End-to-End Learning for Constrained Optimization

by

James Kotary

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

UNIVERSITY OF VIRGINIA

December 2024

THESIS COMMITTEE

THESIS SUPERVISOR

Ferdinando Fioretto
Department of Computer Science
University of Virginia

COMMITTEE MEMBERS

Jundong Li
Departments of Electrical Engineering and Computer Science
University of Virginia
Anil Vullikanti
Department of Computer Science
University of Virginia
Madhav Marathe
Department of Computer Science
University of Virginia
Bartolomeo Stellato
Department of Operations Research and Financial Engineering
Princeton University

End-to-End Learning for Constrained Optimization

by

James Kotary

Submitted to the Department of Computer Science
on December 2, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

In this thesis, we study the integration of constrained optimization algorithms with the training of deep neural networks. In particular, we we primarily interested in end-to-end trainable prediction and decision models composed of differentiable components. The use of such techniques spans several broad application areas, which we divide into two categories. When Learning to Optimize, the goal is to train neural networks to solve or aid in the solution of constrained optimization problems. In the Predict-Then-Optimize setting, the goal is to optimize decisions under uncertainty by estimating unknown coefficients in optimization problems from correlated data. Both frameworks stem from efforts to enhance optimization modeling technology for operations research and decision making tasks. This thesis contributes to both areas, and seeks to combine techniques from each to enhance the expressive and computational ability of models that learn to make decisions.

Acknowledgments

I'm most grateful to my advisor Nando, for his years of investment, support and collaboration. His belief in me over the years has contributed to my own self-belief. I'm also thankful for my lab mates: Cuong, My, Vincenzo, Saswat and Jacob, for their friendship, company, and commisseration. They've contributed greatly to this thesis, while adding many laughs along the way.

Special thanks go to Ján Drgoňa, who has been a second advisor to me through our collaboration at PNNL. His mentorship and support have shaped the way I do research as well as the direction of my career. Many thanks also go to Draguna Vrabie and Mahantesh Halappanavar, for the great opportunities they've opened to me at PNNL.

My appreciation also goes out to my collaborators outside UVA, whose contributions and discussions have shaped the material of this thesis. They include Jayanta Mandi, Senne Berden, Ethan King, and Ziwei Zhao. Great thanks also to my thesis committee, whose support I've been honored to have: Jundong Li, Anil Vullikanti, Madhav Marathe, and Bartolomeo Stellato.

Finally, I wish to acknowledge the great teachers and professors who have inspired my interest in science for many years. Among them I cannot forget Jae-Hun Jung, Brian Spencer, Micheal Cowen, Kenneth Takeuchi, Estie Arkin, Roman Samulyak, Brian Connery, and Joan Terenzetti.

Contents

Chapter 1

Introduction

Machine learning (ML) is a rapidly developing field, with recent developments having transformative impacts on information processing in domains such as natural language processing, computer vision, and web search. A growing subfield is that of Scientific ML, an extention of ML methodology which aims to enhance and augment existing computational methods, to aid in scientific and engineering research. On the other hand, constrained optimization (CO) is a subfield of mathematics which models decision-making problems by the minimization of functions over constrained sets, and poses efficient algorithmic solutions to those problems. This thesis focuses on the methodological integration of modern machine learning methods, with classical tools in constrained optimization, to enhance the efficacy of CO decision models by leveraging pattern recognition in data. The use of such techniques spans several application areas, which we broadly divide into two categories. When Learning to Optimize, the goal is to train neural networks to solve or aid in the solution of constrained optimization problems. In Prediction-Then-Optimize (PtO), the goal is to learn the uncertain coefficients of CO problems from observable data.

A substantial literature has been dedicated in recent years to the use of machine learning to accelerate the solution of optimization problems [89]. This research direction, often termed Learning to Optimize (LtO), aims to develop real-time constrained optimization capabilities, for applications requiring complex decisions to be made under stringent time constraints. These capabilities are increasingly demanded in settings such as job scheduling in manufacturing [90], power grid operation [54], and optimal control [132]. Many approaches within the LtO scope aim at assisting external optimization solvers with information such as learned heuristics [78] and active constraint prediction [108]. This thesis focuses in particular on *end-to-end* LtO, in which fully differentiable models are trained by gradient descent to minimize task-specific optimization objectives.

The *Predict-Then-Optimize* (PtO) framework models decision-making processes as optimization problems whose parameters are only partially known while the remaining, unknown, parameters must be estimated by a machine learning model. The predicted parameters complete the specification of an optimization problem, which is then solved to produce a final decision. Its main application is in optimization under uncertainty, where optimal decisions must be made with imperfect knowledge of the objective function. While conventional predictive modeling can be used to estimate unknown optimization parameters based on correlated data, modern PtO research is dominated by end-to-end learning approaches. By backpropagating gradients through the solution of an optimization problem, predictive ML models can be trained to optimize loss functions defined directly on the downstream decisions which result from their predictions. This thesis proposes novel methods which show how

PtO methodology can be used to enhance predictive modeling in well-known ML settings such as learning to rank and ensemble learning.

Modern approaches both of these problem settings, especially those based on end-to-end learning, often rely on *Differentiable Programming*. Optimization solvers which supply routines for backpropagation of gradients in *addition* to forward propagation of solutions are naturally useful in the design of integrated prediction and optimization models. Although we do not consider Differentiable Programming as a problem setting, the design and analysis of such tools is important enough that it is treated as a third main topic in this thesis. Next, each of our main topics of interest is described in further detail, followed by an overview of research challenges and contributions made in this thesis.

1.1 Problem Settings

This section gives an overview of the Learning-to-Optimize and Predict-Then-Optimize settings, and prefaces the topic of differentiable programming, before the main technical contributions of the thesis are discussed.

Learning to Optimize Consider a generic optimization problem with continuous variables, subject to equality and inequality constraints, all parameterized by a vector of coefficients $\boldsymbol{c} \in \mathbb{R}^c$. From this we may define a mapping from any instance of coefficients $\boldsymbol{c}$ to the resulting optimal solution $\boldsymbol{x}^\star(\boldsymbol{c}) \in \mathbb{R}^n$:

$$\boldsymbol{x}^\star(\boldsymbol{c}) \in \underset{\boldsymbol{x}}{\operatorname{argmin}} \quad f_{\boldsymbol{c}}(\boldsymbol{x}) \tag{1.1a}$$

$$s.t. \quad h_{\boldsymbol{c}}(\boldsymbol{x}) = \boldsymbol{0} \tag{1.1b}$$

$$g_{\boldsymbol{c}}(\boldsymbol{x}) \leq \boldsymbol{0} \tag{1.1c}$$

in which any choice of $\boldsymbol{c}$ specifies an optimization problem by determining functions the $f_{\boldsymbol{c}} : \mathbb{R}^n \to \mathbb{R}$, $g_{\boldsymbol{c}} : \mathbb{R}^n \to \mathbb{R}^m$, and $h_{\boldsymbol{c}} : \mathbb{R}^n \to \mathbb{R}^p$. In turn, this determines a corresponding optimal solution $\boldsymbol{x}^\star(\boldsymbol{c})$.

The goal is to train an optimization *proxy solver* $\mathcal{X}_{\boldsymbol{\theta}} : \mathbb{R}^c \to \mathbb{R}^n$, over a distribution of problem parameters $\boldsymbol{c} \sim C$, which approximates the mapping $\boldsymbol{x}^\star(\boldsymbol{c})$ as defined by equation (1.1). The proxy model $\mathcal{X}_{\boldsymbol{\theta}}$ may consist of a deep neural network $\mathcal{N}_{\boldsymbol{\theta}}$ with *trainable* weights $\boldsymbol{\theta}$, possibly combined with a *non-trainable correction routine* $\mathcal{S}$ to improve solution quality, so that $\mathcal{X}_{\boldsymbol{\theta}} = \mathcal{S} \circ \mathcal{N}_{\boldsymbol{\theta}}$. One may define the following as an ideal training goal for the optimization proxy model $\mathcal{X}_{\boldsymbol{\theta}}$:

$$\min_{\boldsymbol{\theta}} \quad \mathop{\mathbb{E}}_{\boldsymbol{c} \sim C} \left[\; f_{\boldsymbol{c}}\left(\mathcal{X}_{\boldsymbol{\theta}}(\boldsymbol{c})\right) \; \right] \tag{1.2a}$$

$$s.t. \quad h_{\boldsymbol{c}}\left(\mathcal{X}_{\boldsymbol{\theta}}(\boldsymbol{c})\right) = \boldsymbol{0} \quad \forall \boldsymbol{c} \sim C \tag{1.2b}$$

$$g_{\boldsymbol{c}}\left(\mathcal{X}_{\boldsymbol{\theta}}(\boldsymbol{c})\right) \leq \boldsymbol{0} \quad \forall \boldsymbol{c} \sim C. \tag{1.2c}$$

The training goal (1.2) emphasizes that each solution $\mathcal{X}_{\boldsymbol{\theta}}(\boldsymbol{c})$ produced by the proxy solver must be feasible to the constraints of problem (1.1). Subject to these constraints on each output, their mean objective value should thus be minimized. This training goal is evidently an unsupervised learning problem. However, as discussed in this thesis, supervision via labeled information such as precomputed optimal solutions $\boldsymbol{x}^\star(\boldsymbol{c})$ may also be leveraged.

Predict-Then-Optimize In this setting, decision problems are modeled as optimization problems whose parameters are only partially known while the remaining, unknown, parameters must be estimated by a machine learning (ML) model C_θ . The predicted parameters complete the specification of an optimization problem which is then solved to produce a final decision. The problem is posed as estimating the solution $\boldsymbol{x}^\star(c) \in \mathcal{X} \subseteq \mathbb{R}^n$ of a *parametric* optimization problem:

$$\boldsymbol{x}^\star(\boldsymbol{c}) = \underset{\boldsymbol{x}}{\operatorname{argmin}} \;\; f(\boldsymbol{x}, \boldsymbol{c}) \tag{1.3a}$$

$$\textit{such that:} \;\; \boldsymbol{x} \in \mathcal{S}, \tag{1.3b}$$

given that parameters $\boldsymbol{c} \in C \subseteq \mathbb{R}^p$ are unknown, but that a correlated set of observable values $\boldsymbol{z} \in \mathcal{Z}$ are available. Here f is an objective function, and $\boldsymbol{g}$ and $\boldsymbol{h}$ define the set of the problem's inequality and equality constraints. The combined prediction and optimization model is evaluated on the basis of the optimality of its downstream decisions, with respect to f under its ground-truth problem parameters [52]:

$$\min_\theta \;\; \mathbb{E}_{(\boldsymbol{z},\boldsymbol{c})\sim\Omega} \left[f\left(\boldsymbol{x}^\star(\boldsymbol{C}_\theta(\boldsymbol{z})), \boldsymbol{c}\right)\right]. \tag{1.4}$$

This setting is ubiquitous to many real-world applications confronting the task of decision-making under uncertainty, such as planning the shortest route in a city, determining optimal power generation schedules, or managing investment portfolios. For example, a vehicle routing system may aim to minimize a rider's total commute time by solving a shortest-path optimization model (1.3) given knowledge of the transit times $\boldsymbol{c}$ over each individual city block. In absence of that knowledge, it may be estimated by prediction models based on exogenous data $\boldsymbol{z}$, such as weather and traffic conditions. In this context, more accurately predicted transit times $\hat{\boldsymbol{c}}$ tend to produce routing plans $\boldsymbol{x}^\star(\hat{\boldsymbol{c}})$ with shorter commutes, with respect to the true city-block transit times $\boldsymbol{c}$.

However, direct training of predictions from observable features to problem parameters can generalize poorly with respect to the ground-truth optimality achieved by a subsequent decision model [104, 89]. To address this challenge, *End-to-end Predict-Then-Optimize* (EPO) [52] has emerged as a transformative paradigm in data-driven decision making, where predictive models are trained to directly minimize loss functions defined on the downstream optimal solutions $\boldsymbol{x}^\star(\hat{\boldsymbol{c}})$.

Differentiable Programming To integrate optimization solvers with end-to-end trainable prediction models, is often necessary to view them as differentiable functions. In terms of the mapping (1.1), we are interested in optimization solvers which can not only evaluate the function $\boldsymbol{x}^\star(\hat{\boldsymbol{c}})$, but also its derivatives. Such solvers form a key toolset in the design of algorithms for both of the above-described problem settings. Because of their central role in the topics of this thesis, we have dedicated sections to their design and analysis.

1.2 Motivations and Contributions

Within each of its main topics of interest, we preface the motivations behind the research contributions of this thesis. The following research questions establish the point of view of this thesis, framing its aims and goals relative to the current state of the literature.

Research Questions in Learning to Optimize In learning to solve optimization problems (1.2), we consider two key aspects of the machine learning design: **(1)** What information should be learned about an optimal solution, and **(2)** what information should be used to supervise that learning? More precisely, we investigate the following research questions:

1. When a Learning to Optimize method is trained to solve NP-hard problems under the supervision of pre-computed solutions, how can we understand the effects of nonuniqueness and suboptimality in those solutions?

2. What kinds of information can be predicted to assist in accelerating the convergence of an optimization routine?

These questions motivate the research contributions of Chapter 3. In Section 3.1, we propose an analysis on the relationship between generation of target solutions and the resulting accuracy of a learned optimization model. In Section 3.2, we propose the idea of learning non-Euclidean metrics to accelerate the convergence of operator splitting schemes based on proximal operators, and show that the enhanced convergence can be related to a latent prediction of active constraints.

Research Questions in Differentiable Programming As shown in prior work, differentiable optimization solvers can be implemented in a variety of ways. A standard approach known as *unrolling* is to execute an entire optimization algorithm in an automatically differentiable environment, then backpropagate each of its successive operations to compute a derivative. More efficient approaches based on implicit differentiation have since been developed. In Chapter 4, we seek to better understand the mathematics of backpropagation through optimization as well as its potential applications in machine learning. The following research questions are pursued:

- How can we understand the backpropagation of optimization algorithms implemented in automatically differentiable environments?

- How can differentiable programming be used to enhance performance on traditional machine learning tasks?

In Section 4.1, we address the first question by proposing a convergence analysis of unrolling, which reveals its limiting convergence rate and its conceptual connections to implicit differentiation. Then in Section 4.2, we show how discrete masking mechanisms based on differentiable optimization can be used to learn model selection, enhancing performance in ensemble learning.

Research Questions in Predict-Then-Optimize In the Predict-Then-Optimize scope, we are interested in using PtO methodologies to solve conventional machine learning tasks while leveraging the ability to impart hard constraints on their behavior via the constrained optimization component. In particular, we are interested in PtO as a way of imparting algorithmic fairness guarantees on the outputs of ML models. In part, this requires us to extend the PtO methodology to optimization problems with multiple objectives. We pose the following research questions to motivate the contributions of Chapter 5:

1. How can the PtO framework be used extended to handle multiple uncertain objective criteria? In particular, how can it be used to learn decisions that promote fairness over outcomes across multiple parties?

2. How can the PtO framework be leveraged to ensure fair outcomes in widely deployed applications of ML?

The first question is addressed in Section 5.1, which recognizes that fair aggregation functions over multiple objectives are often *nondifferentiable*, requiring novel aproximation methods for their incorporation in end-to-end learning. Then, Section 5.2 establishes a new line of applications for the PtO framework, in the important area of fairness-constrained learning to rank. By combining deep learning with constrained optimization, it proposes integrated models for web ranking optimization with guarantees on fairness in exposure of various content in search results.

Before detailing the original contributions of this thesis work, the next chapter gives a broad overview of preliminary concepts and related research works in the existing literature.

Chapter 2

Background

This chapter provides a broad overview of related work and preliminary concepts. It covers basic concepts of deep learning and constrained optimization, before describing and categorizing modern research papers at the intersection of machine learning and optimization. Consistent with the main interests of this thesis, we focus on those works aimed at designing end-to-end trainable models, particularly in the learning-to-optimize and predict-then-optimize settings. Much of the material in this section is based on the author's published survey paper [89]. A later, more extensive survey of the topic can be found in [104].

2.1 Preliminaries: Constrained Optimization

A constrained optimization (CO) problem poses the task of minimizing an *objective function* $f : \mathbb{R}^n \to \mathbb{R}_+$ of one or more variables $\boldsymbol{z} \in \mathbb{R}^n$, subject to the condition that a set of *constraints* C are satisfied between the variables:

$$O = \underset{\boldsymbol{z}}{\operatorname{argmin}}\, f(\boldsymbol{z}) \quad \text{subject to} \quad \boldsymbol{z} \in C. \tag{2.1}$$

An assignment of values $\boldsymbol{z}$ which satisfies C is called a *feasible solution*; if, additionally $f(\boldsymbol{z}) \leq f(\boldsymbol{w})$ for all feasible $\boldsymbol{w}$, it is called an *optimal solution*.

A well-understood class of optimization problems are *convex* problems, those in which the constrained set C is a convex set, and the objective f is a convex function. Convex problems have the favorable properties of being efficiently solvable with strong theoretical guarantees on the existence and uniqueness of solutions [24].

A particularly common constraint set arising in practical problems takes the form $C = \{\boldsymbol{z} : \boldsymbol{A}\boldsymbol{z} \leq \boldsymbol{b}\}$, where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$. In this case, C is a convex set. If the objective f is an affine function, the problem is referred to as *linear program* (LP). When a linearly constrained problem includes a quadratic objective rather than a linear one, the result is called a *quadratic program* (QP). If, in addition, some subset of a problem's variables are required to take integer values, it is called *mixed integer program* (MIP). While LP and QP with convex objectives belong to the class of convex problems, the introduction of integral constraints ($\boldsymbol{x} \in \mathbb{N}^n$) results in a much more difficult problem. The feasible set in MIP consists of distinct points in $\boldsymbol{x} \in \mathbb{R}^n$, not only nonconvex but also disjoint, and the resulting problem is, in general, NP-Hard. Finally, nonlinear programs (NLPs) are

optimization problems where some of the constraints or the objective function are nonlinear. Many NLPs are nonconvex and can not be efficiently solved [115].

Of particular interest are the *mixed integer linear programs* (MILPs), linear programs in which a subset of variables required to take integral values. This survey is primarily concerned with optimization problems involving linear constraints, linear or quadratic objective, and either continuous or integral variables, or a combination thereof.

Optimization Solving Methods A well-developed theory exists for solving convex problems. Methods for solving LP problems include *simplex* methods [36], *interior point* methods [24] and *Augmented Lagrangian* methods [69, 124]. Each of these methods has a variant that applies when the objective function is convex. Convex problems [24] are in the class of $\boldsymbol{P}$, and can be assumed to be reliably and efficiently solvable in most contexts. For a review on convex optimization and Lagrangian duality, the interested reader is referred to [24].

MILPs require a fundamentally different approach, as their integrality constraints put them in the class of NP-Hard problems. *Branch and bound* is a framework combining optimization and search, representable by a search tree in which a LP *relaxation* of the MILP is formed at each node by dropping integrality constraints, and efficiently solved using solution methods for linear programming. Subsequently, a variable z_i with (fractional) value a_i in the relaxation is selected for branching to two further nodes. In the right node, the constraint $z_i \geq a_i$ is imposed and in the left, $z_i < a_i$; bisecting the search space and discarding fractional values in between the bounds. The solution of each LP relaxtion provides a lower bound on the MILP's optimal objective value. When an LP relaxation happens to admit an integral solution, an upper bound is obtained. The minimal upper bound found thus far is used, at each node, for pruning.

Finally, *Constraint Programming* [128] is an additional effective paradigm adopted to solve industrial-sized MI(L)P and discrete combinatorial programs.

2.2 Preliminaries: Deep Learning

Supervised deep learning can be viewed as the task of approximating a complex non-linear mapping from targeted data. Deep Neural Networks (DNNs) are deep learning architectures composed of a sequence of layers, each typically taking as inputs the results of the previous layer [97]. Feed-forward neural networks are basic DNNs where the layers are fully connected and the function connecting the layer is given by $\boldsymbol{o} = \pi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$, where $\boldsymbol{x} \in \mathbb{R}^n$ and is the input vector, $\boldsymbol{o} \in \mathbb{R}^m$ the output vector, $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ a matrix of weights, and $\boldsymbol{b} \in \mathbb{R}^m$ a bias vector. The function $\pi(\cdot)$ is often non-linear (e.g., a rectified linear unit (ReLU)).

Supervised learning tasks consider datasets $\chi = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$ consisting of N data points with $\boldsymbol{x}_i \in \mathcal{X}$ being a feature vector and $\boldsymbol{y}_i \in \mathcal{Y}$ the associated targets. The goal is to learn a model $\mathcal{M}_\theta : \mathcal{X} \to \mathcal{Y}$, where θ is a vector of real-valued parameters, and whose quality is measured in terms of a nonnegative, and assumed differentiable, *loss function* $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$. The learning task minimizes the empirical risk function (ERM):

$$\min_\theta J(\mathcal{M}_\theta, \chi) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\mathcal{M}_\theta(\boldsymbol{x}_i), \boldsymbol{y}_i). \tag{2.2}$$

Most of the techniques surveyed in this work use (variants of) DNNs whose training conforms to the objective above. Other notable classes of deep learning methods used to solve CO problems are Graph Neural Networks (GNNs), which exploit architectures designed to perform inference on data described by graphs, and Reinforcement Learning (RL), which differs from supervised learning in not requiring labeled input/output pairs and concerns with learning a policy that maximizes some expected reward function. We refer the reader to [159] and [137] for an extensive overview of GNNs and RL, respectively.

2.3 Overview of ML and CO

Current research areas in the synthesis of constrained optimization and machine learning can be categorized into two main directions: *ML-augmented CO*, which focuses on using ML to aid the performance of CO problem solvers, and *End-to-End CO learning*, which focuses on integrating combinatorial solvers or optimization methods into deep learning architectures.

The area related with End-to-End CO learning is the focus of this survey and is concerned with the data-driven prediction of solutions to CO problems. We divide this area into: **(1)** approaches that develop ML architectures to predict fast, approximate solutions to predefined CO problems, further categorized into *learning with constraints* and *learning on graphs*, and **(2)** approaches that exploit CO solvers as neural network layers for the purpose of structured logical inference, referred to here as the *Predict-Then-Optimize* paradigm.

2.4 ML-augmented CO

ML-augmented CO involves the augmentation of already highly-optimized CO solvers with ML inference models. Techniques in this area draw on both supervised and RL frameworks to develop more efficient approaches to various aspects of CO solving for both continuous and discrete combinatorial problems.

In the context of combinatorial optimization, these are broadly categorized into methods that learn to guide the search decisions in branch and bound solvers, and methods that guide the application of primal heuristics within branch and bound. The former include low-cost emulation of expensive branching rules in mixed integer programming [79, 62, 67], prediction of optimal combinations of low-cost variable scoring rules to derive more powerful ones [12], and learning to cut [140] in cutting plane methods within MILP solvers. The latter include prediction of the most effective nodes at which to apply primal heuristics [82], and specification of primal heuristics such as the optimal choice of variable partitions in large neighborhood search [136]. The reader is referred to the excellent surveys [100, 17] for a thorough account of techniques developed within ML-augmented combinatorial optimization.

Techniques in this area have also used ML to improve decisions in continuous CO problems and include learning restart strategies [61], learn rules to ignore some optimization variables leveraging the expected sparsity of the solutions and consequently leading to faster solvers, [113], and learning active constraints [114, 125] to reduce the size of the problem to be fed into a CO solver .

2.5 Learning CO Solutions

A diverse body of work within the end-to-end CO learning literature has focused on developing ML architectures to predict fast, approximate solutions to predefined CO problems *end-to-end* without the use of CO solvers at the time of inference, by observing a set of solved instances or execution traces. These approaches contrasts with those that use ML to augment search-based CO solvers and configure their subroutines to direct the solver to find solutions efficiently. This survey categorizes the literature on *learning CO solutions* into **(1)** methods that incorporate constraints into end-to-end learning, for the prediction of feasible or near-feasible solutions to both continuous and discrete constrained optimization problems, and **(2)** methods that learn combinatorial solutions on graphs, with the goal of producing outputs as combinatorial structures from variable-sized inputs. These two categories, referred to as *learning with constraints* and *learning CO solutions*, respectively, are reviewed next.

2.5.1 Learning with Constraints

The methods below consider datasets $\chi = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N}$ whose inputs $\boldsymbol{x}_i$ describe some problem instance specification, such as matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ describing linear constraints in MILPs, and the labels $\boldsymbol{y}_i$ describe complete solutions to problem O with input $\boldsymbol{x}_i$. Notably, each sample may specify a different problem instance (with different objective function coefficients and constraints).

An early approach to the use of ML for learning CO problem solutions was presented by Hopfield and Tank [70], which used Hopfield Networks (Hopfield [71]) with modified energy functions to emulate the objective of a traveling salesman problem (TSP), and applied Lagrange multipliers to enforce feasibility to the problem's constraints. It was shown in Wilson and Pawley [156] however, that this approach suffers from several weakness, notably sensitivity to parameter initialization and hyperparameters. As noted in Bello et al. [16] , similar approaches have largely fallen out of favor with the introduction of practically superior methods.

Frameworks that exploit Lagrangian duality to guide the prediction of approximate solutions to satisfy the problem's constraints have found success in the context of continuous NLPs including energy optimization [56, 146] as well as constrained prediction on tasks such as transprecision computing and fair classification [54, 142].

Other end-to-end learning approaches have demonstrated success on the prediction of solutions to constrained problems by injecting information about constraints from targeted feasible solutions. Recently, Detassis et al. [38] presented an iterative process of using an external solver for discrete or continuous optimization to adjust targeted solutions to more closely match model predictions while maintaining feasibility, reducing the degree of constraint violation in the model predictions in subsequent iterations.

2.5.2 Learning Solutions on Graphs

By contrast to approaches learning solutions to unstructured CO problems, a variety of methods learn to solve CO cast on graphs. The development of deep learning architectures such as sequence models and attention mechanisms, as well as GNNs, has provided a natural toolset for these tasks and led to substantial improvements in the state of the art.

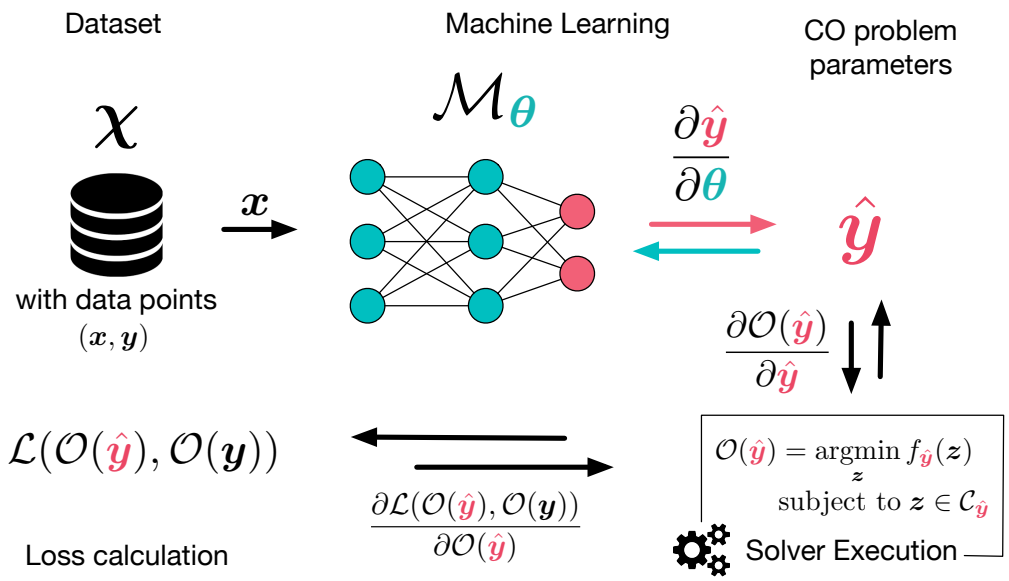


Figure 2.1: Predict-then-optimize framework; gradients of a solver output (solution) must be computed with respect to its input (problem parameters) in order to maximize empirical model performance.

Vinyals et al. [148] introduced the *pointer network*, in which a sequence-to-sequence model uses an encoder-decoder architecture paired with an attention mechanism to produce permutations over inputs of variable size. The resulting model was used to learn solutions to the TSP and the Delaunay triangulation problems from previously solved instances in a supervised manner, and demonstrated some ability to generalize over variable-sized problem instances. This pointer network architecture was also adopted by Bello et al. [16] but developed an improved model for learning the TSP by training it with RL, using tour length as the reward signal. The move from supervised to RL was motivated partly by the difficulties associated with obtaining target solutions that are optimal, and the existence of nonunique optimal solutions. Kool et al. [86] applied an attention-based RL model to the TSP as well as variants of the vehicle routing problem, but with a graph attention network [145] inspired by the Transformer architecture [144]. This neural network design introduces invariance to permutations of the input nodes, improving learning efficiency.

Khalil et al. [81] adopted a different RL approach based on a greedy heuristic framework which determines approximate solutions by the sequential selection of graph nodes. The selection policy is learned by reinforcement learning, using a graph neural network (GNN) to predict actions given a graph embedding representation of the current solution's state. Despite the trend toward RL-oriented frameworks, Nowak et al. [116] discusses a purely supervised learning method for the general quadratic assignment problem based on the use of GNN's trained on representations of individual problem instances and their targeted solutions. Inferences from the model take the form of permutations, which are converted into feasible solutions by a beam search.

These and more approaches are covered in detail in [28], which provides a thorough survey on CO and reasoning with GNNs.

2.6 Predict-Then-Optimize

A burgeoning topic in the intersection of ML and CO is the fusion of prediction (ML) and decision (CO) models, in which decision models are represented by partially defined optimization problems, whose specification is completed by parameters that are predicted from data. The resulting composite models employ constrained optimization as a neural network layer and are trained *end-to-end*, based on the optimality of their decisions. This setting is altogether different in motivation to those previously discussed, in which the goal was to solve predefined CO instances with increased efficiency. The goal here is the synthesis of predictive and prescriptive techniques to create ML systems that learn to make decisions based on empirical data.

