Mathematical Programming Computation (2024) 16:297–335
https://doi.org/10.1007/s12532-024-00255-x

FULL LENGTH PAPER

PyEPO: a PyTorch-based end-to-end predict-then-optimize library for linear and integer programming

Bo Tang[1] · **Elias B. Khalil**[1]

Received: 12 April 2022 / Accepted: 18 April 2024 / Published online: 18 July 2024

Abstract

In deterministic optimization, it is typically assumed that all problem parameters are fixed and known. In practice, however, some parameters may be a priori unknown but can be estimated from contextual information. A typical predict-then-optimize approach separates predictions and optimization into two distinct stages. Recently, end-to-end predict-then-optimize has emerged as an attractive alternative. This work introduces 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 includes various algorithms such as surrogate decision losses, black-box solvers, and perturbated methods. *PyEPO* offers a user-friendly interface for defining new optimization problems, applying state-of-the-art algorithms, and using custom neural network architectures. We conducted experiments comparing various methods on problems such as the Shortest Path, the Multiple Knapsack, and the Traveling Salesperson Problem, discussing empirical insights that may 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 90-04 · 90C11 · 62J05

✉ Elias B. Khalil
khalil@mie.utoronto.ca

Bo Tang
botang@mie.utoronto.ca

[1] Department of Mechanical and Industrial Engineering, University of Toronto, 5 King's College Road, Toronto, ON M5S 3G8, Canada

1 Introduction

Predictive modeling is ubiquitous in real-world decision-making. For instance, in many applications, the objective function coefficients, such as travel time in a routing problem, electricity price in a power system, and asset return in portfolio optimization, are unknown at the time of decision-making. In this work, we focus on the commonly used paradigm of prediction followed by optimization in the context of linear programs or integer linear programs. In this paradigm, it is assumed that a set of features describes an instance of the optimization problem. A regression model maps the features to the (unknown) objective function coefficients. Then, a deterministic optimization problem is 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 garnered increasing attention in recent years.

One natural approach is to proceed in two stages, first training an accurate predictive model and then solving the downstream optimization problem using predicted coefficients. The advantage of the two-stage approach is the direct utilization of existing machine learning methods and optimization solvers. However, prediction errors, such as mean squared error, cannot adequately measure the quality of decisions. While a perfect prediction would always yield an optimal decision, learning a prediction model without errors is impractical. Bengio [5], Ford et al. [19], and Elmachtoub and Grigas [16] reported that training a predictive model based on prediction error leads to worse decisions compared to directly considering the decision error. Consequently, the state-of-the-art alternative is to integrate optimization into prediction, taking into account its 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* [23],*CPLEX* [9] and *COPT* [21] are highly optimized, allowing users to easily express business or academic problems as optimization models without a deep understanding of the underlying theory and algorithms. However, embedding a solver for end-to-end learning necessitates additional computation and integration (e.g., gradient calculation), which current software does not provide.

On the other hand, the field of machine learning has witnessed tremendous growth in recent decades. Breakthroughs in deep learning have led to remarkable improvements in several complex tasks. As a result, neural networks now pervade disparate applications that span computer vision, natural language, and planning, among others. Python-based machine learning frameworks such as *Scikit-Learn* [39], *TensorFlow* [1], *PyTorch* [38], *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 capability to handle constrained optimization such as integer linear programming.

Since Amos and Kolter [4] first introduced a neural network layer for generic mathematical optimization, there have been several prominent attempts to bridge the gap between optimization solvers and deep learning frameworks. A critical component is typically a differentiable block for optimization tasks. With a differentiable optimizer

or a surrogate decision loss function, 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.

Although research code implementing a number of predict-then-optimize training algorithms has been made available for particular classes of optimization problems and/or predictive models [2, 3, 12, 14, 16, 17, 22, 28, 28, 40, 42], there is a pressing need for a generic end-to-end learning framework, particularly for linear and integer programming. In this paper, we propose the open-source software package *PyEPO* that aims to customize and train end-to-end predict-then-optimize for optimization problems with linear objective functions, such as linear and integer programming. Our contributions are as follows:

1. We implement **`SPO+`** ("Smart Predict-then-Optimize+") loss [16], **`DBB`** (differentiable black-box) solver [40], **`DPO`** (differentiable perturbed optimizer), **`PFYL`** (perturbed Fenchel-Young loss), among other 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* facilitates easy integration and usage of any deep neural network.
3. We provide interfaces to the Python-based optimization modeling frameworks *GurobiPy*, *COPTPy* and *Pyomo*. These high-level modeling languages allow non-specialists to formulate optimization models with *PyEPO*.
4. We enable parallel computing for the forward and backward pass in *PyEPO*. Optimizations during training are carried out in parallel, allowing users to leverage 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. The performance of different methods across various datasets demonstrate the competitiveness of end-to-end learning and highlight the surprising effect of relaxations and regularization.

PyEPO focuses on the development of software that offers interfaces for various methodologies. It also provides new benchmark datasets and conducts experiments. In addition to *PyEPO*, there are comprehensive surveys [31, 41] that complement this paper. Sadana et al. [41] provide a theoretical and algorithmic review and taxonomy for more general contextual optimization, including predict-then-optimize approaches. Meanwhile, the survey by Mandi et al. [31] delivers an extensive analysis and experiments on end-to-end predict-then-optimize methods. Notably, the experiments on Mandi et al. [31] differ from *PyEPO*: they test more methods on several different datasets but do not explore the implications of relaxation or coefficients regularization.

2 Related work

In early work on the topic, Bengio [5] introduced a differentiable portfolio optimizer and suggested that direct optimization with financial criteria performs better in neural

networks than the mean squared error of the predicted values. Kao et al. [26] trained a linear regressor with a convex combination of prediction error and decision error, but only considered unconstrained quadratic programming. Domke [13] investigated generic gradient descent methods to minimize unconstrained energy functions.

More recently, interest has shifted towards constrained optimization problems. For instance, Gould et al. [22] allows differentiating bilevel optimization problems with and without constraints. Subsequent studies have comprehensively explored differentiable constraint optimization, covering various types of optimization problems such as quadratic programming, linear programming, and integer linear programming. A comparison of gradient-based methodologies for end-to-end constrained optimization is presented in Tables 1 and 2. Additionally, Elmachtoub et al. [15] proposed a tree-based approach that does not rely on gradients.

2.1 KKT-based implicit differentiation

Gradient-based end-to-end learning requires well-defined first-order derivatives of optimization. The Karush-Kuhn-Tucker (KKT) conditions become an attractive option because they make the optimization problem with hard constraints differentiable.

Amos and Kolter [4] proposed **`OptNet`**, which derives implicit differentiation of constrained quadratic programs from KKT conditions. Building on **`OptNet`**, Donti et al. [14] investigated a general end-to-end framework, **`DQP`**, for learning with constrained quadratic programming, which simultaneously obtains optimal solutions and their gradients. Although linear programming is a special case of quadratic programming, **`DQP`** has no ability to tackle linear objective functions because the optimal solution exhibits piecewise constancy, leading to zero gradients almost everywhere.

Furthermore, Wilder et al. [42] added a small quadratic objective term to **`DQP`** to ensure nonzero gradients for linear programming, resulting in **`QPTL`**. Wilder et al. [42] also discussed relaxation and rounding for the approximation of binary problems. Ferber et al. [17] 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 [28] introduced **`IntOpt`** to compute gradients for linear programming with the log-barrier term instead of the quadratic term of **`QPTL`**. Except for **`MIPaaL`** [17], end-to-end learning approaches for (mixed) integer programming use linear relaxation during training but evaluate with optimal integer solutions at test time.

In addition to **`DQP`** and its extension, Agrawal et al. [2] introduced a generic framework, **`CvxpyLayers`**, which differentiates through KKT conditions of the conic program. The central idea in **`CvxpyLayers`** is to canonicalize disciplined convex programming as conic programming so that it is applicable to a wider range of convex optimization problems. Similarly to **`DQP`**, **`CvxpyLayers`** encounters difficulties when differentiating through linear programs. Thus, a squared solution is used as a heuristic.

Table 1 Methodology comparison

Method	In PyEPO	w/Unk Constr	Disc Var	Lin Obj	Quad Obj
DQP [14] [code]	✗	✓	✗	✗	✓
QPTL [42] [code]	✗	✗	✗	✓	✓
MIPaaL [17]	✗	✗	✓	✓	✗
IntOpt [28] [code]	✗	✗	✗	✓	✗
CvxpyLayers [2] [code]	✗	✓	✗	✓	✓
SPO+ [16] [code]	✓	✗	✓	✓	✗
SPO+ Rel [29] [code]	✓	✗	✓	✓	✗
SPO+ WS [29] [code]	✗	✗	✓	✓	✗
DBB [40] [code]	✓	✗	✓	✓	✗
DPO [6] [code]	✓	✗	✓	✓	✗
PFYL [6] [code]	✓	✗	✓	✓	✗
NCE [35] [code]	✓	✗	✓	✓	✓
LTR [30] [code]	✓	✗	✓	✓	✓

This is a comparison diagram for different methodologies
The first set of methods uses KKT conditions, and the second part is based on differentiable approximations
"In PyEPO" denotes whether the method is available in *PyEPO*
"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

Table 2 Computational cost per gradient calculation for different methodologies

Method	Computation per gradient
DQP [14]	GPU-based interior-point method for quadratic programming
QPTL [42]	GPU-based interior-point method for quadratic programming
MIPaaL [17]	Cutting-plane method + GPU-based interior-point method for quadratic programming
IntOpt [28]	GPU-based interior-point method for quadratic programming
CvxpyLayers [2]	GPU-based interior-point method for conic programming
SPO+ [16]	Linear/integer programming
SPO+ Rel [29]	Linear programming
SPO+ WS [29]	Integer programming with warm starting
DBB [40]	Two Linear/integer programming
DPO [6]	Sampling with multiple linear/integer programming solves with random noise
PFYL [6]	Sampling with multiple linear/integer programming solves with random noise
NCE [35]	Linear/integer programming
LTR [30]	Linear/integer programming

However, these KKT-based methods require a specific quadratic or conic programming solver, and linear programming necessitates additional objective terms. Therefore, the efficiency and accuracy of the above implementations is not comparable to commercial MILP solvers such as *Gurobi* [23] and *CPLEX* [9]. Moreover, these methods are designed for continuous problems and do not naturally support discrete optimization.

2.2 Surrogate loss or gradients

Since KKT conditions may not be ideal for linear objective functions, researchers have also explored the design of surrogate loss or gradients. For example, Elmachtoub and Grigas [16] proposed regret (SPO loss in their paper) to measure decision error. However, the gradient of regret is still zero almost everywhere and is undefined otherwise for the linear objective function. To address this, they introduced **SPO+**, a convex and sub-differentiable loss, to ensure a nonzero subgradient for training. As in previous approaches, **SPO+** solves an optimization problem in each forward pass. Unlike KKT-based methods, **SPO+** is limited to the linear objective function. Because optimization is the computational bottleneck, Mandi et al. [29] applied **SPO+** to combinatorial problems using relaxation and warm starting techniques. They reported that relaxation reduces solving time at the cost of performance.

Pogančić et al. [40] computed a subgradient from continuous interpolation of solutions, an approach they referred to as the “differentiable black-box solver", **DBB**. The interpolation approximation is nonconvex but avoids vanishing gradients. Compared to **SPO+**, **DBB** requires an extra optimization for the backward pass, and the loss of **DBB** is flexible (e.g., the Hamming distance in their paper).

In addition, Berthet et al. [6] applied perturbed optimizer **DPO** by adding random noise to the objective function coefficients so that the piecewise constant solutions are smoothed by probability and further constructed the Fenchel-Young loss **PFYL** by duality. As a sampling-based method, the number of optimization problems that need to be solved for each iteration is the number of samples.

