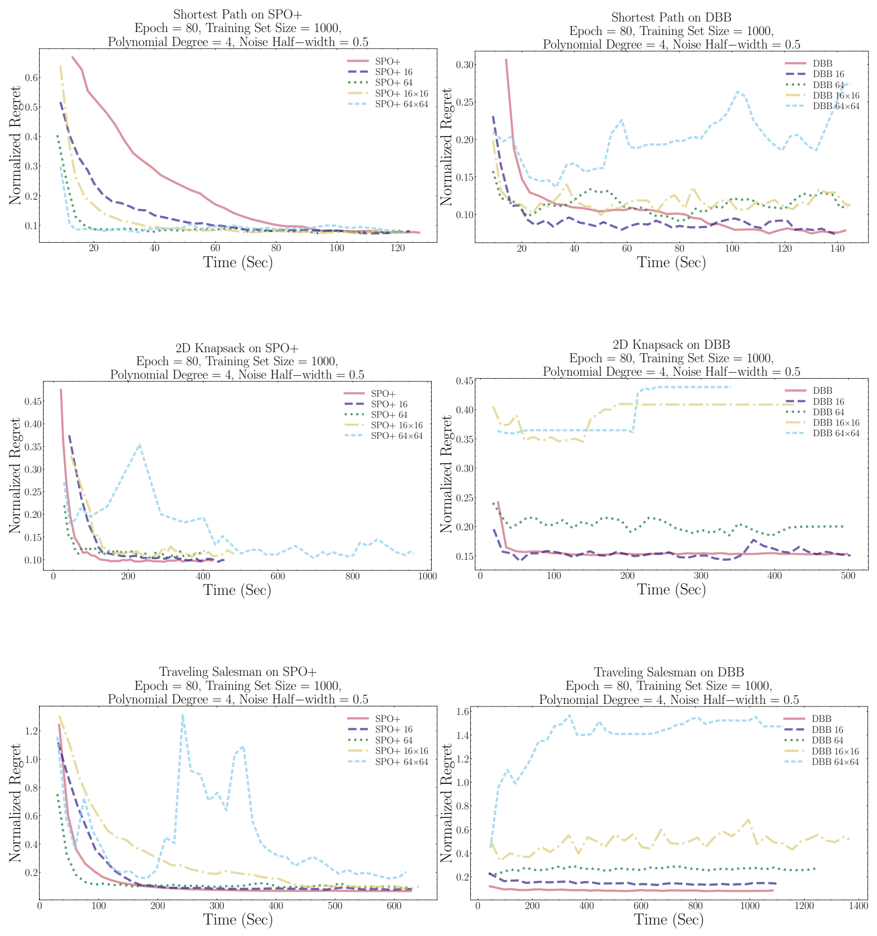


Fig. 18: Learning curve on different neural network architecture: The experiment covers `SPO+` and `DBB` for the shortest path, knapsack, and TSP. The learning curve is about the normalized regret on the test dataset, in which process time is the x-axis.

are conducted with these datasets, and the results shows that the end-to-end methods achieved excellent improvements in decision qualities over two-stage methods in many cases. In addition, the end-to-end models can benefit from the use of relaxations, regularization, and hyperparameter tuning.

The future development efforts of *PyEPO* include

– New applications and new optimization problems, including linear objective functions with mixed-integer variables or non-linear constraints;

- Additional variations and improvements to `SPO+` and `DBB`, such as warm starting and training speed up;
- Novel training methods that leverage the gradient computation features that *PyEPO* provides;
- Other existing end-to-end predict-then-optimize approaches, such as `QPTL` and its variants;
- Richer contextual features including images such as in the shortest path example in [33] or text data such as in news article recommendation [13].

References

1. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, et al. (2016) Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:160304467
2. Agrawal A, Amos B, Barratt S, Boyd S, Diamond S, Kolter JZ (2019) Differentiable convex optimization layers. In: Wallach H, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox E, Garnett R (eds) Advances in Neural Information Processing Systems, Curran Associates, Inc., vol 32
3. Agrawal A, Barratt S, Boyd S, Busseti E, Moursi WM (2019) Differentiating through a cone program. arXiv preprint arXiv:190409043
4. Amos B, Kolter JZ (2017) Optnet: Differentiable optimization as a layer in neural networks. In: International Conference on Machine Learning, PMLR, pp 136–145
5. Bengio Y (1997) Using a financial training criterion rather than a prediction criterion. International Journal of Neural Systems 8(04):433–443
6. Berthet Q, Blondel M, Teboul O, Cuturi M, Vert JP, Bach F (2020) Learning with differentiable perturbed optimizers. arXiv preprint arXiv:200208676
7. Biewald L (2020) Experiment tracking with weights and biases. URL `https://www.wandb.com/`, software available from wandb.com
8. Chen T, Li M, Li Y, Lin M, Wang N, Wang M, Xiao T, Xu B, Zhang C, Zhang Z (2015) Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:151201274
9. Cplex II (2009) V12. 1: User's manual for cplex. International Business Machines Corporation 46(53):157
10. Dantzig G, Fulkerson R, Johnson S (1954) Solution of a large-scale traveling-salesman problem. Journal of the operations research society of America 2(4):393–410
11. Djolonga J, Krause A (2017) Differentiable learning of submodular models. In: Guyon I, Luxburg UV, Bengio S, Wallach H, Fergus R, Vishwanathan S, Garnett R (eds) Advances in Neural Information Processing Systems, Curran Associates, Inc., vol 30
12. Donti P, Amos B, Kolter JZ (2017) Task-based end-to-end model learning in stochastic optimization. In: Guyon I, Luxburg UV, Bengio S, Wallach