The following constrained optimization problem is posed, in which the objective function $f_{\boldsymbol{y}}$ and feasible region $C_{\boldsymbol{y}}$ depend on a parameter vector $\boldsymbol{y}$:

$$O(\boldsymbol{y}) = \underset{\boldsymbol{z}}{\operatorname{argmin}}\, f_{\boldsymbol{y}}(\boldsymbol{z}) \;\text{ subject to }\; \boldsymbol{z} \in C_{\boldsymbol{y}}. \tag{2.3}$$

The goal here is to use supervised learning to predict $\hat{\boldsymbol{y}}$ the unspecified parameters from empirical data. The learning task is performed so that the optimal solution $O(\hat{\boldsymbol{y}})$ best matches a targeted optimal solution $O(\boldsymbol{y})$, relative to some appropriately chosen loss function. The empirical data in this setting is defined abstractly as belonging to a dataset χ, and can represent any empirical observations correlating with targeted solutions to (2.3) for some $\boldsymbol{y}$. See Figure 2.1 for an illustration.

This framework aims to improve on simpler two-stage approaches, in which a conventional loss function (e.g. MSE) is used to target labeled parameter vectors $\boldsymbol{y}$ that are provided in advance, before solving the associated CO problem to obtain a decision. Such approaches are suboptimal in the sense that predictions of $\boldsymbol{y}$ do not take into account the accuracy of the resulting solution $O(\boldsymbol{y})$ during training.

We note that there are two ways to view the predictions that result from these integrated models. If $\hat{\boldsymbol{y}}$ is viewed as the prediction, then the calculation of $O(\hat{\boldsymbol{y}})$ is absorbed into the loss function $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$, which targets the provided parameter vectors. Otherwise, the loss function $\mathcal{L}(O(\hat{\boldsymbol{y}}), O(\boldsymbol{y}))$ is considered separately from the decision model and aims to match computed optimal solutions to targeted ones. One advantage sought in either case is the opportunity to minimize during training the ultimate error in the computed optimal objective values $f_{\hat{\boldsymbol{y}}}(O(\hat{\boldsymbol{y}}))$, relative to those of the target data. This notion of training loss is known as *regret:*

$$\mathit{regret}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = f_{\hat{\boldsymbol{y}}}(O(\hat{\boldsymbol{y}})) - f_{\boldsymbol{y}}(O(\boldsymbol{y})).$$

Otherwise the optimal solution $O(\boldsymbol{y})$ *is targeted and one can use* $\mathrm{regret}(O(\hat{\boldsymbol{y}}), O(\boldsymbol{y}))$*, regardless of whether the corresponding* $\boldsymbol{y}$ *is available.* Depending on the techniques used, it may be possible to minimize the regret without access to ground-truth solutions, as in Wilder et al. [153], since the targeted solutions $O(\boldsymbol{y})$ contribute only a constant term to the overall loss. It is worth mentioning that because the optimization problem in (2.3) is viewed as a function, the existence of nonunique optimal solutions is typically not considered. The implication then is that $O(\boldsymbol{y})$ is directly determined by $\boldsymbol{y}$.

Training these end-to-end models involves the introduction of external CO solvers into the training loop of a ML model, often a DNN. Note that combinatorial problems with discrete state spaces do not offer useful gradients; viewed as a function, the *argmin* of a discrete problem is

piecewise constant. *The challenge of forming useful approximations to* $\frac{\partial \mathcal{L}}{\partial y}$ *is central in this context and must be addressed in order to perform backpropagation*. It may be approximated directly, but a more prevalent strategy is to model $\frac{\partial O(y)}{\partial y}$ and $\frac{\partial \mathcal{L}}{\partial O}$ separately, in which case the challenge is to compute the former term by *differentiation through argmin*. Figure 2.1 shows the role of this gradient calculation in context.

2.6.1 Quadratic Programming

One catalyst for the development of this topic was the introduction of *differentiable optimization layers*, beginning with Amos and Kolter [6] which introduced a GPU-ready QP solver that offers exact gradients for backpropagation by differentiating the KKT optimality conditions of a quadratic program at the time of solving, and utilizing information from the forward pass to solve a linear system for incoming gradients, once the outgoing gradients are known. Subsequently, Donti et al. [47] proposed a *predict-then-optimize* model in which QPs with stochastic constraints were integrated in-the-loop to provide accurate solutions to inventory and power generator scheduling problems specified by empirical data.

2.6.2 Linear Programming

Concurrent with Donti et al. [47], an alternative methodology for end-to-end learning with decision models, called *smart predict-then-optimize* (SPO), was introduced by Elmachtoub and Grigas [52], which focused on prediction with optimization of constrained problems with linear objectives, in which the cost vector is predicted from data and the feasible region C is invariant to the parameter $\boldsymbol{y}$:

$$O(\boldsymbol{y}) = \underset{\boldsymbol{z}}{\operatorname{argmin}}\, \boldsymbol{y}^T \boldsymbol{z} \quad \text{subject to} \quad \boldsymbol{z} \in C. \tag{2.4}$$

The target data in this work are the true cost vectors $\boldsymbol{y}$, and an inexact subgradient calculation is used for the backpropagation of regret loss $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \hat{y}^T(O(\hat{\boldsymbol{y}}) - O(\boldsymbol{y}))$ on the decision task, by first defining a convex surrogate upper bound on regret called the *SPO+loss*, for which it is shown that $O(\boldsymbol{y}) - O(2\hat{\boldsymbol{y}} - \boldsymbol{y})$ is a useful subgradient. Since this work is limited to the development of surrogate loss functions on regret from the optimization task, it does not apply to learning tasks in which the full solution to an optimization problem is targeted. The paper includes a discussion justifying the method's use on problems with discrete constraints in C, as in combinatorial optimization, but experimental results are not provided on that topic. It is, however, demonstrated that the approach succeeds in a case where C is convex but nonlinear.

Wilder et al. [153] introduced an alternative framework to predict-then-optimize linear programming problems, based on exact differentiation of a smoothed surrogate model. While LPs are special cases of QPs, the gradient calculation of Amos and Kolter [6] does not directly apply due to singularity of the KKT conditions when the objective function is purely linear. This is addressed by introducing a small quadratic regularization term to the LP objective $f_{\boldsymbol{y}}(\boldsymbol{z}) = \boldsymbol{y}^T \boldsymbol{z}$ so that the problem in (2.3) becomes

$$O(\boldsymbol{y}) = \underset{\boldsymbol{z}}{\operatorname{argmin}}\, \boldsymbol{y}^T \boldsymbol{z} + \epsilon \|\boldsymbol{z}\|^2 \quad \text{subject to} \quad \boldsymbol{A}\boldsymbol{z} \leq \boldsymbol{b}. \tag{2.5}$$

The resulting problems approximate the desired LP, but have unique solutions that vary smoothly as a function of their parameters, allowing for accurate backpropagation of the result. The integrated

model is trained to minimize the expected optimal objective value across all training samples, which is equivalent to minimizing the regret loss but without the need for a target dataset. This work demonstrated success on problems such as the knapsack (using LP relaxation) and bipartite matching problems where a cost vector is predicted from empirical data (e.g., historical cost data for knapsack items), and is shown to outperform two-stage models which lack integration of the LP problem. We note that although the differentiable QP solving framework of Amos and Kolter [6] is capable of handling differentiation with respect to any objective or constraint coefficient, this work only report results on tasks in which the *cost* vector is parameterized within the learning architecture, and constraints are held constant across each sample. *This limitation is common to all of the works described below, as well.*

Next, Mandi and Guns [102] introduced an altogether different approach to obtaining approximate gradients for the *argmin* of a linear program. An interior point method is used to compute the solution of a homogeneous self-dual embedding with early stopping, and the method's log-barrier term is recovered and used to solve for gradients in the backward pass. Equivalently, this can be viewed as the introduction of a log-barrier regularization term, by analogy to the QP-based approach of Wilder et al. [153]:

$$O(\boldsymbol{y}) = \underset{\boldsymbol{z}}{\operatorname{argmin}}\, \boldsymbol{y}^T \boldsymbol{z} + \lambda \left(\sum_i \ln(z_i) \right) \quad \text{subject to} \quad \boldsymbol{A}\boldsymbol{z} \leq \boldsymbol{b}.$$

Further, the method's performance on end-to-end learning tasks is evaluated against the QP approach of Wilder et al. [153] on LP-based predict-then-optimize tasks, citing stronger accuracy results on energy scheduling and knapsack problems with costs predicted from data.

Berthet et al. [20] detailed an approach based on stochastic perturbation to differentiate the output of linear programs with respect to their cost vectors. The output space of the LP problem is smoothed by adding low-amplitude random noise to the cost vector and computing the expectation of the resulting solution in each forward pass. This can be done in Monte Carlo fashion and in parallel across the noise samples. The gradient calculation views the solver as a black box in this approach, and does not require the explicit solving of LP for operations that can be mathematically formulated as LP, but are simple to perform (e.g., sorting and ranking). Results include a replication of the shortest path experiments presented in [123], in which a model integrated with convolutional neural networks is used to approximate the shortest path through stages in a computer game, solely from images.

2.6.3 Combinatorial Optimization

Ferber et al. [53] extended the approach of Wilder et al. [153] to integrate MILP within the end-to-end training loop, with the aim of utilizing more expressive NP-Hard combinatorial problems with parameters predicted from data. This is done by reducing the MILP with integer constraints to a LP by a method of cutting planes. In the ideal case, the LP that results from the addition of cutting planes has the same optimal solution as its mixed-integer parent. Exact gradients can then be computed for its regularized QP approximation as in Wilder et al. [153]. Although the LP approximation to MILP improves with solving time, practical concerns arise when the MILP problem cannot be solved to completion. Each instance of the NP-Hard problem must be solved in each forward pass of the training loop, which poses clear obstructions in terms of runtime. A

disadvantage of the approach is that cutting-plane methods are generally considered to be inferior in efficiency to staple methods like branch and bound. Improved results were obtained on portfolio optimization and diverse bipartite matching problems, when compared to LP-relaxation models following the approach of Wilder et al. [153].

Mandi et al. [103] investigates the application of the same SPO approach to NP-Hard combinatorial problems. Primary among the challenges faced in this context is, as in [53], the computational cost of solving hard problems within every iteration of the training. The authors found that continuous relaxations of common MILP problems (e.g. knapsack) often offer subgradients of comaparable quality to the full mixed-integer problem with respect to the SPO loss, so that training end-to-end systems with hard CO problems can be simplified in such cases by replacing the full problem with an efficiently solvable relaxation, an approach termed *SPO-relax*. The authors put continuous relaxations into the broader category of "weaker oracles" for the CO solver, which also includes approximation algorithms (e.g. greedy approximation for knapsack). The main results showed that SPO-relax achieves accuracy competitive with the full SPO approach but with shorter training times on a handful of discrete problems. The SPO-relax approach was compared also against the formulation of Wilder et al. [153] on equivalent relaxations, but no clear winner was determined.

Pogančić et al. [123] introduced a new idea for approximating gradients over discrete optimization problems for end-to-end training, which relies on viewing a discrete optimization problem as a function of its defining parameters (in this context coming from previous layers), whose range is piecewise constant. The only requirement is that the objective be linear. The gradient calculation combines the outputs of two calls to an optimization solver; one in the forward pass on initial parameters $\boldsymbol{y}$, and one in the backward pass on perturbed parameters $\bar{\boldsymbol{y}}$. The results are used to construct a piecewise *linear* function which approximates the original solver's output space, but has readily available gradients. Because the gradient calculation is agnostic to the implementation of the solver, it is termed "black-box differentiation". As such, input parameters to the solver do not correspond explicitly to coefficients in the underlying optimization problem. Results on end-to-end learning for the shortest path problem, TSP and min-cost perfect matching are shown. In each case, the discrete problem's specification is implicitly defined in terms of images, which are used to predict parameters of the appropriate discrete problem through deep networks. The optimal solutions coming from blackbox solvers are expressed as binary indicator matrices in each case and matched to targeted optimal solutions using a Hamming distance loss function.

Finally, Wang et al. [152] presented a differentiable solver for the MAXSAT problem, another problem form capable of representing NP-Hard combinatorial problems. Approximate gradients are formed by first approximating the MAXSAT problem as a related semidefinite program (SPD), then differentiating its solution analytically during a specialized coordinate descent method [151] which solves the SDP. The successful integration of MAXSAT into deep learning is demonstrated with a model trained to solve sudoku puzzles represented only by handwritten images.

Chapter 3

Learning to Accelerate the Solution of Optimization Problems

This chapter is dedicated to the original contributions of this thesis in the scope of Learning to Optimize, and is divided in two sections. The first main section addresses the topic of data generation for supervision labels of learned optimizers. It is based on the author's published work [88]. The second section is dedicated to developing a novel framework in which non-Euclidean metrics are learned as a means of accelerating the convergence of operator splitting methods. It is also based on published work, found in [83].

3.1 Data Generation for Learning to Optimize

Two classes of hard optimization problems of particular interest in many fields are (1) *combinatorial optimization problems* and (2) *nonlinear constrained problems*. Combinatorial optimization problems are characterized by discrete search spaces and have solutions that are combinatorial in nature, involving for instance, the selection of subsets or permutations, and the sequencing or scheduling of tasks. Nonlinear constrained problems may have continuous search spaces but are often characterized by highly nonlinear constraints, such as those arising in electrical power systems whose applications must capture physical laws such as Ohm's law and Kirchhoff's law in addition to engineering operational constraints. Such CO problems are often NP-Hard and may be computationally challenging in practice, especially for large-scale instances.

While the AI and Operational Research communities have contributed fundamental advances in optimization in the last decades, the complexity of some problems often prevents them from being adopted in contexts where many instances must be solved over a long-term horizon (e.g., multi-year planning studies) or when solutions must be produced under time constraints. Fortunately, in many practical cases, including the scheduling and energy problems motivating this work, one is interested in solving many problem instances sharing similar patterns. Therefore, the application of deep learning methods to aid the solving of computationally challenging constrained optimization problems appears to be a natural approach and has gained traction in the nascent area at the intersection between CO and ML [17, 89, 147]. In particular, supervised learning frameworks can train a model using pre-solved CO instances and their solutions. However, learning the underlying combinatorial structure of the problem or learning approximations of optimization problems with

hard physical and engineering constraints may be an extremely difficult task. While much of the recent research at the intersection of CO and ML has focused on learning good CO approximations in jointly training prediction and optimization models [12, 78, 111, 116, 148] and incorporating optimization algorithms into differentiable systems [6, 123, 153, 103], learning the combinatorial structure of CO problems remains an elusive task.

Beside the difficulty of handling hard constraints, which will almost always exhibit some violations, two interesting challenges have emerged: the presence of multiple, often symmetric, solutions, and the learning of approximate solution methods. The first challenge recognizes that an optimization problem may not have a unique solution. This challenge is illustrated in Figure 3.1, where the various $\boldsymbol{y}^{(i)}$ represent *optimal* solutions to CO instances $\boldsymbol{x}^{(i)}$ and C the feasible space. As a result, a combinatorial number of possible datasets may be generated. While equally valid as optimal solutions, some sets follow patterns which are more meaningful and recognizable. Symmetry breaking is of course a major area of combinatorial optimization and may alleviate some of these issues. But different instances may not break symmetries in the same fashion, thus creating datasets that are harder to learn.

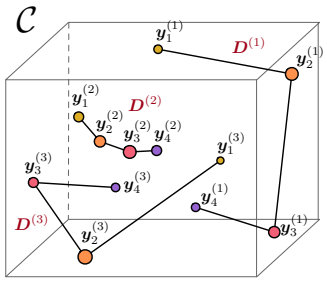


Figure 3.1: Co-optimal datasets due to symmetries.

The second challenge comes from realities in the application domain. Because of time constraints, the solution technique may return a sub-optimal solution. Moreover, modern combinatorial optimization techniques often use randomization and large neighborhood search to produce high-quality solutions quickly. Although these are widely successful, different runs for the same, or similar, instances may produce radically different solutions. As a result, learning the solutions returned by these approximations may be inherently more difficult. These effects may be viewed as a source of noise that obscures the relationships between training data and their target outputs. Although this does not raise issues for optimization systems, it creates challenging learning tasks.

This section demonstrates these relations, connects the variation in the training data to the ability of a model to approximate it, and proposes a method for producing (exact or approximate) solutions to optimization problems that are more amenable to supervised learning tasks. More concretely, we make the following contributions:

1. It shows that the existence of co-optimal or approximated solutions obtained by solving hard CO problems to construct training datasets challenges the learnability of the task.
2. To overcome this limitation, it introduces the problem of optimal dataset design, which is cast as a bilevel optimization problem. The optimal dataset design problem is motivated using theoretical insights on the approximation of functions by neural networks, relating the properties of a function describing a training dataset to the model capacity required to represent it.
3. It introduces a tractable algorithm for the generation of datasets that are amenable to learning, and empirical demonstration of marked improvements to the accuracy of trained models, as well as the ability to satisfy constraints at inference time.
4. Finally, it provides state-of-the-art accuracy results at vastly enhanced computational runtime on learning two challenging optimization problems: Job Shop Scheduling problems and Optimal Power Flow problems for energy networks.

To the best of the authors' knowledge, this work is the first to highlight the issue of learnability

in the face of co-optimal or approximate solutions obtained to generate training data for learning to approximate hard CO problems.

3.1.1 Preliminaries

A constrained optimization (CO) problem poses the task of minimizing an *objective function* $f : \mathcal{Y} \times \mathcal{X} \to \mathbb{R}_+$ of one or more variables $\boldsymbol{y} \in \mathcal{Y} \subseteq \mathbb{R}^n$, subject to the condition that a set of *constraints* $C_{\boldsymbol{x}}$ are satisfied between the variables and where $\boldsymbol{x} \in \mathcal{X} \subseteq \mathbb{R}^m$ denotes a vector of input data that specifies the problem instance:

$$O(\boldsymbol{x}) = \underset{\boldsymbol{y}}{\operatorname{argmin}}\, f(\boldsymbol{y}, \boldsymbol{x}) \;\; \text{subject to:} \;\; \boldsymbol{y} \in C_{\boldsymbol{x}}. \tag{3.1}$$

An assignment of values $\boldsymbol{y}$ which satisfies $C_{\boldsymbol{x}}$ is called a *feasible solution*; if, additionally $f(\boldsymbol{y}, \boldsymbol{x}) \leq f(\boldsymbol{w}, \boldsymbol{x})$ for all feasible $\boldsymbol{w}$, it is called an *optimal solution*.

A particularly common constraint set arising in practical problems takes the form $C = \{\boldsymbol{y} : \boldsymbol{A}\boldsymbol{y} \leq \boldsymbol{b}\}$, where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$. In this case, C is a convex set. If the objective f is an affine function, the problem is referred to as *linear program* (LP). If, in addition, some subset of a problem's variables are required to take integer values, it is called *mixed integer program* (MIP). While LPs with convex objectives belong to the class of convex problems, and can be solved efficiently with strong theoretical guarantees on the existence and uniqueness of solutions [24], the introduction of integral constraints ($\boldsymbol{y} \in \mathbb{N}^n$) results in a much more difficult problem. The feasible set in MIP consists of distinct points in $\boldsymbol{y} \in \mathbb{R}^n$, not only nonconvex but also disjoint, and the resulting problem is, in general, NP-Hard. Finally, nonlinear programs (NLPs) are optimization problems where some of the constraints or the objective function are nonlinear. Many NLPs are nonconvex and can not be efficiently solved [115].

The methodology introduced in this section is illustrated on hard MIP and nonlinear program instances.

3.1.2 Problem setting and goals

This section focuses on learning approximate solutions to problem (3.1) via supervised learning. The task considers datasets $\chi = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^N$ consisting of N data points with $\boldsymbol{x}^{(i)} \in \mathcal{X}$ being a vector of input data, as defined in equation (3.1), and $\boldsymbol{y}^{(i)} \in O(\boldsymbol{x}^{(i)})$ being a solution of the optimization task. A desirable, but not always achievable, property is for the solutions $\boldsymbol{y}^{(i)}$ to be optimal.

The goal is to learn a model $f_\theta : \mathcal{X} \to \mathcal{Y}$, where θ is a vector of real-valued parameters, and whose quality is measured in terms of a nonnegative, and assumed differentiable, *loss function* $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$. The learning task minimizes the empirical risk function (ERM):

$$\min_\theta J(f_\theta; \chi) = \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(\boldsymbol{x}^{(i)}), \boldsymbol{y}^{(i)}), \tag{3.2}$$

with the desired goal that the predictions also satisfy the problem constraints: $f_\theta(\boldsymbol{x}^{(i)}) \in C_{\boldsymbol{x}}$.

While the above task is difficult to achieve due to the presence of constraints, the section adopts a Lagrangian approach [54], which has been shown successful in learning constrained representations, along with projection operators, commonly applied in constrained optimization to

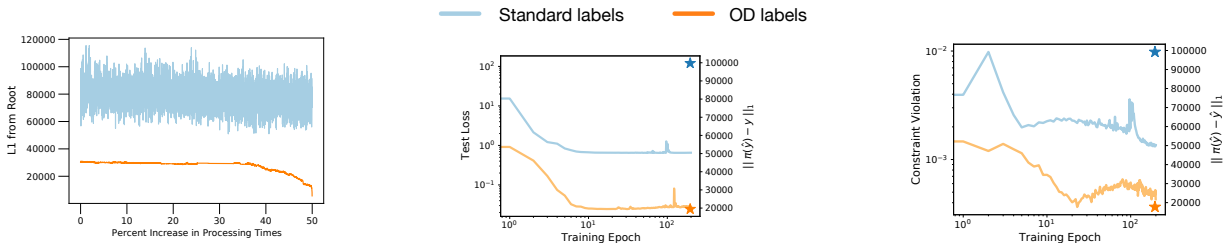


Figure 3.2: Dataset solution $\boldsymbol{y}^{(i)}$ comparison: L1 Distance from a reference solution (left); Test loss (center); and Constraint violations (right).

ensure constraint satisfaction of an assignment. For a given point $\hat{\boldsymbol{y}}$, e.g., representing the model prediction, a projection operator $\pi_C(\hat{\boldsymbol{y}})$ finds the closest feasible point $\boldsymbol{y} \in C$ to $\hat{\boldsymbol{y}}$ under a p-norm:

$$\pi_C(\hat{\boldsymbol{y}}) \stackrel{\text{def}}{=} \underset{\boldsymbol{y}}{\operatorname{argmin}} \|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_p \quad \text{subject to:} \quad \boldsymbol{y} \in C.$$

3.1.3 Challenges in learning hard combinatorial problems

One of the challenges arising in this area comes from the recognition that a problem instance may admit a variety of disparate optimal solutions for each input $\boldsymbol{x}$. To illustrate this challenge, the section uses a set of scheduling instances that differ only in the time required to process tasks on some machine. A standard approach to the generation of dataset in this context would consist in solving each instance independently using some SoTA optimization solver. However, this may create some significant issues that are illustrated in Figure 3.2 (more details on the problem are provided in Section 3.1.6). The blue curve in Figure 3.2 (left) illustrates the behavior of this natural approach. In the figure, the processing times in the instances increase from left to right and the blue curve represents the L_1-distance between the obtained solution to each instance (i.e., the start times of the tasks) and a reference optimal solution for some instance. The volatile curve shows that meaningful patterns can be lost, including the important relationship between an increase in processing times and the resulting solutions. Figure 3.2 (center) shows that, while the solution patterns induced by the target labels appear volatile, the ERM problem appears well behaved, in the face of minimizing the test loss. However, when training loss converges, accuracy (measured as the distance between the projection of the prediction $\pi_C(\hat{\boldsymbol{y}})$ and the real label $\boldsymbol{y}$) remains poor in models trained on such data (blue star). Figure 3.2 (right) shows the average magnitude of the constraints violation during training, corresponding to the two target solution sets of Figure 3.2 (left), along with a comparison of the objective of the projection operator applied to the prediction: $\|\pi_C(\hat{\boldsymbol{y}}) - \hat{\boldsymbol{y}}\|$. It is worth emphasizing that these issues are further exacerbated when time constraints prevent the solver from obtaining optimal solutions. Moreover, similar patterns can also be observed for the data generated while solving optimal power flow instances that exhibit

Additionally, observations collected on motivating applications of this work show that, even when the model complexity (i.e., the dimensionality of the model parameters θ) is increased arbitrarily, the resulting learned models tend to have low-variance. This is illustrated in Figure 3.3, where the orange and blue curves depict, respectively, a function interpolating the training labels and the associated learned

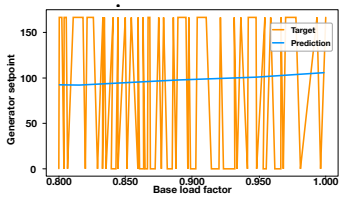


Figure 3.3: Approximating highly volatile function results in low-variance models.

solutions.

The goal of this work is to construct datasets that are well-suited
for learning the optimal (or near-optimal) solutions to optimization problems. The benefit of such an approach is illustrated by the orange curves and stars in Figure 3.2, which were obtained using the data generation methodology proposed in this work. They considered the same instances and obtained (different) optimal solutions, but exhibit much enhanced behavior on all metrics.

3.1.4 Theoretical justification of the data generation

Although the data generation strategy is necessarily heuristic, it relies on key theoretical insights on the nature of optimization problems and the representation capabilities on neural networks. This section reviews these insights.

First, observe that, as illustrated in the motivating Figure 3.3, the solution trajectory associated with the problem instances on various input parameters can often be naturally approximated by piecewise linear functions. This approximation is in fact exact for linear programs when the inputs capture incremental changes to the objective coefficients or the right-hand side of the constraints. Additionally, ReLU neural networks, used in this study to approximate the optimization solutions, have the ability to capture piecewise linear functions [72]. While these models are thus compatible with the task of predicting the solutions of an optimization problem, the model capacity required to represent a target piecewise linear function exactly depends directly on the number of constituent pieces.

Theorem 1 (Model Capacity [9])**.** *Let $f : \mathbb{R}^d \to \mathbb{R}$ be a piecewise linear function with p pieces. If f is represented by a ReLU network with depth $k+1$, then it must have size at least $\frac{1}{2}kp^{\frac{1}{k}} - 1$. Conversely, any piecewise linear function f that is represented by a ReLU network of depth $k+1$ and size at most s, can have at most $\left(\frac{2s}{k}\right)^k$ pieces.*

The solution trajectories may be significantly different depending on how the data is generated. Hence, the more volatile the trajectory, the harder it will be to learn. Moreover, for a network of fixed size, the more volatile the trajectory, the larger the approximation error will be in general. The data generation proposed in this section will aim at generating solution trajectories that are approximated by piecewise linear functions with fewer distinct pieces. The following theorem bounds the approximation error when using continuous piecewise linear functions: it connects the approximation errors of a piecewise linear function with the *total variation in its slopes*.

Theorem 2. *Suppose a piecewise linear function $f_{p'}$, with p' pieces each of width h_k for $k \in [p']$, is used to approximate a piecewise linear f_p with p pieces, where $p' \leq p$. Then the approximation error*

$$\|f_p - f_{p'}\|_1 \leq \frac{1}{2} h_{\max}^2 \sum_{1 \leq k \leq p} |L_{k+1} - L_k|,$$

holds where L_k is the slope of f_p on piece k and $h_{\max}$ is the maximum width of all pieces.

Proof. Firstly, the proof proceeds with considering the special case in which f_p conincides in slope and value with $f_{p'}$ at some point, and that each piece of $f_{p'}$ overlaps with at most 2 distinct pieces of

f_p. This is always possible when $p' \geq \frac{p}{2}$. Call I_k the interval on which $f_{p'}$ is defined by its k^{th} piece. If I_k overlaps with only one piece of f_p, then for $x \in I_k$,

$$|f_p(x) - f_{p'}(x)| = 0 \tag{3.3}$$

If I_k overlaps with pieces k and $k+1$ of f_p, then for $x \in I_k$,

$$|f_p(x) - f_{p'}(x)| \leq h_k |L_{k+1} - L_k| \tag{3.4}$$

Each of the above follows from the assumption that f_p and $f_{p'}$ are equal in their slope and value at some point within I_k. From this it follows that on I_k,

$$\|f_p - f_{p'}\|_1 = \int_{I_k} |f_p - f_{p'}| \leq \frac{1}{2} \sum_{1 \leq i \leq p} h_k^2 |L_{k+1} - L_k| \leq \frac{1}{2} h_{\max}^2 \sum_{1 \leq k \leq p} |L_{k+1} - L_k|, \tag{3.5}$$

so that on the entire domain of f_p and $f_{p'}$,

$$\|f_p - f_{p'}\|_1 \leq \frac{1}{2} h_{\max}^2 \sum_{1 \leq k \leq p} |L_{k+1} - L_k|. \tag{3.6}$$

Since removing the initial simplifying assumptions tightens this upper bound, the result holds. □

The final observation that justifies the proposed approach is the fact that optimization problems often satisfy a local Lipschitz condition, i.e., if the inputs of two instances are close, then they admit solutions that are close as well, i.e., there exist $\boldsymbol{y}^{\star(i)} \in O(\boldsymbol{x}^{(i)})$ and $\boldsymbol{y}^{\star(j)} \in O(\boldsymbol{x}^{(j)})$, where

$$\| \boldsymbol{y}^{\star(i)} - \boldsymbol{y}^{\star(j)} \| \leq C \|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|, \tag{3.7}$$

for some $C \geq 0$ and $\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\| \leq \epsilon$, where ϵ is a small value. This is obviously true in linear programming when the inputs vary in the objective coefficients or the right-hand side of the constraints, but it also holds locally for many other types of optimization problems. That observation suggests that, when this local Lipschitz condition holds, it may be possible to generate solution trajectories that are well-behaved and can be approximated effectively. Note that Lipschitz functions can be nicely approximated by neural networks as the following result indicates.

Theorem 3 (Approximation [31])**.** *If $f : [0,1]^n \to \mathbb{R}$ is L-Lipschitz continuous, then for every $\epsilon > 0$, there exists some single-layer neural network ρ of size N such that $\|f - \rho\|_\infty < \epsilon$, where $N = \binom{n + \frac{3L}{\epsilon}}{n}$.*

The result above illustrates that the model capacity required to approximate a given function depends to a non-negligible extent on the Lipschitz constant value of the underlying function.

Note that the results in this section are bounds on the ability of neural networks to represent generic functions. In practice, these bounds themselves rarely guarantee the training of good approximators, as the ability to minimize the empirical risk problem in practice is often another significant source of error. In light of these results however, it is to be expected that datasets which exhibit less variance and have small Lipschitz constants will be better suited to learning good function approximations. The following section presents a method for dataset generation motivated by these considerations.

