PyEPO: A PyTorch-based End-to-End Predict-then-Optimize Library with Linear Objective Function

Bo Tang BOTANG@MIE.UTORONTO.CA
Elias B. Khalil KHALIL@MIE.UTORONTO.CA
Department of Mechanical % Industrial Engineering, University of Toronto, Toronto, ON M5S 3G8
SCALE AI Research Chair in Data-Driven Algorithms for Modern Supply Chains

Abstract

In many practical settings, some parameters of an optimization problem may be a priori unknown but can be estimated from historical data. Recently, end-to-end predict-then-optimize has emerged as an attractive alternative to the two-stage approach of separately fitting a predictive model for the unknown parameters, then optimizing. In this work, we present the *PyEPO* package, a *PyTorch*-based end-to-end predict-then-optimize library in Python for linear and integer programming. It provides two base algorithms: the first is based on the convex surrogate loss function from the seminal work of Elmachtoub and Grigas [8], and the second is based on the differentiable black-box solver approach of Pogančić et al. [20]. *PyEPO* provides a simple interface for the definition of new optimization problems, the implementation of state-of-the-art predict-then-optimize training algorithms, the use of custom neural network architectures, and the comparison of end-to-end approaches with the two-stage approach. *PyEPO* and its documentation are available at https://github.com/khalil-research/PyEPO.

Keywords: Data-driven optimization, Mixed integer programming, Machine learning

1. Introduction

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

Bengio [5], Ford et al. [11], and Elmachtoub and Grigas [8] reported that training a predictive model based on prediction error leads to worse decisions than directly considering decision error. Thus, compared to independent prediction and optimization, to integrate optimization into prediction becomes an attractive alternative. Since Amos and Kolter [4] first introduced a neural network layer for mathematical optimization, there have been some prominent attempts to bridge the gap between mathematical optimization and deep learning. The critical component is typically a dif-

ferentiable block for optimization tasks. With a differentiable optimizer, neural network packages enable the computation of gradients for optimization operations and then update predictive model parameters based on a loss function that depends on decision quality.

While research code implementing a number of predict-then-optimize training algorithms have been made available for particular classes of optimization problems and/or predictive models [2, 3, 6–9, 14, 14, 20, 21], there is a dire need for a generic end-to-end learning framework, especially for linear and integer programming. In this paper, we propose the open-source software package *PyEPO* which aims to customize and train end-to-end predict-then-optimize for linear and integer programming. We implement `SPO+` ("Smart Predict-then-Optimize+") loss [8], and `DBB` (differentiable black-box) solver [20] with parallel computing, which are two typical end-to-end methods for linear and integer programming. We also provide interfaces to the Python-based optimization modeling frameworks *GurobiPy* and *Pyomo*, allowing non-specialists to formulate optimization models with *PyEPO*.

2. Preliminaries

2.1. Definitions and Notation

For the sake of convenience, we define the following linear programming problem without loss of generality, where the decision variables are $\boldsymbol{w} \in \mathbb{R}^d$ and all $w_i \geq 0$, the cost coefficients are $\boldsymbol{c} \in \mathbb{R}^d$, the constraint coefficients are $\boldsymbol{A} \in \mathbb{R}^{k \times d}$, and the right-hand sides of the constraints are $\boldsymbol{b} \in \mathbb{R}^k$. When some variables w_i are restricted to be integers, we obtain a (mixed) integer program:

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

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

2.2. Gradient-based End-to-end Predict-then-Optimize

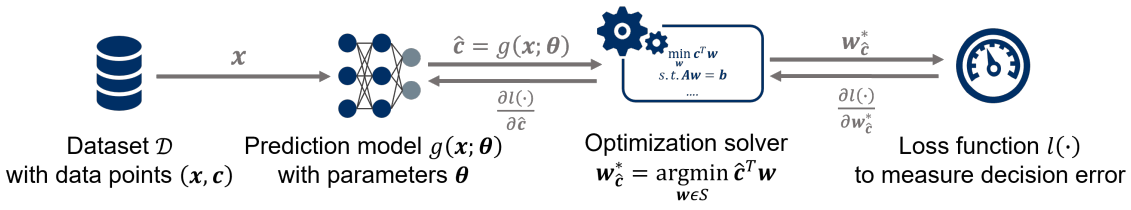


Figure 1: Illustration of the end-to-end predict-then-optimize framework

The end-to-end predict-then-optimize method in Figure 1 attempts to minimize the decision error. Consistent with deep learning terminology, we will use the term "backward pass" to refer to