Moreover, Mulamba et al. [35] adopted the noise-contrastive estimate (**NCE**) to compute a surrogate loss. This approach treats a subset of suboptimal feasible solutions as negative samples and maximizes the gap between the optimal solution and these negative samples. Inspired by **NCE**, Mandi et al. [30] converted the predict-then-optimize task into a "Learning to Rank" (**LTR**) problem [27], ranking the subset of feasible solutions based on the objective value. Both **NCE** and **LTR** offer the flexibility of being unrestricted in terms of the type of optimization problem, which presents a significant advantage. However, to our knowledge, these methods have not been tested on non-linear problems.

As the key algorithms of *PyEPO*, **SPO+**, **DBB**, **DPO**, and **PFYL** are further discussed in Sect. 3.

2.3 Code for predict-then-optimize

2.3.1 Research code

Table 1 lists and links to the codebases that will be discussed next. Amos and Kolter [4] developed a *PyTorch*-based solver **qpth** for **OptNet**, which efficiently solves quadratic programs and computes their gradients. The solver was based on a primal-dual interior-point method [33] and can handle batches of quadratic programs on a GPU. Using this solver, Donti et al. [14] provided an open-source repository to reproduce the **DQP** experiments; the repository was specifically designed for inventory, power scheduling, and battery storage problems. Wilder et al. [42] provided code for **QPTL** for budget allocation, bipartite matching, and diverse recommendation. **MIPaaL** [17] relied on **qpth** and used *CPLEX* to generate cutting planes, but open-source code is not available. Mandi and Guns [28] released **IntOpt** code for knapsack, shortest path, and power scheduling.

Berthet et al. [6] contributed the *TensorFlow*-based **DPO** and **PFYL** implementation, which provided universal functions for end-to-end predict-then-optimize but required users to create additional helper methods for tensor operations. Elmachtoub and Grigas [16] provided an implementation of **SPO+** in Julia, which contained the shortest path and portfolio optimization problems, while Mandi et al. [29] implemented **SPO+** with Python for the knapsack and power scheduling problems. The code for **DBB** [40] is applied to the shortest path, the travel salesperson, the ranking, perfect matching, and graph matching are available. Additionally, **NCE** [35] covers the knapsack, energy-cost scheduling, and bipartite matching, while **LTR** [30] covers the shortest path, energy-cost scheduling, and bipartite matching problems.

Except for **DPO** and **PFYL**, the above contributions provided solutions to specific optimization problems, and Berthet et al. [6] did not package **DPO** and **PFYL** as a generic library. In conclusion, they were confined to research-grade code aimed at reproducing experimental results.

2.3.2 Software package

CvxpyLayers [2] is the first generic end-to-end predict-then-optimize learning framework. Unlike the aforementioned codes, it requires modeling with a domain-specific language *CVXPY*, which is embedded into a differentiable layer in a straightforward manner. 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 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 its applicability to many real-world decision-making problems.

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 objective function coefficients (cost vector) 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 $\boldsymbol{b} \in \mathbb{R}^k$:

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

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

$$\begin{aligned} \min_{\boldsymbol{w}} \quad & \boldsymbol{c}^\intercal \boldsymbol{w} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{w} \leq \boldsymbol{b} \\ & \boldsymbol{w} \geq \boldsymbol{0} \\ & w_i \in \mathbb{Z} \quad \forall i \in I \subseteq \{1, 2, ..., d\} \end{aligned} \tag{2}$$

For linear and integer programming, let S be the feasible region, $z^*(\boldsymbol{c})$ be the optimal objective value with respect to the objective function coefficients $\boldsymbol{c}$, 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 above, some coefficients are unknown and need to be predicted prior to optimization. Here, we assume that only the objective function coefficients $\boldsymbol{c}^i$ are unknown, but they correlate with a feature vector $\boldsymbol{x}^i \in \mathbb{R}^p$. Given a training dataset $\mathcal{D} = \{(\boldsymbol{x}^1, \boldsymbol{c}^1), (\boldsymbol{x}^2, \boldsymbol{c}^2), ..., (\boldsymbol{x}^n, \boldsymbol{c}^n)\}$ or $\mathcal{D} = \{(\boldsymbol{x}^1, \boldsymbol{w}^*(\boldsymbol{c}^1)), (\boldsymbol{x}^2, \boldsymbol{w}^*(\boldsymbol{c}^2)), ..., (\boldsymbol{x}^n, \boldsymbol{w}^*(\boldsymbol{c}^n))\}$, a machine learning predictor $g(\cdot)$ can be trained to minimize a loss function $\mathcal{L}(\cdot)$, where $\boldsymbol{\theta}$ are predictor parameters and $\hat{\boldsymbol{c}}^i = g(\boldsymbol{x}^i; \boldsymbol{\theta})$ is the prediction of the objective function coefficients $\boldsymbol{c}^i$.

3.2 The two-stage method

As Fig. 1 shows, the two-stage approach trains a predictor $g(\cdot)$ by minimizing a loss function with respect to the true coefficients $\boldsymbol{c}$, such as mean squared error (MSE), $\mathcal{L}_{\text{MSE}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \frac{1}{n}\|\hat{\boldsymbol{c}} - \boldsymbol{c}\|^2$. Following training and given an instance with feature vector $\boldsymbol{x}$, the predictor produces coefficients $\hat{\boldsymbol{c}} = g(\boldsymbol{x}; \boldsymbol{\theta})$, which is then used to solve the optimization problem. It decomposes the predict-then-optimize problem into a traditional regression and then an optimization. However, it introduces a mismatch between the decision error and the prediction error.

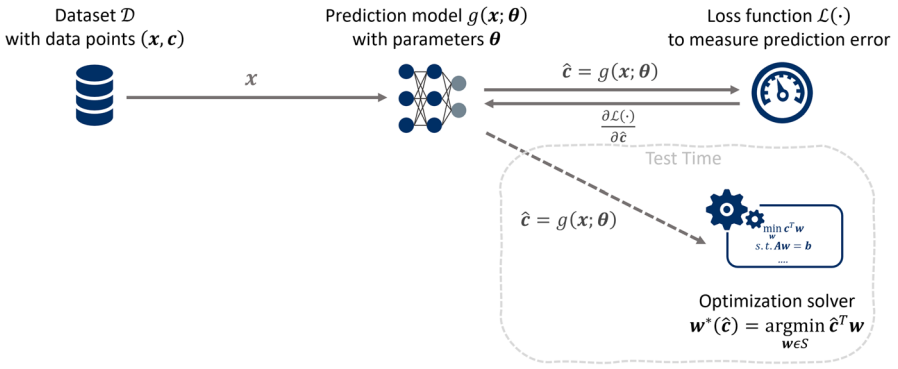

Fig. 1 Illustration of the two-stage predict-then-optimize framework: a labeled dataset $\mathcal{D}$ of $(\boldsymbol{x}, \boldsymbol{c})$ pairs is used to fit a machine learning predictor that minimizes prediction error. At test time (grey box), the predictor is used to estimate the parameters of an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

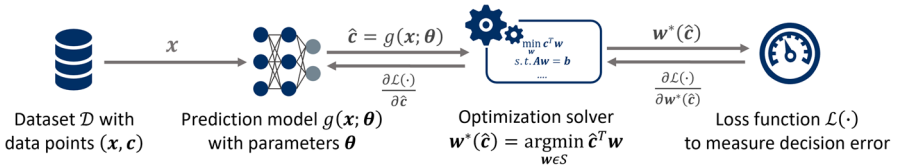

Fig. 2 Illustration of the end-to-end predict-then-optimize framework: a labeled dataset $\mathcal{D}$ of $(\boldsymbol{x}, \boldsymbol{c})$ (or $(\boldsymbol{x}, \boldsymbol{w}^*(\boldsymbol{c}))$) pairs is used to fit a machine learning predictor that directly minimizes decision error. The critical 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 Fig. 1

3.3 Gradient-based end-to-end predict-then-optimize

The main drawbacks of the two-stage approach are that it does not account for decision error during training and that it always requires the true coefficients as labels for supervised learning. In contrast, the end-to-end predict-then-optimize method depicted in Fig. 2 aims to minimize the decision error and has the potential to learn with only optimal solutions. In line with deep learning terminology, we will use the term "backward pass" to refer to the gradient computation and parameter updating 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 predictor parameters.

For an appropriately defined loss function that penalizes decision error, the chain rule can be used to calculate the gradient of the loss with respect to the predictor parameters as follows:

$$\frac{\partial \mathcal{L}(\hat{\boldsymbol{c}}, \cdot)}{\partial \boldsymbol{\theta}} = \frac{\partial \mathcal{L}(\hat{\boldsymbol{c}}, \cdot)}{\partial \hat{\boldsymbol{c}}} \frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}} \tag{3}$$

$$= \frac{\partial \mathcal{L}(\hat{\boldsymbol{c}}, \cdot)}{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})} \frac{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}{\partial \hat{\boldsymbol{c}}} \frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}} \tag{4}$$

Algorithm 1 End-to-end Gradient Descent

Require: coefficient matrix $\boldsymbol{A}$, right-hand side $\boldsymbol{b}$, data $\mathcal{D}$
Initialize predictor parameters $\boldsymbol{\theta}$ for predictor $g(\boldsymbol{x}; \boldsymbol{\theta})$
for epochs **do**
  for each batch of training data $(\boldsymbol{x}, \boldsymbol{c})$ or $(\boldsymbol{x}, \boldsymbol{w}^*(\boldsymbol{c}))$ **do**
    Sample batch of the $\boldsymbol{c}$ or $\boldsymbol{w}^*(\boldsymbol{c})$ with the corresponding features $\boldsymbol{x}$
    Predict coefficients using predictor $\hat{\boldsymbol{c}} := g(\boldsymbol{x}; \boldsymbol{\theta})$
    Forward pass to compute optimal solution $\boldsymbol{w}^*(\hat{\boldsymbol{c}}) := \text{argmin}_{\boldsymbol{w} \in S} \hat{\boldsymbol{c}}^\intercal \boldsymbol{w}$
    Forward pass to compute decision loss $\mathcal{L}(\hat{\boldsymbol{c}}, \cdot)$
    Backward pass from loss $\mathcal{L}(\hat{\boldsymbol{c}}, \cdot)$ to update parameters $\boldsymbol{\theta}$ with gradient
  end for
end for

$$\text{Note: } \frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}} = \frac{\partial g(\boldsymbol{x}; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{5}$$

The last term $\frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}}$ is the gradient of the predictions with respect to the model parameters, which is trivial to calculate in modern deep learning frameworks. The challenging part is to compute the a surrogate decision loss function $\frac{\partial \mathcal{L}(\hat{\boldsymbol{c}}, \cdot)}{\partial \hat{\boldsymbol{c}}}$ or constrained optimization gradient $\frac{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}{\partial \hat{\boldsymbol{c}}}$. Since the mapping from coefficients $\boldsymbol{c}$ to solution vector $\boldsymbol{w}^*$ is piecewise constant for linear and integer programming, the predictor parameters cannot be updated with gradient descent. Thus, both **SPO+** and **PFYL** are surrogate loss functions that adhere to Eq. 3, which derive the gradient of decision loss $\frac{\partial \mathcal{L}(\hat{\boldsymbol{c}}, \cdot)}{\partial \hat{\boldsymbol{c}}}$, while **DBB** and **DPO**, follow Eq. 4, approximate $\frac{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}{\partial \hat{\boldsymbol{c}}}$. Furthermore, **DBB** and **DPO** provide flexibility to customize loss functions, and the use of loss functions in our experiments is detailed in Tables 3 and 5.