3.1.5 Optimal CO training data design

Given a set of input data $\{\boldsymbol{x}i\}_{i=1}^{N}$, the goal is to construct the associated pairs $\boldsymbol{y}i$ for each $i \in [N]$, that solve the following problem

$$\min_{\theta, \boldsymbol{y}^{(i)}} \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(\boldsymbol{x}i), \boldsymbol{y}i) \tag{3.8a}$$

$$\text{subject to : } \boldsymbol{y}i \in \operatorname*{argmin}_{\boldsymbol{y} \in C_{\boldsymbol{x}i}} f(\boldsymbol{y}, \boldsymbol{x}i). \tag{3.8b}$$

One often equips the data point set $\{\boldsymbol{x}i\}_{i=1}^{N}$ with an ordering relation $\preceq$ such that $\boldsymbol{x} \preceq \boldsymbol{x}' \Rightarrow \|\boldsymbol{x}\|_p \leq \|\boldsymbol{x}'\|_p$ for some p-norm. For example, in the scheduling domain, the data points $\boldsymbol{x}$ represent task start times and the training data are often generated by "slowing down" some machine, which simulates some unexpected ill-functioning component in the scheduling pipeline. In the energy domain, $\boldsymbol{x}$ represent the load demands and the training data are generated by increasing or decreasing these demands, simulating the different power load requests during daily operations in a power network. For simplicity, we assume the existence of such a useful ordering over the entire set of instances in a learning task.

From the space of co-optimal solutions $\boldsymbol{y}i$ to each problem instance $\boldsymbol{x}i$, the goal is to generate solutions which coincide, to the extent possible, with a target function of low total variation and Lipschitz factor, as well as a low number of constituent linear pieces in the case of discrete optimization. While it may not be possible to produce a target set that simultaneously optimizes each of these metrics, they are confluent and can be improved simultaneously. Natural heuristics are available which reduce these metrics substantially when compared with naive approaches.

One heuristic aimed at satisfying the aforementioned properties reduces to the problem of determining a solution set $\{\boldsymbol{y}i\}_{i=1}^{N}$ for the inputs $\{\boldsymbol{x}i\}_{i=1}^{N}$ of problem (3.1) that minimizes their total variation:

$$\textit{minimize} \quad TV\left(\{\boldsymbol{y}^{(i)}\}_{i=1}^{N}\right) = \frac{1}{2} \sum_{i=1}^{N-1} \|\boldsymbol{y}^{(i+1)} - \boldsymbol{y}^{(i)}\|_p \tag{3.9a}$$

$$\text{subject to : } \boldsymbol{y}^{(i)} = \operatorname*{argmin}_{\boldsymbol{y} \in C_{\boldsymbol{x}^{(i)}}} f(\boldsymbol{y}, \boldsymbol{x}^{(i)}). \tag{3.9b}$$

Algorithm 1: *Opt. Data Generation*

input : $\{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$: Input data

$\boldsymbol{y}^{(N)} \leftarrow \overset{\star}{\boldsymbol{y}}{}^{(N)} \in \tilde{O}(\boldsymbol{x}^{(N)})$

for $i = N-1$ *down to* 1 **do**

  $\boldsymbol{y}^{(i)} \leftarrow \overset{\star}{\boldsymbol{y}}{}^{(i)} \in \tilde{O}(\boldsymbol{x}^{(i)})$

  $$\boldsymbol{y}^{(i)} \in \begin{cases} \operatorname{argmin}_{\boldsymbol{y}} \quad \|\boldsymbol{y} - \boldsymbol{y}^{(i+1)}\|_p \\ \text{subject to: } \boldsymbol{y} \in C_{\boldsymbol{x}^{(i)}} \\ \qquad\qquad f(\boldsymbol{y}) \leq f(\overset{\star}{\boldsymbol{y}}{}^{(i)}) \end{cases}$$

return $\chi = \left\{\left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right)\right\}_{i=1}^{N}$

In practice, this bi-level minimization cannot be achieved, due partially to its prohibitive size. It is possible, however, to minimize the individual terms of (3.9a), each subject to the result of

the previous, by solving individual instances sequentially. Algorithm 1 ensures that solutions to subsequent instances have minimal distance with respect to the chosen p-norm (the experiments of Section 3.1.6 use $p = 1$). This method approximates a set of solutions with minimal total variation, while ensuring that the maximum magnitude of change between subsequent instances is also small. When the data represent the result of a discrete optimization, this coincides naturally with a representative function which requires less pieces, with less extreme changes in slope. The method starts by solving a target instance, e.g., the last in the given ordering $\preceq$ (line 1). Therein, $\tilde{O}$ denotes the solution set of a (possibly approximated) minimizer for problem (3.1). In the case of the job shop scheduling, for example, $\tilde{O}$ represents a local optimizer with a suitable time-limit. The process then generates the next dataset instance $\boldsymbol{y}^{(i)}$ in the ordering $\preceq$ by solving the optimization problem given in line 3. The problem finds a solution to problem $\boldsymbol{x}^{(i)}$ that is close to adjacent solution $\boldsymbol{y}^{(i+1)}$ while preserving *optimality*, i.e., the objective of the sought $\boldsymbol{y}$ is constrained to be at most that of $\boldsymbol{\tilde{y}}^{(i)} \in \tilde{O}(\boldsymbol{x}^{(i)})$.

When the difficulty of the underlying optimization instances makes the sequential solving of Algorithm 1 impractical, structural properties of the CO problem may be exploited to increase efficiency, i.e., by the use of warm-starts or solution-guided search. In the Job Shop Scheduling case study, $\boldsymbol{\tilde{y}}^{(i+1)}$ is feasible to the subsequent problem and may be used to warm-start its solution, carrying forward solution progress between iterations of Algorithm 1. Therefore, a byproduct of this data-generation approach is that the optimization problem in line 3 can be well-approximated within a short timeout. When such exploits are not available, the secondary optimization of line 4 may be applied over independently pre-solved instances in an analogous way.

In addition to providing enhanced efficacy for learning, this method of generating target instances is generally preferable from a modeling point of view. When predicted solutions to related decision problems are close together, the resulting small changes are often more practical and actionable, and thus highly preferred in practice. For example, a small change in power demands should result in an updated optimal power network configuration which is easy to achieve given its previous state.

3.1.6 Application to case studies

The concepts introduced above are applied in this section to two representative case studies, *Job Shop Scheduling* (JSS) and *Optimal Power Flow* (OPF). Both are of interest in the optimization and machine learning communities as practical problems which must be routinely solved, but are difficult to approximate under stringent time constraints. The JSS problem represents the class of combinatorial problems, while the OPF problem is continuous but nonlinear and non convex.

Job shop scheduling

Job Shop Scheduling (JSS) assumes a set of J jobs, each consisting of a list of M tasks to be completed in a specified order. Each task has a fixed processing time and is assigned to one of M machines, so that each job assigns one task to each machine. The objective is to find a schedule with minimal *makespan*, or time taken to process all tasks. The *no-overlap* condition requires that for any two tasks assigned to the same machine, one must be complete before the other begins. See the problem specification (A.5). The objective of the learning task is to predict the start times of all tasks given a JSS problem specification (task duration, machine assignments).

Instance	Size	Prediction Error		Constraint Violation		Optimality Gap (%)		Time SoTA Eq. (s)	
	$J \times M$	Standard	OD	Standard	OD	Standard	OD	Standard	OD
ta25	20×20	193.9	**23.4**	180.0	**45.5**	10.3	**4.0**	24	**550**
yn02	20×20	153.2	**38.9**	124.9	**70.3**	9.1	**4.5**	27	**45**
swv03	20×10	309.4	**12.4**	206.9	**31.6**	18.0	**2.2**	15	**65**
swv07	20×15	330.4	**19.9**	280.1	**67.2**	17.0	**3.0**	15	**60**
swv11	50×10	1090.0	**51.2**	906.4	**151.7**	28.5	**4.5**	13	**100**

Table 3.1: Standard vs OD training data: prediction errors, constraint violations, and optimality gap (the smaller the better), Time SoTA Eq. (the larger the better). Best results are highlight in bold.

Data Generation Algorithms The experiments examine the proposed models on a variety of problems from the JSPLIB library [138]. The ground truth data are constructed as follows: different input data $\boldsymbol{x}^{(i)}$ are generated by simulating a machine slowdown, i.e., by altering the time required to process the tasks on that machine by a constant amount which depends on the instance i. Each training dataset associated with a JSS benchmark is composed of a total of 5000 instances. Increasing the processing time of selected tasks may also change the difficulty of the scheduling. The method of sequential solving outlined in Section 3.1.5 is particularly well-suited to this context. Individual problem instances can be ordered relative to the amount of extension applied to those processing times, so that when $d_{jt}^{(i)}$ represents the time required to process task t of job j in instance i, $d_{jt}^{(i)} \leq d_{jt}^{(i+1)} \quad \forall j, t$. In this case, any solution to instance $d_{(i+1)}$ is feasible to instance d_i (tasks in a feasible schedule cannot overlap when their processing times are reduced, and start times are held constant). As such, the method can be made efficient by passing the solution between subsequent instances as a warm-start.

The analysis compares two datasets: One consisting of target solutions generated independently with a solving time limit of 1800 seconds using the state-of-the-art IBM CP Optimizer constraint programming software (denoted as Standard), and one whose targets are generated according to algorithm 1, called the Optimal Design dataset (denoted as OD).

Figure 3.4 presents a comparison of the total variation resulting from the two datasets. Note that the OD datasets have total variation which is orders of magnitude lower than their Standard counterparts. Recall that a small total variation is perceived as a notion of well-behaveness from the perspective of function approximation. Additionally, it is noted that the total computation time required to generate the OD dataset is at least an order of magnitude smaller than that required to generate the standard dataset (13.2h vs. 280h).

Instance	Size	Total Variation ($\times 10^6$)	
	$J \times M$	Standard Data	OD Data
ta25	20×20	67.8	**0.194**
yn02	20×20	55.0	**0.483**
swv03	20×10	109.4	**0.424**
swv07	20×15	351.2	**0.100**
swv11	50×10	352.0	**1.376**

Figure 3.4: Standard vs OD training data: Total Variation.

Prediction Errors and Constraint Violations Table 3.1 reports the prediction errors as L_1-distance between the (feasible) predicted variables, i.e., the projections $\pi(\hat{\boldsymbol{y}})$ and their original ground-truth quantities ($\boldsymbol{y}$), the average constraint violation degrees, expressed as the L_1-distance between the predictions and their projections, and the optimality gap, which is the relative difference in makespan (or, equivalently objective values) between the predicted (feasible) schedules and target schedules. All these metrics are averaged over all perturbed instances of the dataset and expressed in percentage. In the case of the former two metrics, values are reported as a percentage of the average

task duration per individual instance. Notice that for all metrics the methods trained using the OD datasets result in drastic improvements (i.e., one order of magnitude) with respect to the baseline method. Additionally, Table 3.1 (last column) reports the runtime required by CP-Optimizer to find a value with the same makespan as the one reported by the projected predictions (projection times are also included). The values are to be read as the larger the better, and present a remarkable improvement over the baseline method. It is also noted that the worst average time required to obtain a feasible solution from the predictions is 0.02 seconds. Additional experiments, reported in Appendix A.4 also show that the observations are robust over a wide range of hyper-parameters adopted to train the learning models.

The results show that the OD data generation can drastically improve predictions qualities while reducing the effort required by a projection step to satisfy the problem constraints.

AC Optimal Power Flow

Optimal Power Flow (OPF) is the problem of finding the best generator dispatch to meet the demands in a power network. The OPF is solved frequently in transmission systems around the world and is increasingly difficult due to intermittent renewable energy sources. The problem is required to satisfy the AC power flow equations, that are non-convex and nonlinear, and are a core building block in many power system applications. The objective function captures the cost of the generator dispatch, and the Constraint set describes the power flow operational constraints, enforcing generator output, line flow limits, Kirchhoff's Current Law and Ohm's Law for a given load demand. The OPF receives its input from unit-commitment algorithms that specify which generators will be committed to deliver energy and reserves during the 24 hours of the next day. Because many generators are similar in nature (e.g., wind farms or solar farms connected to the same bus), the problem may have a large number of symmetries. If a bus has two symmetric generators with enough generator capacities, the unit commitment optimization may decide to choose one of the symmetric generators or to commit both and balance the generation between both of them. The objective of the learning task is to predict the generator setpoints (power and voltage) for all buses given the problem inputs (load demands).

Data Generation Algorithms The experiments compare this commitment strategy and its effect on learning on Pegase-89, which is a coarse aggregation of the French system and IEEE-118 and IEEE-300, from the NESTA library [32]. All base instances are solved using the Julia package PowerModels.jl [34] with the nonlinear solver IPOPT [149]. Additional data is reported in Appendix A.3. A number of renewable generators are duplicated at each node to disaggregate the generation capabilities. The test cases vary the load data by scaling the (input) loads from 0.8 to 1.0 times their nominal values. Instances with higher load pattern are typically infeasible. The unit-commitment strategy sketched above can select any of the symmetric generators at a given bus (Standard data). The optimal objective values for a given load are the same, but the optimal solutions vary substantially. Note that, when the unit-commitment algorithm commits generators by removing symmetries (OD data), the solutions for are typically close to each other when the loads are close. As a result, they naturally correspond to the generation procedure proposed in this section.

Prediction Errors and Constraint Violations As shown in Table 3.2, the OD approach to data generation results in predictions that are closer to their optimal target solutions (error expressed in MegaWatt (MW)), reduce the constraint violations (expressed as L_1-distance between the predictions

Instance	Size	Prediction Error		Constraint Violation		Optimality Gap (%)	
	No. buses	Standard	OD	Standard	OD	Standard	OD
Pegase-89	89	198.3	**5.02**	1.21	**1.01**	20.1	**7.0**
IEEE-118	118	172.5	**13.31**	2.12	**1.98**	32.5	**9.6**
IEEE-300	300	194.2	**24.12**	2.63	**2.04**	44.2	**15.2**

Table 3.2: Standard vs OD training data: prediction errors, constraint violations, and optimality gap.

and their projections), and improve the optimality gap, which is the relative difference in objectives between the predicted (feasible) solutions and the target ones.

3.1.7 Limitations and Conclusions

This section was motivated by engineering applications where problem instances share an underlying "infrastructure" (e.g., the power grid in optimal power flow or the manufacturing floor in job shop scheduling) which is stable and does not evolve too rapidly. These problems are pervasive in applications ranging from supply chains and logistics, to electricity grids, and manufacturing, to name a few. Note that the approach to data generation presented in this section is not intended for direct application to every task of learning to solve constrained optimization. In general, data generation approaches should be tailored to exploit the properties of their respective optimization problems and experimental settings, but the algorithms and insights demonstrated above may be exploited in a variety of settings.

While the proposed methodology is limited to machine learning tasks in which optimization problem instances admit useful orderings with respect to their parameters, this property may hold only locally over a particular distribution of problems. For example, in timetabling applications, as in the design of employees shifts, a desired condition may be for shifts to be diverse for different but similar inputs. The proposed methodology may, in fact, induce a learner to predict similar solutions across similar input data. A possible solution to this problem may be that of generating various *trajectories* of solutions, learn from them with independent models, and then randomize the model selection to generate a prediction. These more complex settings may require composition or extension of the solutions presented in this section, and generalizations of the approach are an avenue for future work.

3.2 Learning Metrics to Accelerate Operator-Splitting Methods

A substantial literature has been dedicated to the use of machine learning to aid in the fast solution of constrained optimization problems. This interest is driven by an increasing need for real-time decision-making capabilities, in which decision processes modeled by optimization problems must be resolved faster than can be met by traditional optimization methods. Such capabilities are of interest in various application settings such as job scheduling in manufacturing [90, 88], power grid operation [54], and optimal control [132].

A prominent application of machine learning in accelerating optimization is to learn the parameters of a standard solution algorithm, such that iterations to convergence are minimized. Examples include gradient stepsizes [8], and initial solution estimates [132]. This section proposes an alternative approach by parametrizing the underlying metric space of an optimization algorithm which relies on *proximal operators*. Proximal operators, which include projections, are based on a notion of distance within a metric space and employed in many practical optimization methods. While most methods that employ proximal algorithms are typically based on the standard Euclidean metric, it is well-known that many such methods are also guaranteed to converge for non-Euclidean metrics defined as general quadratic forms over the continuous space of positive definite matrices [15].

The possibility of accelerating convergence by selecting non-Euclidean metrics within that space has been noted [64], but no known method has shown to be effective over a general class of optimization problems. For limited classes of problems, *optimal* metric choices have been modeled as the solution to an auxiliary optimization problem. But for many problems including general quadratic programming (QP) problems, such models are yet unknown [65]. Theoretical insights have been used to suggest *heuristic* metric choices for QP problems [64, 65], but the potential for improvement over these heuristic rules has not been fully explored.

This section proposes differentiable programming to both explore the potential, and to overcome the challenges of metric selection for more general classes of optimization problems. Specifically, we propose a system of end-to-end learning for proximal optimization, which trains machine learning models to predict metrics that empirically minimize solution error over a prescribed number of iterations on a given problem instance. Enhanced convergence of two proximal optimization methods is demonstrated on Quadratic Programming (QP) problems, including test cases where theoretically prescribed heuristic metric choices perform poorly.

We demonstrate that while prior heuristic models of optimal metric selection can *fail* in the presence of active constraints at the optimal solution, our learned metrics are *correlated* with the active constraints at optima, and can accelerate convergence by ignoring the inactive constraints. This leads to an interpretation of metric selection as a problem which incorporates active set prediction, whose difficulty may approach that of solving the optimization problem itself. The proposed integration of optimization and learning thus shows advantages in both accuracy and efficiency over theoretical approaches to metric selection in proximal optimization.

3.2.1 Related Work

This section's topic is at the intersection of learning to accelerate optimization, and metric selection in proximal optimization. Before proceeding to the main contributions, related work is summarized with respect to both areas.

Learning to Accelerate Optimization Various systems for learning fast solutions to optimization problems have been proposed. For example, several works have shown how to learn heuristics such as branching rules [12, 67, 78] and cutting planes [122] in mixed-integer programming. An early survey [17] provides a comprehensive summary of machine learning in combinatorial optimization. Further surveys on learning to branch [101] and learning to cut [39] provide even more detail on the topic. To enhance the resolution of optimization problems with continuous variables, several works have also considered simplifying an optimization problem by first learning its active constraints [21, 108]. An altogether different paradigm aims to train deep neural networks to produce solutions to optimization problem directly. For example, several works consider end-to-end learning of solutions to combinatorial problems [16, 80, 85, 148]. Other works have shown how to learn solutions to problems with general nonlinear constraints, either by leveraging Lagrangian duality [57, 90, 119], differentiable constraint corrections [48], or reparametrization of the feasible space [84]. Another closely related direction [132] focuses on learning warm-starts to proximal algorithms for quadratic programming.

Metric Selection in Proximal Optimization The potential for accelerating the convergence of a proximal algorithm by optimizing its underlying proximal metric has been theoretically demonstrated in previous works. The authors in [65] derived the optimal choice of metric for ADMM and Douglas-Rachford splitting algorithms on a limited class of problems. Based on this result, they also suggested heuristic methods for selecting an appropriate metric for problems outside of that class. Similar results were shown in [64] for a fast dual forward-backward splitting method. Unfortunately, these theoretical results do not extend to many problems of practical interest, including generic Quadratic Programming (QP) problems. Furthermore, when the optimal metric can be computed, it typically requires solution of a difficult semidefinite program, reducing its practical benefit in accelerating the solution of problems. This section demonstrates the use of end-to-end machine learning to derive models whose predicted proximal metrics can outperform the heuristic theory-based models of [65] on several QP problems.

3.2.2 Preliminaries

Let $\mathcal{S}^n_{++}$ be the set of $n \times n$ positive definite matrices. For $M \in \mathcal{S}^n_{++}$ let $\mathbb{R}^n_M$ be the Hilbert space on $\mathbb{R}^n$ with the corresponding inner product and norm defined respectively for all $x, y \in \mathbb{R}^n$ as

$$\langle x, y \rangle_M = x^T M y, \qquad \text{and} \qquad \|x\|_M^2 = x^T M x \,.$$

We will denote by $\Gamma(\mathbb{R}^n_M)$ the set of functions $f : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ that are proper, closed and convex, where $\mathbb{R} \cup \{\infty\}$ represents the extended reals. For $f, g \in \Gamma(\mathbb{R}^n_M)$, *Douglas-Rachford* splitting (DR) considers optimization problems of the form

$$\min_{x \in \mathbb{R}^n} f(x) + g(x) \,, \tag{3.10}$$

and computes optimal solutions by following the iterations

$$y_k = \mathrm{prox}_{\gamma g}(x_k) \,, \tag{3.11a}$$
$$z_k = \mathrm{prox}_{\gamma f}(2y_k - x_k) \,, \tag{3.11b}$$
$$x_{k+1} = x_k + z_k - y_k \,, \tag{3.11c}$$

where the $\text{prox}_\gamma f$ is the *proximal operator* defined with respect to the space $\mathbb{R}^n_M$ as

$$\text{prox}_\gamma f(x) = \underset{z\in\mathbb{R}^n}{\text{argmin}}\, f(z) + \frac{1}{\gamma}\|x - z\|_M^2\,, \tag{3.12}$$

for $\gamma > 0$. It can be shown that if a solution of (3.10) exists then the DR iterations will converge, in particular the sequence $\text{prox}_\gamma g(x_k)$ will converge weakly to a solution and under additional mild assumptions will converge strongly. Various alternative formulations and relaxations of DR exist but here we will primarily restrict consideration to the formulation (3.11); for proofs and additional details see for example [13].

This formulation naturally gives rise to the question of the selection of a positive-definite matrix M to define a metric in (3.12) for a given problem to improve convergence of the iterations (3.11). In [65] Giselsson and Boyd study the optimal metric choice to improve convergence rate for DR applied to the Fenchel dual of (3.10), which can be shown to be equivalent to employing the alternating direction method of multipliers (ADMM) on the primal problem [65, 60, 50]. In this case, any choice of M other than the identity is equivalent to the use of preconditioning in ADMM. A standard formulation for problems to be solved by ADMM is given by

$$\min_{x\in\mathbb{R}^n, y\in\mathbb{R}^m} \quad f(x) + g(y) \tag{3.13a}$$

$$\text{subject to:} \quad Ax + By = c \tag{3.13b}$$

for $f \in \Gamma(\mathbb{R}^n_M)$, $g \in \Gamma(\mathbb{R}^m_I)$, $A \in \mathbb{R}^{m,n}$, $B \in \mathbb{R}^{m,m}$, and $c \in \mathbb{R}^m$. ADMM applied to the preconditioned primal problem

$$\min_{x\in\mathbb{R}^n, y\in\mathbb{R}^m} \quad f(x) + g(y) \tag{3.14a}$$

$$\text{subject to:} \quad MAx + MBy = Mc\,, \tag{3.14b}$$

is equivalent to DR applied to its Fenchel dual when utilizing the metric M. Thus dual DR using metric M can be implemented by the primal ADMM iterations

$$x_{k+1} = \underset{x\in\mathbb{R}^n}{\text{argmin}}\{f(x) + \gamma 2\|M(Ax + By_k - c) + u_k\|_2^2\}\,, \tag{3.15a}$$

$$y_{k+1} = \underset{y}{\text{argmin}}\{g(y) + \frac{\gamma}{2}\|M(Ax_{k+1} + By - c) + u_k\|_2^2\}\,, \tag{3.15b}$$

$$u_{k+1} = u_k + M(Ax_{k+1} + By_{k+1} - c)\,. \tag{3.15c}$$

In [65] the authors show how to calculate the matrix M which optimizes the convergence rate of ADMM applied to (3.14), under the additional assumptions that f is strongly convex and smooth, and that A has full row rank. In such cases, an optimal metric M can be modeled as the solution to a related semidefinite programming problem (SDP). However, those requisite assumptions exclude many practical optimization problems, including general-form quadratic programming (QP) problems. For QP forms outside the scope of these assumptions, the authors of [65] suggest heuristic models of metric selection.

Significant work has been done to improve ADMM convergence using preconditioning, for instance in [63, 18]. As pointed out in [65], in the case of QP problems these methods ultimately amount to reconditioning the quadratic objective function. The same is true of the heuristic method

they propose, which equates to selecting the optimal metric for a related *unconstrained* QP. In this section, we explore the potential for empirically learning metrics to enhance convergence of proximal algorithms on QP problems which do not satisfy the assumptions required for metric optimization presented in [65]. We investigate solution of QP problems using learned metrics with DR applied to both the primal and dual problems, implementing ADMM for solution of the dual as given in (3.15).

3.2.3 Learning Metrics to Accelerate Quadratic Programming

The proposed system for metric learning in proximal optimization leverages a reformulation of general QP problems which renders the metrics easier to learn. In this section, we first introduce the problem reformulation before describing details of the end-to-end learning approach. In brief, a neural network model is trained to predict positive definite matrices M as a function of the parameters which define an optimization problem instance. Solution error after a fixed number of iterations (3.11) or (3.15) is treated as a loss function and minimized, by backpropagation through the solver iterations in stochastic gradient descent training. While the system is general and can in principle be applied to any problem of the form (3.10), the scope of this work is limited to demonstration on QP problems.

Problem Reformulation

Let $Q \in \mathbb{R}^{n,n}$ be a positive semi-definite matrix, $q \in \mathbb{R}^n$, $L \in \mathbb{R}^{m,n}$, $b \in \mathbb{R}^m$, $W \in \mathbb{R}^{k,n}$, and $c \in \mathbb{R}^k$. We consider QP problems of the form

$$\underset{x \in \mathbb{R}^n}{\operatorname{argmin}} \quad \frac{1}{2} x^T Q x + q^T x \tag{3.16a}$$

$$s.t. \quad Lx = b \tag{3.16b}$$

$$Wx + c \leq 0. \tag{3.16c}$$

For implementation with both primal DR and ADMM we introduce slack variables $s \in \mathbb{R}^m$ and reformulate the problem as

$$\underset{z \in \mathbb{R}^{n+k}}{\operatorname{argmin}} \quad \frac{1}{2} z^T I_x^T Q I_x z + q^T I_x z \tag{3.17a}$$

$$s.t. \quad Rz + r = 0 \tag{3.17b}$$

$$I_s z \geq 0, \tag{3.17c}$$

where I^n is the $n \times n$ identity matrix, and

$$z = \begin{bmatrix} x \\ s \end{bmatrix}, \quad R = \begin{bmatrix} L & 0 \\ W & I^k \end{bmatrix}, \quad r = \begin{bmatrix} -b \\ c \end{bmatrix},$$
$$I_s = \begin{bmatrix} 0 & 0 \\ 0 & I^k \end{bmatrix}, \quad I_x = \begin{bmatrix} I^n & 0 \\ 0 & 0 \end{bmatrix},$$

where the zero entries in the matrices are presumed to be of dimension appropriate to make (3.17) coherent.

Note that even if the inequalities (3.16c) represent simple box constraints on the variables x (for example $x < 0$) we still introduce corresponding slack variables. This is done to construct a splitting for both primal DR and ADMM with the intent to increase the impact M can have on convergence, as described below.

QP Splitting For implementation of both primal DR and ADMM on problem (3.17) we use the splitting

$$f(z) = z^T I_x^T Q I_x z + q^T I_x z + i_{\{z \in \mathbb{R}^{n+k} \,:\, Rz+r=0\}}(z) \tag{3.18a}$$

$$g(z) = i_{\{z \in \mathbb{R}^{n+k} \,:\, I_s z \geq 0\}}(z) \tag{3.18b}$$

where for a set $\mathcal{S}$ we define the indicator function on $\mathcal{S}$ to be

$$i_S(x) = \begin{cases} 0 & x \in S \\ \infty & x \notin S \end{cases}.$$

With this splitting we implement the primal DR iteration as given in (3.11), and implement ADMM as in (3.15), with $A = I$, $B = -I$, and $c = 0$. Minimization steps with respect to f can be accomplished with for example the corresponding Karush-Kuhn-Tucker conditions. Minimization steps with respect to g for both algorithms equate to projections onto the positive orthant. Slack variables are initialized at zero throughout.

For both ADMM and DR iterations using the splitting (3.18), the proximal operators as given in (3.11b) and (3.15a) equate to projections onto the relaxed constraint set (3.17b) with respect to the underlying metric. With the slack variables for each inequality constraint initialized at zero, a relatively large corresponding weight in the metric matrix M will bias the non-Euclidean projection to maintain those slacks near zero. Hence if M has relatively large weights for just the slacks corresponding to the active constraints of a problem, the projection can approximate projection onto the active set. Indeed, the learned metrics exhibit this expected behavior as illustrated in Figure 3.6.

End-to-End Learning Framework

As suggested by the results of Section (3.2.4), the optimal metric for solving of an instance of (3.17) can be closely related to the active constraints at its optimal solution. Thus, it is expected that learning the optimal metrics for solving a class of problems may be nearly as difficult as learning their optimal solutions. As is common in prior works on learning to solve optimization problems, the metric learning problem is formulated relative to a *parametric* optimization problem

$$x^\star(p) = \underset{x \in \mathbb{R}^n}{\operatorname{argmin}} \, f_p(x) + g_p(x)\,, \tag{3.19}$$

and we learn to predict metrics for parametric problem instances within a limited distribution. In the QP problem (3.16), this corresponds to the elements Q, q, L, b, W, and c each being potential functions of p. The metric M which best solves problem (3.19) when formulated as in (3.18) is then learned as a function of the problem's parameters $p \in \mathbb{R}^v$. This learned function takes the form of a neural network $\mathcal{N}_\omega : \mathbb{R}^v \to \mathcal{S}^n_{++}$ with weights ω, so that $M = \mathcal{N}_\omega(p)$. It is trained over a distribution of problem parameters $p \sim \mathcal{P}$, for which a finite dataset of instances $\{p_i\}_{i \in \mathcal{T}}$ are drawn. A target dataset $\{x^*(p_i)\}_{i \in \mathcal{T}}$ contains the corresponding optimal solutions, as per (3.19).

To define a loss function for training $\mathcal{N}_\omega$ on these data, first define the following function. Let $\mathcal{D}_k : \mathbb{R}^v \times \mathcal{S}^n_{++} \times \mathbb{R}^n \to \mathbb{R}^n$ denote the application of k iterations of DR or ADMM, on problem (3.19) with parameters p using metric M, starting from initial variable values x_0. It yields a solution estimate x_k; that is, $\mathcal{D}_k(p, M, x_0) = x_k$. The metric prediction model $\mathcal{N}_\omega$ is then trained to minimize the overall loss function

$$\min_\omega \frac{1}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} \|\mathcal{D}_k(p_i, \mathcal{N}_\omega(p_i), \mathcal{E}_\theta(p_i)) - x^*(p_i)\|^2 \tag{3.20}$$

by stochastic gradient descent. This requires backpropagation of gradients through the solver iterations which constitute $\mathcal{D}_k$. In this work, backpropogation is performed by automatic differentiation in PyTorch [120].