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

Algorithm 1 End-to-end Gradient Descent

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

For an appropriately defined loss function, i.e., one that penalizes decision error, the chain rule can be used to calculate the following gradient of the loss w.r.t. predictor parameters. Due to lack of nonzero gradients in linear and integer programs, computing $\frac{\partial l(\cdot)}{\partial \hat{\boldsymbol{c}}}$ or $\frac{\partial \boldsymbol{w}^*_{\hat{\boldsymbol{c}}}}{\partial \hat{\boldsymbol{c}}}$ is challenging.

$$\begin{aligned} &\frac{\partial l(\cdot)}{\partial \boldsymbol{\theta}} = \frac{\partial l(\cdot)}{\partial \hat{\boldsymbol{c}}}\frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}} = \frac{\partial l(\cdot)}{\partial \boldsymbol{w}^*_{\hat{\boldsymbol{c}}}}\frac{\partial \boldsymbol{w}^*_{\hat{\boldsymbol{c}}}}{\partial \hat{\boldsymbol{c}}}\frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}} \\ &\text{Note: } \frac{\partial \hat{\boldsymbol{c}}}{\partial \boldsymbol{\theta}} = \frac{\partial g(\boldsymbol{x};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \end{aligned} \tag{2}$$

2.2.1. DECISION LOSS

To measure the error in decision-making, the notion of regret (also called SPO Loss [8]) has been proposed and is defined as the difference in objective value between an optimal solution (using the true but unknown cost vector) and one obtained using the predicted cost vector:

$$l_{\text{Regret}}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \boldsymbol{c}^T\boldsymbol{w}^*_{\hat{\boldsymbol{c}}} - z^*_{\boldsymbol{c}}. \tag{3}$$

2.3. Methodologies

2.3.1. SMART PREDICT-THEN-OPTIMIZE

To make the decision error differentiable, Elmachtoub and Grigas [8] proposed `SPO+`, a convex upper bound on the regret:

$$l_{SPO+}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = \min_{\boldsymbol{w}\in S}\{(2\hat{\boldsymbol{c}} - \boldsymbol{c})^T\boldsymbol{w}\} + 2\hat{\boldsymbol{c}}^T\boldsymbol{w}^*_{\boldsymbol{c}} - z^*_{\boldsymbol{c}}. \tag{4}$$

One proposed subgradient for this loss writes as follows:

$$2(\boldsymbol{w}^*_{\boldsymbol{c}} - \boldsymbol{w}^*_{2\hat{\boldsymbol{c}}-\boldsymbol{c}}) \in \frac{\partial l_{\text{SPO+}}(\hat{\boldsymbol{c}}, \boldsymbol{c})}{\partial \hat{\boldsymbol{c}}} \tag{5}$$

To accelerate the `SPO+` training, Mandi et al. [15] employed continous relaxations (`SPO+ Rel`) and warm starting (`SPO+ WS`) to speed-up the optimization.

2.3.2. DIFFERENTIABLE BLACK-BOX SOLVER

DBB was developed by Pogančić et al. [20] to estimate gradients from interpolation, replacing the zero gradient in $\frac{\partial \boldsymbol{w}^*_{\boldsymbol{c}}}{\partial \boldsymbol{c}}$.

Algorithm 2 DBB Forward Pass

Data: $\hat{\boldsymbol{c}}$
Result: $\boldsymbol{w}^*_{\hat{\boldsymbol{c}}}$
Solve $\boldsymbol{w}^*_{\hat{\boldsymbol{c}}}$
Save $\hat{\boldsymbol{c}}$ and $\boldsymbol{w}^*_{\hat{\boldsymbol{c}}}$ for backward pass

Algorithm 3 DBB Backward Pass

Data: $\frac{\partial l(\cdot)}{\partial \boldsymbol{w}^*_{\hat{\boldsymbol{c}}}}$, λ
Result: $\frac{\partial \boldsymbol{w}^*_{\hat{\boldsymbol{c}}}}{\partial \hat{\boldsymbol{c}}}$
Load $\hat{\boldsymbol{c}}$ and $\boldsymbol{w}^*_{\hat{\boldsymbol{c}}}$ from forward pass
$\boldsymbol{c}' := \hat{\boldsymbol{c}} + \lambda \frac{\partial l(\cdot)}{\partial \boldsymbol{w}^*(\hat{\boldsymbol{c}})}$
Solve $\boldsymbol{w}^*_{\boldsymbol{c}'}$
$\frac{\partial \boldsymbol{w}^*_{\hat{\boldsymbol{c}}}}{\partial \hat{\boldsymbol{c}}} := \frac{1}{\lambda}(\boldsymbol{w}^*_{\boldsymbol{c}'} - \boldsymbol{w}^*_{\hat{\boldsymbol{c}}})$