3.3.1 Decision loss

The concept of regret (also known as SPO loss [16]) has been introduced to quantify the error in decision-making. It is defined as the gap of the objective value between the true optimal solution $\boldsymbol{w}^*(\boldsymbol{c})$ and the optimal solution acquired using the predicted coefficients $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$. Formally, it is represented as:

$$\mathcal{L}_{\text{Regret}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \boldsymbol{c}^\intercal \boldsymbol{w}^*(\hat{\boldsymbol{c}}) - z^*(\boldsymbol{c}). \tag{6}$$

Given the coefficients of the objective function $\hat{\boldsymbol{c}}$, there may be multiple optimal solutions to $\min_{\boldsymbol{w} \in S} \hat{\boldsymbol{c}}^\intercal \boldsymbol{w}$. To address this, Elmachtoub and Grigas [16] devised the "unambiguous" regret (also called an unambiguous SPO loss): $\mathcal{L}_{\text{URegret}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \max_{\boldsymbol{w} \in W^*(\boldsymbol{c})} \boldsymbol{w}^\intercal \boldsymbol{c} - z^*(\boldsymbol{c})$. This loss considers the worst case among all optimal solutions with respect to the predicted coefficients.

An example of the worst-case scenario is when the predicted coefficients $\hat{\boldsymbol{c}}$ converge to all zeros, making all feasible solutions optimal. Consequently, there is a small likelihood that the solver always selects the particular optimal solution $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$ equivalent to the true optimal solution $\boldsymbol{w}^*(\boldsymbol{c})$, leading to zero loss with zero prediction.

PyEPO provides an evaluation module (Sect. 4.7) that includes both regret and unambiguous regret. However, as Fig. 3 shows, regret and unambiguous regret are

Table 3 Methods compared in the experiments

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* [18] with 10 minutes time limit and uses MSE as metric
SPO+	Linear model with SPO+ loss [16]
PFYL	Linear model with perturbed Fenchel-Young loss [6]
DBB	Linear model with differentiable black-box optimizer [40] and regret loss
SPO+ Rel	Linear model with SPO+ loss [16], using linear relaxation for training
PFYL Rel	Linear model with perturbed Fenchel-Young loss [6], using linear relaxation for training
DBB Rel	Linear model with differentiable black-box optimizer [40] and regret loss, using linear relaxation for training
SPO+ L1	Linear model with SPO+ loss [16], using l_1 regularization for coefficients
SPO+ L2	Linear model with SPO+ loss [16], using l_2 regularization for coefficients
PFYL L1	Linear model with perturbed Fenchel-Young loss [6], using l_1 regularization for coefficients
PFYL L2	Linear model with perturbed Fenchel-Young loss [6], using l_2 regularization for coefficients
DBB L1	Linear model with differentiable black-box optimizer [40] and regret loss, using l_1 regularization for coefficients
DBB L2	Linear model with differentiable black-box optimizer [40] and regret loss, using l_2 regularization for coefficients

Table 4 Problem parameters for performance comparison

Problem	Parameters	Feature size	Label size
Shortest path	Height of the grid is 5	5	40
	Width of the grid is 5		
Knapsack	Dimension of resource is 2	5	32
	Number of items is 32		
	Capacity is 20		
Traveling salesperson	Number of nodes is 20	10	190

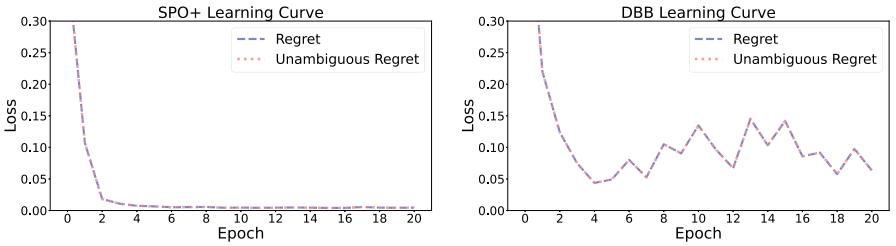

Fig. 3 As shown for the learning curves of the training of **SPO+** (left) and **DBB** (right) on the shortest path, regret and unambiguous regret in the various tasks overlap almost exactly

almost identical throughout the training procedures. Therefore, although unambiguous regret is more theoretically rigorous, it is not necessary to consider it in practice.

In addition to regret, the decision error can also be defined as the difference between the true solution and its prediction, such as the Hamming distance of the solutions [40] and the squared error of the solutions [6]. Moreover, Dalle et al.[10] also considered treating the objective value $\boldsymbol{c}^\intercal \boldsymbol{w}^*_{\hat{\boldsymbol{c}}}$ itself as a loss.

3.4 Methodologies

3.4.1 Smart predict-then-optimize [16]

To make the decision error differentiable, Elmachtoub and Grigas [16] proposed **SPO+**, a convex upper bound on the regret:

$$\mathcal{L}_{\text{SPO+}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = -\min_{\boldsymbol{w}\in S}\{(2\hat{\boldsymbol{c}} - \boldsymbol{c})^\intercal \boldsymbol{w}\} + 2\hat{\boldsymbol{c}}^\intercal \boldsymbol{w}^*(\boldsymbol{c}) - z^*(\boldsymbol{c}). \tag{7}$$

One proposed subgradient for this loss is as follows:

$$2\boldsymbol{w}^*(\boldsymbol{c}) - 2\boldsymbol{w}^*(2\hat{\boldsymbol{c}} - \boldsymbol{c}) \in \frac{\partial \mathcal{L}_{\text{SPO+}}(\hat{\boldsymbol{c}}, \boldsymbol{c})}{\partial \hat{\boldsymbol{c}}} \tag{8}$$

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

To accelerate **SPO+** training, Mandi et al. [29] employed relaxations (**SPO+ Rel**) and warm starting (**SPO+ WS**) to speed up optimization. The idea of **SPO+ Rel** is to use the continuous relaxation of the integer program during training. This simplification greatly reduces training time at the expense of model performance. Compared to **SPO+**, the improvement in **SPO+ Rel** in training efficiency is not negligible. For example, linear programming can be solved in polynomial time, while integer programming is worst-case exponential. In Sect. 6, we will further discuss this performance-efficiency trade-off. For **SPO+ WS**, Mandi et al. [29] suggested using previous solutions as a starting point for the integer programming solver, which potentially improves efficiency by narrowing down the search space.

3.4.2 Differentiable black-box solver [40]

DBB was developed by Pogančić et al. [40] to estimate gradients from finite difference, replacing the zero gradients in $\frac{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}{\partial \hat{\boldsymbol{c}}}$. For the predicted coefficients $\hat{\boldsymbol{c}}$, Pogančić et al. [40] performed a linear interpolation of the loss function $\mathcal{L}(\hat{\boldsymbol{c}}, \cdot)$ between $\hat{\boldsymbol{c}}$ and $\hat{\boldsymbol{c}} + \lambda \frac{\partial \mathcal{L}(\hat{\boldsymbol{c}}, \cdot)}{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}$. Thus, the substitute function becomes piecewise affine. Therefore, when computing $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$, a useful nonzero gradient is obtained at the cost of faithfulness. The forward and backward passes are shown in Algorithms 2 and 3. The hyperparameter $\lambda > 0$ controls the degree of interpolation. However, compared to **SPO+**, the approximation function of **DBB** is nonconvex, 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: $\frac{\partial \mathcal{L}(\hat{\boldsymbol{c}}, \cdot)}{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}$, λ
1: Load $\hat{\boldsymbol{c}}$ and $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$ from forward pass
2: $\boldsymbol{c}' := \hat{\boldsymbol{c}} + \lambda \frac{\partial \mathcal{L}(\hat{\boldsymbol{c}}, \cdot)}{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}$
3: Solve $\boldsymbol{w}^*(\boldsymbol{c}')$
4: **return** $\frac{\partial \mathcal{L}(\hat{\boldsymbol{c}}, \cdot)}{\partial \hat{\boldsymbol{c}}} := \frac{1}{\lambda}(\boldsymbol{w}^*(\boldsymbol{c}') - \boldsymbol{w}^*(\hat{\boldsymbol{c}}))$

Similarly to **SPO+**, **DBB** requires solving the optimization problem in each training iteration. Thus, utilizing a relaxation approach may also work for **DBB**. However, Pogančić et al. [40] did not consider this option. Given the potential efficiency gains that continuous relaxation can bring, we also conducted experiments for **DBB Rel** in section 6. The same applies to **DPO** and **PFYL** below.

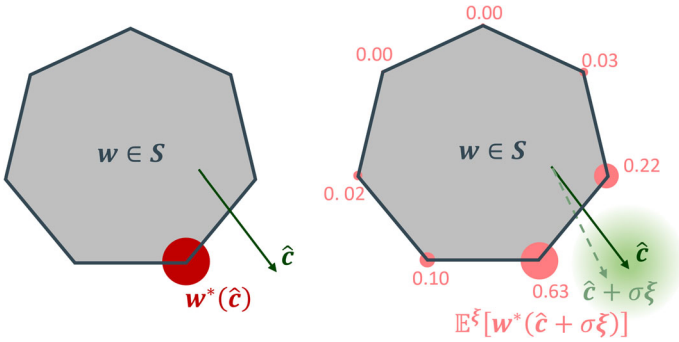

Fig. 4 Illustration of optimal solution (left) and expected optimal solution under perturbation (right): while the optimal solution $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$ remains static or shifts to a neighboring extreme point as the predicted coefficient $\hat{\boldsymbol{c}}$ changes, the perturbed expectation undergoes a smooth variation

3.4.3 Differentiable perturbed optimizer [6]

DPO [6] uses Monte-Carlo samples to estimate solutions, in which the predicted coefficients $\hat{\boldsymbol{c}}$ are perturbed by Gaussian noise $\boldsymbol{\xi} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$. Based on the K samples of perturbed coefficients $\hat{\boldsymbol{c}}+\sigma\boldsymbol{\xi}$, it returns the estimate of the optimal solution expectation

$$\mathbb{E}_{\boldsymbol{\xi}}[\boldsymbol{w}^*(\hat{\boldsymbol{c}} + \sigma\boldsymbol{\xi})] \approx \frac{1}{K}\sum_{\kappa}^{K} \boldsymbol{w}^*(\hat{\boldsymbol{c}} + \sigma\boldsymbol{\xi}_{\kappa}).$$

DPO outputs the expectation of the optimal solution $\mathbb{E}_{\boldsymbol{\xi}}[\boldsymbol{w}^*(\hat{\boldsymbol{c}}+\sigma\boldsymbol{\xi})]$ under a perturbed coefficients with random noise $\hat{\boldsymbol{c}} + \sigma\boldsymbol{\xi}$, rather than the optimal solution $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$ under a fixed coefficients $\hat{\boldsymbol{c}}$. As visualized in Fig. 4, the expectation can be regarded as a convex combination of different feasible solutions.