In equation (3.20), the function $\mathcal{E}_\theta$ is an oracle which returns a starting point x_0 for any parameter vector p. As part of an overall mechanism for producing fast solutions to (3.16), it is a neural network trained to produce direct estimates of the optimal solution to (3.19) by mean square error regression:

$$\min_\theta \frac{1}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} \|\mathcal{E}_\theta(p_i) - x^*(p_i)\|^2 . \tag{3.21}$$

Metric Representation

Finally, we describe the manner of representation used to predict the metrics M via the model $M = \mathcal{N}_\omega(p)$. As a neural network, $\mathcal{N}_\omega$ produces a vector of values $m \in \mathbb{R}^n$ which is then scaled between predefined upper and lower bounds $[m_{\min}, m_{\max}]$. Finally, a scalar parameter $\rho \in \mathbb{R}$ is predicted and also scaled to fit within predefined bounds $[\rho_{\min}, \rho_{\max}]$. The final metric is constructed as the diagonal matrix $M = diag(\rho \cdot m)$. In this work we explore only diagonal metrics for both ease of training and interpretability, however wider classes of positive definite matrices could be searched over and could provide additional improvements in convergence.

3.2.4 Numerical Results

For illustrative purposes, this section begins with a reductive two-dimensional problem on which some effects of metric learning are most easily observed. Then, we demonstrate the effect of metric learning on convergence for larger examples consisting of a portfolio optimization problem, and a model predictive control problem.

In the following experiments, predictive models $\mathcal{N}_\omega$ and $\mathcal{E}_\theta$ are fully connected neural networks with rectified linear unit (ReLU) activation functions. The values $\rho_{\min}, \rho_{\max}, m_{\min}, m_{\max}$ can be treated as hyperparameters; in practice, it is found that effective metrics can be learned by searching over $\rho_{\max}$ while the others remain fixed. All numerical test cases in this work are implemented using NeuroMANCER, an open source differentiable programming library built on top of Pytorch [49].

Active Set Prediction

This section illustrates how metric learning correlates with active set prediction in inequality-constrained problems, by assigning higher metric weights to the coordinates which correspond to

slack variables on the problem's active constraints. As an illustrative example, we consider a simple QP problem:

$$\min_{x,y} \; x^2 + y^2 \tag{3.22a}$$

subject to:

$$-x - y + p_1 \leq 0 \tag{3.22b}$$

$$x + y - p_1 - 1 \leq 0 \tag{3.22c}$$

$$x - y + p_2 - 1 \leq 0 \tag{3.22d}$$

$$-x + y - p_2 \leq 0 \tag{3.22e}$$

for parameters $p_1, p_2 \in [-2, 2]$. The constraint set defines a box which is translated around the origin according to the parameter choices as shown in Figure 3.6. After assigning slack variables s, the constraints become

$$-x - y + p_1 + s_1 = 0 \tag{3.23a}$$

$$x + y - p_1 - 1 + s_2 = 0 \tag{3.23b}$$

$$x - y + p_2 - 1 + s_3 = 0 \tag{3.23c}$$

$$-x + y - p_2 + s_4 = 0 \tag{3.23d}$$

$$s_1, \; s_2, \; s_3, \; s_4 \; \geq 0 \tag{3.23e}$$

We train both ADMM and DR metrics over the parameter space. The neural network map $\mathcal{N}_\omega$ returns diagonal metrics with diagonals of the form

$$diag([w_y, w_x, w_1, w_2, w_3, w_4])$$

where the weights w_x and w_y correspond to the primal variables, and the weights $\{w_1, w_2, w_3, w_4\}$ correspond to the slack variables for each of the constraints.

Settings The initial prediction model $\mathcal{E}_\theta$ uses hidden dimension 80. For $\mathcal{N}_\omega$ we use hidden dimension 20, with bounds $\rho_{\min} = 0.05$, $\rho_{\max} = 1.0$, $m_{\max} = 5.0$, and $m_{\min} = 0.2$. We sample 2000 parameters uniformly at random to construct a training set, and an additional 2000 parameters uniformly at random for a test set. The initial solution estimator $\mathcal{E}_\theta$ was trained for 200 epochs at learning rate of 0.001. Both ADMM and DR were run for 10 steps during training, and $\mathcal{N}_\omega$ was trained for 100 epochs at a learning rate of 0.001 for both.

Results As shown in Figure 3.5, DR with a trained metric converges the fastest, while ADMM using the heuristic metric as presented in [65] converges most slowly. Computation of the heuristic metric effectively ignores the inequality constraints, and computes a metric that achieves the fastest convergence with respect to the objective, were no constraints present this metric would achieve the best possible convergence. However, the results show that in the presence of constraints it can be detrimental.

Conversely, the learned metrics appear to achieve faster convergence by incorporating information about the active constraints. Figure 3.6 and Figure 3.7 highlight the close correspondence between the metric weights corresponding to slack variables and whether the associated constraints are active at the optimum for a problem.

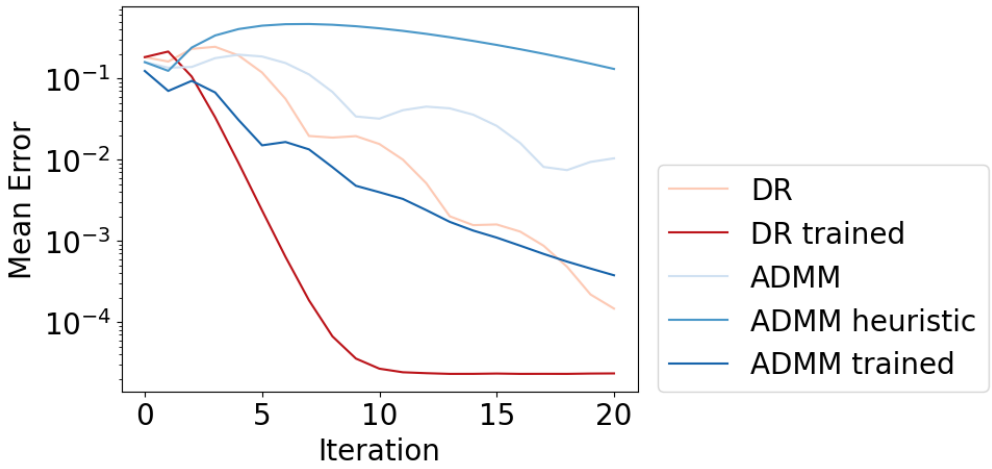
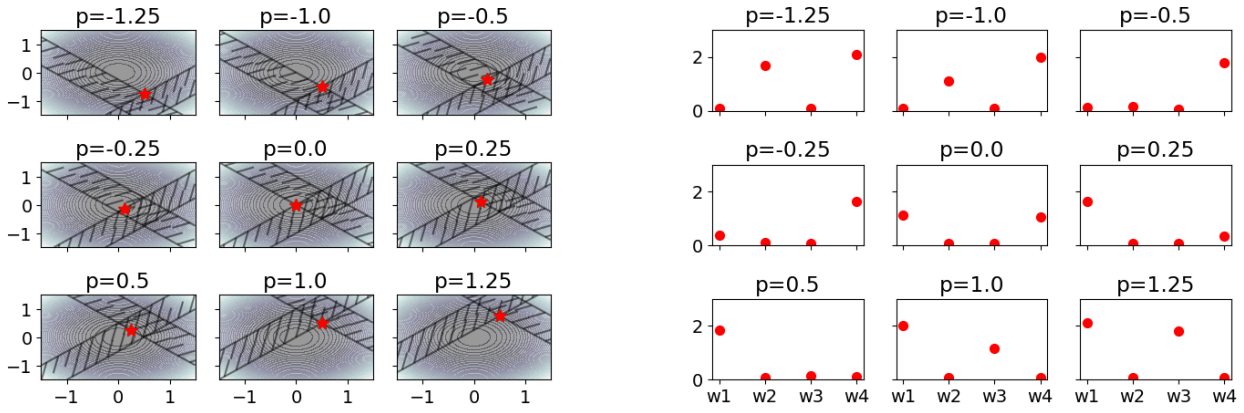


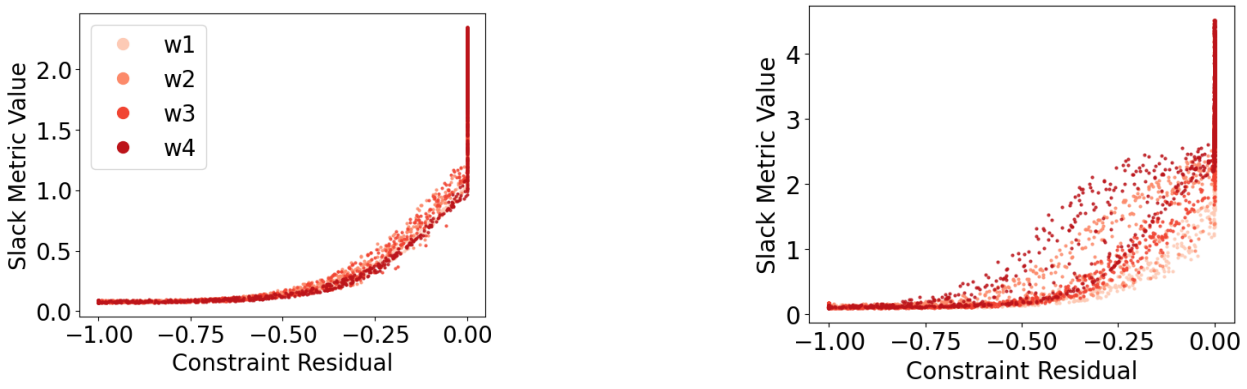
Figure 3.5: Comparison of convergence of DR and ADMM using trained metrics versus not, as well as comparison to use of a heuristic metric choice. Reported values are the mean error at each iteration on 2000 test set problems.



(a) Optimal Solution

(b) Metric weight for each slack variable

Figure 3.6: Plot (a) shows the optimal solution of (3.22) for parameter choice $p_1 = p_2 = p$, for values of p ranging from -1.25 to 1.25. As the feasible set is translated around the origin the optimal solution marked by a red star can be seen to trace along the constraints. Correspondingly, plot (b) shows that the learned DR metric weights corresponding to the slack variables for the inequality constraints are near zero when the constraint is not active and larger when the constraint is active, showing a clear correspondence between the active set and metric weights on slack variables.



(a) DR

(b) ADMM

Figure 3.7: Plots of the learned metric weight for slack variables and the corresponding constraint residual at the optimum on two thousand test problems for metrics learned using DR and ADMM. As constraint residuals becomes zero the constraints become active and the metric weight for the corresponding slack increases.

Portfolio Optimization Problem

This experiment models the optimal allocation of assets in an investment portfolio as a quadratic programming problem. Given n investment assets, their future price differentials $p \in \mathbb{R}^n$ are treated as parameters in the following QP:

$$x^*(p) = \underset{x}{\text{argmin}}\ x^T \Sigma x - p^T x \tag{3.24a}$$

$$\text{subject to:}\quad 1^T x = 1 \tag{3.24b}$$

$$x \geq 0, \tag{3.24c}$$

where Σ represents a constant covariance matrix. The objective (3.24a) balances maximization of future profit with minimization of price covariance as a measure of risk. Constraints (3.24b,3.24c) define a valid proportional allocation.

Settings Data on price action per asset are collected from the Nasdaq online database [112]. A training dataset of 5000 observations for the future price differential p are generated by adding Gaussian random noise to this data, plus an additional 500 each for the validation and testing sets. A 5-layer neural network with hidden layer dimension equal to the problem size n is used to predict the elements of a diagonal metric matrix M as a function of p. The following parameters are fixed: $m_{\min} = 0.01$, $m_{\max} = 1.0$, $\rho_{\min} = 0.01$. A search over the upper bound $\rho_{\max} \in \{1.0, 5.0, 10.0, 50.0, 100.0, 500.0\}$ shows that the best results occur for $\rho_{\max} = 100.0$ but remain similar for higher values of $\rho_{\max}$.

Results Figure 3.8 illustrates convergence of the DR and ADMM algorithms in solving (3.24), under metric prediction trained with k iterations in the loop, for $k \in \{5, 10, 15, 20, 25, 30\}$. At test time, 100 iterations of DR and 150 iterations ADMM are applied regardless of k. The plotted values

represent mean relative solution error in the $L2$ norm. Dotted black curves correspond to a baseline in which the standard Euclidean metric is used.

The following observations apply to both DR (at left) and ADMM (at right). The metric prediction model which is trained using k solver iterations always attains the best accuracy at exactly k solver iterations. Additionally, the models trained using more solver iterations k at each training step perform better as more solver iterations are performed at test time. This implies that an accelerated DR or ADMM model intended for exactly k solver iterations should be trained using k solver iterations, while a model intended for solver iteration until convergence should train using large values of k.

Note additionally that in this particular experiment, training for small k comes at a cost of slower long-term convergence in the case of DR. In ADMM, models trained with larger k generally perform equally or better than with smaller k. An exception is observed for $k = 30$ in ADMM, in which error is minimized at iteration 30 at the cost of higher error in both earlier and later iterations. In all but one case, the learned metrics outperform the Euclidean metric at test time.

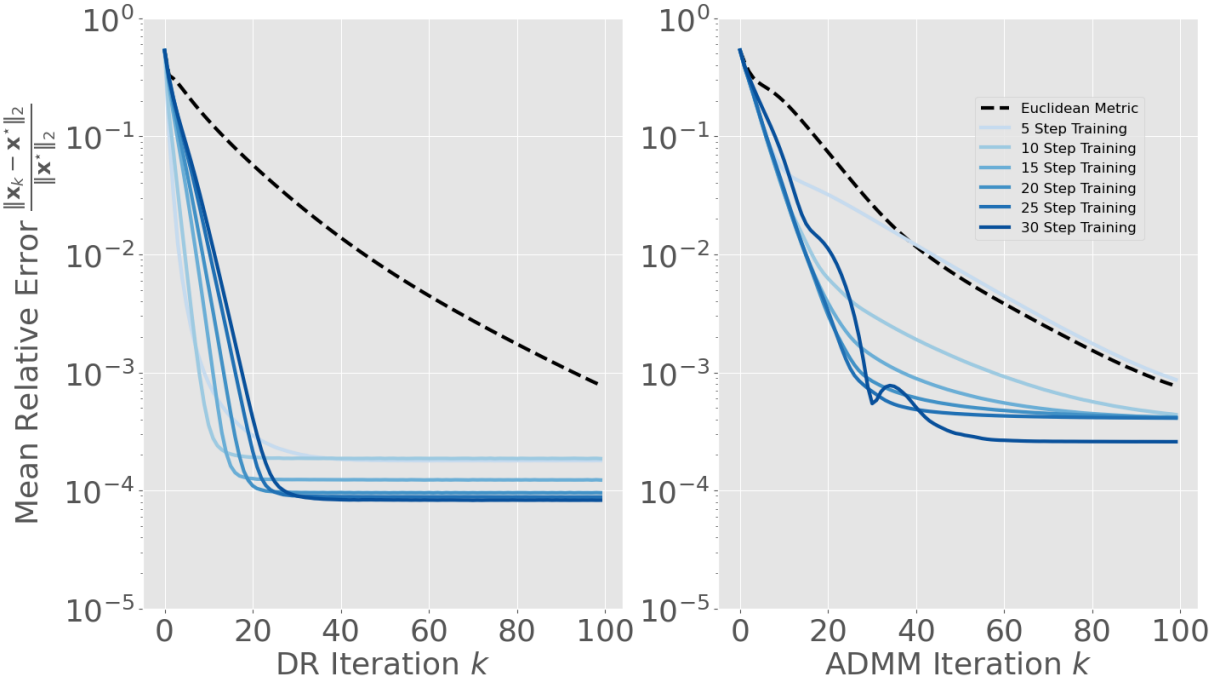


Figure 3.8: Results of training proximal metrics for DR and ADMM on Portfolio Optimization, to minimize error at increments of 5 iterations.

Comparison with Heuristic Metric Selection In order to compare the proposed metric learning for ADMM against the theoretically prescribed heuristic metric choice given in [65] we test the metric learning experiment on problem (3.24) with a reduced size of $n = 20$ assets. Problem size is reduced to allow a heuristic metric to be calculated efficiently.

To illustrate the effect of active constraints on the viability of the heuristic metric, results for ADMM are compared on two variants of problem (3.24). The first is as given in (3.24), and the second replaces the equality constraint (3.24b) with a scaled asset allocation budget $1^T x = 10$. This

increase in budget has the effect of reducing the percentage of active constraints over problem parameters. Figure 3.9 shows the results due to each allocation budget, at left and right respectively. In the larger budget case few constraints are active on average over the test set. This approximates a problem setting with no constraints where the heuristic metric would be optimal, and it can be seen to improve convergence in comparison to the Euclidean metric. Further, the learned optimal metrics perform similarly when trained for more than $k = 5$ iterations. On the other hand, the smaller budget leads to more active constraints and the heuristic metric performs poorly. Conversely, the trained metric achieves a greater improvement in convergence rate with respect to the Euclidean metric than in the high budget case, highlighting the capacity for improvement by incorporating information about active constraints.

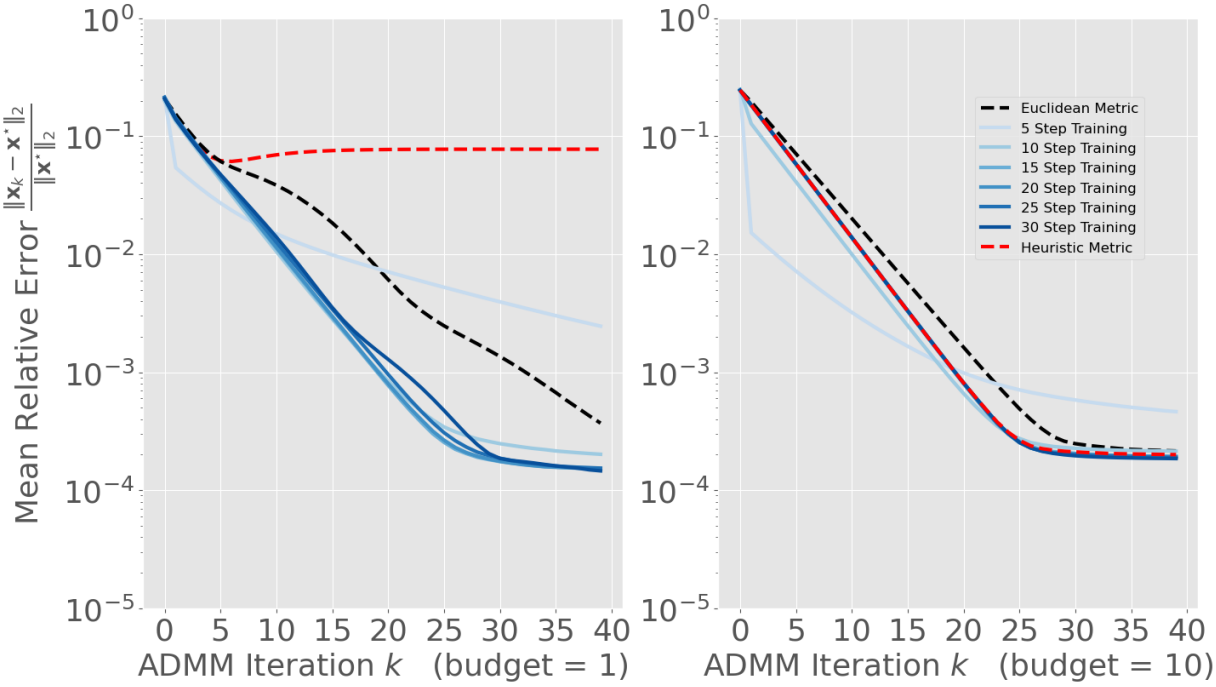


Figure 3.9: Comparison of trained and heuristic metrics for ADMM for Portfolio Optimization. The left plot presents a case where problem constraints are routinely active over problem parameters, while the right shows a case in which problem constraints are more rarely active.

Quadcopter Control

We also test metric learning on a standard reference tracking model predictive control problem implemented for a linear discrete time dynamical model of a quadcopter. Let the dynamics be defined by $A \in \mathbb{R}^{n,n}$, $B \in \mathbb{R}^{n,m}$, and the cost function be defined by positive semi-definite matrices $Q \in \mathbb{R}^{n,n}$ and $R \in \mathbb{R}^{m,m}$. Here we take the parameter $p \in \mathbb{R}^n$ to be the initial state of the system and

solve

$$u^*(p), x^*(p) = \underset{x,u}{\operatorname{argmin}} \sum_{k=0}^{N} (x_{k+1} - r)^T Q(x_{k+1} - r) + u_k^T R u_k \tag{3.25a}$$

subject to:

$$x_1 = Ap + Bu_0\,, \tag{3.25b}$$

$$x_{k+1} = Ax_k + Bu_k\,, \tag{3.25c}$$

$$u_a \leq u_k \leq u_b\,, \tag{3.25d}$$

$$x_a \leq x_k \leq x_b\,, \tag{3.25e}$$

$$\forall k \in \{0, 1, 2, \ldots N\}\,. \tag{3.25f}$$

The optimal solution produces control actions $u_k \in \mathbb{R}^m$ that drive the state of the system $x_k \in \mathbb{R}^n$ from a given initial state p towards the reference point $r \in \mathbb{R}^n$ over a finite number of time steps $k \in \{1, 2, \ldots N\}$. The control actions must also keep the state within the bounds $x_a, x_b \in \mathbb{R}^n$ while staying within the control bounds $u_a, u_b \in \mathbb{R}^m$. In practice the problem (3.25) is solved iteratively with the control implemented for just the initial time step, then the problem is re-solved from the new system state at the next time step. Thus the problem can be understood to be parameterized by the initial state and reference point.

Settings The quadcopter model has a 12 dimensional state and 4 dimensional control input. Only the first two state variables are constrained, and are restricted to the interval $[-\pi/6\ ,\ \pi/6]$. To generate training data we take the reference point r to be the origin and we generate initial states p uniformly at random with the first two variables sampled from $[-\pi/6\ ,\ \pi/6]$ and the rest from $[-0.8, 0.8]$. This choice was made such that generated problems were feasible, and state constraints were routinely active at solutions. We solve the problem (3.25) over a 10 step horizon, resulting in $n = 304$ variables with 132 inequality constraints, and 132 equality constraints. Diagonal elements of a metric matrix M are predicted on a per-instance basis using a 5-layer ReLU network of hidden layer size 400. As in Section 3.2.4, the parameter choices: $m_{\min} = 0.01$, $m_{\max} = 1.0$, $\rho_{\min} = 0.01$ are fixed. A search over the upper bound $\rho_{\max} \in \{1.0, 5.0, 10.0, 50.0, 100.0, 500.0\}$ shows that the best results occur for $\rho_{\max} = 50.0$ and remain similar for higher values of $\rho_{\max}$.

Results Convergence of both DR and ADMM due to the various trained metric prediction models are illustrated in Figure 3.10. Prediction models are trained using k steps of each algorithm for $\{5, 10, 15, 20, 25, 30, 35, 40\}$. Solution error over the full horizon needed for convergence is not shown, to make visible the effects of training up to the first 40 iterations. This is consistent with the intended application in real-time optimization, which demands solutions within stringent time constraints.

With regards to the effect of k, similar observations apply as in the portfolio optimization experiments. Models trained to minimize error at iteration k consistently perform best after exactly k iterations. Meanwhile, training with larger k generally benefits long-term convergence. Note that nearly all trained models reach a relative error of 1×10^{-2} in a fraction of the iterations required by the standard variant with a Euclidean metric.

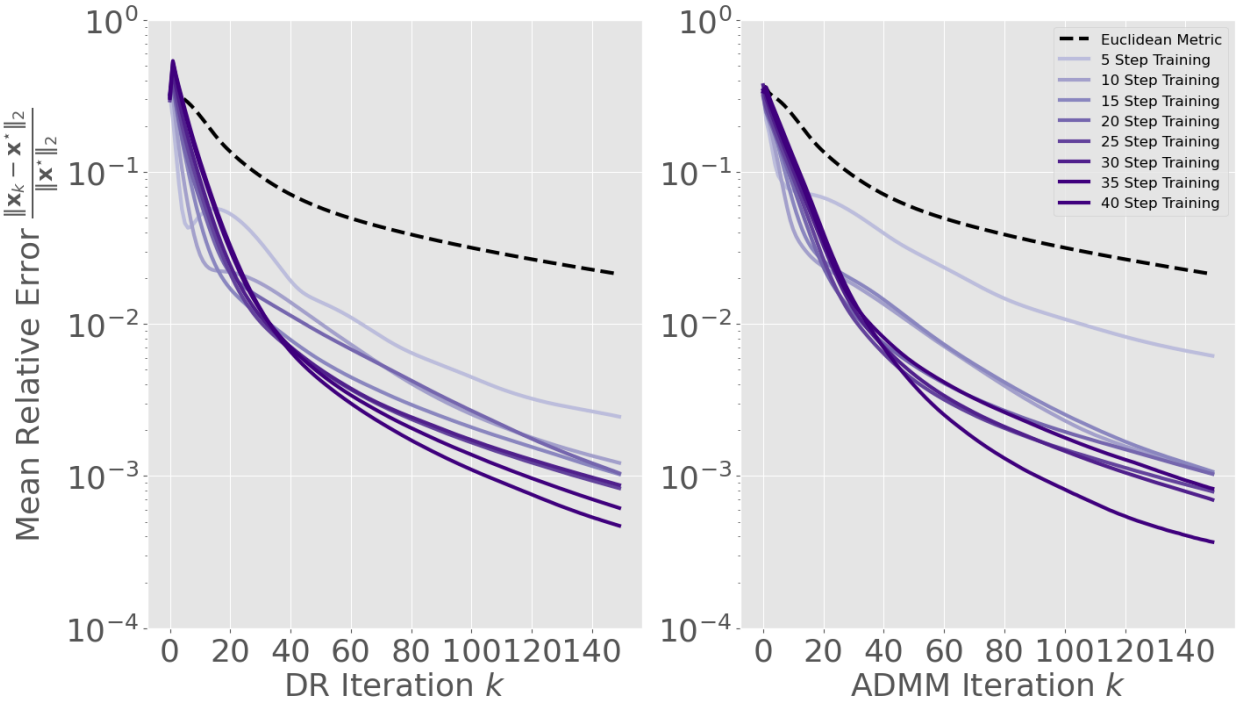


Figure 3.10: Results of training proximal metrics for DR and ADMM on Quadcopter Control, to minimize error at increments of 5 iterations.

3.2.5 Conclusions

Metric learning as presented here for parametric QP problems can consistently result in orders of magnitude improvements in solution accuracy at low a number of iterations. The most benefit is observed in problem settings with a significant number of inequality constraints relative to the problem size that are routinely active over parameters of interest. Notably, this is exactly the case in which the theoretically prescribed heuristic metrics are not guaranteed to be optimal. Future work is needed to understand the capacity for metric learning to reduce solution time on large-scale problems. By relying on a problem reformulation with slack variables, the total number of variables is expanded resulting in a larger overall problem size. On the other hand, it allows for metric learning to potentially significantly reduce the number of iterations required to achieve a given accuracy as seen here. Because it is independent of other strategies for learning to accelerate optimization, it may have significant potential to be combined with previously proposed techniques. Combining the proposed metric learning with nonoverlapping strategies such as prediction of solution warmstarts may further reduce overall solution times.

Chapter 4

Differentiable Solution of Optimization Problems

This chapter is dedicated to original research on the topic of differentiable programming. Of the two main sections, the first is methodologically driven, focused on analyzing the backpropagation routine that results from automatic differentiation through an optimization algorithm. The analysis sheds light on the convergence properties of that backpropagation, while making connections to the related topic of implicit differentiation. Section 4.2 focuses on an application of differentiable programming with the machine learning domain, in which the selection of models within an ensemble is optimized end-to-end to enhance performance on classification tasks. The two sections, respectively, are based on the author's published works [94, 93].

4.1 Analyzing and Enhancing the Backpropagation of Optimization Methods

The integration of optimization problems as components in neural networks has shown to be an effective framework for enforcing structured representations in deep learning. A parametric optimization problem defines a mapping from its unspecified parameters to the resulting optimal solutions, and this mapping is treated as a layer of a neural network. Outputs of the layer are guaranteed to obey the problem's constraints, which may be predefined or learned [89].

Using optimization as a layer can offer enhanced accuracy and efficiency on specialized learning tasks by imparting task-specific structural knowledge. For example, it has been used to design efficient multi-label classifiers and sparse attention mechanisms [106], learning to rank based on optimal matching [2, 91], accurate model selection protocols [93], and enhanced models for optimal decision-making under uncertainty [154].

While constrained optimization mappings can be used as components in neural networks in a similar manner to linear layers or activation functions [6], a prerequisite is their differentiation, for the backpropagation of gradients in end-to-end training by stochastic gradient descent.

This poses unique challenges, partly due to their lack of a closed form, and modern approaches typically follow one of two strategies. In *unrolling*, an optimization algorithm is executed entirely on the computational graph, and backpropagated by automatic differentiation from optimal solutions to the underlying problem parameters. The approach is adaptable to many problem classes, but

has been shown to suffer from time and space inefficiency, as well as vanishing gradients [109]. *Analytical differentiation* is a second strategy that circumvents those issues by forming implicit models for the derivatives of an optimization mapping and solving them exactly. However, current frameworks have rigid requirements on the form of the optimization problems, such as relying on transformations to canonical convex cone programs before applying a standardized procedure for their solution and differentiation, based on cone programming [4]. This system precludes the use of specialized solvers that are best-suited to handle various optimization problems, and inherently restricts itself only to convex problems.