3. Implementation

The core module of *PyEPO* is an "autograd" function which is inherited from *PyTorch* [17]. These functions implement a forward pass that yields optimal solutions to the optimization problem and a backward pass to obtain non-zero gradients such that the prediction model can learn from the decision error or its surrogates. Thus, our implementation extends *PyTorch*, which facilitates the deployment of end-to-end predict-then-optimize tasks using any neural network that can be implemented in *PyTorch*.

We choose *GurobiPy* [12] and *Pyomo* [13] to build optimization models, which provide a natural way to express mathematical programming models in *PyEPO*. Users without specialized optimization knowledge can easily build and maintain optimization models through high-level algebraic representations. Besides *GurobiPy* and *Pyomo*, *PyEPO* also allows users to construct optimization models from scratch using any algorithm and solver. Modeling details and an example are in appendix A.

4. Empirical Evaluation

In this section, we present experimental results for the 2D-knapsack and TSP datasets in Appendix B. In these experiments, We examine the training time and the normalized regret on a test set with a sample size of $n_{\text{test}} = 1000$. Recall that the regret was defined in (3). We define the normalized regret by

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

The methods we compare include the two-stage approach and SPO+/DBB with a linear model (only a fully-connected layer). The predictors of two-stage method include linear regression, random forest, *Auto-Sklearn* [10] with MSE metric and 10 minutes time limit. For the sake of consistency, we only use regret (3) as the loss for DBB.

All the numerical experiments were conducted in Python v3.7.9 with Intel E5-2683 v4 Broadwell CPU processors and 8GB memory. Specifically, we used *PyTorch* [18] v1.10.0 for training end-to-end models, and *Scikit-Learn* [19] v0.24.2 and *Auto-Sklearn* [10] v0.14.6 as predictor of the two-stage method. *Gurobi* [12] v9.1.2 was the optimization solver in the background.

We compare the performance between two-stage methods, SPO+, and DBB with varying training data size $n \in \{100, 1000\}$, polynomial degree $deg \in \{1, 2, 4, 6\}$, and noise half-width $\bar{\epsilon}$ is 0.5. For the 2D-knapsack, number of items is 32 and capacity is 20. For the TSP, number of nodes is 20. We repeated all experiments 10 times, each with a different random vectors and matrix to generate 10 different training/validation/test datasets.

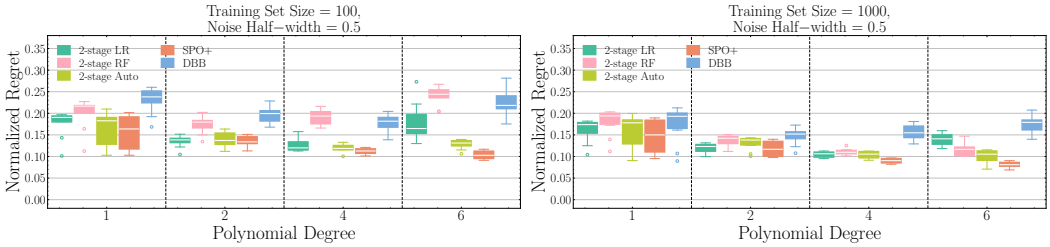


Figure 2: Normalized regret for the 2D knapsack problem on the test set, lower is better.

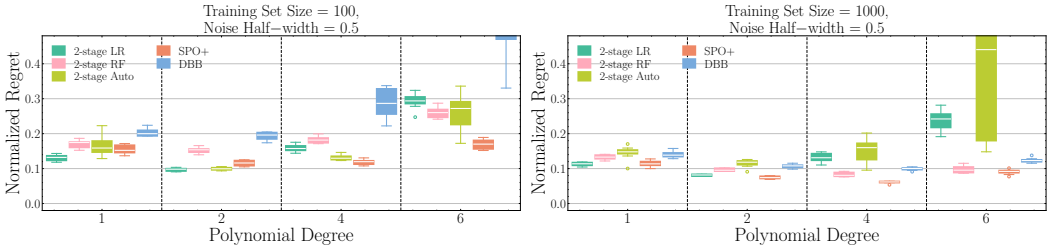


Figure 3: Normalized regret for the TSP problem on the test set, lower is better.