Thus, unlike the piecewise constant function $\boldsymbol{w}^*(\hat{\boldsymbol{c}})$, $\mathbb{E}_{\boldsymbol{\xi}}[\boldsymbol{w}^*(\hat{\boldsymbol{c}} + \sigma\boldsymbol{\xi})]$ varies the proportions in response to the change of $\hat{\boldsymbol{c}}$, providing a nonzero Jacobian matrix of $\hat{\boldsymbol{c}}$:

$$\frac{\partial \mathbb{E}_{\boldsymbol{\xi}}[\boldsymbol{w}^*(\hat{\boldsymbol{c}} + \sigma\boldsymbol{\xi})]}{\partial \hat{\boldsymbol{c}}} \approx \frac{1}{K}\sum_{\kappa}^{K} \boldsymbol{w}^*(\hat{\boldsymbol{c}} + \sigma\boldsymbol{\xi}_{\kappa})\boldsymbol{\xi}_{\kappa}^{\mathsf{T}}.$$

The forward and backward pass are as follows:

Algorithm 4 DPO Forward Pass

Require: $\hat{\boldsymbol{c}}$, K, σ
1: **for** sample $\kappa \in \{1, ..., K\}$ **do**
2: Generate Gaussian noise $\boldsymbol{\xi}_{\kappa}$
3: Solve: $\boldsymbol{w}_{\kappa}^{\boldsymbol{\xi}} := \boldsymbol{w}^*(\hat{\boldsymbol{c}} + \sigma\boldsymbol{\xi}_{\kappa})$
4: Save $\boldsymbol{w}_{\kappa}^{\boldsymbol{\xi}}$ and $\boldsymbol{\xi}_{\kappa}$ for backward pass
5: **end for**
6: **return** $\frac{1}{K}\sum_{\kappa}^{K} \boldsymbol{w}_{\kappa}^{\boldsymbol{\xi}}$

Algorithm 5 DPO Backward Pass

Require: $\frac{\partial \mathcal{L}(\hat{\boldsymbol{c}},\cdot)}{\partial \mathbb{E}_{\boldsymbol{\xi}}[\boldsymbol{w}^*]}$, K

1: Load $\boldsymbol{w}_\kappa^{\boldsymbol{\xi}}$ and $\boldsymbol{\xi}_\kappa$ from forward pass

2: Compute $\frac{\partial \mathbb{E}_{\boldsymbol{\xi}}[\boldsymbol{w}^*]}{\partial \hat{\boldsymbol{c}}} := \frac{1}{K}\sum_\kappa^K \boldsymbol{w}_\kappa^{\boldsymbol{\xi}} \boldsymbol{\xi}_\kappa^\intercal$

3: Compute $\frac{\mathcal{L}(\hat{\boldsymbol{c}},\cdot)}{\partial \hat{\boldsymbol{c}}} := \frac{\partial \mathcal{L}(\hat{\boldsymbol{c}},\cdot)}{\partial \mathbb{E}_{\boldsymbol{\xi}}[\boldsymbol{w}^*]} \frac{\partial \mathbb{E}_{\boldsymbol{\xi}}[\boldsymbol{w}^*]}{\partial \hat{\boldsymbol{c}}}$

4: **return** $\frac{\mathcal{L}(\hat{\boldsymbol{c}},\cdot)}{\partial \hat{\boldsymbol{c}}}$

3.5 Perturbed Fenchel-Young loss [6]

Instead of using an arbitrary loss for **DPO**, Berthet et al. [6] further constructed the Fenchel-Young loss [7] to directly compute the decision error $\mathcal{L}_{\text{FY}}(\hat{\boldsymbol{c}}, \boldsymbol{w}^*(\boldsymbol{c}))$ and its gradient $\frac{\partial \mathcal{L}_{\text{FY}}(\hat{\boldsymbol{c}}, \boldsymbol{w}^*(\boldsymbol{c}))}{\partial \hat{\boldsymbol{c}}}$. Compared to **DPO**, **PFYL** avoids the inefficient calculation of the Jacobian matrix $\nabla \boldsymbol{w}^*(\hat{\boldsymbol{c}})$ and includes a theoretically sound loss function.

The loss of **PFYL** is based on the Fenchel duality: The expectation of the perturbed minimizer is defined as $F(\boldsymbol{c}) = \mathbb{E}_{\boldsymbol{\xi}}[\min_{\boldsymbol{w}\in S}\{(\boldsymbol{c}+\sigma\boldsymbol{\xi})^\intercal \boldsymbol{w}\}]$, and the dual of $F(\boldsymbol{c})$, denoted by $\Omega(\boldsymbol{w}^*(\boldsymbol{c}))$, is utilized to define the Fenchel-Young loss to reduce duality gap:

$$\mathcal{L}_{\text{FY}}(\hat{\boldsymbol{c}}, \boldsymbol{w}^*(\boldsymbol{c})) = \hat{\boldsymbol{c}}^\intercal \boldsymbol{w}^*(\boldsymbol{c}) - F(\hat{\boldsymbol{c}}) - \Omega(\boldsymbol{w}^*(\boldsymbol{c})),$$

where the gradient of the loss is

$$\frac{\partial \mathcal{L}_{\text{FY}}(\hat{\boldsymbol{c}}, \boldsymbol{w}^*(\boldsymbol{c}))}{\partial \hat{\boldsymbol{c}}} = \boldsymbol{w}^*(\boldsymbol{c}) - \frac{\partial F(\hat{\boldsymbol{c}})}{\partial \hat{\boldsymbol{c}}} = \boldsymbol{w}^*(\boldsymbol{c}) - \mathbb{E}_{\boldsymbol{\xi}}[\boldsymbol{w}^*(\hat{\boldsymbol{c}}+\sigma\boldsymbol{\xi})].$$

Similar to **DPO**, we can estimate the well-defined gradient as

$$\frac{\partial \mathcal{L}_{\text{FY}}(\hat{\boldsymbol{c}}, \boldsymbol{w}^*(\boldsymbol{c}))}{\partial \hat{\boldsymbol{c}}} \approx \boldsymbol{w}^*(\boldsymbol{c}) - \frac{1}{K}\sum_\kappa^K \boldsymbol{w}^*(\hat{\boldsymbol{c}}+\sigma\boldsymbol{\xi})$$

4 Implementation and modeling

The core module of *PyEPO* is an "autograd" function inherited from *PyTorch* [37]. Such functions perform a forward pass to yield optimal solutions or decision losses and a backward pass to obtain nonzero gradients so that the prediction model can learn from the decision error. Thus, our implementation extends *PyTorch*, which facilitates the deployment of end-to-end predict-then-optimize tasks using any neural network that can be built with *PyTorch*.

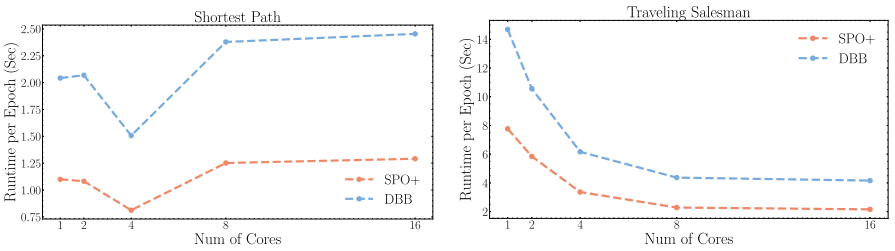

Fig. 5 Parallel efficiency: although creating a new process incurs additional overhead, parallel computing with an appropriate number of processors can effectively reduce the training time for **SPO+** and **DBB**

4.1 Modeling language

We choose *GurobiPy* [23], *COPTPy* [21] or *Pyomo* [25] to build optimization models. All of these are algebraic modeling languages (AMLs) written in Python.

GurobiPy and *COPTPy* are Python interfaces to *Gurobi* and *COPT*, which combine 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* for modeling, and *COPTPy* offers similar functionality. Recognizing that users might not have access to a *Gurobi* or *COPT* license, we also implemented a *Pyomo* interface as an alternative. *Pyomo* is an open-source optimization modeling language that supports a variety of solvers, including *Gurobi* and *GLPK*.

Each of these tools provides a natural way to express mathematical programming models. Therefore, users without specialized optimization knowledge can easily build and maintain optimization models through high-level algebraic representations.

Moreover, *PyEPO* also allows users to construct optimization models from scratch coding with any algorithm and solver, facilitating fast and flexible model customization for both research and production purposes.

4.2 Parallel computing

In addition, *PyEPO* supports parallel computing. For methods such as **SPO+**, **DBB**, **DPO**, and **PFYL**, the computational cost of repeated optimization is the major challenge, especially for integer programs. These methods require solving a batch of optimization problems per iteration to obtain the solution and the gradient. Parallel computing helps mitigate this issue by distributing the computational load across multiple processors.

Figure 5 illustrates the average running time per epoch for a mini-batch gradient descent algorithm with a batch size of 32, plotted against the number of cores. The decrease in running time per epoch is sublinear with respect to the number of cores. This may be due to the overhead associated with initializing additional cores, which can dominate computation costs. For example, in Fig. 5, for the shortest path, the easily solvable polynomial problem, the running time actually increases when the number of cores exceeds 4. In contrast, the more complicated $\mathcal{NP}$-complete problem, TSP,

continues to benefit slightly from additional cores. Overall, we believe this feature to be crucial for large-scale predict-then-optimize tasks.

4.3 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 objective function coefficients, it is necessary to instantiate an optimization model, `optModel`, with fixed constraints and variable coefficients. Such an optimization model would accept different objective function coefficients and find the corresponding optimal solutions under identical constraints. The construction of `optModel` is separated from the autograd functions (see Sect. 4.4), and then the instance of `optModel` will be passed as an argument to the autograd functions.

4.3.1 Optimization model from scratch

In *PyEPO*, the `optModel` functions as a black-box, which means that it does not rely on a specific algorithm or solver. This design grants users the flexibility to customize their tasks. To construct an `optModel` from scratch, users must override abstract methods `_getModel` to build a model and retrieve its variables, `setObj` to set the objective function with given coefficients, and `solve` to find an optimal solution. In addition, `optModel` provides an attribute `modelSense` to indicate whether the problem is minimization or maximization. The following shortest path example uses the Python library *NetworkX* [24] and its built-in Dijkstra's algorithm:

```
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 _getModel(self):
        """
        A method to build a 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 the 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

    def _getArcs(self):
        """
        A helper method to get a list of arcs for the 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

optmodel = myShortestPathModel()
```

4.3.2 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. To illustrate, consider the following optimization model (9), where c_i represents an unknown objective function coefficient:

$$\begin{aligned} \max_{x} \quad & \sum_{i=0}^{4} c_i x_i \\ s.t. \quad & 3x_0 + 4x_1 + 3x_2 + 6x_3 + 4x_4 \leq 12 \\ & 4x_0 + 5x_1 + 2x_2 + 3x_3 + 5x_4 \leq 10 \\ & 5x_0 + 4x_1 + 6x_2 + 2x_3 + 3x_4 \leq 15 \\ & \forall x_i \in \{0, 1\} \end{aligned} \tag{9}$$

Inheriting from `optGrbModel` provides a 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. Furthermore, there is no need to manually assign a value to the attribute `modelSense`. An example of the model (9) 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
        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()
```

The same applies to `optCoptModel` with *COPT*.

4.3.3 Optimization Model with Pyomo

Similarly, `optOmoModel` allows modeling mathematical programs with *Pyomo*. Unlike `optGrbModel`, `optOmoModel` requires explicitly setting

the `modelSense`. Since *Pyomo* supports multiple solvers, instantiating an `optOmoModel` requires a parameter `solver` to specify the solver. The following is an implementation of the model (9):

```
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")
```

4.4 Autograd Functions

Training neural networks with modern deep learning libraries such as *TensorFlow* [1] or *PyTorch* [38] requires gradient calculations for backpropagation. This is achieved through the numerical technique of automatic differentiation [37]. For example, *PyTorch* provides autograd functions.

The autograd functions are the core modules of *PyEPO* that solve and back-propagate the optimization problems with predicted coefficients. These functions can be integrated with different neural network architectures to achieve end-to-end predict-then-optimize for various tasks.

4.4.1 Function SPOPlus

The function `SPOPlus` directly calculates the SPO+ loss, which measures the decision error of an optimization solved with predicted coefficients. This optimization is represented as an instance of `optModel` and passed into `SPOPlus` as an argument. As shown below, `SPOPlus` also requires `processes` to specify the number of processes for parallel computation.

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

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

- `pred_cost`: a batch of predicted coefficient vectors, one vector per instance;
- `true_cost`: a batch of true coefficient 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 = spop(pred_cost, true_cost,
    true_sol,
true_obj)
```

4.4.2 Function blackboxOpt

`SPOPlus` directly computes a loss, whereas `blackboxOpt` provides a solution as a layer. Thus, `blackboxOpt` makes it possible to use various loss functions. Compared to `SPOPlus`, `blackboxOpt` requires an additional parameter `lambd`, which is the interpolation degree λ for finite difference. According to Pogančić et al. [40], 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)
# set some loss
l1 = nn.L1Loss()
```

Since `blackboxOpt` functions as a differentiable optimizer, there is only one parameter `pred_cost` for the forward pass. As shown in the code below, the output is the optimal solution for the given predicted coefficients:

```
# find the optimal solution
pred_sol = dbb(pred_cost)
# calculate additional loss
loss = l1(pred_sol, true_sol)
```

4.4.3 Function perturbedOpt

Same as `blackboxOpt`, `perturbedOpt` is a differentiable optimizer and provides an "expected solution". The hyperparameters for `perturbedOpt` include the number K of Monte-Carlo sample, "`n_sample`" and the amplitude parameter σ of the perturbation, "`sigma`".