Contributions. To address these limitations, this section proposes a novel analysis of unrolled optimization, which results in efficiently-solvable models for the backpropagation of unrolled optimization. Theoretically, the result is significant because it establishes an equivalence between unrolling and analytical differentiation, and allows for convergence of the backward pass to be analyzed in unrolling. Practically, it allows for the forward and backward passes of unrolled optimization to be disentangled and solved separately, using blackbox implementations of specialized algorithms. More specifically, we make the following novel contributions: **(1)** A theoretical analysis of unrolling that leads to an efficiently solvable closed-form model, whose solution is equivalent to the backward pass of an unrolled optimizer. **(2)** Building on this analysis, it proposes a system for generating analytically differentiable optimizers from unrolled implementations, accompanied by a Python library called `fold-opt` to facilitate automation. **(3)** Its efficiency and modeling advantages are demonstrated on a diverse set of end-to-end optimization and learning tasks, including the first demonstration of decision-focused learning with *nonconvex* decision models.

4.1.1 Setting and Goals

In this section, the goal is to differentiate mappings that are defined as the solution to an optimization problem. Consider the parameterized problem (4.1) which defines a function from a vector of parameters $\mathbf{c} \in \mathbb{R}^p$ to its associated optimal solution $\mathbf{x}^\star(\mathbf{c}) \in \mathbb{R}^n$:

$$\mathbf{x}^\star(\mathbf{c}) = \underset{\mathbf{x}}{\operatorname{argmin}}\ f(\mathbf{x}, \mathbf{c}) \tag{4.1a}$$

$$\text{subject to: } g(\mathbf{x}, \mathbf{c}) \leq \mathbf{0}, \tag{4.1b}$$

$$h(\mathbf{x}, \mathbf{c}) = \mathbf{0}, \tag{4.1c}$$

in which f is the objective function, and g and h are vector-valued functions capturing the inequality and equality constraints of the problem, respectively. The parameters $\mathbf{c}$ can be thought of as a prediction from previous layers of a neural network, or as learnable parameters analogous to the weights of a linear layer, or as some combination of both. It is assumed throughout that for any $\mathbf{c}$, the associated optimal solution $\mathbf{x}^\star(\mathbf{c})$ can be found by conventional solution methods, within some tolerance in solver error. This coincides with the "forward pass" of the mapping in a neural network. *The primary challenge is to compute its backward pass*, which amounts to finding the Jacobian matrix of partial derivatives $\frac{\partial \mathbf{x}^\star(\mathbf{c})}{\partial \mathbf{c}}$.

Backpropagation. Given a downstream task loss $\mathcal{L}$, backpropagation through $\mathbf{x}^\star(\mathbf{c})$ amounts to computing $\frac{\partial \mathcal{L}}{\partial \mathbf{c}}$ given $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^\star}$. In deep learning, backpropagation through a layer is typically accomplished

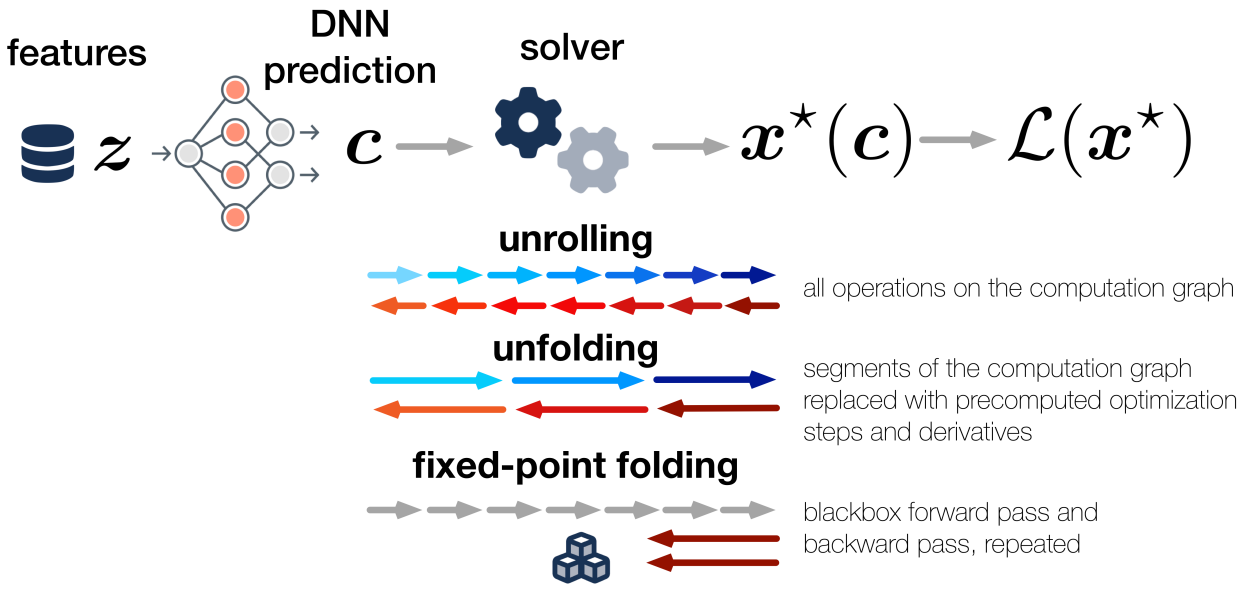


Figure 4.1: Compared to unrolling, unfolding requires fewer operations on the computational graph by replacing inner loops with Jacobian-gradient products. Fixed-point folding models the unfolding analytically, allowing for blackbox optimization implementations.

by automatic differentiation (AD), which propagates gradients through the low-level operations of an overall composite function by repeatedly applying the multivariate chain rule. This can be performed automatically given a forward pass implementation in an AD library such as PyTorch. However, it requires a record of all the operations performed during the forward pass and their dependencies, known as the *computational graph*.

Jacobian-gradient product (JgP). The *Jacobian* matrix of the vector-valued function $\mathbf{x}^\star(\mathbf{c}) : \mathbb{R}^p \to \mathbb{R}^n$ is a matrix $\frac{\partial \mathbf{x}^\star}{\partial \mathbf{c}}$ in $\mathbb{R}^{n \times p}$, whose elements at (i, j) are the partial derivatives $\frac{\partial x_i^\star(\mathbf{c})}{\partial c_j}$. When the Jacobian is known, backpropagation through $\mathbf{x}^\star(\mathbf{c})$ can be performed simply by computing the product

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^\star} \cdot \frac{\partial \mathbf{x}^\star(\mathbf{c})}{\partial \mathbf{c}}. \tag{4.2}$$

Folded Optimization: Overview. The problem (4.1) is most often solved by iterative methods, which refine an initial *starting point* $\mathbf{x}_0$ by repeated application of a subroutine, which we view as a function. For optimization variables $\mathbf{x} \in \mathbb{R}^n$, the *update function* is a vector-valued function $\mathcal{U} : \mathbb{R}^n \to \mathbb{R}^n$:

$$\mathbf{x}_{k+1}(\mathbf{c}) = \mathcal{U}(\mathbf{x}_k(\mathbf{c}),\ \mathbf{c}). \tag{U}$$

The iterations (U) *converge* if $\mathbf{x}_k(\mathbf{c}) \to \mathbf{x}^\star(\mathbf{c})$ as $k \to \infty$. When *unrolling*, the iterations (U) are computed and recorded on the computational graph, and the function $\mathbf{x}^\star(\mathbf{c})$ can be thereby be backpropagated by AD without explicitly representing its Jacobian. However, unrolling over many iterations often faces time and space inefficiency issues due to the need for graph storage and traversal [109]. In the following sections, we show how the backward pass of unrolling can be

analyzed to yield equivalent analytical models for the Jacobian of $\mathbf{x}^\star(\mathbf{c})$. The analysis recognizes two key challenges in modeling the backward pass of unrolling iterations (U).

First, it often happens that evaluation of $\mathcal{U}$ in (U) requires the solution of another optimization subproblem, such as a projection or proximal operator, which must also be unrolled. Section 4.1.2 introduces ***unfolding*** as a variant of unrolling, in which the unrolling of such inner loops is circumvented by analytical differentiation of the subproblem, allowing the analysis to be confined to a single unrolled loop.

Second, the backward pass of an unrolled solver is determined by its forward pass, whose trajectory depends on its (potentially arbitrary) starting point and the convergence properties of the chosen algorithm. Section 4.1.3 shows that the backward pass converges correctly even when the forward pass iterations are initialized at a precomputed optimal solution. This allows for separation of the forward and backward passes, which are typically *intertwined* across unrolled iterations, greatly simplifying the backward pass model and allowing for modular implementations of both passes.

Section 4.1.4 uses these concepts to show that the backward pass of unfolding (U) follows exactly the solution of the linear system for $\frac{\partial \mathbf{x}^\star(\mathbf{c})}{\partial \mathbf{c}}$ which arises by differentiating the fixed-point conditions of (U). Section 4.1.5 then outlines ***fixed-point folding***, a system for generating Jacobian-gradient products through optimization mappings from their unrolled solver implementations, based on efficient solution of the models proposed in Section 4.1.4. The main differences between unrolling, unfolding, and fixed-point folding are illustrated in Figure 4.1.

4.1.2 From Unrolling to Unfolding

For many optimization algorithms of the form (U), the update function $\mathcal{U}$ is composed of closed-form functions that are relatively simple to evaluate and differentiate. In general though, $\mathcal{U}$ may itself employ an optimization subproblem that is nontrivial to differentiate. That is,

$$\mathcal{U}(\mathbf{x}_k) \coloneqq \mathcal{T}\left(\, \mathcal{O}(\mathcal{S}(\mathbf{x}_k)),\ \ \mathbf{x}_k \,\right), \tag{O}$$

wherein the differentiation of $\mathcal{U}$ is complicated by an *inner optimization* sub-routine $\mathcal{O} : \mathbb{R}^n \to \mathbb{R}^n$. Here, $\mathcal{S}$ and $\mathcal{T}$ aggregate any operations preceding or following the inner optimization (such as gradient steps), viewed as differentiable functions. In such cases, unrolling (U) would also require unrolling $\mathcal{O}$. If the Jacobians of $\mathcal{O}$ can be found, then backpropagation through $\mathcal{U}$ can be completed, free of unrolling, by applying a chain rule through Equation (O), which may be assisted by automatic differentiation of $\mathcal{T}$ and $\mathcal{S}$.

Then, only the outermost iterations (U) need be unrolled on the computational graph for backpropagation. This partial unrolling, which allows for backpropagating large segments of computation at a time by leveraging analytically differentiated subroutines, is henceforth referred to as *unfolding*. It is made possible when the update step $\mathcal{U}$ is easier to differentiate than the overall optimization mapping $\mathbf{x}^\star(\mathbf{c})$.

Definition 1 (Unfolding)**.** *An* unfolded *differentiable optimization of the form (U) is one in which the backpropagation of $\mathcal{U}$ at each step does not require unrolling an iterative algorithm.*

Unfolding of a solver is distinguished from its more general unrolling by the presence of only a single unrolled loop. This definition sets the stage for Section 4.1.4, which shows how

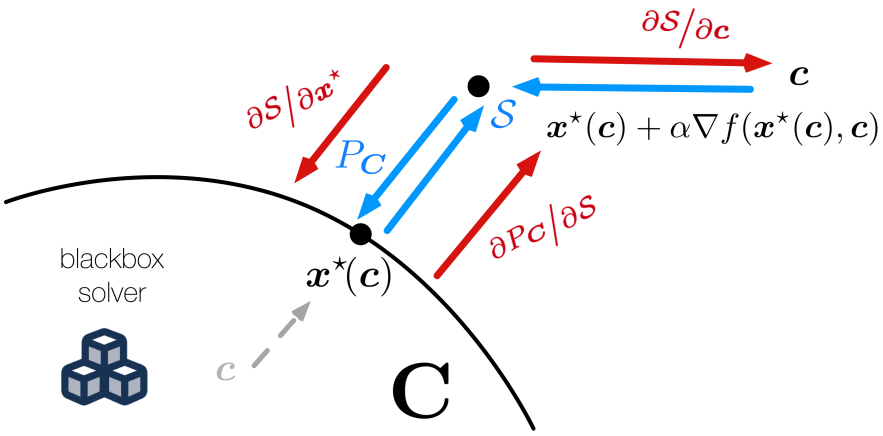


Figure 4.2: Unfolding Projected Gradient Descent at $\mathbf{x}^\star$ consists of alternating gradient step $\mathcal{S}$ with projection $\mathcal{P}_\mathbf{C}$. Each function's forward and backward pass are in blue and red, respectively.

the backpropagation of an unrolled loop can be modeled with a Jacobian-gradient product. Thus, unfolded optimization is a precursor to the complete replacement of backpropagation through loops in unrolled solver implementations by JgP.

When $\mathcal{O}$ has a closed form and does not require an iterative solution, there is no distinction between unrolling and unfolding of (U). When $\mathcal{O}$ is nontrivial to solve but has known Jacobians, they can be used to produce an unfolding. For example, in the case when $\mathcal{O}$ is a Quadratic Program (QP), a JgP-based differentiable QP solver called `qpth` is provided by [6]. Alternatively, the replacement of unrolled loops by JgP's proposed in Section 4.1.4 can be applied recursively to $\mathcal{O}$. These concepts are illustrated in the following examples, highlighting the roles of $\mathcal{U}$, $\mathcal{O}$ and $\mathcal{S}$. Each will be used to create differentiable folded optimization mappings for a variety of end-to-end learning tasks in Section 4.1.5.

Projected gradient descent. Given a problem

$$\min_{\mathbf{x} \in \mathbf{C}} \; f(\mathbf{x}) \tag{4.3}$$

where f is differentiable and $\mathbf{C}$ is the feasible set, Projected Gradient Descent (PGD) follows the update function

$$\mathbf{x}_{k+1} = \mathcal{P}_\mathbf{C}(\mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)), \tag{4.4}$$

where $\mathcal{O} = \mathcal{P}_\mathbf{C}$ is the Euclidean projection onto $\mathbf{C}$, and $\mathcal{S}(\mathbf{x}) = \mathbf{x} - \alpha \nabla f(\mathbf{x})$ is a gradient descent step. Many simple $\mathbf{C}$ have closed-form projections to facilitate unfolding of (4.4) (see [15]). Further, when $\mathbf{C}$ is linear, $\mathcal{P}_\mathbf{C}$ is a quadratic programming (QP) problem for which a differentiable solver `qpth` is available from [6].

Figure 4.2 shows one iteration of unfolding projected gradient descent, with the forward and backward pass of each recorded operation on the computational graph illustrated in blue and red, respectively.

Proximal gradient descent. More generally, to solve

$$\min_{\mathbf{x}} \; f(\mathbf{x}) + g(\mathbf{x}) \tag{4.5}$$

where f is differentiable and g is a closed convex function, proximal gradient descent follows the update function

$$\mathbf{x}_{k+1} = \text{Prox}_{\alpha_k g}\left(\mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)\right). \tag{4.6}$$

Here O is the proximal operator, defined as

$$\text{Prox}_g(\mathbf{x}) = \underset{y}{\text{argmin}} \left\{ g(\mathbf{y}) + \frac{1}{2}\|\mathbf{y} - \mathbf{x}\|^2 \right\}, \tag{4.7}$$

and its difficulty depends on g. Many simple proximal operators can be represented in closed form and have simple derivatives. For example, when $g(\mathbf{x}) = \lambda\|\mathbf{x}\|_1$, then $\text{Prox}_g = \mathcal{T}_\lambda(\mathbf{x})$ is the soft thresholding operator.

Sequential quadratic programming. Sequential Quadratic Programming (SQP) solves the general optimization problem (4.1) by approximating it at each step by a QP problem, whose objective is a second-order approximation of the problem's Lagrangian function, subject to a linearization of its constraints. SQP is well-suited for unfolded optimization, as it can solve a broad class of convex and nonconvex problems and can readily be unfolded by differentiating its QP subproblem with the `qpth` differentiable QP solver.

Quadratic programming by ADMM. A differentiable QP solver based on the alternating direction of multipliers is constructed in Section 4.1.5. Its inner optimization step O is an easier equality-constrained QP; its solution is equivalent to solving a linear system of equations, which has a simple derivative rule in PyTorch.

Given an unfolded QP solver by ADMM, its unrolled loop can be replaced with backpropagation by JgP as shown in Section 4.1.4. The resulting differentiable QP solver can then take the place of `qpth` in the examples above. Subsequently, *this technique can be applied recursively* to the resulting unfolded PGD and SQP solvers. This exemplifies the intermediate role of unfolding in converting unrolled, nested solvers to fully JgP-based implementations, detailed in Section 4.1.5.

From the viewpoint of unfolding, the analysis of backpropagation in unrolled solvers can be simplified by accounting for only a single unrolled loop. The next section identifies a further simplification: *that the backpropagation of an unfolded solver can be completely characterized by its action at a fixed point of the solution's algorithm.*

4.1.3 Unfolding at a Fixed Point

Optimization procedures of the form (U) generally require a starting point $\mathbf{x}_0$, which is often chosen arbitrarily, since convergence $\mathbf{x}_k \to \mathbf{x}^\star$ of iterative algorithms is typically guaranteed regardless of starting point. It is natural to then ask how the choice of $\mathbf{x}_0$ affects the convergence of the backward pass. We define backward-pass convergence as follows:

Definition 2. *Suppose that an unfolded iteration (U) produces a convergent sequence of solution iterates* $\lim_{k\to\infty} \mathbf{x}_k = \mathbf{x}^\star$ *in its forward pass. Then convergence of the backward pass is*

$$\lim_{k\to\infty} \frac{\partial \mathbf{x}_k}{\partial \mathbf{c}}(\mathbf{c}) = \frac{\partial \mathbf{x}^\star}{\partial \mathbf{c}}(\mathbf{c}). \tag{4.8}$$

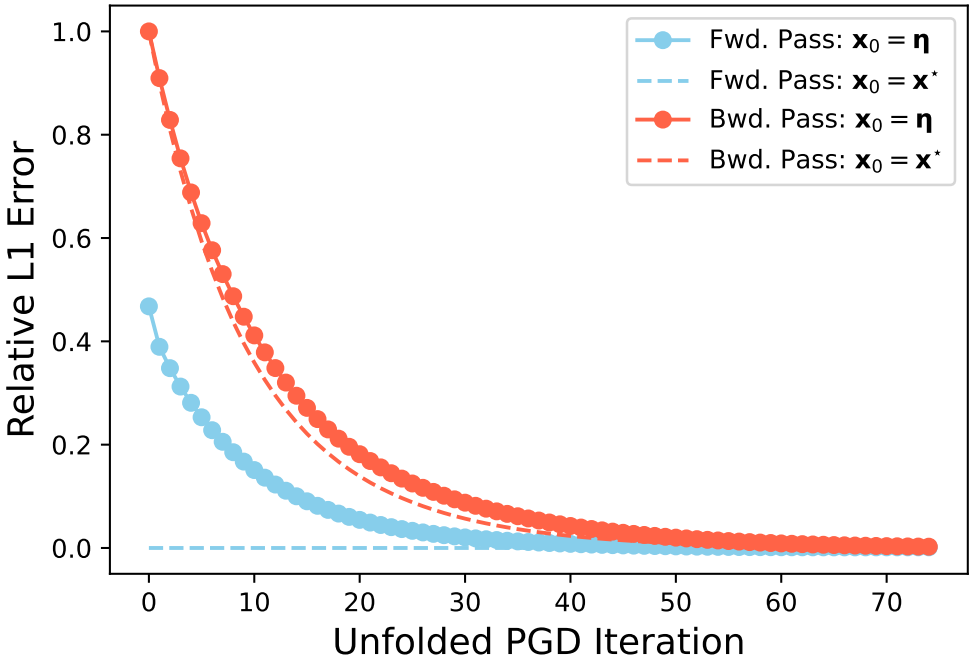


Figure 4.3: Forward and backward pass error in unfolding PGD

Effect of the starting point on backpropagation. Consider the optimization mapping (4.26) which maps feature embeddings to smooth top-k class predictions, and will be used to learn multilabel classification later in Section 4.1.5. A loss function $\mathcal{L}$ targets ground-truth top-k indicators, and the result of the backward pass is the gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{c}}$. To evaluate backward pass convergence in unfolded projected gradient descent, we measure the relative L_1 errors of the forward and backward passes, relative to the equivalent result after full convergence. We consider two starting points: the precomputed optimal solution $\mathbf{x}_0^a = \mathbf{x}^\star$, and a uniform random vector $\mathbf{x}_0^b = \boldsymbol{\eta} \sim \mathbf{U}(0, 1)$. The former case is illustrated in Figure 4.2, in which $\mathbf{x}_k$ remains stationary at each step.

Figure 4.3 reports the errors of the forward and backward pass at each iteration of the unfolded PGD under these two starting points. The figure shows that when starting the unfolding from the precomputed optimal solution $\mathbf{x}_0^a$, the forward pass error remains within error tolerance to zero. This is because $\mathbf{x}^\star(\mathbf{c}) = \mathcal{U}(\mathbf{x}^\star(\mathbf{c}), \mathbf{c})$ is a *fixed point* of (U). Interestingly though, the backward pass also converges, and at a faster rate than when starting from a random vector $\mathbf{x}_0^b$.

We will see that this phenomenon holds in general: when an unfolded optimizer is iterated at a precomputed optimal solution, its backward pass converges. This has practical implications which can be exploited to improve the efficiency and modularity of differentiable optimization layers based on unrolling. These improvements will form the basis of our system for converting unrolled solvers to JgP-based implementations, called *folded optimization*, and are discussed next.

Fixed-Point Unfolding: Forward Pass. Note first that backpropagation by unfolding at a fixed point must assume that a fixed point has already been found. This is generally equivalent to finding a local optimum of the optimization problem which defines the forward-pass mapping (4.1) [15]. Since the calculation of the fixed point itself does not need to be backpropagated, it can be furnished by a *blackbox* solver implementation. Furthermore, when $\mathbf{x}_0 = \mathbf{x}^\star$ is a fixed point of the iteration

(U), we have $\mathcal{U}(\mathbf{x}_k) = \mathbf{x}_k = \mathbf{x}^\star$, $\forall k$. Hence, *there is no need to re-evaluate the forward pass* of $\mathcal{U}$ in each unfolded iteration of (U) at a fixed point $\mathbf{x}^\star$.

This enables the use of any specialized method to compute the forward pass optimization (4.1), which can be different from unfolded algorithm used for backpropagation, assuming it shares the same fixed point. It also allows for highly optimized software implementations such as Gurobi [68], and is a major advantage over existing differentiable optimization frameworks such as `cvxpy`, which requires converting the problem to a convex cone program before solving it with a specialized operator-splitting method for conic programming [4], rendering it inefficient for many optimization problems.

Fixed-Point Unfolding: Backward Pass. While the forward pass of each unfolded update step (U) need not be recomputed at a fixed point, the dotted curves of Figure 4.3 illustrate that its backward pass must still be iterated until convergence. However, since $\mathbf{x}_k = \mathbf{x}^\star$, we also have $\frac{\partial \mathcal{U}(\mathbf{x}_k)}{\partial \mathbf{x}_k} = \frac{\partial \mathcal{U}(\mathbf{x}^\star)}{\partial \mathbf{x}^\star}$ at each iteration. Therefore the backward pass of $\mathcal{U}$ need only be computed *once*, and iterated until backpropagation of the full optimization mapping (4.1) converges.

Next, it will be shown that this process is equivalent to iteratively solving a linear system of equations. We identify the iterative method first, and then the linear system it solves, before proceeding to prove this fact. The following textbook result can be found, e.g., in [126].

Lemma 1. *Let* $\mathbf{B} \in \mathbb{R}^{n\times n}$ *and* $\mathbf{b} \in \mathbb{R}^n$. *For any* $\mathbf{z}_0 \in \mathbb{R}^n$, *the iteration*

$$\mathbf{z}_{k+1} = \mathbf{B}\mathbf{z}_k + \mathbf{b} \tag{LFPI}$$

converges to the solution $\mathbf{z}$ *of the linear system* $\mathbf{z} = \mathbf{B}\mathbf{z} + \mathbf{b}$ *whenever* $\mathbf{B}$ *is nonsingular and has spectral radius* $\rho(\mathbf{B}) < 1$. *Furthermore, the asymptotic convergence rate for* $\mathbf{z}_k \to \mathbf{z}$ *is*

$$-\log\ \rho(\mathbf{B}). \tag{4.9}$$

Linear fixed-point iteration (LFPI) is a foundational iterative linear system solver, and can be applied to any linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ by rearranging $\mathbf{z} = \mathbf{B}\mathbf{z} + \mathbf{b}$ and identifying $\mathbf{A} = \mathbf{I} - \mathbf{B}$.

Next, we exhibit the linear system which is solved for the desired gradients $\frac{\partial \mathbf{x}^\star}{\partial \mathbf{c}}(\mathbf{c})$ by unfolding at a fixed point. Consider the fixed-point conditions of the iteration (U):

$$\mathbf{x}^\star(\mathbf{c}) = \mathcal{U}(\mathbf{x}^\star(\mathbf{c}),\ \mathbf{c}) \tag{FP}$$

Differentiating (FP) with respect to $\mathbf{c}$,

$$\frac{\partial \mathbf{x}^\star}{\partial \mathbf{c}}(\mathbf{c}) = \underbrace{\frac{\partial \mathcal{U}}{\partial \mathbf{x}^\star}(\mathbf{x}^\star(\mathbf{c}),\ \mathbf{c})}_{\mathbf{\Phi}} \cdot \frac{\partial \mathbf{x}^\star}{\partial \mathbf{c}}(\mathbf{c}) + \underbrace{\frac{\partial \mathcal{U}}{\partial \mathbf{c}}(\mathbf{x}^\star(\mathbf{c}),\ \mathbf{c})}_{\mathbf{\Psi}}, \tag{4.10}$$

by the chain rule and recognizing the implicit and explicit dependence of $\mathcal{U}$ on the independent parameters $\mathbf{c}$. Equation (4.10) will be called the *differential fixed-point conditions*. Rearranging (4.10), the desired $\frac{\partial \mathbf{x}^\star}{\partial \mathbf{c}}(\mathbf{c})$ can be found in terms of $\mathbf{\Phi}$ and $\mathbf{\Psi}$ as defined above, to yield the system (DFP) below.

The results discussed next are valid under the assumptions that $\mathbf{x}^\star : \mathbb{R}^n \to \mathbb{R}^n$ is differentiable in an open set C, and Equation (FP) holds for $\mathbf{c} \in C$. Additionally, $\mathcal{U}$ is assumed differentiable on an open set containing the point $(\mathbf{x}^\star(\mathbf{c}), \mathbf{c})$.

Lemma 2. *When* $\mathbf{I}$ *is the identity operator and* $\boldsymbol{\Phi}$ *nonsingular,*

$$(\mathbf{I} - \boldsymbol{\Phi})\frac{\partial \mathbf{x}^\star}{\partial \mathbf{c}} = \boldsymbol{\Psi}. \tag{DFP}$$

The result follows from the Implicit Function theorem [110]. It implies that the Jacobian $\frac{\partial \mathbf{x}^\star}{\partial \mathbf{c}}$ can be found as the solution to a linear system once the prerequisite Jacobians $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}$ are found; these correspond to backpropagation of the update function $\mathcal{U}$ at $\mathbf{x}^\star(\mathbf{c})$.

4.1.4 Folded Optimization

We are now ready to discuss the central result of the section. Informally, it states that the backward pass of an iterative solver (U), unfolded at a precomputed optimal solution $\mathbf{x}^\star(\mathbf{c})$, is equivalent to solving the linear equations (DFP) using linear fixed-point iteration, as outlined in Lemma 1.

This has significant implications for unrolling optimization. It shows that backpropagation of unfolding is computationally equivalent to solving linear equations using a specific algorithm and does not require automatic differentiation. It also provides insight into the convergence properties of this backpropagation, including its convergence rate, and highlights that more efficient algorithms can be used to solve (DFP) in favor of its inherent LFPI implementation in unfolding.

The following results hold under the assumptions that the parameterized optimization mapping $\mathbf{x}^\star$ converges for certain parameters $\mathbf{c}$ through a sequence of iterates $\mathbf{x}_k(\mathbf{c}) \to \mathbf{x}^\star(\mathbf{c})$ using algorithm (U), and that $\boldsymbol{\Phi}$ is nonsingular with a spectral radius $\rho(\boldsymbol{\Phi}) < 1$.

Theorem 4. *The backward pass of an unfolding of algorithm (U), starting at the point* $\mathbf{x}_k = \mathbf{x}^\star$*, is equivalent to linear fixed-point iteration on the linear system (DFP), and will converge to its unique solution at an asymptotic rate of*

$$-\log \rho(\boldsymbol{\Phi}). \tag{4.11}$$

Proof. Since $\mathcal{U}$ converges given any parameters $\mathbf{c} \in C$, Equation (FP) holds for any $\mathbf{c} \in C$. Together with the assumption the $\mathcal{U}$ is differentiable on a neighborhood of $(\mathbf{x}^\star(\mathbf{c}),\ \mathbf{c})$,

$$(\mathbf{I} - \boldsymbol{\Phi})\frac{\partial \mathbf{x}^\star}{\partial \mathbf{c}} = \boldsymbol{\Psi} \tag{4.12}$$

holds by Lemma 2. When (U) is unfolded, its backpropagation rule can be derived by differentiating its update rule:

$$\frac{\partial}{\partial \mathbf{c}}\left[\ \mathbf{x}_{k+1}(\mathbf{c})\ \right] = \frac{\partial}{\partial \mathbf{c}}\left[\ \mathcal{U}(\mathbf{x}_k(\mathbf{c}),\ \mathbf{c})\ \right] \tag{4.13a}$$

$$\frac{\partial \mathbf{x}_{k+1}}{\partial \mathbf{c}}(\mathbf{c}) = \frac{\partial \mathcal{U}}{\partial \mathbf{x}_k}\frac{\partial \mathbf{x}_k}{\partial \mathbf{c}} + \frac{\partial \mathcal{U}}{\partial \mathbf{c}}, \tag{4.13b}$$

where all terms on the right-hand side are evaluated at $\mathbf{c}$ and $\mathbf{x}_k(\mathbf{c})$. Note that in the base case $k = 0$, since in general $\mathbf{x}_0$ is arbitrary and does not depend on $\mathbf{c}$, $\frac{\partial \mathbf{x}_0}{\partial \mathbf{c}} = \mathbf{0}$ and

$$\frac{\partial \mathbf{x}_1}{\partial \mathbf{c}}(\mathbf{c}) = \frac{\partial \mathcal{U}}{\partial \mathbf{c}}(\mathbf{x}_0, \mathbf{c}). \tag{4.14}$$

This holds also when $\mathbf{x}_0 = \mathbf{x}^\star$ w.r.t. backpropagation of (U), since $\mathbf{x}^\star$ is precomputed outside the computational graph of its unfolding. Now since $\mathbf{x}^\star$ is a fixed point of (U),

$$\mathbf{x}_k(\mathbf{c}) = \mathbf{x}^\star(\mathbf{c}) \quad \forall k \geq 0, \tag{4.15}$$

which implies

$$\frac{\partial \mathcal{U}}{\partial \mathbf{x}_k}(\mathbf{x}_k(\mathbf{c}),\ \mathbf{c}) = \frac{\partial \mathcal{U}}{\partial \mathbf{x}^\star}(\mathbf{x}^\star(\mathbf{c}),\ \mathbf{c}) = \mathbf{\Phi}, \quad \forall k \geq 0 \tag{4.16a}$$

$$\frac{\partial \mathcal{U}}{\partial \mathbf{c}}(\mathbf{x}_k(\mathbf{c}),\ \mathbf{c}) = \frac{\partial \mathcal{U}}{\partial \mathbf{c}}(\mathbf{x}^\star(\mathbf{c}),\ \mathbf{c}) = \mathbf{\Psi}, \quad \forall k \geq 0. \tag{4.16b}$$

Letting $\mathbf{J}_k \coloneqq \frac{\partial \mathbf{x}_k}{\partial \mathbf{c}}(\mathbf{c})$, the rule (4.13b) for unfolding at a fixed-point $\mathbf{x}^\star$ becomes, along with initial conditions (4.14),

$$\mathbf{J}_0 = \mathbf{\Psi} \tag{4.17a}$$

$$\mathbf{J}_{k+1} = \mathbf{\Phi}\mathbf{J}_k + \mathbf{\Psi}. \tag{4.17b}$$

The result then holds by direct application of Lemma 1 to (4.17), recognizing $\mathbf{z}_k = \mathbf{J}_k$, $\mathbf{B} = \mathbf{\Phi}$ and $\mathbf{z}_0 = \mathbf{b} = \mathbf{\Psi}$. □

The following is a direct result from the proof of Theorem 4.