SPO+ shows its advantage: it performs best, or at least relatively well, in all cases. SPO+ is comparable to linear regression under low polynomial degree and depends less on the sample size than random forest. At high polynomial degrees, SPO+ outperforms *Auto-Sklearn*, which exposes the limitations of the two-stage approach.

5. Conclusion

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

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

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J. Zico Kolter. Differentiable convex optimization layers. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[3] Akshay Agrawal, Shane Barratt, Stephen Boyd, Enzo Busseti, and Walaa M Moursi. Differentiating through a cone program. *arXiv preprint arXiv:1904.09043*, 2019.

[4] Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, pages 136–145. PMLR, 2017.

[5] Yoshua Bengio. Using a financial training criterion rather than a prediction criterion. *International Journal of Neural Systems*, 8(04):433–443, 1997.

[6] Josip Djolonga and Andreas Krause. Differentiable learning of submodular models. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[7] Priya Donti, Brandon Amos, and J. Zico Kolter. Task-based end-to-end model learning in stochastic optimization. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[8] Adam N. Elmachtoub and Paul Grigas. Smart "predict, then optimize". *Management Science*, 0(0), 2021.

[9] Aaron Ferber, Bryan Wilder, Bistra Dilkina, and Milind Tambe. Mipaal: Mixed integer program as a layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1504–1511, 2020.

[10] Matthias Feurer, Aaron Klein, Jost Eggensperger, Katharina Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems 28 (2015)*, pages 2962–2970, 2015.

[11] Benjamin Ford, Thanh Nguyen, Milind Tambe, Nicole Sintov, and Francesco Delle Fave. Beware the soothsayer: From attack prediction accuracy to predictive reliability in security games. In *International Conference on Decision and Game Theory for Security*, pages 35–56. Springer, 2015.

[12] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021. URL https://www.gurobi.com.

[13] William E Hart, Carl D Laird, Jean-Paul Watson, David L Woodruff, Gabriel A Hackebeil, Bethany L Nicholson, John D Siirola, et al. *Pyomo-optimization modeling in python*, volume 67. Springer, 2017.

[14] Jayanta Mandi and Tias Guns. Interior point solving for lp-based prediction+optimisation. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 7272–7282. Curran Associates, Inc., 2020.

[15] Jayanta Mandi, Peter J Stuckey, Tias Guns, et al. Smart predict-and-optimize for hard combinatorial optimization problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1603–1610, 2020. doi: 10.1609/aaai.v34i02.5521.

[16] Hector Ortega-Arranz, Diego R Llanos, and Arturo Gonzalez-Escribano. The shortest-path problem: Analysis and comparison of methods. *Synthesis Lectures on Theoretical Computer Science*, 1(1):1–87, 2014.

[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS 2017 Autodiff Workshop*, 2017.

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[20] Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*, 2019.

[21] Bryan Wilder, Bistra Dilkina, and Milind Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1658–1665, 2019.

Appendix A. Code for Modeling

A.1. Optimization Model

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

In *PyEPO*, the **optModel** works as a black-box, which means that we do not specifically require a certain algorithm or a certain solver. This design is intended to give the users more freedom to customize their tasks. In addition, we provide more convenient API to create an optimization model with *GurobiPy* and *Pyomo*.

Let us use the following optimization model 6 with *Gurobi* as an example, where c_i is an unknown cost coefficient:

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

Inheriting **optGrbModel** is the convenient way to use *Gurobi* with *PyEPO*. The only implementation required is to override **_getModel** and return a *Gurobi* model and the corresponding decision variables. In addition, there is no need to assign a value to the attribute **modelSense** in **optGrbModel** manually. An example for Equation 6 is as follows:

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

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

optmodel = myModel()
```

A.2. Autograd Functions

Training neural networks with modern deep learning libraries such as *TensorFlow* [1] or *PyTorch* [18] requires gradient calculations for backpropagation. For this purpose, the numerical technique of automatic differentiation [17] is used. For example, *PyTorch* provides autograd functions, so that users are allowed to utilize or create functions that automatically compute partial derivatives.

Autograd functions are the core modules of *PyEPO* that solve and backpropagate the optimization problems with predicted costs. These functions can be integrated with different neural network architectures to achieve end-to-end predict-then-optimize for various tasks. In *PyEPO*, autograd functions include **SPOPlus** [8] and **blackboxOpt** [20].

A.2.1. FUNCTION SPOPLUS

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

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

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

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

A.2.2. FUNCTION BLACKBOXOPT

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

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

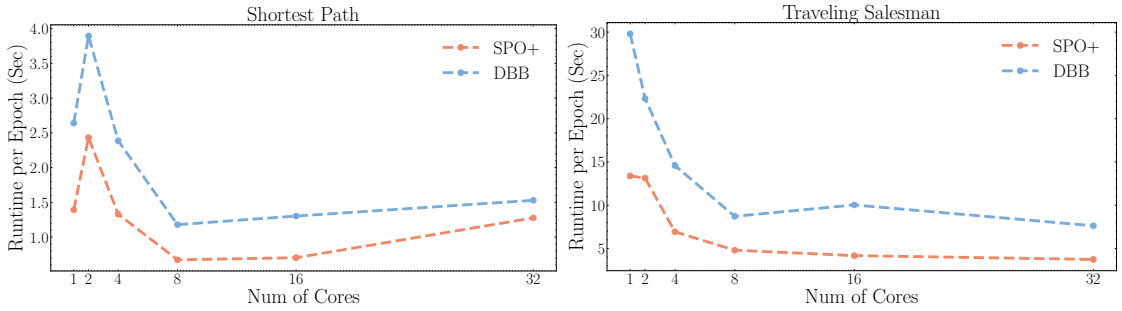
Since **blackboxOpt** works as a solver, there is only one parameter **pred_cost** for the forward pass. As in the code below, the output is the optimal solution for the given predicted cost:

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

A.3. Parallel Computation

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

Figure 4 shows the average running time per epoch for a mini-batch gradient descent algorithm with a batch size of 32 as a function of the number of cores. It was conducted in Python v3.7.9 with AMD Ryzen-7-5800X 8-Core Processor and 16GB memory. Specifically, we used *pathos* v0.2.8 for multiprocessing. The decrease in running time per epoch is sublinear in the number of



(*a*) (*b*)

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

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

Appendix B. Benchmark Datasets

In *PyEPO*, we generated new benchmark datasets for the task of end-to-end predict-then-optimize. Overall, we generate datasets in a similar way to Elmachtoub and Grigas [8]. The synthetic dataset $\mathcal{D}$ includes features $\boldsymbol{x}$ and cost coefficients $\boldsymbol{c}$: $\mathcal{D} = \{(\boldsymbol{x}_1, \boldsymbol{c}_1), (\boldsymbol{x}_2, \boldsymbol{c}_2), ..., (\boldsymbol{x}_n, \boldsymbol{c}_n)\}$. The feature vector $\boldsymbol{x}_i \in \mathbb{R}^p$ follows a standard multivariate Gaussian distribution $\mathcal{N}(0, \boldsymbol{I}_p)$ and the corresponding cost vector $\boldsymbol{c}_i \in \mathbb{R}^d$ comes from a (possibly nonlinear) polynomial function of $\boldsymbol{x}_i$ with additional random noise. $\epsilon_{ij} \sim \boldsymbol{U}(1 - \bar{\epsilon}, 1 + \bar{\epsilon})$ is the multiplicative noise term for c_{ij}, the j^{th} element of cost $\boldsymbol{c}_i$.

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

B.1. Shortest Path

We consider a $h \times w$ grid network and the goal is to find the shortest path [16] from northwest to southeast. We generate a random matrix $\mathcal{B} \in \mathbb{R}^{d \times p}$, where $\mathcal{B}_{ij}$ follows Bernoulli distribution with probability 0.5. Then, the cost vector $\boldsymbol{c}_i$ is almost the same as in [8], and is generated from

$$\left[\frac{1}{3.5^{deg}\sqrt{p}}((\mathcal{B}\boldsymbol{x}_i)_j + 3)^{deg} + 1\right] \cdot \epsilon_{ij}. \tag{7}$$

B.2. Multi-Dimensional Knapsack

Because we assume that the uncertain coefficients exist only in the objective function, the weights of items are fixed throughout the data. We use k to denote the number of resources; the number of items is same as the dimension of the cost vector d. The weights $\mathcal{W} \in \mathbb{R}^{k \times m}$ are sampled from 3 to 8 with a precision of 1 decimal place. With the same $\mathcal{B} \in \mathbb{R}^{d \times p}$ as in Section B.1, cost c_{ij} is calculated according to (7).

B.3. Traveling Salesperson

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