```
import pyepo
# init DPO solver
dpo = pyepo.func.perturbedOpt(optmodel, n_samples=3, sigma
    =1.0, processes=8)
# set some loss
l1 = nn.L1Loss()
```

Given predicted coefficients $\hat{\boldsymbol{c}}$, `perturbedOpt` outputs an expected solution by averaging the solutions of K randomly perturbed optimization problems:

```
# find the expected optimal solution
exp_sol = dpo(pred_cost)
# calculate loss
loss = l1(exp_sol, true_sol)
```

4.4.4 Function perturbedFenchelYoung

`perturbedFenchelYoung` uses a predicted coefficients $\hat{\boldsymbol{c}}$ and a true solution $\boldsymbol{w}^*(\boldsymbol{c})$ to compute the Perturbed Fenchel-Young loss $\mathcal{L}_{\text{FY}}(\hat{\boldsymbol{c}}, \boldsymbol{w}^*(\boldsymbol{c}))$; it requires the same hyperparameters as `perturbedOpt`.

```
import pyepo
# init Fenchel-Young loss
pfyl = pyepo.func.perturbedFenchelYoung(optmodel,
    n_samples=3, sigma=1.0, processes=8)
```

The below code block illustrates the calculation of Fenchel-Young loss:

```
# calculate Fenchel-Young loss
loss = pfyl(pred_cost, true_sol)
```

4.5 The `optDataset` class for managing data

The use of decision losses, such as **`SPO+`** and **`PFYL`**, requires access to true optimal solutions. To facilitate convenient training and testing in *PyEPO*, an auxiliary component `optDataset` has been introduced, though it is not strictly indispensable. `optDataset` stores the features and their associated coefficients of the objective function and solves optimization problems to obtain optimal solutions and corresponding objective values.

`optDataset` is extended from *PyTorch* Dataset. In order to obtain optimal solutions, `optDataset` requires the corresponding `optModel` (see in Sect. 4.3). The parameters for the `optDataset` are as follows:

- `model`: an instance of `optModel`;
- `feats`: data features;
- `costs`: corresponding objective function coefficients;

Then, *PyTorch* DataLoader receives an `optDataset`, wraps the data samples, and acts as a sampler that provides an iterable over the given dataset. The batch size is required as the number of training samples that will be used in each update of the model parameters.

```
import pyepo
from torch.utils.data import DataLoader

# build dataset
dataset = pyepo.data.dataset.optDataset(optmodel, feats,
    costs)
# get data loader
dataloader = DataLoader(dataset, batch_size=32, shuffle=
    True)
```

By iterating over the DataLoader, we can obtain a batch of features, true objective function coefficients, optimal solutions, and the corresponding objective values:

```
for x, c, w, z in dataloader:
    # a batch of features
    print(x)
```

```
    # a batch of true coefficients
    print(c)
    # a batch of true optimal solutions
    print(w)
    # a batch of true optimal objective values
    print(z)
```

4.6 End-to-end training

The core capability of *PyEPO* is to build an optimization model and then embed it into a *PyTorch* neural network for end-to-end training. Here, we illustrate this with a simple linear regression model in PyTorch as an example:

```
from torch import nn

# construct linear model
class linearRegression(nn.Module):

    def __init__(self):
        super(linearRegression, self).__init__()
        # fully-connected layer without activation
        self.linear = nn.Linear(num_feat, len_cost)

    def forward(self, x):
        out = self.linear(x)
        return out

# init model
predmodel = linearRegression()
```

We can then train the prediction model with SPO+ loss to predict unknown coefficients, make decisions, and compute decision errors. The prediction model is trained using a stochastic gradient descent (SGD) optimizer. Leveraging the automatic differentiation capabilities of *PyTorch*, the loss gradients with respect to the model parameters $\boldsymbol{\theta}$ are calculated and used to update the model parameters during training.