H, Fergus R, Vishwanathan S, Garnett R (eds) Advances in Neural Information Processing Systems, Curran Associates, Inc., vol 30
13. Elmachtoub A, Liang JCN, McNellis R (2020) Decision trees for decision-making under the predict-then-optimize framework. In: International Conference on Machine Learning, PMLR, vol 119, pp 2858–2867
14. Elmachtoub AN, Grigas P (2021) Smart "predict, then optimize". Management Science 0(0)
15. Ferber A, Wilder B, Dilkina B, Tambe M (2020) Mipaal: Mixed integer program as a layer. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol 34, pp 1504–1511
16. Feurer M, Klein A, Eggensperger J Katharina Springenberg, Blum M, Hutter F (2015) Efficient and robust automated machine learning. In: Advances in Neural Information Processing Systems 28 (2015), pp 2962–2970
17. Ford B, Nguyen T, Tambe M, Sintov N, Delle Fave F (2015) Beware the soothsayer: From attack prediction accuracy to predictive reliability in security games. In: International Conference on Decision and Game Theory for Security, Springer, pp 35–56
18. Gavish B, Graves SC (1978) The travelling salesman problem and related problems
19. Gurobi Optimization, LLC (2021) Gurobi Optimizer Reference Manual. URL `https://www.gurobi.com`
20. Hagberg A, Swart P, S Chult D (2008) Exploring network structure, dynamics, and function using networkx. Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States)
21. Hart WE, Laird CD, Watson JP, Woodruff DL, Hackebeil GA, Nicholson BL, Siirola JD, et al. (2017) Pyomo-optimization modeling in python, vol 67. Springer
22. Kao Yh, Roy B, Yan X (2009) Directed regression. In: Bengio Y, Schuurmans D, Lafferty J, Williams C, Culotta A (eds) Advances in Neural Information Processing Systems, Curran Associates, Inc., vol 22
23. Kingma DP, Ba J (2014) Adam: A method for stochastic optimization. arXiv preprint arXiv:14126980
24. Mandi J, Guns T (2020) Interior point solving for lp-based prediction+optimisation. In: Larochelle H, Ranzato M, Hadsell R, Balcan MF, Lin H (eds) Advances in Neural Information Processing Systems, Curran Associates, Inc., vol 33, pp 7272–7282
25. Mandi J, Stuckey PJ, Guns T, et al. (2020) Smart predict-and-optimize for hard combinatorial optimization problems. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol 34, pp 1603–1610, DOI 10.1609/aaai.v34i02.5521
26. Martello S, Toth P (1990) Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc.
27. Mattingley J, Boyd S (2012) Cvxgen: A code generator for embedded convex optimization. Optimization and Engineering 13(1):1–27

28. Miller CE, Tucker AW, Zemlin RA (1960) Integer programming formulation of traveling salesman problems. Journal of the ACM (JACM) 7(4):326–329
29. Ortega-Arranz H, Llanos DR, Gonzalez-Escribano A (2014) The shortest-path problem: Analysis and comparison of methods. Synthesis Lectures on Theoretical Computer Science 1(1):1–87
30. Paszke A, Gross S, Chintala S, Chanan G, Yang E, DeVito Z, Lin Z, Desmaison A, Antiga L, Lerer A (2017) Automatic differentiation in pytorch. In: NIPS 2017 Autodiff Workshop
31. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) Pytorch: An imperative style, high-performance deep learning library. In: Wallach H, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox E, Garnett R (eds) Advances in Neural Information Processing Systems 32, Curran Associates, Inc., pp 8024–8035
32. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12:2825–2830
33. Pogančić MV, Paulus A, Musil V, Martius G, Rolinek M (2019) Differentiation of blackbox combinatorial solvers. In: International Conference on Learning Representations
34. Wilder B, Dilkina B, Tambe M (2019) Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol 33, pp 1658–1665

8 Statements and Declarations

8.1 Funding

This work was supported by funding from a SCALE AI Research Chair and an NSERC Discovery Grant.

8.2 Author Contributions

Tang and Khalil contributed to the conception and design of the project. Software development, data generation, software testing, and benchmarking experiments were performed by Tang. Tang wrote the first draft of the submission. Tang and Khalil contributed to finalizing the submission. Both authors read and approved the final manuscript.

8.3 Competing Interests

The authors have no relevant financial or non-financial interests to disclose.