Corollary 1. *Backpropagation of the fixed-point unfolding is equivalent to a Jacobian computed as:*

$$\mathbf{J}_0 = \mathbf{\Psi} \tag{4.18a}$$

$$\mathbf{J}_{k+1} = \mathbf{\Phi}\mathbf{J}_k + \mathbf{\Psi}, \tag{4.18b}$$

where $\mathbf{J}_k \coloneqq \frac{\partial \mathbf{x}_k}{\partial \mathbf{c}}(\mathbf{c})$.

The proof illustrates that in the LFPI applied through fixed-point unfolding, the initial iterate is $\mathbf{J}_0 = \mathbf{\Psi}$. In any case, convergence to the true Jacobian is guaranteed regardless of the initial iterate.

Theorem 4 specifically applies to the case where the initial iterate is the precomputed optimal solution, $\mathbf{x}_0 = \mathbf{x}^\star$. However, it also has implications for the general case where $\mathbf{x}_0$ is arbitrary. As the forward pass optimization converges, i.e. $\mathbf{x}_k \to \mathbf{x}^\star$ as $k \to \infty$, this case becomes identical to the one proved in Theorem 4 and a similar asymptotic convergence result applies. If $\mathbf{x}_k \to \mathbf{x}^\star$ and $\mathbf{\Phi}$ is a nonsingular operator with $\rho(\mathbf{\Phi}) < 1$, the following result holds.

Corollary 2. *When the parametric problem (4.1) can be solved by an iterative method of the form (U) and the forward pass of the unfolded algorithm converges, the backward pass converges at an asymptotic rate that is bounded by* $-\log \rho(\mathbf{\Phi})$.

The result above helps explain why the forward and backward pass in the experiment of Section 4.1.3 converge at different rates. If the forward pass converges faster than $-\log\ \rho(\mathbf{\Phi})$, the *overall convergence rate of an unfolding is limited by that of the backward pass*.

Fixed-Point Folding. Building on the above findings, we propose to replace unfolding at the fixed point $\mathbf{x}^\star$ with the equivalent Jacobian-gradient product following the solution of (DFP). This leads to *fixed-point folding*, a system for converting *any* unrolled implementation of an optimization method (U) into a differentiable *folded optimization* that *eliminates unrolled loops*. By leveraging AD through a single step of the unrolled solver, but avoiding the use of AD to unroll through multiple iterations on the computational graph, it enables backpropagation of optimization layers by JgP using a seamless integration of automatic and analytical differentiation.

This section introduces `fold-opt`, a Python library which supplies routines for constructing and solving the requisite linear systems, and integrating the resulting Jacobian-gradient products into the computational graph of PyTorch. To do so, it requires an AD implementation of the update function $\mathcal{U}$ in Pytorch for an appropriately chosen optimization routine. Once constructed, the linear system (DFP) may be solved by a blackbox linear solver. However, note that for backpropagation it is often most efficient *not to solve* for the Jacobian matrix $\frac{\partial \mathbf{x}^\star}{\partial \mathbf{c}}$, but rather to solve directly for the desired JgP vector $\frac{\partial \mathcal{L}}{\partial \mathbf{c}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^\star} \cdot \frac{\partial \mathbf{x}^\star(\mathbf{c})}{\partial \mathbf{c}}$. Solution by iterative methods can be implemented without explicit matrix-vector products by utilizing the backward pass of (U) in place of its equivalent JgP.

The purpose of the `fold-opt` library is to facilitate the conversion of unfolded optimization code into JgP-based differentiable optimization, by leveraging automatic differentiation in Pytorch. It relies on the fact that backpropagation of a (gradient) vector $\boldsymbol{g}$ through the computational graph of a function $\boldsymbol{x} \to \boldsymbol{f}(\boldsymbol{x})$ by reverse-mode automatic differentiation is equivalent to computing the JgP product $\boldsymbol{g} \cdot \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}}$.

We erite the backpropagation of the loss gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^\star}$ through k unfolded steps of (U) at the fixed point $\boldsymbol{x}^\star$ as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^\star}^T \left(\frac{\partial \mathbf{x}^k(\mathbf{c})}{\partial \mathbf{c}} \right). \tag{4.19}$$

We seek to compute the limit $\frac{\partial \mathcal{L}}{\partial \mathbf{c}} = \boldsymbol{g}^T \boldsymbol{J}$ where $\boldsymbol{g} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^\star}$, $\boldsymbol{J} \coloneqq \lim_{k\to\infty} \boldsymbol{J}_k$, and $\boldsymbol{J}_k = \frac{\partial \mathbf{x}^k(\mathbf{c})}{\partial \mathbf{c}}$. Following the backpropagation rule (4.18), the expression (4.19) is equal to

$$\begin{aligned} \boldsymbol{g}^T \boldsymbol{J}_k &= \boldsymbol{g}^T \left(\boldsymbol{\Phi} \boldsymbol{J}_{k-1} + \boldsymbol{\Psi} \right) && \text{(4.20a)} \\ &= \boldsymbol{g}^T \left(\boldsymbol{\Phi}^k \boldsymbol{\Psi} + \boldsymbol{\Phi}^{k-1} \boldsymbol{\Psi} + \ldots + \boldsymbol{\Phi} \boldsymbol{\Psi} + \boldsymbol{\Psi} \right) && \text{(4.20b)} \end{aligned}$$

This expression can be rearranged as

$$\boldsymbol{g}^T \boldsymbol{J}_k = \boldsymbol{v}_k^T \boldsymbol{\Psi} \tag{4.21}$$

where

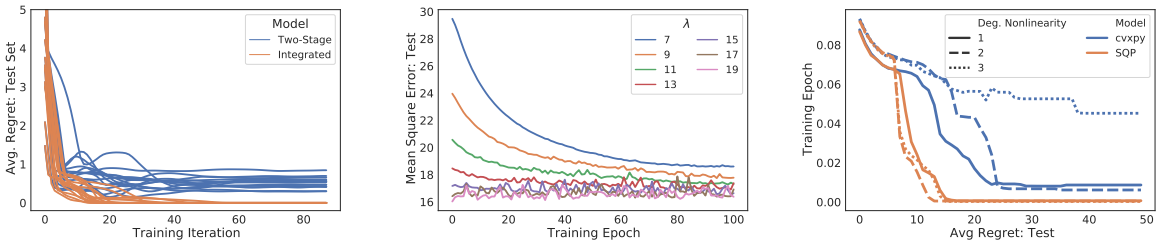
$$\boldsymbol{v}_k^T \coloneqq \left(\boldsymbol{g}^T \boldsymbol{\Phi}^k + \boldsymbol{g}^T \boldsymbol{\Phi}^{k-1} + \ldots + \boldsymbol{g}^T \boldsymbol{\Phi} + \boldsymbol{g}^T \right). \tag{4.22}$$

The sequence $\boldsymbol{v}_k$ can be computed most efficiently as

$$\boldsymbol{v}_k^T = \boldsymbol{v}_{k-1}^T \boldsymbol{\Phi} + \boldsymbol{g}^T \tag{4.23}$$

which identifies $\boldsymbol{v} \coloneqq \lim_{k\to\infty} \boldsymbol{v}_k$ as the solution of the linear system

$$\boldsymbol{v}^T (\boldsymbol{I} - \boldsymbol{\Phi}) = \boldsymbol{g}^T \tag{4.24}$$



(a) (b) (c)

Figure 4.4: Bilinear Programming (a), Enhanced Denoising with *f-FDPG* (b), and Portfolio optimization (c).

under the conditions of Lemma (1), after transposing both sides of (4.23) and (4.24) .

Once $\boldsymbol{v}^T$ is calculated by (4.23), the desired JgP is

$$\boldsymbol{g}^T \boldsymbol{J} = \boldsymbol{v}^T \boldsymbol{\Psi}. \tag{4.25}$$

The left matrix-vector product with respect to $\boldsymbol{\Phi}$ in (4.23) and $\boldsymbol{\Psi}$ in (4.25) can be computed by backpropagation through the computational graph of the update function $\mathcal{U}(\boldsymbol{x}^\star(\boldsymbol{c}),\ \boldsymbol{c})$, backward to $\boldsymbol{x}^\star(\boldsymbol{c})$ and $\boldsymbol{c}$ respectively.

Notice that in contrast to unfolding, this backpropagation method requires to store the computational graph only for a single update step, rather than for an entire optimization routine consisting of many iterations.

Having reduced the calculation of $\boldsymbol{g}^T \boldsymbol{J}$ to the solution of a linear system (4.24) followed by a matrix-vector product (4.25), it is clear how efficiency can be improved by replacing the LFPI iterations (4.23) with a faster-converging linear solution scheme based on matrix-vector products, such as Krylov subspace methods. This emphasizes the inherently sub-optimal convergence rate of backpropagation in unfolded solvers, and such upgrades will be planned for future versions of `fold-opt`.

Nested application As per Definition 1, the innermost optimization loop of a nested unrolling can be considered an unfolded loop, and can be backpropagated by JgP with fixed-point folding. Subsequently, the next innermost loop can be considered unfolded, and the same process applied until all unrolled optimization loops are replaced with their respective JgP's. Nested application of fixed-point folding is exemplified by *f-PGDb* (see Section 6), which applies successive fixed-point folding through ADMM and PGD to compose a JgP-based differentiable layer for any optimization problem with a smooth objective function and linear constraints.

Optimization Parameters Note that before it can be used to generate JgP's in folded optimization, a differentiable solver step $\mathcal{U}$ often requires specification of parameters such as gradient descent stepsizes. This can affect $\rho(\boldsymbol{\Phi})$, and hence the backward pass convergence and its rate by Theorem 4.

4.1.5 Experiments

This section evaluates folded optimization on four end-to-end optimization and learning tasks. It is primarily evaluated against `cvxpy`, which is the preeminent general-purpose differentiable optimization solver. Two crucial limitations of `cvxpy` are its efficiency and expressiveness. This is due to its reliance on transforming general optimization programs to convex cone programs, before applying a standardized operator-splitting cone program solver and differentiation scheme. This precludes the incorporation of problem-specific solvers in the forward pass and limits its use to convex problems only. One major benefit of `fold-opt` is the modularity of its forward optimization pass, which can apply any black-box algorithm to produce $\mathbf{x}^\star(\mathbf{c})$. In each experiment below, this is used to demonstrate a different advantage.

Implementations. The experiments test four `fold-opt` implementations: **(1)** *f-PGDa* applies to optimization mappings with linear constraints, and is based on fixed-point folding of projected gradient descent steps, where each inner projection is a QP solved by the differentiable QP solver `qpth` [6]. **(2)** *f-PGDb* is a variation on the former, in which the inner QP step is differentiated by fixed-point folding of an ADMM method. **(3)** *f-SQP* applies to optimization with nonlinear constraints and uses folded SQP with the inner QP differentiated by `qpth`. **(4)** *f-FDPG* comes from fixed-point folding of the Fast Dual Proximal Gradient Descent (FDPG) method. The inner Prox is a soft thresholding operator, whose simple closed form is differentiated by AD in PyTorch.

Decision-focused learning with nonconvex bilinear programming. The first experiment showcases the ability of folded optimization to be applied in decision-focused learning with *nonconvex* optimization. In this experiment, we predict the coefficients of a *bilinear* program

$$\begin{aligned}\mathbf{x}^\star(\mathbf{c},\mathbf{d}) = \underset{\mathbf{0}\le\mathbf{x},\mathbf{y}\le\mathbf{1}}{\operatorname{argmax}} \;& \mathbf{c}^T\mathbf{x} + \mathbf{x}^T\mathbf{Q}\mathbf{y} + \mathbf{d}^T\mathbf{y} \\ \texttt{s.t.} \;& \sum \mathbf{x} = p,\ \sum \mathbf{y} = q,\end{aligned}$$

in which two separable linear programs are confounded by a nonconvex quadratic objective term $\boldsymbol{Q}$. Costs $\boldsymbol{c}$ and $\boldsymbol{d}$ are predicted by a 5-layer network, while p and q are constants. Such programs have numerous industrial applications such as optimal mixing and pooling in gas refining [11]. Here we focus on the difficulty posed by the problem's form and propose a task to evaluate *f-PGDb* in learning with nonconvex optimization. Feature and cost data are generated by the process described in Appendix B.1, along with 15 distinct $\mathbf{Q}$ for a collection of nonconvex decision models.

It is known that PGD converges to local optima in nonconvex problems [10], and this folded implementation uses the Gurobi nonconvex QP solver to find a global optimum. Since no known general framework can accommodate nonconvex optimization mappings in end-to-end models, we benchmark against the *two-stage* approach, in which the costs $\mathbf{c}$, and $\mathbf{d}$ are targeted to ground-truth costs by MSE loss and the optimization problem is solved as a separate component from the learning task. The integrated *f-PGDb* model minimizes solution regret (i.e., suboptimality) directly. [52]. Notice in Figure 4.4(a) how *f-PGDb* achieves much lower regret for each of the 15 nonconvex objectives.

Enhanced Total Variation Denoising. This experiment illustrates the efficiency benefit of incorporating problem-specific solvers. The optimization models a denoiser

$$\mathbf{x}^\star(\mathbf{D}) = \underset{\mathbf{x}}{\text{argmin}} \quad \frac{1}{2}||\mathbf{x} - \mathbf{d}||^2 + \lambda||\mathbf{D}\mathbf{x}||_1,$$

which seeks to recover the true signal $\mathbf{x}^\star$ from a noisy input $\mathbf{d}$ and is often best handled by variants of Dual Proximal Gradient Descent. Classically, $\mathbf{D}$ is a differencing matrix so that $||\mathbf{D}\mathbf{x}||_1$ represents total variation. Here we initialize $\mathbf{D}$ to this classic case and *learn* a better $\mathbf{D}$ by targeting a set of true signals with MSE loss and adding Gaussian noise to generate their corresponding noisy inputs. Figure 4.4(b) shows test MSE throughout training due to *f-FDPG* for various choice of λ. Appendix B.2 shows comparable results from the framework of [6], which converts the problem to a QP form in order to differentiate the mapping analytically with `qpth`. Small differences in these results likely stem from solver error tolerance in the two methods. However, *f-FDPG computes* $\mathbf{x}^\star(\mathbf{D})$ *up to* 40 *times faster*.

Mutilabel Classification on CIFAR100. Since gradient errors accumulate at each training step, we ask how precise are the operations performed by `fold-opt` in the backward pass. This experiment compares the backpropagation of both *f-PGDa* and *f-SQP* with that of `cvxpy`, by using the forward pass of `cvxpy` in each model as a control factor.

This experiment, adapted from [19], implements a smooth top-5 classification model on noisy CIFAR-100. The optimization below maps image feature embeddings $\mathbf{c}$ from DenseNet 40-40 [73], to smoothed top-k binary class indicators (see Appendix B.1 for more details):

$$\mathbf{x}^\star(\mathbf{c}) = \underset{\mathbf{0}\leq\mathbf{x}\leq\mathbf{1}}{\text{argmax}} \quad \mathbf{c}^T\mathbf{x} + \sum_i x_i \log x_i \quad s.t. \quad \sum \mathbf{x} = k \tag{4.26}$$

Appendix B.2 shows that all three models have indistinguishable classification accuracy throughout training, indicating the backward pass of both `fold-opt` models is precise and agrees with a known benchmark even after 30 epochs of training on 45k samples. On the other hand, the more sensitive test set shows marginal accuracy divergence after a few epochs.

Portfolio Prediction and Optimization. Having established the equivalence in performance of the backward pass across these models, the final experiment describes a situation in which `cvxpy` makes non negligible errors in the forward pass of a problem with nonlinear constraints:

$$\mathbf{x}^\star(\mathbf{c}) = \underset{\mathbf{0}\leq\mathbf{x}}{\text{argmax}}\, \mathbf{c}^T\mathbf{x} \quad s.t.\ \mathbf{x}^T\mathbf{V}\mathbf{x} \leq \gamma, \ \sum \mathbf{x} = 1. \tag{4.27}$$

This model describes a risk-constrained portfolio optimization where $\boldsymbol{V}$ is a covariance matrix, and the predicted cost coefficients $\mathbf{c}$ represent assets prices [52]. A 5-layer ReLU network is used to predict future prices $\boldsymbol{c}$ from exogenous feature data, and trained to minimize regret (the difference in profit between optimal portfolios under predicted and ground-truth prices) by integrating Problem (4.27). The folded *f-SQP* layer used for this problem employs Gurobi QCQP solver in its forward pass. This again highlights the ability of `fold-opt` to accommodate a highly optimized blackbox solver. Figure 4.4(c) shows test set regret throughout training, three synthetically generated datasets of different nonlinearity degrees. Notice the accuracy improvements of `fold-opt` over `cvxpy`.

Such dramatic differences can be explained by non-negligible errors made in `cvxpy`'s forward pass optimization on some problem instances, which occurs regardless of error tolerance settings. In contrast, Gurobi agrees to machine precision with a custom SQP solver, and solves about 50% faster than `cvxpy`. This shows the importance of highly accurate optimization solvers for accurate end-to-end training.

4.1.6 Conclusions

This section introduced folded optimization, a framework for generating analytically differentiable optimization solvers based on unrolled implementations. Theoretically, folded optimization was justified by a novel analysis of unrolling at a precomputed optimal solution, which showed that its backward pass is equivalent to solution of a solver's differential fixed-point conditions, specifically by fixed-point iteration on the resulting linear system. This allowed for the convergence analysis of the backward pass of unrolling, and evidence that the backpropagation of unrolling can be improved by using superior linear system solvers. We showed that folded optimization offers substantial advantages over existing differentiable optimization frameworks, including modularization of the forward and backward passes and the ability to handle nonconvex optimization.

4.2 Differentiable Model Selection Layers for Ensemble Learning

Model selection involves the process of identifying the most suitable models from a set of candidates for a given learning task. The chosen model should ideally generalize well to unseen data, with the complexity of the model playing a crucial role in this selection process. However, striking a balance between underfitting and overfitting is a significant challenge.

A variety of techniques have been presented in the machine learning literature to address this issue. Of particular relevance, *ensemble learning* [157] is a meta-algorithm that combines the outputs of individually pre-trained models, known as *base learners*, to improve overall performance. Despite being trained to perform the same task, these base learners may exhibit error diversity, meaning they fail on different samples, and their accuracy profiles complement each other across an overall distribution of test samples. The potential effectiveness of an ensemble model strongly depends on the correlation between the base learners' errors across input samples and their accuracy; those with higher accuracy and error diversity have a higher potential for improved ensemble accuracy [107].

However, the task of identifying the optimal aggregation of ensemble model predictions for any particular input sample is nontrivial. Traditional approaches often aggregate predictions across all base learners of an ensemble, aiming to make predictions more robust to the error of individual base learners. While these techniques could be enhanced by selectively applying them to a subset of base learners known to be more reliable on certain inputs, the design of algorithms that effectively select and combine the base learners' individual predictions remains a complex endeavor. Many consensus rule-based methods apply aggregation schemes that combine or exclude base learners' predictions based on static rules, thereby missing an opportunity to inform the ensemble selection based on a particular input's features.

Recently, the concept of differentiable model selection has emerged, aiming to incorporate the model selection process into the training process itself [46, 133, 59]. This approach leverages gradient-based methods to optimize model selection, proving particularly beneficial in areas like neural architecture search. The motivation behind differentiable model selection lies in the potential to automate and optimize the model selection process, thereby leading to superior models and more efficient selection procedures. Despite its promises, however, it remains non-trivial how to design effective differentiable model selection strategies and the use of gradient-based methods alone further enhances the risk of converging to local optima which can lead to suboptimal model selection.

In light of these challenges, this section proposes a novel framework for differentiable model selection specifically tailored for ensemble learning. The framework integrates machine learning and combinatorial optimization to learn the selection of ensemble members by modeling the selection process as an optimization problem leading to optimal selections within the prescribed context.

Contributions. In more detail, this section makes the following contributions: **(1)** It proposes *end-to-end Combinatorial Ensemble Learning (e2e-CEL)*, a novel ensemble learning framework that exploits an integration of ML and combinatorial optimization to learn specialized consensus rules *for a particular input sample*. **(2)** It shows how to cast the selection and aggregation of ensemble base learner predictions as a differentiable optimization problem, which is parameterized

by a deep neural network and trained end-to-end within the ensemble learning task. **(3)** An analysis of challenging learning tasks demonstrates the strengths of this idea: e2e-CEL outperforms models that attempt to select individual ensemble members, such as the optimal weighted combination of the individual ensemble members' predictions as well as conventional consensus rules, implying a much higher ability to leverage error diversity.

These results demonstrate the integration of constrained optimization and learning to be a key enabler to enhance the effectiveness of model selection in machine learning tasks.

4.2.1 Setting and Goals

The section considers *ensembles* as a collection of n models or *base learners* represented by functions $f_i, 1 \leq i \leq n$, trained independently on separate (but possibly overlapping) datasets $(\mathcal{X}_i,\mathcal{Y}_i)$, all on the same intended *classification* task. On every task studied, it assumed that $(\mathcal{X}_i,\mathcal{Y}_i)$ are given, along with a prescription for training each base learners, so that f_i are assumed to be pre-configured. This setting is common in federated analytic contexts, where base learners are often trained on diverse datasets with skewed distributions [77], and in ML services, where providers offer a range of pre-trained models with different architectural and hyper-parametrization choices [127].

Let $n \in \mathbb{N}$ be the number of base learners, $c \in \mathbb{N}$ the number of classes and $d \in \mathbb{N}$ the input feature size. Given a sample $z \in \mathbb{R}^d$, each base learner $f_j \colon \mathbb{R}^d \to \mathbb{R}^c$ computes $f_j(z) = \hat{y}_j$. For the classification tasks considered in this work, each $\hat{y}_j$ is the direct output of a *softmax* function $\mathbb{R}^c \to \mathbb{R}^c$,

$$\texttt{softmax}(c)_i = \frac{e^{c_i}}{\sum_{k=1}^{c} e^{c_k}}. \tag{4.28}$$

Explicitly, each classifier $f_i(\phi_i, x)$ is trained with respect to its parameters ϕ_i to minimize a classification loss $\mathcal{L}$ as

$$\min_{\phi_i} \mathbb{E}_{(x,y)\sim(\mathcal{X}_i,\mathcal{Y}_i)} \left[\mathcal{L}(f_i(\phi_i, x), y)\right]. \tag{4.29}$$

The goal is then to combine the base learners into an *ensemble*, whose aggregated classifier g performs the same task, but with greater overall accuracy on a *master dataset* $(\mathcal{X},\mathcal{Y})$, where $\mathcal{X}_i \subset \mathcal{X}$ and $\mathcal{Y}_i \subset \mathcal{Y}$ for all i with $0 \leq i \leq n$:

$$\min_{\theta} \mathbb{E}_{(x,y)\sim(\mathcal{X},\mathcal{Y})} \left[\mathcal{L}(g(\theta, x), y)\right]. \tag{4.30}$$

As is typical in ensemble learning, the base learners may be trained in a way that increases test-error diversity among f_i on $\mathcal{X}$ — see Section 4.2.3. In each dataset there is an implied train/test/validation split, so that evaluation of a trained model is always performed on its test portion. Where this distinction is needed, the symbols $\mathcal{X}_{\texttt{train}}$, $\mathcal{X}_{\texttt{valid}}$, $\mathcal{X}_{\texttt{test}}$ are used. A list of symbols used in the section to describe various aspects of the computation, along with their meanings is provided in [92], Table 4.

4.2.2 End-to-end Combinatorial Ensemble Learning

Ideally, given a pretrained ensemble $f_i, 1 \leq i \leq n$ and a sample $z \in \mathcal{X}$, one would select from the ensemble a classifier which is known to produce an accurate class prediction for z. However,

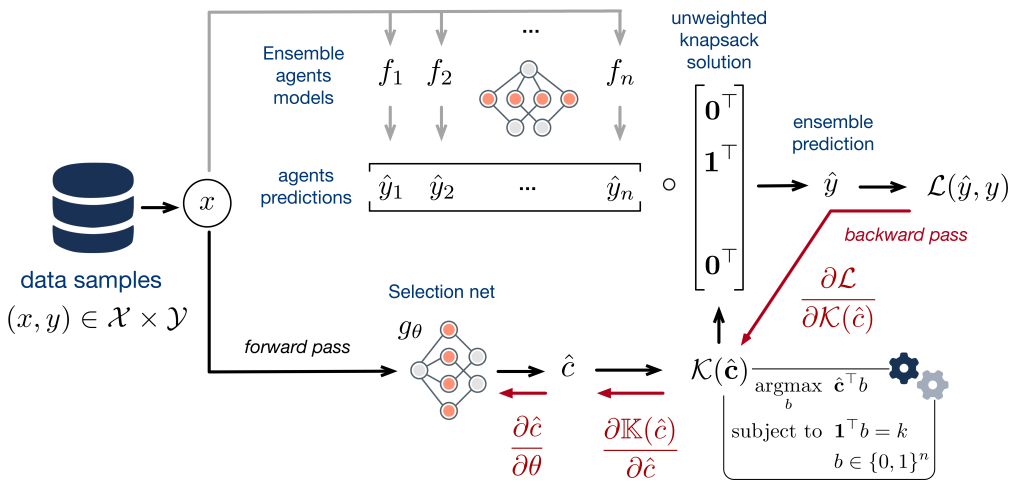


Figure 4.5: End-to-end Ensemble Learning scheme: Black and red arrows illustrate forward and backward operations, respectively.

a performance assessment for each base learners' predictions is not available at test time. Thus, conventional ensemble learning schemes resort to selection criteria such as *plurality voting* (see Section 4.2.3 for a description of the aggregation rules here used as a benchmark).

The end-to-end learning scheme in this work is based on the idea that a more accurate ensemble prediction can be made by using predictions based on z, and that selecting a well-chosen subset of the ensemble, rather than the entire ensemble, can provide more reliable results than a single base learner. The size of the subset, k, is treated as a hyperparameter. While it may seem logical to only choose the best predicted base learner for a given input sample (setting $k = 1$), it is consistently observed in Section 4.2.3 that the optimal performance is achieved for $1 < k < n$.

The proposed mechanism casts the sub-ensemble selection as an optimization program that is end-to-end differentiable and can thus be integrated with a learning model g_θ to select a reliable subset of the ensemble base learners to combine for predictions. An *end-to-end Smart Selection Ensemble* (e2e-SSE), or simply, *smart ensemble*, consists of an ensemble of base learners along with a module that is trained by e2e-CEL to select the *sub-ensemble* of size k, which produces the most accurate combined prediction for a given input. The model g is called the *selection net*, and the end-to-end ensemble model is trained by optimizing its parameters θ.

E2e-CEL overview. E2e-CEL is composed of three main steps:

1. Predict a vector of scores $g_\theta(z) = \hat{c}$, estimating the prediction accuracy for each base learner on sample z.
2. Identify the base learner indices $\mathcal{E} \subset [n]$ which correspond the the top k predicted scores.
3. Collect the predictions of the selected sub-ensemble $f_j(z)$ and perform an approximate majority voting scheme over those predictions to determine the z's class.

By training on the master set $\mathcal{X}_{\texttt{train}}$, the smart ensemble learns to make better predictions by virtue of learning to select better sub-ensembles to vote on its input samples. However, note that subset selection and plurality voting are discrete operations, and in plain form do not offer useful gradients for backpropagation and learning. The next sections discuss further details of the e2e-CEL

framework, including differentiable approximations for each step of the overall model.

Figure 4.5 illustrates the e2e-CEL model and its training process in terms of its component operations. Backpropagation is shown with red arrows, and it only applies to the operations downstream from the selection net g, since the e2e-CEL is parameterized by the parameters of g alone.

Differentiable Model Selection

The e2e-CEL system is based on learning to select $k < n$ predictions from the master ensemble, given a set of input features. This can be done by way of a structured prediction of binary values, which are then used to mask the individual base learner predictions.

Consider the unweighted knapsack problem

$$\mathcal{K}(\hat{c}) = \underset{b}{\text{argmax}} \quad \hat{c}^\top b \tag{4.31a}$$

$$\text{subject to} \quad \mathbf{1}^\top b = k, \tag{4.31b}$$

$$b \in \{0,1\}^n, \tag{4.31c}$$

which can be viewed as a selection problem whose optimal solution assigns the value 1 to the elements of b associated to the top k values of $\hat{c}$. Relaxing constraint (4.31c) to $0 \leq b \leq 1$ results in an equivalent linear program (LP) with discrete optimal solutions $b \in \{0,1\}^n$, despite being both convex and composed of continuous functions. This useful property holds for any LP with totally unimodular constraints and integer right-side coefficients [14].