```
import torch

# set SGD optimizer
optimizer = torch.optim.SGD(predmodel.parameters(), lr=1e
    -3)

# training
for epoch in range(num_epochs):
    # iterare features, coefficients, solutions, and
    objective values
    for x, c, w, z in dataloader:
        # forward pass
        cp = predmodel(x) # predict coefficients
        loss = spop(cp, c, w, z) # calculate SPO+ loss
        # backward pass
        optimizer.zero_grad() # reset gradients to 0
        loss.backward() # compute gradients
        optimizer.step() # update model parameters
```

4.7 Metrics

PyEPO provides evaluation functions to measure model performance, in particular the two metrics mentioned in Sect. 3.3.1: regret and unambiguous regret. Additionally, we define the normalized (unambiguous) regret by

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

Both require the following parameters:

- `predmodel`: a regression neural network for coefficients prediction.
- `optModel`: a *PyEPO* optimization model.
- `dataloader`: a *PyEPO* data loader.

Assume that we have trained a prediction model `predmodel` for an optimization problem `optModel`. To evaluate the performance of `predmodel` on a dataset, `dataloader`, containing instances of `optModel`, the following suffices:

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

5 Benchmark datasets

5.1 Benchmark datasets from PyEPO

This section describes our new datasets designed for the end-to-end predict-then-optimize task. In general, we generate data sets in a way similar to Elmachtoub and Grigas [16]. The synthetic dataset $\mathcal{D}$ includes features $\boldsymbol{x}$ and objective function coefficients $\boldsymbol{c}$ (e.g. $\mathcal{D} = \{(\boldsymbol{x}^1, \boldsymbol{c}^1), (\boldsymbol{x}^2, \boldsymbol{c}^2), ..., (\boldsymbol{x}^n, \boldsymbol{c}^n)\}$). 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 coefficient $\boldsymbol{c}^i \in \mathbb{R}^d$ comes from a (possibly nonlinear) polynomial function of $\boldsymbol{x}^i$ with additional random noise. $\epsilon_j^i \sim \boldsymbol{U}(1-\bar{\epsilon}, 1+\bar{\epsilon})$ is the multiplicative noise term for c_j^i, the j^{th} element of coefficients $\boldsymbol{c}^i$.

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 for generating these data with adjustable parameters: data size n, number of features p, coefficients vector dimension d, polynomial degree deg, and noise half-width $\bar{\epsilon}$.

5.1.1 Shortest path

Following Elmachtoub and Grigas [16], we consider a $h \times w$ grid network, and the goal is to find the shortest path [36] from the northwest to the southeast. We generate

a random matrix $\mathcal{B} \in \mathbb{R}^{d \times p}$, where $\mathcal{B}_{ij}$ follows the Bernoulli distribution with probability 0.5. Then, the coefficients $\boldsymbol{c}^i$ is almost the same as in [16], and is generated from

$$\left[\frac{1}{3.5^{deg}}\left(\frac{1}{\sqrt{p}}(\mathcal{B}\boldsymbol{x}^i)_j + 3\right)^{deg} + 1\right] \cdot \epsilon^i_j. \tag{10}$$

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

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

5.1.2 Multi-dimensional knapsack

The multi-dimensional knapsack problem [32] is one of the most well-known integer programming models. It maximizes the value of the selected items under multiple resource constraints. Due to its computational complexity, solving this problem can be challenging as the number of variables (items) and constraints (resources) grows.

Assuming that the uncertain coefficients exist only in the objective function, the weights of the items and the capacity of resources remain fixed throughout the data. We denote the number of resources by k and the number of items by d. The weights $\mathcal{W} \in \mathbb{R}^{k \times m}$ are sampled from 3 to 8 with a precision of one decimal place. Using the same $\mathcal{B} \in \mathbb{R}^{d \times p}$ as in Sect. 5.1.1, coefficient c_{ij} is calculated according to Equation (10).

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

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

5.1.3 Traveling salesperson

As one of the most renowned combinatorial optimization problems, the traveling salesperson problem (TSP) seeks to determine the shortest possible tour that visits every node exactly once. Here, we introduce the symmetric TSP (the distance between two nodes is the same in both directions) with the number of nodes to be visited v.

PyEPO generates costs from a distance matrix, where the distance is composed of two parts: one derived from the Euclidean distance, and the other from a polynomial function of the features. For the Euclidean distance, two-dimensional coordinates are created from the mixture of the Gaussian distribution $\mathcal{N}(0, I)$ and the uniform distribution $\boldsymbol{U}(-2, 2)$. For the function of features, the polynomial kernel function is

$$\frac{1}{3^{deg-1}}\left(\frac{1}{\sqrt{p}}(\mathcal{B}\boldsymbol{x}^i)_j + 3\right)^{deg} \cdot \epsilon^i_j,$$

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 follows:

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

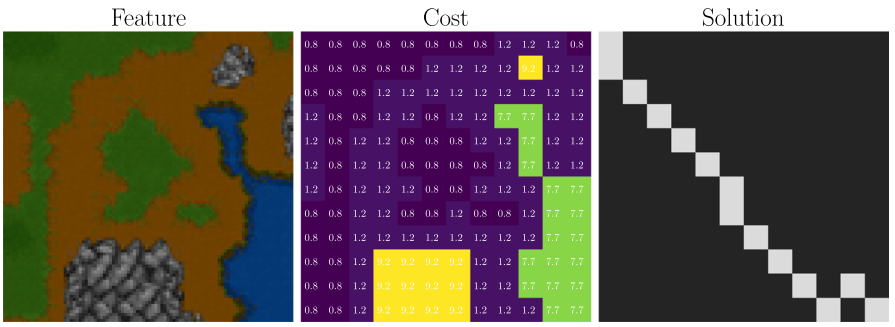

Fig. 6 Warcraft terrain shortest path dataset: (left) each input feature is a $k \times k$ terrain map image as a grid of tiles; (middle) the respective weights is a matrix indicating traveling costs; (right) the corresponding binary matrix represents the shortest path from top left to bottom right

5.2 Warcraft terrain map images

Pogančić et al. [40] proposed a dataset of maps from the popular video game Warcraft (see Fig. 6), designed for learning the shortest path from RGB terrain images. This dataset stands out as a significant benchmark because it employs image input, a modality that was not explored in other work in this area. In accordance with the experiment of Pogančić et al. [40] and Berthet et al. [6], we utilize 96×96 RGB images for determining the shortest path on 12×12 grid networks.

6 Empirical evaluation for *PyEPO* datasets

This section presents experimental results for the benchmark datasets described in Sect. 5.1. The experiments aimed to investigate training time and normalized regret, as defined in Sect. 4.7, on a test set of size $n_{\text{test}} = 1000$. As Table 3 shows, the methods we compare include the two-stage approach with different predictors as well as end-to-end methods such as **SPO+**/**DBB**/**PFYL** with a linear prediction model $g(\boldsymbol{x}; \boldsymbol{\theta})$. Notably, the result of **DPO** was excluded due to its consistently subpar performance.

Unlike direct decision loss functions **SPO+** and **PFYL**, **DBB** and **DPO** allow the use of customized loss functions, offering flexibility for various problems. In the original paper, Pogančić et al. [40] used the Hamming distance between the true optimum and the solution from prediction, whereas Berthet et al. [6] applied the square difference between solutions. However, in our experiments, compared to 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 (6) as a loss for **DBB** and **DPO**. In addition, l_1/l_2 regularization means that the MAE / MSE of the predicted coefficients $\hat{\boldsymbol{c}}$ is added to the loss function.

All numerical experiments were conducted in Python v3.7.9 with 32 Intel E5-2683 v4 Broadwell CPUs and 32GB memory. Specifically, we used *PyTorch* [38] v1.10.0 to train end-to-end models, and *Scikit-Learn* [39] v0.24.2 and *Auto-Sklearn* [18] v0.14.6

Table 5 Methods compared in the experiments

Method	Description
2S	Two-stage method where the predictor is a truncated ResNet18
SPO+	Truncated ResNet18 with SPO+ loss [16]
DBB	Truncated ResNet18 with differentiable black-box optimizer and Hamming distance loss [40]
DPO	Truncated ResNet18 with differentiable perturbed optimizer and squared error loss [6]
PFYL	Truncated ResNet18 with perturbed Fenchel-Young loss [6]

as predictors for the two-stage method. *Gurobi* [23] v9.1.2 was the optimization solver used throughout. The version of *PyEPO* is v0.2.4[1].

6.1 Performance comparison between different methods

We compare the performance between two-stage methods, **SPO+**, **PFYL**, and **DBB** between varying training data size $n \in \{100, 1000, 5000\}$, polynomial degree $deg \in \{1, 2, 4, 6\}$, and noise half-width $\bar{\epsilon} \in \{0.0, 0.5\}$. Each experiment was repeated 10 times, using different $\boldsymbol{x}$, $\mathcal{B}$, and ϵ to generate 10 distinct training/validation/test datasets. We use boxplots to summarize the statistical outcomes.

We also carried out small-scale experiments on the validation set to select gradient descent hyperparameters, namely batch size, learning rate, and momentum for the shortest path, knapsack, and TSP in **SPO+**, **PFYL** and **DBB**. The hyperparameter tuning used a limited random search in the space of possible configurations, thus not guaranteeing the best performance in the results. For **PFYL**, we arbitrarily set the number of samples $K = 1$ and the perturbation amplitude $\sigma = 1.0$.

We generate synthetic datasets using the parameters outlined in Table 4, so the label sizes (the number of unknown objective function coefficients) d are 40, 32, and 190, respectively. For the TSP, we use the Dantzig-Fulkerson-Johnson (DFJ) formulation [11] because it is faster to solve than the alternatives.

Figures 7, 8, and 9 summarize the performance comparison for the shortest path problem, the 2D knapsack problem, and the traveling salesperson problem. These figures should be interpreted as follows: The left column is for noise-free coefficients (easier), while the right column includes noise. Each row of figures is for a training set size in increasing order. Within each figure, the degree of the polynomial that generates the coefficients from the feature vector increases from left to right. For each such polynomial degree, the different methods' boxplots are shown, summarizing the test set normalized regret results across the 10 different experiments; lower is better.

[1] *PyEPO* is an actively maintained open-source library. To ensure the reproducibility of the results, a static branch has been established and can be accessed at https://github.com/khalil-research/PyEPO/blob/MPC.

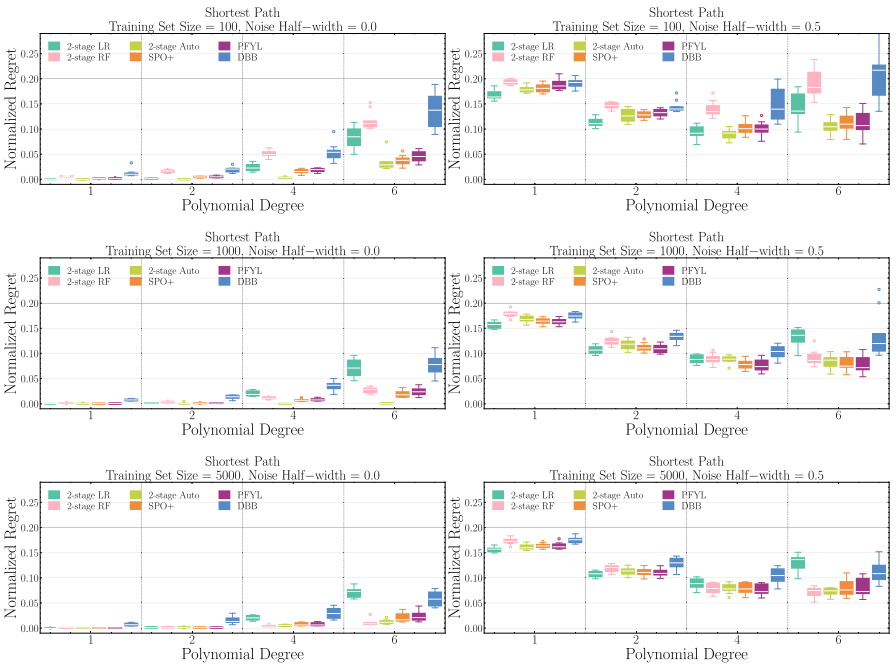

Fig. 7 Normalized regret for the shortest path problem on the test set: the size of the grid network is 5×5. The methods in the experiment include two-stage approaches with linear regression, random forest, and *Auto-Sklearn* and end-to-end learning such as **SPO+**, **PFYL**, and **DBB**. The normalized regret is visualized under different sample sizes, noise half-width, and polynomial degrees. For normalized regret, lower is better

The 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, the random forest achieves the best performance in many cases. The two-stage method with automated hyperparameter tuning using the *Auto-Sklearn* tool [18] (discussed further in Sect. 6.2) is notable: 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, as the output dimensions increase (e.g. TSP in Fig. 9), *Auto-Sklearn* struggles to remain competitive.

SPO+ and **PFYL** show their advantage: they perform best, or at least relatively well, in all cases. These methods are comparable to linear regression at low polynomial degrees and depend less on the sample size than a random forest. At high polynomial degrees, **SPO+** and **PFYL** outperform *Auto-Sklearn*, exposing the limitations of the two-stage approach. Furthermore, the **PFYL** method offers the benefit of not requiring the presence of true coefficients $\boldsymbol{c}$ within the training dataset, setting it apart from **SPO+**.

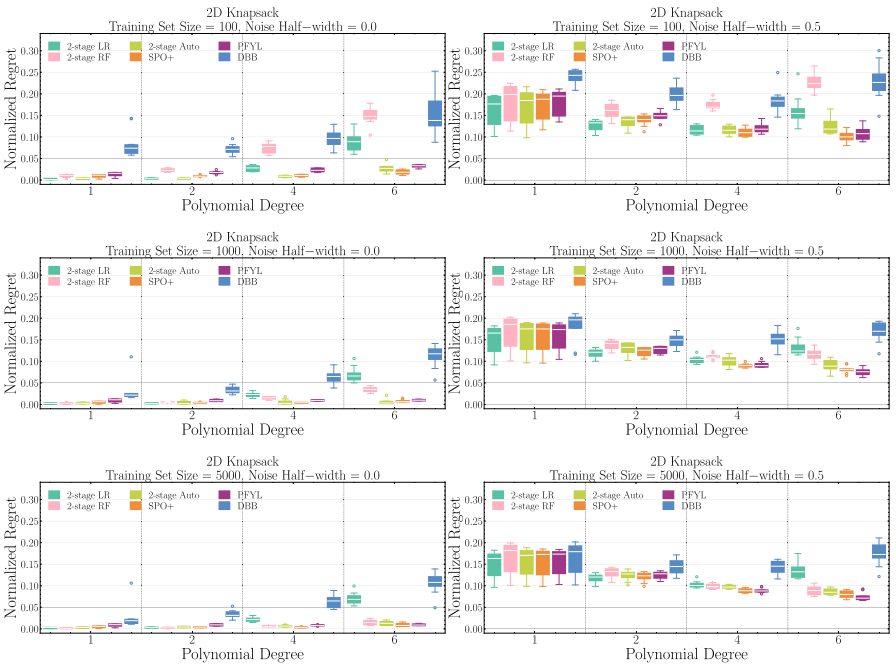

Fig. 8 Normalized regret for the 2D knapsack problem on the test set: there are 32 items, and the capacity of the two resources is 20. The methods in the experiment include two-stage approaches with linear regression, random forest, and *Auto-Sklearn* and end-to-end learning such as **`SPO+`**, **`PFYL`** and **`DBB`**. The normalized regret is visualized under different sample sizes, noise half-width, and polynomial degrees. For normalized regret, lower is better

Finding #1

`SPO+` and **`PFYL`** can robustly achieve relatively good decisions under different scenarios, often outperforming two-stage baselines.

6.2 Two-stage method with automated hyperparameter tuning

This method leverages the sophisticated *Auto-Sklearn* [18] tool that uses Bayesian optimization methods for the automated hyperparameter tuning of the *Scikit-Learn* regression models. The metric of "2-stage Auto" is the mean squared error of the predicted coefficients, which does not directly reduce the decision error. Due to the limitation of multi-output regression in *Auto-Sklearn* v0.14.6, the choices of the predictor in 2-stage Auto are restricted to five models: k-nearest neighbor (KNN), decision tree, random forest, extra trees, and Gaussian process. Despite these constraints, *Auto-Sklearn* can achieve low regret. Although 2-stage Auto training is time-consuming, it remains a competitive method.

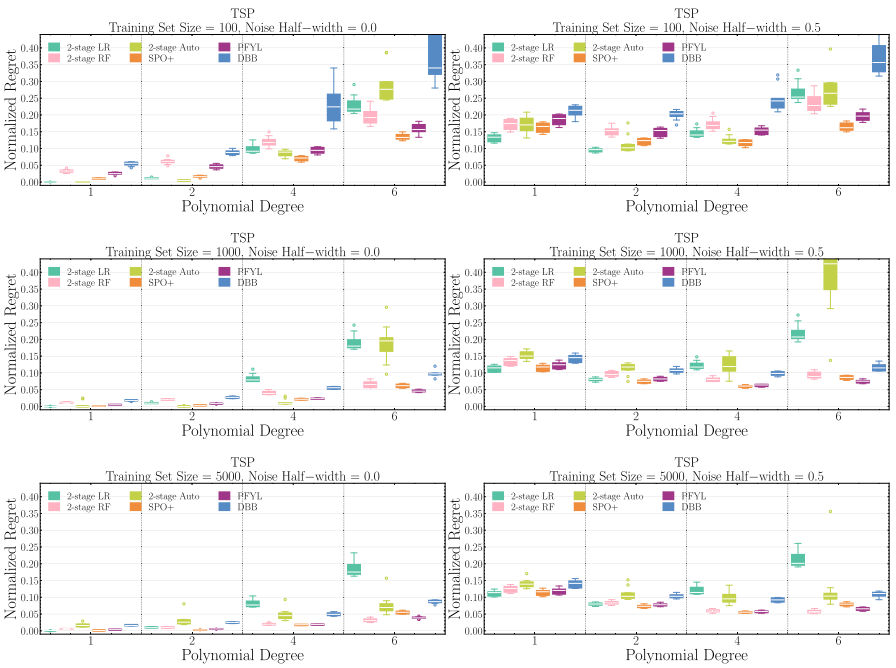

Fig. 9 Normalized regret for the TSP problem on the test set: there are 20 nodes to visit. The methods in the experiment include two-stage approaches with linear regression, random forest, and *Auto-Sklearn* and end-to-end learning such as **`SPO+`**, **`PFYL`**, and **`DBB`**. The normalized regret is visualized under different sample sizes, noise half-width, and polynomial degrees. For normalized regret, lower is better

Finding #2

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

6.3 Exact method and relaxation

Training **`SPO+`**, **`PFYL`**, and **`DBB`** with a linear relaxation instead of solving the integer program improves the computational efficiency. However, the use of a "weaker" solver theoretically undermines the decision quality. Therefore, an important question arises about the trade-off when using linear relaxation in training. To this end, we compare the performance of end-to-end approaches with their relaxation using the 2D knapsack and TSP as case studies. We ensure consistency by using the same instances, models, and hyperparameters as in our previous experiments.

There are several integer programming formulations for the TSP. In addition to DFJ, we have also implemented the Miller–Tucker–Zemlin (MTZ) formulation [34] and the Gavish–Graves (GG) formulation [20]. Although all formulations yield the

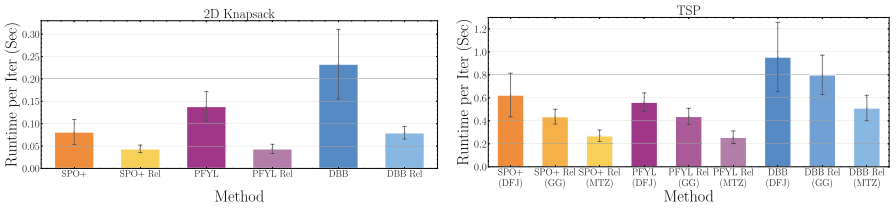

Fig. 10 Average training time per iteration for exact and relaxation methods with standard deviation error bars: we visualized the mean training time with standard deviation for the Knapsack and TSP; lower is better

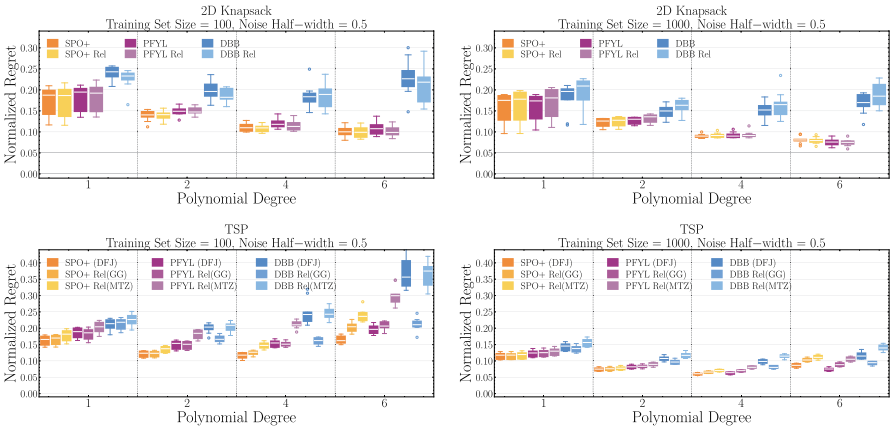

Fig. 11 Normalized regret for the 2D knapsack (at the top) and TSP (at the bottom) on the test set: the methods in the experiment include **SPO+**, **PFYL** and **DBB** w/o relaxation. Then, we visualize the normalized regret under different sample sizes and polynomial degrees to investigate the impact of the relaxation method. For normalized regret, lower is better

same integer solution, they differ in computational efficiency and linear relaxations. Since DFJ requires column generation to handle 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+**, **PFYL**, and **DBB**, and MTZ and GG for relaxation to investigate the effect of the quality of the solution on regret.

According to Fig. 10, the use of linear relaxation significantly reduces training time. As shown in Fig. 11, the impact on the performance of the knapsack is almost negligible. Interestingly, relaxed methods have the potential to perform better on small data, perhaps because linear relaxation acts as a regularization mechanism that prevents overfitting. For TSP, Fig. 11 demonstrates that a tighter bound does reduce the regret more, and **DBB Rel** with GG formulation shows advantages over **DBB**. Overall, employing relaxations achieves commendable performance with enhanced computational efficiency. Moreover, formulations with tighter linear relaxation lead to better performance.

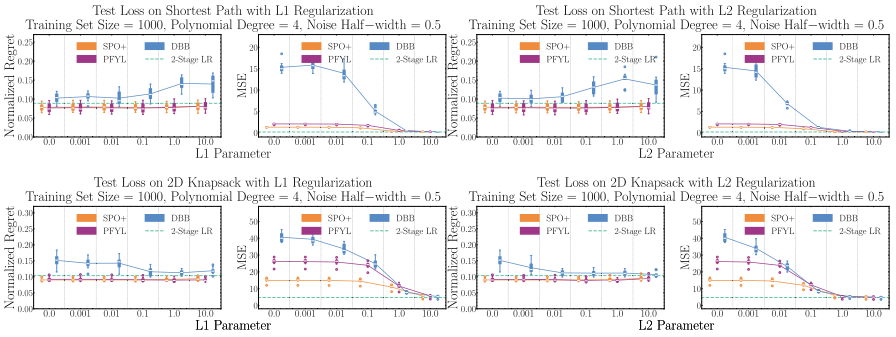

Fig. 12 Normalized regret and MSE on the test set: in these experiments, we compare the performance on **SPO+**, **PFYL**, and **DBB** with different levels of l_1 or l_2 regularization and show the trend line with increasing regularization, while 2-Stage LR (green line) is served as the baseline. The normalized regret and MSE under different problems are shown; lower is better (colour figure online)

Finding #3

End-to-end predict-then-optimize with relaxation has excellent potential to improve computation efficiency at a slight degradation in performance, particularly when the true coefficients-generating function is not very nonlinear.

6.4 Prediction regularization

As proposed in Elmachtoub and Grigas [16], the mean absolute error $\mathcal{L}_{\text{MAE}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \frac{1}{n}\sum_i^n \|\hat{\boldsymbol{c}}^i - \boldsymbol{c}^i\|_1$ or the mean squared error $\mathcal{L}_{\text{MSE}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \frac{1}{2n}\sum_i^n \|\hat{\boldsymbol{c}}_i - \boldsymbol{c}^i\|_2^2$ of the predicted coefficient 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 ϕ_1 and the l_2 regularization parameter ϕ_2 logarithmically from 0.001 to 10. For the experiments, we use the same instances, model, and hyperparameters as before, while the number of training samples n, the noise half-width $\bar{\epsilon}$, and the polynomial degree deg are fixed at 1000, 0.5 and 4.

Based on Fig. 12, regularization appears to be a viable strategy to reduce MSE. With an increase in the regularization parameter, all end-to-end methods achieve an MSE comparable to that of the two-stage linear regression approach. In terms of regret, the impact on **SPO+** and **PFYL** is minor, while more regularization of **DBB** has the potential to produce improvements.

Finding #4

l_1 and l_2 regularizations are considered appealing techniques for enhancing prediction accuracy while simultaneously preserving decision quality.

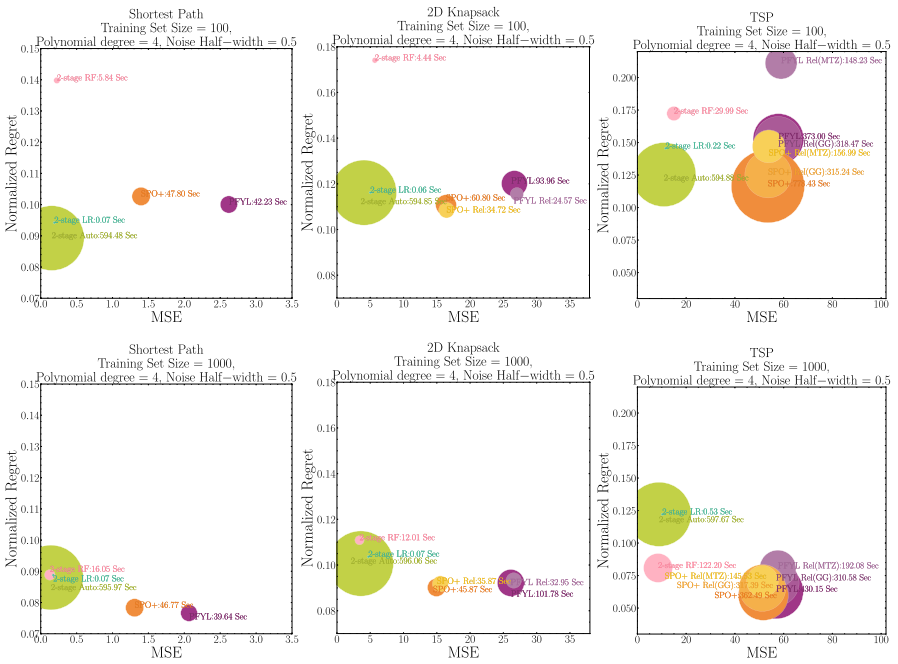

Fig. 13 MSE versus regret: the result covers different two-stage methods, **SPO+**, **PFYL**, and their relaxations. **DBB** is omitted because it is far from others. The size of the circles is proportional to the training time (s), so the smaller is better

6.5 Trade-offs between MSE and regret

Figure 13 examines the trade-off between prediction and decision on the 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 over 10 repeated random experiments. Apart from this, the mean training time is also annotated, with circle sizes proportional to the time. The circles of **DBB** have been removed as they are located far in the top right corner and perform significantly worse than other methods.

Figure 13 shows that **SPO+** and **PFYL** can reach low decision errors, particularly with large training sets, at the cost of higher prediction errors. Moreover, the MSE of the predictions of **PFYL** is significantly higher than that of **SPO+**. Further examination reveals that the higher prediction errors of **SPO+** and **PFYL** are mainly due to scale shifts in the predicted coefficient values, which does not alter the optimals of an optimization problem with a linear objective function. In addition, training with *Auto-Sklearn*, which involves automated algorithm selection and hyperparameter tuning, is time-consuming but provides both high-quality prediction error and decision error. However, compared to **SPO+** and **PFYL**, even the competitive 2-stage Auto model does not have an advantage in decision-making with 1000 training samples.

Inspired by Sec 6.4, we further incorporated l_2 regularization with parameter $\phi_2 = 1.0$ for **SPO+** and **PFYL**, refering to these as **SPO+ L2** and **PFYL L2**. Figure 14

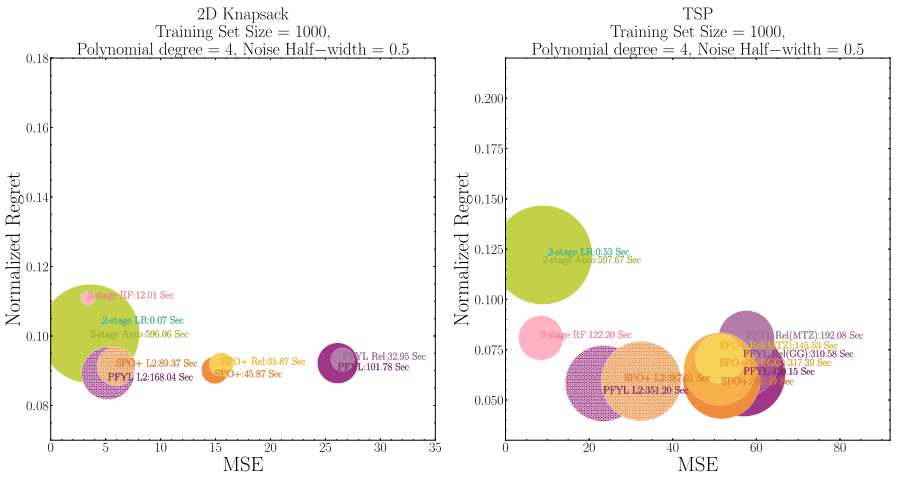

Fig. 14 MSE versus regret: the result covers different two-stage methods, **SPO+**, **PFYL**, and their regularization. **DBB** is omitted because it is far away from others. The size of the circles is proportional to the training time (s), so the smaller is better

shows that the appropriate amount of prediction regularization can achieve a favorable balance between MSE and regret, occasionally even reducing regret more.

Finding #5

Generally, **SPO+** and **PFYL** can achieve good decisions , albeit with a trade-off in prediction accuracy. For a more balanced approach between decision quality and prediction accuracy, an end-to-end method with prediction regularization might be preferable.

7 Empirical evaluation for image-based shortest path

Following Pogančić et al. [40] and Berthet et al. [6], we employed a truncated ResNet18 convolutional neural network (CNN) architecture consisting of the first five layers for Warcraft terrain images (see Sect. 5.2). As Table 5 shows, the methods we compare include a two-stage method, **SPO+**, **DBB**, **DPO**, and **PFYL**, all using truncated ResNet18. The CNN is trained over 50 epochs with batches of size 70. The learning rate is set to 0.0005, decaying at the epochs 30 and 40, and the hyperparameters $K = 1, \sigma = 1$ for **DPO** and **PFYL**, $\lambda = 10$ for **DBB**. We use the Hamming distance for **DBB** and the squared error of solutions for **DPO**, as specified in their original papers.

The sample size of the test set n_{test} is 1000. To evaluate our methods, we compute the relative regret $\frac{\boldsymbol{c}^\top \boldsymbol{w}^*(\hat{\boldsymbol{c}}) - z^*(\boldsymbol{c})}{z^*(\boldsymbol{c})}$ and path accuracy $\frac{\sum_{j=1}^{d} \mathbb{1}(z^*(\boldsymbol{c})_j = z^*(\hat{\boldsymbol{c}})_j)}{d}$ per instance on the test set; the latter is simply the proportion of edges in the "predicted" solution that also appear in the optimal solution.