This optimization problem can be viewed as a mapping from $\hat{c}$ to a binary vector indicating its top k values, and represents thus a natural candidate for selection of the optimal sub-ensemble of size k given the individual base learners' predicted scores, seen as $\hat{c}$. However, the outputs of Problem (4.31) define a piecewise constant function, $\mathcal{K}(\hat{c})$, which does not admit readily informative gradients, posing challenges to differentiability. For integration into the end-to-end learning system, the function $\mathcal{K}(\hat{c})$ must provide informative gradients with respect to $\hat{c}$. In this work, this challenge is overcome by smoothing $\mathcal{K}(\hat{c})$ based on perturbing $\hat{c}$ with random noise.

As observed by [20], any continuous, convex linear programming problem can be used to define a *differentiable perturbed optimizer*, which yields approximately the same solutions but is differentiable with respect to its linear objective coefficients. Given a random noise variable Z with probability density $p(z) \propto \exp(-v(z))$ where v is a twice differentiable function,

$$\mathbb{K}(\hat{c}) = \mathbb{E}_{z\sim Z} \quad [\mathcal{K}(\hat{c} + \epsilon z)], \tag{4.32}$$

is a differentiable perturbed optimizer associated to $\mathcal{K}$. The temperature parameter $\epsilon > 0$ controls the sensitivity of its gradients (or properly, Jacobian matrix), which can itself be represented by the expected value [1]:

$$\frac{\partial \mathbb{K}(\hat{c})}{\partial \hat{c}} = \mathbb{E}_{z\sim Z} \quad [\mathcal{K}(\hat{c} + \epsilon z) \;\; v'(z)^\top]. \tag{4.33}$$

In this work, Z is modeled as a standard Normal random variable. While these expected values are analytically intractable (due to the constrained argmax operator within the knapsack problem $\mathcal{K}$), they can be estimated to arbitrary precision by sampling in Monte Carlo fashion. This procedure is a generalization of the Gumbel Max Trick [66].

Note that simulating Equations (4.32) and (4.33) requires solving Problem (4.31) for potentially many values of z. However, although the theory of perturbed optimizers requires the underlying problem to be a linear program, only a blackbox implementation is required to produce $\mathcal{K}(\hat{c})$ [20], allowing for an efficient algorithm to be used in place of a (more costly) LP solver. The complexity of evaluating the differentiable perturbed optimizer $\mathbb{K}(\hat{c})$ is discussed next.

Theorem 5. *The total computation required for solving Problem* (4.31) *is* $O(n \log k)$*, where* n *and* k *are, respectively, the ensemble and sub-ensembles sizes.*

Proof. This result relies on the observation that $\mathcal{K}(\hat{c})$ can be computed efficiently by identifying the top k values of $\hat{c}$ in $O(n \log k)$ time using a divide-and-conquer technique. See, for example, [35]. □

Generating m such solutions for gradient estimation then requires $O(m\, n \log k)$ operations. Note, however, that these can be performed in parallel across samples, allowing for sufficient noise samples to be generated for computing accurate gradients, especially when GPU computing is available.

For clarity, note also that the function $\mathcal{K}$, as a linear program mapping, has a discrete output space since any linear program takes its optimal solution at a vertex of its feasible region [14], which are finite in number. As such, it is a piecewise constant function and is differentiable except on a set of measure zero [58]. However, $\partial\mathcal{K}/\partial\hat{c} = 0$ everywhere it is defined, so the derivatives lack useful information for gradient descent [154]. While $\partial\mathbb{K}/\partial\hat{c}$ is not the true derivative of $\mathcal{K}$ at $\hat{c}$, it supplies useful information about its direction of descent.

In practice, the forward optimization pass is modeled as $\mathcal{K}(\hat{c})$, and the backward pass is modeled as $\partial\mathbb{K}(\hat{c})/\partial\hat{c}$. This allows further downstream operations, and their derivatives, to be evaluated at $\mathcal{K}(\hat{c})$ without approximation, which improves training and test performance [89]. These forward and backward passes together are henceforth referred to as the *Knapsack Layer*. Its explicit backward pass is computed as

$$\frac{\partial \mathbb{K}(\hat{c})}{\partial \hat{c}} \approx \frac{1}{m} \sum_{i=1}^{m} \left[\mathcal{K}(\hat{c} + \epsilon z_i) \;\; v'(z_i)^\top \right], \tag{4.34}$$

where $z_i \sim \mathcal{N}(0,1)^n$ are m independent samples each drawn from a standard normal distribution.

Combining Predictions

Denote as $P \in \mathbb{R}^{c \times n}$ the matrix whose $j-th$ column is the softmax vector $\hat{y}_j$ of base learner j,

$$P = \begin{pmatrix} \hat{y}_1 & \hat{y}_2 \ldots \hat{y}_n. \end{pmatrix} \tag{4.35}$$

For the purpose of combining the ensemble agent predictions, $\mathcal{K}(\hat{c})$ is treated as a binary masking vector $b \in \{0,1\}^n$, which selects the subset of agents for making a prediction. Denote as $B \in \{0,1\}^{c \times n}$ the matrix whose i^{th} column is $B_i = \overrightarrow{\mathbf{1}}\, b_i$; i.e.,

$$B = \begin{bmatrix} b_1 & \ldots & b_n \end{bmatrix}^\top.$$

This matrix is used to mask the agent models' softmax predictions P by element-wise multiplication. Next, define

$$\begin{aligned} P_k &= B \circ P \\ &= \begin{bmatrix} b_1 & \dots & b_n \end{bmatrix}^\top \circ \begin{bmatrix} \hat{y}_1 & \dots & \hat{y}_n \end{bmatrix} \end{aligned} \tag{4.36}$$

Doing so allows to compute the sum of predictions over the selected sub-ensemble $\mathcal{E}$, but in a way that is automatically differentiable, that is:

$$\hat{v} \coloneqq \sum_{i \in \mathcal{E}} \hat{y}_i = \sum_{i=1}^{n} P_k^{(i)}. \tag{4.37}$$

The e2e-CEL prediction comes from applying softmax to this sum:

$$\hat{y} = \texttt{softmax}(\hat{v}) = \texttt{softmax}\left(\sum_{i=1}^{n} P_k^{(i)} \right), \tag{4.38}$$

viewing the softmax as a smooth approximation to the argmax function as represented with one-hot binary vectors. This function is interpreted as a smoothed majority voting to determine a class prediction: given one-hot binary class indicators h_i, the majority vote is equal to $\text{argmax}(\sum_i h_i)$. An illustration of the process is given in Figure 4.5.

At test time, class predictions are calculated as

$$\underset{1 \le i \le c}{\text{argmax}} \quad \hat{y}_i(x). \tag{4.39}$$

Combining predictions in this way allows for an approximated majority voting over a selected sub-ensemble, but in a differentiable way so that selection net parameters θ can be directly trained to produce selections that minimize the classification task loss, as detailed in the next section.

Learning Selections

The smart ensemble mechanism learns accurate class predictions by learning to select better subensembles to vote on its input samples. In turn, this is done by predicting better coefficients $\hat{c}$ which parameterize the Knapsack Layer.

The task of predicting $\hat{c}$ based on input z is itself learned by the *selection net*, a neural network model g so that $\hat{c} = g(z)$. Since g acts on the same input samples as each f_i, it should be capable of processing inputs from z at least as well as the base learners' models; in Section 4.2.3, the selection net in each experiment uses the same CNN architecture as that of the base learner models. Its predicted values are viewed as scores over the ensemble members, rather than over the possible classes. High scores correspond to base learners which are well-qualified to vote on the sample in question.

In practice, the selection net's predictions $\hat{c}$ are normalized before input to the mapping $\mathcal{K}$:

$$\hat{c} \leftarrow \frac{\hat{c}}{\|\hat{c}\|_2}. \tag{4.40}$$

This has the effect of standardizing the magnitude of the linear objective term in (4.31a), and tends to improve training. Since scaling the objective of an optimization problem has no effect on its solution, this is equivalent to standardizing the relative magnitudes of the linear objective and random noise perturbations in Equations (4.32) and (4.33), preventing ϵ from being effectively absorbed into the predicted $\hat{c}$.

For training input x, let $\hat{y}_\theta(x)$ represent the associated e2e-SSE prediction given the selection net parameters θ. During training, the model minimizes the classification loss between these predictions and the ground-truth labels:

$$\min_\theta \mathbb{E}_{(x,y)\sim(\mathcal{X},\mathcal{Y})} \left[\mathcal{L}(\hat{y}_\theta(x), y)\right]. \tag{4.41}$$

Generally, the loss function $\mathcal{L}$ is chosen to be the same as the loss used to train the base learner models, as the base learners are trained to perform the same classification task.

e2e-CEL Algorithm Details

Algorithm 2 summarizes the e2e-CEL procedure for training a selection net. Note that only the parameters of the selection net are optimized in training, and so only its downstream computations are backpropagated. This is done by the standard automatic differentiation employed in machine learning libraries [121], except in the case of the Knapsack Layer, whose gradient transformation is analytically specified by Equation (4.34).

For clarity, Algorithm 2 is written in terms of operations that apply to a single input sample. In practice, however, minibatch gradient descent is used. Each pass of the training begins evaluating the base learner models (line 4) and sampling standard Normal noise vectors (line 5). The selection net predicts from input features x a vector of base learner scores $g_\theta(x)$, which defines an unweighted knapsack problem $\mathcal{K}(g_\theta(x))$ that is solved to produce the binary mask b (line 6).

Masking is applied to the base learner predictions before being summed and softmaxed for a final ensemble prediction $\hat{y}$ (line 8). The classification loss $\mathcal{L}$ is evaluated with respect to the label y and backpropagated in 3 steps: **(1)** The gradient $\frac{\partial \mathcal{L}}{\partial b}$ is computed by automatic differentiation backpropagated to the Knapsack Layer's output (line 9). **(2)** The chain rule factor $\frac{\partial b}{\partial \hat{c}}$ is analytically computed by the methodology of Section 4.2.2 (line 10). **(3)** The remaining chain rule factor $\frac{\partial \hat{c}}{\partial \theta}$ is computed by automatic differentiation (line 11). Note that as each chain rule factor is computed, it is also applied toward computing $\frac{\partial \mathcal{L}}{\partial \theta}$ (line 12). Finally, a SGD step [130] or one of its variants ([41], [165]) is applied to update θ (line 13).

The next section evaluates the accuracy of ensemble models trained with this algorithm, on classification tasks using deep neural networks.

4.2.3 e2e-CEL Evaluation

The e2e-CEL training is evaluated on several vision classification tasks: digit classification on *MNIST dataset* [37], age-range estimation on *UTKFace dataset* [167], image classification on *CIFAR10 dataset* [96], and emotion detection on *FER2013* dataset [99].

Being an optimized aggregation rule, e2e-CEL is compared with state-of-the-art Super Learner algorithm [76] along with the following widely adopted baseline aggregation rules when paired with a pre-trained ensemble :

Algorithm 2: *Training the Selection Net*

input : $\mathcal{X}, \mathcal{Y}, \alpha, k, m, epsilon$

for *epoch* $k = 0, 1, \ldots$ **do**

  foreach $(x, y) \leftarrow (\mathcal{X}, \mathcal{Y})$ **do**

    $\hat{y}_i \leftarrow f_i(x) \qquad \forall\ 1 \le i \le n$

    $z_i \sim \mathcal{N}(0,1)^n \quad \forall\ 1 \le i \le m$

    $(b, \hat{c}) \leftarrow \left(\mathcal{K}(g_\theta(x)), \frac{g_\theta(x)}{\|g_\theta(x)\|_2}\right)$

    $P_k \leftarrow [b, \ldots, b]^\top \circ [\hat{y}_1, \ldots, \hat{y}_n]$

    $\hat{p} \leftarrow \texttt{softmax}(\sum_{i=1}^{n} P_k^{(i)})$

    $\partial \mathcal{L}(\hat{p}, y) / \partial b \leftarrow \texttt{autodiff}$

    $\partial b / \partial \hat{c} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \left[\mathcal{K}(\hat{c} + \epsilon z_i) \ \ v'(z_i)^\top\right]$

    $\partial \hat{c} / \partial \theta \leftarrow \texttt{autodiff}$

    $\partial \mathcal{L}(\hat{p}, y) / \partial \theta \leftarrow \frac{\partial \mathcal{L}(\hat{p}, y)}{\partial b} \cdot \frac{\partial b}{\partial \hat{c}} \cdot \frac{\partial \hat{c}}{\partial \theta}$

    $\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}(\hat{p}, y)}{\partial \theta}$

- *Super Learner*: a fully connected neural network that, given the base learners' predictions, learns the optimal weighted combinations specialized for any input sample.
- *Unweighted Average*: averages all the base learners' softmax predictions and then compute the index of the corresponding highest label score as the final prediction.
- *Plurality Voting*: makes a discrete class prediction from each base learner and then returns the most-predicted class.
- *Random Selection*: randomly selects a size-k sub-ensemble of base learners for making prediction and then applies the unweighted average rule to the selected base learners' soft predictions.

Experimental settings. Ensemble learning schemes are most effective when base learner models are accurate and have high error diversity. In this work, base learners are deliberately trained to have high error diversity with respect to input samples belonging to different classes. This is done by composing for each base learner model f_i $(1 \le i \le n)$ a training set $\mathcal{X}_i$ in which a subset of classes is over-represented, resulting base learners that specialize in identifying those classes. The exact class composition of each dataset $\mathcal{X}_i$ depends on the particular classification task and on the base learner's intended specialization.

For each task, each base learner is designed to be specialized for recognizing either one or two particular classes. To this end, the training set of each base learner is partitioned to have a majority of samples belonging to a particular class, while the remaining part of the training dataset is uniformly distributed across all other classes by random sampling. Specifically, to compose the *smart ensemble* for each task, a single base learner is trained to specialize on each possible class, and on each pair of classes (e.g., digits 1 and 2 in MNIST). When c is the number of classes, the experimental smart ensemble then consists of $c + \binom{c}{2}$ total base learners. Training a specialized base learner in this way generally leads to high accuracy over its specialty classes, but low accuracy over all other classes. Therefore in this experimental setup, no single base learner is capable of achieving high overall accuracy on the master test set $\mathcal{X}_{\texttt{test}}$. This feature is also recurrent in federated analytic

Dataset	Accuracy (%) Specialized	Complimentary	Overall
MNIST	97.5	86.8	89.6
UTKFACE	93.2	25.2	51.2
FER2013	79.4	38.1	47.8
CIFAR10	76.3	24.8	31.1

Table 4.1: Specialized base learner model test accuracy

models [77].

Table 4.1 shows the average accuracy of individual base learner models on their specialty classes and their non-specialty classes; reported, respectively as *specialized accuracy* and *complementary accuracy*. The reported *overall* accuracy is measured over the entire master test set $\mathcal{X}_{\texttt{test}}$. This sets the stage for demonstrating the ability of e2e-CEL training to compose a classifier that substantially outperforms its base learner models on $\mathcal{X}_{\texttt{test}}$ by adaptively selecting sub-ensembles based on input features; see Section 4.2.3.

Note that, in each experiment, the base learner models' architecture design, hyperparameter selection, and training methods have not been chosen to fully optimize classification accuracy, which is not the direct goal of this work. Instead, the base learners have been trained to maximize error diversity, and demonstrate the ability of e2e-CEL to leverage error diversity and compose highly accurate ensemble models from far less accurate base learner models, in a way that is not shared by conventional aggregation rules. Note also that improving base learner model accuracies would, of course, tend to improve the accuracy of the resulting ensemble classifiers. In each case, throughout this section, the e2e-SSE selection net is given the same CNN architecture as the base learner models which form its ensemble.

Datasets and Settings

For each task, the base learners are trained to specialize in classifying one or two particular classes, which allows the selection program to leverage their error diversity. Additional details about the base learners' models and the dataset split can be found in [92], Appendix B.

Digit classification. MNIST is a dataset of 28x28 pixel greyscale images of handwritten digits. It contains 60000 images for training and 10000 images for testing. The ensemble consists of 55 base learners, 10 of which specialize on one class and $\binom{10}{2} = 45$ of which specialize on two classes.

Image classification. CIFAR10 is a 32x32 pixel color images dataset in 10 classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. It contains 6000 images of each class. The ensemble consists of 55 base learners, 10 of which specialize on 1 class and $\binom{10}{2} = 45$ of which specialize on 2 classes.

	Accuracy (%)				
Dataset	**e2e-CEL**	**SL**	**UA**	**PV**	**RS**
MNIST	**98.55**	96.88	96.81	95.99	96.83
UTKFACE	**90.97**	85.07	84.60	80.78	84.60
FER2013	**66.31**	64.95	63.89	63.15	63.89
CIFAR10	**64.09**	60.13	60.59	60.35	60.59

Table 4.2: e2e-CEL vs super learner (SL), unweighted average (UA), plurality voting (MV), and random selection (RS), using specialized base learners.

Age estimation. UTKFace is a face images dataset consisting of over 20000 samples and different version of images format. Here 9700 cropped and aligned images are split in 5 classes: baby (up to 5 years old), teenager (from 6 to 19), young (from 20 to 33), adult (from 34 to 56) and senior (more than 56 years old). The classes are not uniformly distributed per number of ages, but each class contains the same number of samples. The goal is to estimate a person's age given the face image. The ensemble consists of 15 base learners, 5 of which specialize on 1 class and $\binom{5}{2} = 10$ on 2 classes.

Emotion detection. Fer2013 is a dataset of over 30000 48x48 pixel grayscale face images, which are grouped in 7 classes: angry, disgust, fear, happy, neutral, sad, and surprised. The goal is to categorize the emotion shown in the facial expression into one category. The ensemble consists of 21 base learners, 7 of which specialize on 1 class and $\binom{7}{2} = 21$ of which specialize on 2 classes.

e2e-CEL Analysis

The *e2e-CEL* strategy is tested on each experimental task for sub-ensemble size k varying between 1 and n, and compared to the baseline methods described above. Note in each case that accuracy is defined as the percentage of correctly classified samples over the master test set.

Table 4.2 reports the best accuracy over all the ensemble sizes k of ensembles trained by e2e-CEL along with that of each baseline ensemble model, where each are formed using the same pre-trained base learners. Note how the proposed e2e-CEL scheme outperforms all the baseline methods, in each task, for all but the lowest values of k.

Figure 4.6 reports the test accuracy found by e2e-CEL along with ensembles based on the Super Learner, weighted average, majority voting, or random selection scheme. We make two key observations: **(1)** Note from each subplot in Figure 4.6 that smart ensembles of size $k > 1$ provide more accurate predictions than baseline models that randomly select sub-ensembles of the same size, a trend that diminishes as k increases and base learner selections have less consequence (the two perform equally when $k = n$). **(2)** In every case, the sub-ensemble size which results in optimal performance is strictly between 1 and n. *Importantly, this illustrates the motivating intuition of the e2e-CEL ensemble training. Neither the full ensemble ($k = n$), nor smart selection of a single base learner model ($k = 1$) can outperform models that use smart selection of a sub-ensemble of any size.*

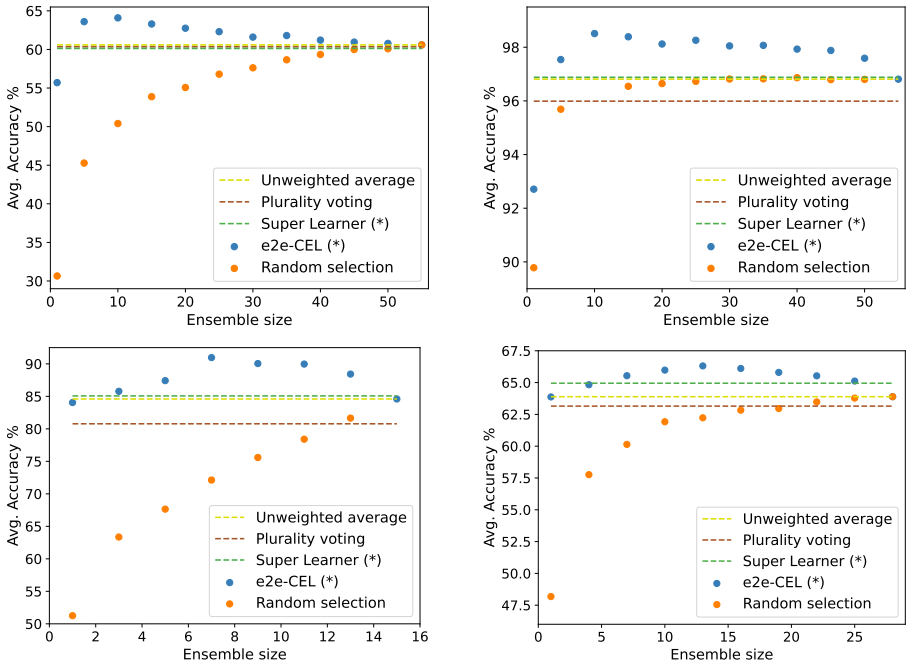


Figure 4.6: Comparison between e2e-CEL and other ensemble models at varying of the sub-ensemble size k on image classification–CIFAR10–(top left), digit classification–MNIST–(top right), age estimation–UTKFace–(bottom left), and emotion detection–FER2013–(bottom right) The (*) in the label identifies methods that use specialized aggregation rules for every input sample..

A well-selected sub-ensemble has higher potential accuracy than the master ensemble, and is, on average, more reliable than a well-selected single base learner.

Next, Table 4.3 (left) reports the accuracy of the e2e-SSE model trained on each task, along with the sub-ensemble size that resulted in highest accuracy. In two cases, for the digit classification and the image classification task, the e2e-SSE performs best when the sub-ensemble size is equal to the number of classes. In the remaining tasks, this observation holds approximately. *This is intuitive, since the number of base learners specializing on any class is equal to the number of classes, and e2e-CEL is able to increase ensemble accuracy by learning to select these base learners for prediction.*

Finally, observe the accuracy of e2e-CEL in Table 4.3 (left) and the performance of the individual base learners predictors of the ensemble tested on both the labels in which their training was specialized as well as the other labels. Note how e2e-SSE predictions outperform their constituent base learners by a wide margin on each task. For example, on UTKFace, the e2e-SSE ensemble reaches an accuracy 40 percentage points higher than its average constituent base learner. *This illustrates the ability of e2e-CEL to leverage the error diversity of base learners to form accurate classifiers by composing them based on input features*, even when the individual base learner's accuracies are poor.

Dataset	Classes	Best k	Accuracy (%) e2e-CEL	Base learners
MNIST	10	10	**98.55**	89.6
UTKFACE	5	7	**90.97**	51.2
FER2013	7	13	**66.31**	47.8
CIFAR10	10	10	**64.09**	31.1

Table 4.3: Left: Best ensemble size (Best k) and associated e2e-CEL test accuracy attained on each dataset. Right: Average accuracy for the constituent ensemble base learners.

4.2.4 Conclusions

This study addresses a significant issue in model selection and ensemble learning: determining the best models for the classification of distinct input samples. The presented solution is an innovative approach for differentiable model selection, and tailored to ensemble learning, merging machine learning with combinatorial optimization. This framework constructs precise predictions, adaptively selecting sub-ensembles based on input samples.

The study show how to transform the ensemble learning task into a differentiable selection process, trained cohesively within the ensemble learning model. This approach allows the proposed framework to compose accurate classification models even from ensemble base learners with low accuracy, a feature not shared by existing ensemble learning approaches. The results on various tasks demonstrate the versatility and effectiveness of the proposed framework, substantially outperforming state-of-the-art and conventional consensus rules in a variety of settings.

This work demonstrates that the integration of machine learning and combinatorial optimization is a valuable toolset for not only enhancing but also combining machine learning models. This work contributes to the ongoing efforts to improve the efficiency and effectiveness of model selection in machine learning, particularly in the context of ensemble learning, and hopes to motivate new solutions where decision-focused learning may be used to improve the capabilities of machine learning systems.

Chapter 5

Learning the Parameters of Optimal Decision Models

This chapter is dedicated to the Predict-Then-Optimize setting, in which an optimization problem must be solved without full knowledge of its objective function. With its numerous practical applications, this setting is a prime example to demonstrate the benefits of differentiable programming. By differentiating through the optimal solution that results from a prediction of its objective parameters, the objective value of a downstream optimization problem can be used directly as a loss function for training its prediction model.

Our contributions in this area are focused on algorithmic fairness of decisions made by combined prediction and optimization models. Section 5.1 is methodologically oriented, addressing the modeling challenges that arise when differentiating through optimizations with nondifferentiable objectives. In particular, we propose a predict-then-optimize framework for the optimization of Ordered Weighted Averaging objectives, known for their fair guarantees over multiple objective functions. This work is based on the author's publications [43, 42]. Then, Section 5.2 develops a practical application of PtO in the fairness domain, also based on published works [92, 44]. It proposes the integration of learning to rank models end-to-end with fair ranking optimization programs. This allows us to optimize web search results subject to guarantees on fairness of exposure to users, across arbitrarily defined categories of content.

5.1 Differentiable Approximations of Fair OWA Optimization

The *Predict-Then-Optimize* (PtO) framework [104] models decision-making processes as optimization problems with unspecified parameters $\boldsymbol{c}$, which must be estimated by a machine learning (ML) model, given correlated features $\boldsymbol{z}$. An estimation of $\boldsymbol{c}$ completes the problem's specification, whose solution defines a mapping:

$$\boldsymbol{x}^\star(\boldsymbol{c}) = \underset{\boldsymbol{x}\in\mathcal{S}}{\operatorname{argmax}} \quad f(\boldsymbol{x}, \boldsymbol{c}) \tag{5.1}$$

The goal is to learn a model $\hat{\boldsymbol{c}} = \mathcal{M}_\theta(\boldsymbol{z})$ from observable features $\boldsymbol{z}$, such that the objective value $f(\boldsymbol{x}^\star(\hat{\boldsymbol{c}}), \boldsymbol{c})$ under ground-truth parameters $\boldsymbol{c}$ is maximized on average. This is common in many applications requiring decision-making under uncertainty, like planning the fastest route through a city with unknown traffic delays or predicting optimal power generation schedules based on demand forecasts.

Optimization of multiple objectives is crucial in contexts requiring a balance of competing goals, especially when fairness is essential in fields like energy systems [141], urban planning [131], and multi-objective portfolio optimization [74, 30]. A common approach is using Ordered Weighted Averaging (OWA) [161] to achieve Pareto-optimal solutions that fairly balance each objective. However, optimizing an OWA objective in PtO is challenging due to its nondifferentiability, which prevents backpropagation through $\boldsymbol{x}^\star(\boldsymbol{c})$ within machine learning models trained by gradient descent. To our knowledge, no prior PtO models encounter a non-differentiable objective, making this challenge novel.

5.1.1 Preliminaries

Fair OWA and its Optimization

The *Ordered Weighted Average* (OWA) operator [161] is used in various decision-making fields to fairly aggregate multiple objective criteria [162]. Let $\mathbf{y} \in \mathbb{R}^m$ be a vector of m distinct criteria, and $\tau : \mathbb{R}^m \to \mathbb{R}^m$ be the sorting map that orders $\mathbf{y}$ in increasing order. For any $\mathbf{w}$ satisfying $\mathbf{w} \in \mathbb{R}^m$, $\sum_i w_i = 1$, and $\mathbf{w} \geq 0$, the OWA aggregation with weights $\mathbf{w}$ is piecewise-linear in $\mathbf{y}$ [117]:

$$\text{OWA}_{\mathbf{w}}(\mathbf{y}) = \mathbf{w}^T \tau(\mathbf{y}), \tag{5.2}$$

We use its concave version, *Fair OWA* [118], characterized by weights in descending order: $w_1 > \ldots > w_n > 0$.

The following three properties of Fair OWA functions are crucial for fairly optimizing multiple objectives: **(1)** *Impartiality*: Permutations of a utility vector are equivalent solutions. **(2)** *Equitability*: Marginal transfers from a higher value criterion to a lower one increase the OWA aggregated value. **(3)** *Monotonicity*: $\text{OWA}_{\mathbf{w}}(\mathbf{y})$ is an increasing function of each element of $\mathbf{y}$. This ensures that solutions optimizing the OWA objectives are Pareto Efficient, meaning no criterion can be improved without worsening another [117]. Optimization of aggregation functions that possess these properties leads to *equitably efficient solutions*, which satisfy a rigorously defined notion of fairness [87].

Predict-Then-Optimize Learning

Our problem setting fits within the PtO framework. Generally, a parametric optimization problem (5.1) models an optimal decision $\boldsymbol{x}^\star(\boldsymbol{c})$ with respect to unknown parameters $\boldsymbol{c}$ drawn from a distribution $\boldsymbol{c} \sim C$. While the true value of $\boldsymbol{c}$ is unknown, correlated feature values $\boldsymbol{z} \sim \mathcal{Z}$ can be observed. The goal is to learn a predictive model $\mathcal{M}_\theta : \mathcal{Z} \to C$ from features $\boldsymbol{z}$ to estimate problem parameters $\hat{\boldsymbol{c}} = \mathcal{M}_\theta(\boldsymbol{z})$, by maximizing the empirical objective value of the resulting solution under ground-truth parameters. That is,

$$\underset{\theta}{\text{argmax}} \quad \mathbb{E}_{(\boldsymbol{z},\boldsymbol{c})\sim\Omega}\, f\left(\boldsymbol{x}^\star(\mathcal{M}_\theta(\boldsymbol{z})), \boldsymbol{c}\right), \tag{5.3}$$

where Ω represents the joint distribution between $\mathcal{Z}$ and C.

The above training goal is often achieved by maximizing empirical *Decision Quality* as a loss function [104], defined:

$$\mathcal{L}_{DQ}(\hat{\boldsymbol{c}}, \boldsymbol{c}) = f\left(\boldsymbol{x}^\star(\hat{\boldsymbol{c}}), \boldsymbol{c}\right). \tag{5.4}$$

Gradient descent training of (5.3) with $\mathcal{L}_{DQ}$ requires a model of gradient $\frac{\partial \mathcal{L}_{DQ}}{\partial \hat{\boldsymbol{c}}}$, either directly or through chain-rule composition $\frac{\partial \mathcal{L}_{DQ}}{\partial \hat{\boldsymbol{c}}} = \frac{\partial \boldsymbol{x}^\star(\hat{\boldsymbol{c}})}{\partial \hat{\boldsymbol{c}}} \cdot \frac{\partial \mathcal{L}_{DQ}}{\partial \boldsymbol{x}^\star}$. When $\boldsymbol{x}^\star$ is not differentiable, as in OWA

optimizations, smooth approximations are required, such as those developed in the next section.