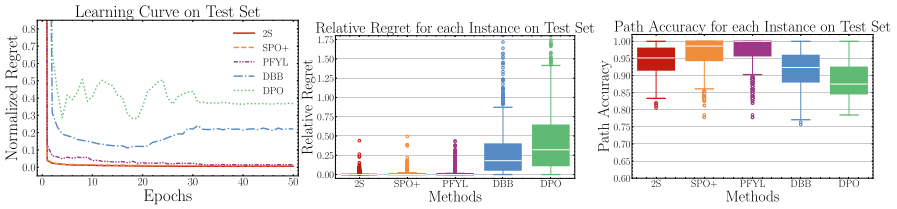

Fig. 15 Learning curve, relative regret, and path accuracy for the shortest path problem on the test set: the methods in the experiment include a two-stage neural network, **SPO+**, **DBB**, **DPO**, and **PFYL**. The learning curve shows relative regret on the test set, and the box plot demonstrates the distribution of relative regret and path accuracy on the test set. For relative regret, lower is better. For path accuracy, higher is better

As shown in Fig. 15, the two-stage method, **SPO+** and **PFYL** achieve comparable performance in predicting the shortest path on the Warcraft terrain. Among these methods, **PFYL** achieves the highest path accuracy, indicating that its solutions align most closely with the given optimal ones. It seems that the Warcraft shortest path problem may not require end-to-end learning. However, it is noteworthy that **PFYL**, despite the lack of knowledge of the true coefficients, produces a competitive result. This finding encourages researchers to explore a broader range of applications for end-to-end learning.

Finding #6

End-to-end learning is effective in rich contextual features such as images. Moreover, the study highlights that **PFYL** can achieve impressive performance levels even without access to labeled coefficients during training.

8 Conclusion

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

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, *PyEPO* enables users to leverage a vast array of state-of-the-art deep learning models and techniques implemented in *PyTorch*.

While *PyEPO* is currently tailored to linear objective functions and has been primarily tested on linear and linear integer programming, its underlying architecture holds the potential for broader problems. Specifically, it can be extended to accommodate nonlinear constraints and even nonlinear objective functions. For example, **SPO+** in *PyEPO* requires convex constraints rather than linear ones.

PyEPO allows users to build optimization problems as black boxes or by leveraging interfaces to *GurobiPy*, *COPTPy* and *Pyomo*, thus providing broad compatibility with high-level modeling languages and both commercial and open-source solvers.

With the *PyEPO* framework, we generate three synthetic datasets and incorporate one image dataset from the literature. Comprehensive experiments and analyses are conducted with these datasets. The results show that the end-to-end methods achieved excellent improvement in decision quality over two-stage methods in many cases. In addition, end-to-end models can benefit from using relaxations and regularization. The code repository includes step-by-step tutorials, enabling new users to replicate the experiments presented in this paper or use them as a starting point for their applications. Future development efforts of *PyEPO* include:

- New applications and new optimization problems, including linear objective functions with mixed-integer variables or nonlinear constraints;
- Novel training methods that leverage the gradient computation features that *PyEPO* provides;
- Innovative approaches that accommodate nonlinear constraints and/or objective function;
- Additional variations and improvements to current approaches such as warm starting and training speed up;
- Other existing end-to-end predict-then-optimize approaches, such as **`QPTL`** and its variants.

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.

Funding This work was supported by funding from a SCALE AI Research Chair and an NSERC Discovery Grant to Elias B. Khalil.

Availability of data and materials All data analyzed during this study are publicly available.

Code availability The complete source code used in this study is open source and can be freely accessed and reviewed. We have utilized several open-source packages for our study. Specific references to these packages can be found within the code and article.

Declarations

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

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv preprint arXiv:1603.04467 (2016)
2. Agrawal, A., Amos, B., Barratt, S., Boyd, S., Diamond, S., Kolter, J.Z.: 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, vol. 32. Curran Associates Inc., Glasgow (2019)
3. Agrawal, A., Barratt, S., Boyd, S., Busseti, E., Moursi, W.M.: Differentiating Through a Cone Program. arXiv preprint arXiv:1904.09043 (2019)
4. Amos, B., Kolter, J.Z.: Optnet: differentiable optimization as a layer in neural networks. In: International Conference on Machine Learning, PMLR, pp. 136–145 (2017)
5. Bengio, Y.: Using a financial training criterion rather than a prediction criterion. Int. J. Neural Syst. **8**(04), 433–443 (1997)
6. Berthet, Q., Blondel, M., Teboul, O., Cuturi, M., Vert, J.P., Bach, F.: Learning with differentiable perturbed optimizers. arXiv preprint arXiv:2002.08676 (2020)
7. Blondel, M., Martins, A.F., Niculae, V.: Learning with Fenchel-Young losses. J. Mach. Learn. Res. **21**(35), 1–69 (2020)
8. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., Zhang, Z.: Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
9. Cplex, I.I.: V12.1: user's manual for complex. Int. Bus. Mach. Corp. **46**(53), 157 (2009)
10. Dalle, G., Baty, L., Bouvier, L., Parmentier, A.: Learning with combinatorial optimization layers: a probabilistic approach. arXiv preprint arXiv:2207.13513 (2022)
11. Dantzig, G., Fulkerson, R., Johnson, S.: Solution of a large-scale traveling-salesman problem. J. Oper. Res. Soc. Am. **2**(4), 393–410 (1954)
12. Djolonga, J., Krause, A.: Differentiable learning of submodular models. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 30. Curran Associates Inc., Glasgow (2017)
13. Domke, J.: Generic methods for optimization-based modeling. In: Artificial Intelligence and Statistics, PMLR, pp 318–326 (2012)
14. Donti, P., Amos, B., Kolter, J.Z.: Task-based end-to-end model learning in stochastic optimization. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 30. Curran Associates Inc., Glasgow (2017)
15. Elmachtoub, A., Liang, J.C.N., McNellis, R.: Decision trees for decision-making under the predict-then-optimize framework. In: International Conference on Machine Learning, PMLR, vol. 119, pp. 2858–2867 (2020)
16. Elmachtoub, A.N., Grigas, P.: Smart predict, then optimize. Manag. Sci. **68**(1), 9–26 (2021)
17. Ferber, A., Wilder, B., Dilkina, B., Tambe, M.: Mipaal: mixed integer program as a layer. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 1504–1511 (2020)
18. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. Adv. Neural. Inf. Process. Syst. **28**, 2962–2970 (2015)
19. Ford, B., Nguyen, T., Tambe, M., Sintov, N., Delle Fave, F.: Beware the soothsayer: From attack prediction accuracy to predictive reliability in security games. In: International Conference on Decision and Game Theory for Security, pp. 35–56. Springer, Berlin (2015)
20. Gavish, B., Graves, S.C.: The travelling salesman problem and related problems. Operations Research Center Working Paper; OR 078-78 (1978)
21. Ge, D., Huangfu, Q., Wang, Z., Wu, J., Ye, Y.: Cardinal Optimizer (COPT) user guide. https://guide.coap.online/copt/en-doc (2022)
22. Gould, S., Fernando, B., Cherian, A., Anderson, P., Cruz, R.S., Guo, E.: On differentiating parameterized argmin and argmax problems with application to bi-level optimization. arXiv preprint arXiv:1607.05447 (2016)
23. Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual. https://www.gurobi.com (2021)
24. Hagberg, A., Swart, P., Chult, D.: Exploring network structure, dynamics, and function using networkX. Tech. rep., Los Alamos National Lab. (LANL), Los Alamos, NM (United States) (2008)
25. Hart, W.E., Laird, C.D., Watson, J.P., Woodruff, D.L., Hackebeil, G.A., Nicholson, B.L., Siirola, J.D., et al.: Pyomo-Optimization Modeling in Python, vol. 67. Springer, Berlin (2017)
26. Kao, Yh., Roy, B., Yan, X.: Directed regression. In: Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C., Culotta, A. (eds.) Advances in Neural Information Processing Systems, vol. 22. Curran Associates Inc., Glasgow (2009)
27. Liu, T.Y., et al.: Learning to rank for information retrieval. Found. Trends Inf. Retr. **3**(3), 225–331 (2009)

28. Mandi, J., Guns, T.: Interior point solving for lp-based prediction+optimisation. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) Advances in Neural Information Processing Systems, vol. 33, pp. 7272–7282. Curran Associates Inc., Glasgow (2020)
29. Mandi, J., Stuckey, P.J., Guns, T., et al.: Smart predict-and-optimize for hard combinatorial optimization problems. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 1603–1610 (2020). https://doi.org/10.1609/aaai.v34i02.5521
30. Mandi, J., Bucarey, V., Tchomba, M.M.K., Guns, T.: Decision-focused learning: through the lens of learning to rank. In: International Conference on Machine Learning, PMLR, pp. 14935–14947 (2022)
31. Mandi, J., Kotary, J., Berden, S., Mulamba, M., Bucarey, V., Guns, T., Fioretto, F.: Decision-focused learning: Foundations, state of the art, benchmark and future opportunities. arXiv preprint arXiv:2307.13565 (2023)
32. Martello, S., Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. Wiley, New York (1990)
33. Mattingley, J., Boyd, S.: Cvxgen: a code generator for embedded convex optimization. Optim. Eng. **13**(1), 1–27 (2012)
34. Miller, C.E., Tucker, A.W., Zemlin, R.A.: Integer programming formulation of traveling salesman problems. J. ACM **7**(4), 326–329 (1960)
35. Mulamba, M., Mandi, J., Diligenti, M., Lombardi, M., Bucarey, V., Guns, T.: Contrastive losses and solution caching for predict-and-optimize. arXiv preprint arXiv:2011.05354 (2020)
36. Ortega-Arranz, H., Llanos, D.R., Gonzalez-Escribano, A.: The shortest-path problem: analysis and comparison of methods. Synth. Lect. Theor. Comput. Sci. **1**(1), 1–87 (2014)
37. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch. In: NIPS 2017 Autodiff Workshop (2017)
38. 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.: 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, vol. 32, pp. 8024–8035. Curran Associates Inc., Glasgow (2019)
39. 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.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
40. Pogančić, M.V., Paulus, A., Musil, V., Martius, G., Rolinek, M.: Differentiation of blackbox combinatorial solvers. In: International Conference on Learning Representations (2019)
41. Sadana, U., Chenreddy, A., Delage, E., Forel, A., Frejinger, E., Vidal, T.: A survey of contextual optimization methods for decision making under uncertainty. arXiv preprint arXiv:2306.10374 (2023)
42. Wilder, B., Dilkina, B., Tambe, M.: 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 (2019)