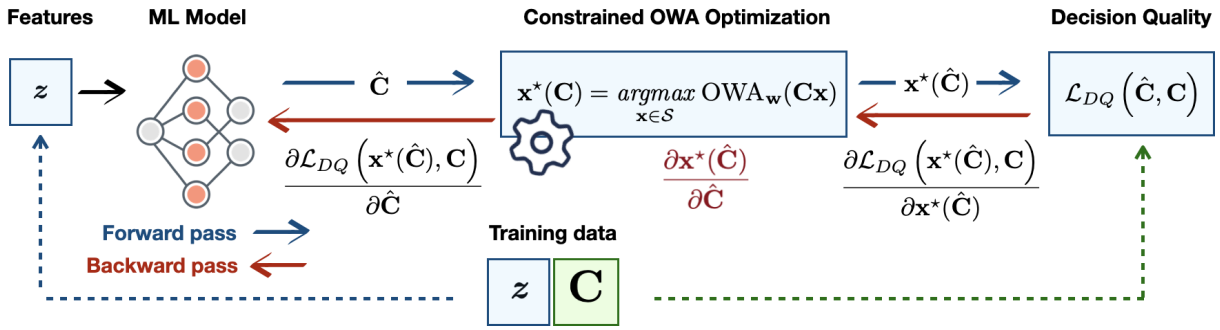


Figure 5.1: Predict-Then-Optimize for OWA Optimization.

5.1.2 End-to-End Learning with Fair OWA Optimization

This section focuses on scenarios where the objective function f is an ordered weighted average of m linear objective functions, each parameterized by a row of a matrix $\boldsymbol{C} \in \mathbb{R}^{m\times n}$ so that $f(\boldsymbol{x}, \boldsymbol{C}) = \mathrm{OWA}_{\boldsymbol{w}}(\boldsymbol{C}\boldsymbol{x})$ and

$$\boldsymbol{x}^\star(\boldsymbol{C}) = \underset{\boldsymbol{x}\in\mathcal{S}}{\mathrm{argmax}} \quad \mathrm{OWA}_{\boldsymbol{w}}(\boldsymbol{C}\boldsymbol{x}). \tag{5.5}$$

Note that this methodology extends to cases where the OWA objective is combined with additional smooth terms. For simplicity, the exposition primarily focuses on the pure OWA objective as shown in equation (5.5).

The goal is to learn a prediction model $\hat{\boldsymbol{C}} = \mathcal{M}_\theta(\boldsymbol{z})$ that maximizes decision quality through gradient descent on problem (5.3), which requires obtaining its gradients w.r.t. $\hat{\boldsymbol{C}}$:

$$\frac{\partial \mathcal{L}_{DQ}(\hat{\boldsymbol{C}}, \boldsymbol{C})}{\partial \hat{\boldsymbol{C}}} = \underbrace{\frac{\partial \boldsymbol{x}^\star}{\partial \hat{\boldsymbol{C}}}}_{\boldsymbol{J}} \cdot \underbrace{\frac{\partial \mathrm{OWA}_{\boldsymbol{w}}(\boldsymbol{C}\boldsymbol{x}^\star)}{\partial \boldsymbol{x}^\star}}_{\boldsymbol{g}}, \tag{5.6}$$

where $\boldsymbol{x}^\star$ is evaluated at $\hat{\boldsymbol{C}}$. The main strategy involves determining the OWA function's gradient $\boldsymbol{g}$ and then computing $\boldsymbol{J}\boldsymbol{g}$ by backpropagating $\boldsymbol{g}$ through $\boldsymbol{x}^\star$.

While nondifferentiable, the class of OWA functions is *subdifferentiable*, with subgradients as follows:

$$\frac{\partial}{\partial \boldsymbol{y}} \mathrm{OWA}_{\boldsymbol{w}}(\boldsymbol{y}) = \boldsymbol{w}_{(\sigma^{-1})} \tag{5.7}$$

where σ are the sorting indices on $\boldsymbol{y}$ [45]. Based on this formula, computing an overall subgradient $\boldsymbol{g} = {}^{\partial}/_{\partial \boldsymbol{x}}\, \mathrm{OWA}_{\boldsymbol{w}}(\boldsymbol{C}\boldsymbol{x})$ is a routine application of the chain rule (via automatic differentiation). We apply the differentiable approximations proposed next to enable its backpropagation through OWA optimization. A schematic illustration highlighting the forward and backward steps required for this process is provided in Figure 5.1.

5.1.3 Differentiable Approximate OWA Optimization

This section introduces two differentiable approximations of the OWA optimization mapping (5.5). Section 5.1.3 adapts a quadratic smoothing technique [154, 7] for a discontinuous linear programming model of OWA. Then, Section 5.1.3 presents an efficient alternative by employing OWA's Moreau envelope approximation. To the best of the author's knowledge, this is the first instance of using objective smoothing via the Moreau envelope as an effective technique for approximating nondifferentiable optimization programs in end-to-end learning.

OWA LP with Quadratic Smoothing

In [117], it's observed that the OWA optimization (5.5) can have the following LP formulation when $\boldsymbol{x} \in \mathcal{S}$ is linear:

$$\boldsymbol{x}^\star(\boldsymbol{C}) = \operatorname{argmax}_{\boldsymbol{x}\in\mathcal{S},\boldsymbol{y},z} \quad z \tag{5.8a}$$

$$\text{s.t.:} \quad \boldsymbol{y} = \boldsymbol{C}\boldsymbol{x} \tag{5.8b}$$

$$z \leq \boldsymbol{w}_\tau \cdot \boldsymbol{y} \quad \forall \tau \in \mathcal{P}_m. \tag{5.8c}$$

This LP problem is typically solvable with a simplex method. However, its constraints (5.8c) grows factorially as $m!$, where m is the number of criteria aggregated by OWA.

Our first approach to differentiable OWA optimization combines this LP transformation with the smoothing technique of [154], which forms differentiable approximations to linear programs by adding a scaled Euclidean norm term $\epsilon|\boldsymbol{x}|^2$ to the objective function. This results in a continuous mapping $\boldsymbol{x}^\star(\boldsymbol{c}) = \operatorname{argmax}_{\boldsymbol{A}\boldsymbol{x}\leq\boldsymbol{b}} \ \boldsymbol{c}^T\boldsymbol{x} + \epsilon\|\boldsymbol{x}\|^2$, a quadratic program (QP) which can be differentiated implicitly via its KKT conditions as in [5].

Smoothing by the scaled norm of joint variables $\boldsymbol{x}, \boldsymbol{y}, z$ in 5.8a leads to a differentiable QP approximation, viable when m is small. This optimization can be solved and differentiated using techniques from [5] or a generic differentiable optimization solver such as [4]:

$$\boldsymbol{x}^\star(\boldsymbol{C}) = \operatorname*{argmax}_{\boldsymbol{x}\in\mathcal{S},\boldsymbol{y},z} \ z + \epsilon\left(\|\boldsymbol{x}\|_2^2 + \|\boldsymbol{y}\|_2^2 + z^2\right) \tag{5.9a}$$

$$\text{subject to:} \quad (5.8b), (5.8c). \tag{5.9b}$$

While problem (5.8) does not fit the exact LP form due to its parameterized constraints (5.8b), the need for quadratic smoothing (5.9a) is illustrated experimentally in Section 5.1.4. The main *disadvantage* of this method is poor scalability in the number of criteria m, due to constraints (5.8c). Another drawback is that the transformed QP is much harder to solve than its original associated LP problems since quadratic smoothing increases the difficulty of an OWA-equivalent LP problem. These drawbacks motivate the next smoothing method, which yields a tractable optimization problem by replacing the OWA objective with a smooth approximation.

Moreau Envelope Smoothing

Instead of adding a quadratic term as in (5.9), we replace the piecewise linear function $\text{OWA}_{\boldsymbol{w}}$ in (5.5) with its Moreau envelope, defined for a convex function f as:

$$f^\beta(\mathbf{x}) = \min_{\mathbf{v}} \ f(\mathbf{v}) + \frac{1}{2\beta}\|\mathbf{v} - \mathbf{x}\|^2. \tag{5.10}$$

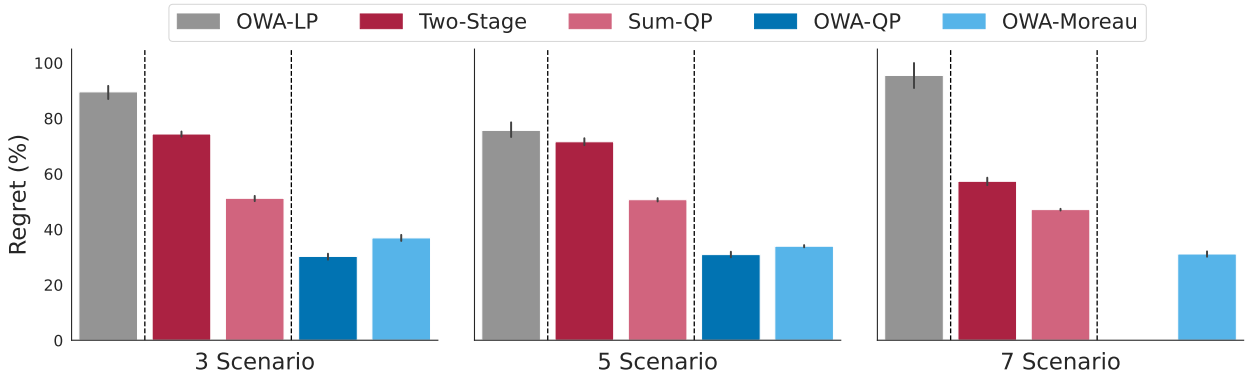


Figure 5.2: Percentage OWA regret (lower is better) on test set, on the robust portfolio problem over 3,5,7 scenarios.

Compared to its underlying function f, the Moreau envelope is $\frac{1}{\beta}$ smooth while sharing the same optima [15]. The Moreau envelope-smoothed OWA optimization problem is

$$\boldsymbol{x}^\star(\boldsymbol{C}) = \text{argmax}_{\boldsymbol{x}\in\mathcal{S}} \ \text{OWA}_{\boldsymbol{w}}^{\beta}(\boldsymbol{C}\boldsymbol{x}). \tag{5.11}$$

With its smooth objective function, problem (5.11) can be solved by gradient-based optimization methods (see Section 5.1.4), and also differentiated for backpropagation.

Differentiation of (5.11) is nontrivial since its objective function lacks a closed form. We model its Jacobian by differentiating the fixed-point conditions of a gradient-based solver. To proceed, we first note from [45] that the gradient of the Moreau envelope $\text{OWA}_{\boldsymbol{w}}^{\beta}$ is equal to a projection:

$$\frac{\partial}{\partial \boldsymbol{x}}\text{OWA}_{\boldsymbol{w}}^{\beta}(\mathbf{x}) = proj_{C(\tilde{\boldsymbol{w}})}\left(\frac{\boldsymbol{x}}{\beta}\right), \tag{5.12}$$

where $\tilde{\boldsymbol{w}} = -(w_m, \ldots, w_1)$ and the permutahedron $C(\tilde{\boldsymbol{w}})$ is the convex hull of all permutations of $\tilde{\boldsymbol{w}}$.

The following approach to differentiation of (5.11) requires differentiation of the function (5.12). For this, we leverage the differentiable permutahedral projection framework of [23], which was originaly used to implement a soft sorting model. This allows evaluation and differentiation of (5.12) in $O(m \log m)$ time, via isotonic regression.

Letting $\mathcal{U}(\boldsymbol{x}, \boldsymbol{C}) = proj_{\mathcal{S}}(\boldsymbol{x} - \alpha \cdot \frac{\partial}{\partial \boldsymbol{x}}\text{OWA}_{\boldsymbol{w}}^{\beta}(\boldsymbol{x}, \boldsymbol{C}))$, a projected gradient descent step on (5.11) is $\boldsymbol{x}^{k+1} = \mathcal{U}(\boldsymbol{x}^k, \boldsymbol{C})$. Differentiating the fixed-point conditions of convergence where $\boldsymbol{x}^k = \boldsymbol{x}^{k+1} = \boldsymbol{x}^\star$, and rearranging terms yields a linear system for $\frac{\partial \boldsymbol{x}^\star}{\partial \boldsymbol{C}}$:

$$\left(I - \underbrace{\frac{\partial \mathcal{U}(\boldsymbol{x}^\star, \boldsymbol{C})}{\partial \boldsymbol{x}^\star}}_{\boldsymbol{\Phi}}\right)\frac{\partial \boldsymbol{x}^\star}{\partial \boldsymbol{C}} = \underbrace{\frac{\partial \mathcal{U}(\boldsymbol{x}^\star, \boldsymbol{C})}{\partial \boldsymbol{C}}}_{\boldsymbol{\Psi}} \tag{5.13}$$

The partial Jacobian matrices $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}$ above can be found given a differentiable implementation of $\mathcal{U}$. This is achieved by computing the inner gradient $\frac{\partial}{\partial \boldsymbol{x}}\text{OWA}_{\boldsymbol{w}}^{\beta}(\boldsymbol{x}, \boldsymbol{C})$ via the differentiable permutahedral projection (5.12), and solving the outer projection mapping $proj_{\mathcal{S}}$ using a generic differentiable solver such as `cvxpy` [4]. As such, applying $\mathcal{U}$ at a precomputed solution $\boldsymbol{x}^\star(\boldsymbol{C})$ allows $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}$ to be extracted in PyTorch, in order to solve (5.13); this process is efficiently implemented via the `fold-opt` library [94].

5.1.4 Experiments

This section extends the PtO framework to scenarios with multiple uncertain objectives jointly learned and fairly optimized through OWA aggregation. The *Robust Markowitz Portfolio Optimization* evaluates the differentiable approximations from Section 5.1.3 against baseline methods.

The training goal is to maximize empirical decision quality with respect to their Fair OWA aggregation:

$$\mathcal{L}_{DQ}(\hat{\boldsymbol{C}}, \boldsymbol{C}) = \mathrm{OWA}_{\boldsymbol{w}}\left(\boldsymbol{C}\boldsymbol{x}^{\star}(\hat{\boldsymbol{C}})\right). \tag{5.14}$$

Evaluations Each model is evaluated based on its ability to train a model $\hat{\boldsymbol{C}} = \mathcal{M}_{\theta}(z)$ to achieve high decision quality (5.14) for the OWA-aggregated objective. Results are reported using the *regret* metric of suboptimality, whose 0 corresponds to maximum decision quality:

$$\mathit{regret}(\hat{\boldsymbol{C}}, \boldsymbol{C}) = \mathrm{OWA}^{\star}_{\boldsymbol{w}}(\boldsymbol{C}) - \mathrm{OWA}_{\boldsymbol{w}}\left(\boldsymbol{C}\boldsymbol{x}^{\star}(\hat{\boldsymbol{C}})\right) \tag{5.15}$$

where $\mathrm{OWA}^{\star}_{\boldsymbol{w}}(\boldsymbol{C})$ is the true optimal value of (5.5). This experiment evaluates the proposed differentiable approximations (5.9) and (5.11), named *OWA-QP* and *OWA-Moreau*. Two common baselines are compared against our methods: **(1)** *Two-stage*: This standard baseline for PtO (5.3) [104] trains the prediction model $\hat{\boldsymbol{C}} = \mathcal{M}_{\theta}(\boldsymbol{z})$ by minimizing MSE $\mathcal{L}_{TS}(\hat{\boldsymbol{C}}, \boldsymbol{C}) = \|\hat{\boldsymbol{C}} - \boldsymbol{C}\|^2$ without considering the downstream optimization model, used only at test time. **(2)** *Unweighted sum* (*UWS*): This baseline (Sum-QP) uses an LP model: $\boldsymbol{x}^{\star}(\boldsymbol{C}) = \mathrm{argmax}_{\boldsymbol{x}\in\mathcal{S}}\, \mathbf{1}^T(\boldsymbol{C}\boldsymbol{x})$ in end-to-end training, with quadratic smoothing [154] in Section 5.1.4.

OWA Optimization Under Uncertainty: Robust Markowitz Portfolio Problem

The classic Markowitz portfolio problem is concerned with constructing an optimal investment portfolio, given future returns $\boldsymbol{c} \in \mathbb{R}^n$ on n assets, which are unknown and predicted from exogenous data. An alternative risk-aware approach considers robustness over scenarios. In [26], m future price scenarios are represented by a matrix $\boldsymbol{C} \in \mathbb{R}^{m\times n}$, where the i^{th} row contains per-asset prices for the i^{th} scenario. Thus, an optimal allocation is modeled as:

$$\boldsymbol{x}^{\star}(\boldsymbol{C}) = \underset{\boldsymbol{x}\in\Delta_n}{\mathrm{argmax}} \quad \mathrm{OWA}_{\boldsymbol{w}}(\boldsymbol{C}\boldsymbol{x}). \tag{5.16}$$

This experiment integrates robust portfolio optimization (5.16) end-to-end with per-scenario price prediction $\hat{\boldsymbol{C}} = \mathcal{M}_{\theta}(\boldsymbol{z})$. This experiment's setting is detailed in Appendix C.1.

Results. Figure 5.2 shows average percent regret in the OWA objective over the test set (lower is better). The end-to-end training *Sum-QP* outperforms *Two-stage* approach. However, both *OWA-QP* and *OWA-Moreau* achieve substantially higher decision quality. While *OWA-QP* performs slightly better, it cannot scale past 5 scenarios, highlighting the importance of the Moreau envelope smoothing. More results on an alternate dataset can be found in Appendix C.1.

OWA-LP uses the OWA's equivalent LP as a differentiable optimization without smoothing. Grey bars show non-smoothed OWA LP results implemented with implicit differentiation in `cvxpylayers` [4]. This comparison highlights the accuracy improvement due to quadratic smoothing in OWA-QP. The poor performance of OWA subgradient training under non-smoothed OWA-LP demonstrates the necessity of the proposed approximations in Section 5.1.3.

Runtimes of the smoothed models (5.9) and (5.11) are compared in Figure 5.3. Moreau envelope smoothing keep runtimes low as m increases, while the QP approximation suffers past $m = 5$ and

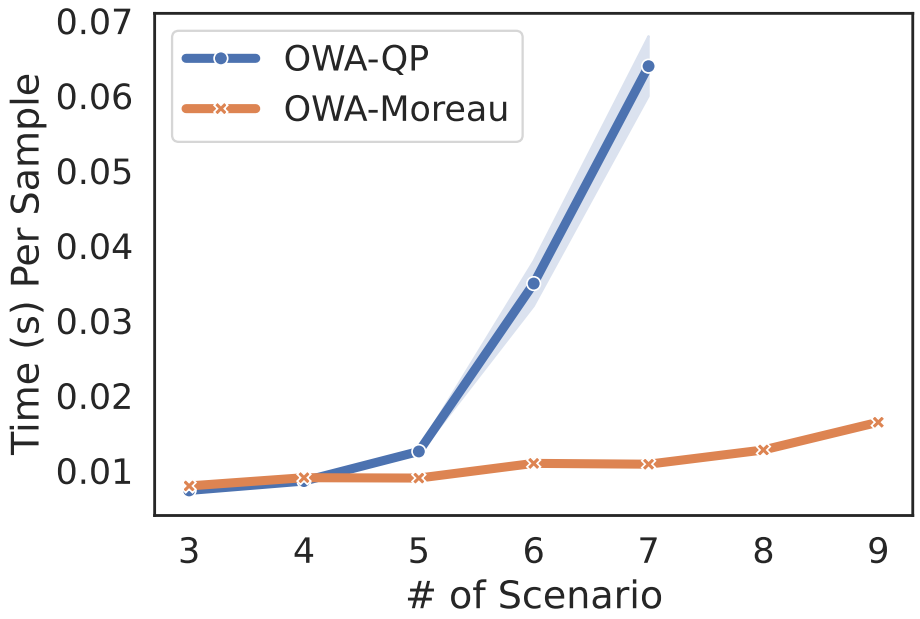


Figure 5.3: Average solving time of 2 smoothed OWA optimization models, on Robust Portfolio Optimization, over 1000 input samples. Missing data points past 7 scenarios are due to memory overflow as the QP model grows factorially.

encounters memory overflow beyond $m = 6$.

5.1.5 Conclusions

This work has presented a methodology for incorporating Fair OWA optimization end-to-end with predictive modeling. These developments were used to demonstrate the potential of Fair OWA optimization in data-driven decision making, with results not previously possible on important problems, such as robust resource allocation. Furthermore, we believe that some insights developed here may generalize to broader classes of problems - in particular, approximating an optimization of nonsmooth objectives by that of its Moreau envelope may have potential for generalization to a wider variety of problem settings.

5.2 End-to-End Learning for Fairness-Constrained Learning to Rank

Ranking systems are a pervasive aspect of our everyday lives: They are an essential component of online web searches, job searches, property renting, streaming content, and even potential friendships. In these systems, the items to be ranked are products, job candidates, or other entities associated with societal and economic benefits, and the relevance of each item is measured by implicit feedback from users (click data, dwell time, etc.). It has been widely recognized that the position of an item in the ranking has a strong influence on its exposure, selection, and, ultimately economic success.

The algorithms used to learn these rankings are typically oblivious to their potential disparate impact on the exposure of different groups of items. For example, it has been shown that in a job candidate ranking system, a small difference in relevance can incur a large difference in exposure for candidates from a minority group [51]. Similarly, in an image search engine, a disproportionate number of males may be shown in response to the query 'CEO' [134].

Ranking systems that ignore fairness considerations, or are unable to bound these effects, are prone to the "rich-get-richer" dynamics that exacerbate the disparate impacts. The resulting biased rankings can be detrimental to users, ranked items, and ultimately society. *There is thus a pressing need to design learning-to-rank (LRT) systems that can deliver accurate ranking outcomes while controlling disparate impacts.*

Current approaches to fairness in learning-to-rank systems rely on using a loss function representing a weighted combination of expected task performance and fairness. This strategy is effective in improving the fairness of predicted rankings on average, but has three key shortcomings: **(1)** The resulting rankings, even when fair in expectation across all queries, can admit large fairness disparities for some queries. This aspect may contribute to exacerbate the rich-get-richer dynamics, while giving a false sense of controlling the system's disparate impacts. **(2)** While a tradeoff between fairness and ranking utility is usually desired, these models cannot be directly controlled through the specification of an allowed magnitude for the violation of fairness. **(3)** A large hyperparameter search is required to find the weights of the loss function that deliver the desired performance tradeoff. Furthermore, each of these issues becomes worse as the number of protected groups increases.

This section addresses these issues and proposes the first fair learning to rank system–named Smart Predict and Optimize for Fair Ranking (SPOFR)–that guarantees satisfaction of fairness in the resulting rankings. The proposed framework uses a unique integration of a constrained optimization model within a deep learning pipeline, which is trained end-to-end to produce optimal fair ranking policies with respect to empirical relevance scores.

Contributions We make the following contributions: **(1)** It proposes *SPOFR*, a Fair LTR system that predicts and optimizes through an end-to-end composition of differentiable functions, guaranteeing the satisfaction of user-specified group fairness constraints. **(2)** Due to their discrete structure, imposing fairness constraints over ranking challenges the computation and back-propagation of gradients. To overcome this challenge, *SPOFR* introduces a novel training scheme which allows direct optimization of empirical utility metrics on predicted rankings using efficient back-propagation through constrained optimization programs. **(3)** The model ensures uniform fairness guarantees over all queries, a directly controllable fairness-utility tradeoff, and guarantees for multi-group fairness.

Symbol	Semantic
N	Size of the training dataset
n	Number of items to rank, for each query
$\boldsymbol{x}_q = (x_q^i)_{i=1}^n$	List of items to rank for query q
$\boldsymbol{a}_q = (a_q^i)_{i=1}^n$	protected groups associated with items x_q^i
$\boldsymbol{y}_q = (y_q^i)_{i=1}^n$	relevance scores (1-hot labels)
σ	Permutation of the list $[n]$ (individual rankings)
Π	Ranking policy
$\boldsymbol{v} = (v_i)_{i=1}^n$	Position bias vector
$\boldsymbol{w} = (w_i)_{i=1}^n$	Position discount vector

Table 5.1: Common symbols

(4) These unique aspects are demonstrated on two LTR datasets in the partial information setting. Additionally, SPOFR is shown to significantly improve on current state-of-the-art fair LTR systems with respect to established performance metrics.

5.2.1 Related Work

The imposition of fairness constraints over discrete rankings can require nontrivial optimizations. To address this challenge, multiple notions of fairness in ranking have been developed. Celis et al. [29] propose to directly require fair representation between groups within each prefix of a ranking, by specifying a mixed integer programming problem to solve for rankings of the desired form. Zehlike et al. [164] design a greedy randomized algorithm to produce rankings which satisfy fairness up to a threshold of statistical significance. The approach taken by Singh and Joachims [134] also constructs a randomized ranking policy by formalizing the ranking policy as a solution to a linear optimization problem with constraints ensuring fair exposure between groups in expectation.

Fairness in learning-to-rank is studied by Zehlike and Castillo [163], which adopts the LTR approach of Cao et al. [27] and introduces a penalty term to the loss function to account for the violation of group fairness in the top ranking position. Stronger fairness results are reported by Yadav et al. [160] and Singh and Joachims [135], which apply a policy gradient method to learn fair ranking policies. The notion of fairness is enforced by a penalty to its violation in the loss function, forming a weighted combination of terms representing fairness violation and ranking utility over rankings sampled from the learned polices using a REINFORCE algorithm [155].

5.2.2 Settings and Goals

The LTR task consists in learning a mapping between a list of n *items* and a permutation σ of the list $[n]$, which defines the order in which the items should be ranked in response to a user query. The LTR setting considers a training dataset $\boldsymbol{D} = (\boldsymbol{x}_q, \boldsymbol{a}_q, \boldsymbol{y}_q)_{q=1}^N$ where the $\boldsymbol{x}_q \in \mathcal{X}$ describe lists $(x_q^i)_{i=1}^n$ of n items to rank, with each item x_q^i defined by a feature vector of size k. The $\boldsymbol{a}_q = (a_q^i)_{i=1}^n$

elements describe protected group attributes in some domain $\mathcal{G}$ for each item x_q^i. The $\boldsymbol{y}_q \in \mathcal{Y}$ are supervision labels $(y_q^i)_{i=1}^n$ that associate a non-negative value, called *relevance scores*, with each item. Each sample $\boldsymbol{x}_q$ and its corresponding label $\boldsymbol{y}_q$ in $\boldsymbol{D}$ corresponds to a unique *query* denoted q. For example, on a image web-search context, a query q denotes the search keywords, e.g., "nurse", the feature vectors x_q^i in $\boldsymbol{x}_q$ encode representations of the items relative to q, the associated protected group attribute a_q^i may denote gender or race, and the label y_q^i describes the relevance of item i to query q.

The goal of learning to rank is to predict, for any query q, a distribution of rankings Π, called a *ranking policy*, from which individual rankings can be sampled. The utility U of a ranking policy Π for query q is defined as

$$U(\Pi, q) = \mathbb{E}_{\sigma\sim\Pi}\left[\Delta(\sigma, \boldsymbol{y}_q)\right], \tag{5.17}$$

where Δ measures the utility of a given ranking with respect to given relevance scores $\boldsymbol{y}_q$.

Let $\mathcal{M}_\theta$ be a machine learning model, with parameters θ, which takes as input a query and returns a ranking policy. The LTR goal is to find parameters θ^* that maximize the empirical risk:

$$\theta^* = \operatorname*{argmax}_\theta \frac{1}{N}\sum_{q=1}^{N} U(\mathcal{M}_\theta(\boldsymbol{x}_q), \boldsymbol{y}_q). \tag{P}$$

This description refers to the *Full-Information* setting [75], in which all target relevance scores are assumed to be known.

Fairness This section aims at learning ranking policies that satisfy group fairness. It considers a predictor $\mathcal{M}$ satisfying some group fairness notion on the learned ranking policies with respect to protected attributes $\boldsymbol{a}_q$. A desired property of fair LTR models is to ensure that, for a given query, items associated with different groups receive equal exposure over the ranked list of items. The exposure $\mathcal{E}(i, \sigma)$ of item i within some ranking σ is a function of only its position, with higher positions receiving more exposure than lower ones. Thus, similar to [135], this exposure is quantified by $\mathcal{E}(i, \sigma) = v_{\sigma_i}$, where the *position bias* vector $\boldsymbol{v}$ is defined with elements $v_j = 1/(1+j)^p$, for $j \in [n]$ and with $p > 0$ being an arbitrary power.

For ranking policy $\mathcal{M}_\theta(\boldsymbol{x}_q)$ and query q, fairness of exposure requires that, for every group indicator $g \in \mathcal{G}$, $\mathcal{M}$'s rankings are statistically independent of the protected attribute g:

$$\mathbb{E}_{\substack{\sigma\sim\mathcal{M}_\theta(\boldsymbol{x}_q)\\ i\sim[n]}}\left[\mathcal{E}(i,\sigma)\,|a_q^i = g\right] = \mathbb{E}_{\substack{\sigma\sim\mathcal{M}_\theta(\boldsymbol{x}_q)\\ i\sim[n]}}\left[\mathcal{E}(i,\sigma)\right]. \tag{5.18}$$

We consider bounds on fairness defined as the difference between the group and population level terms, i.e.,

$$\nu(\mathcal{M}_\theta(\boldsymbol{x}_q), g) = \mathbb{E}\left[\mathcal{E}(i,\sigma)|a_q^i = g\right] - \mathbb{E}\left[\mathcal{E}(i,\sigma)\right]. \tag{5.19}$$

Definition 3 (δ-fairness)**.** *A model $\mathcal{M}_\theta$ is δ-fair, with respect to exposure, if for any query $q \in [N]$ and group $g \in \mathcal{G}$:*

$$\left|\nu(\mathcal{M}_\theta(\boldsymbol{x}_q), g)\right| \leq \delta.$$

In other words, the fairness violation on the resulting ranking policy is upper bounded by $\delta \geq 0$.

Our goal is to design accurate LTR models that guarantee δ-fairness for any prescribed fairness level $\delta \geq 0$. As noted by Agarwal et al. [3] and Fioretto et al. [55], several fairness notions, including those considered in this work, can be viewed as linear constraints between the properties of each

group with respect to the population. While the above description focuses on exposure, the methods discussed here can handle any fairness notion that can be formalized as a (set of) linear constraints, including *merit weighted fairness*, introduced in Section 5.2.4. A summary of the common adopted symbols is provided in Table 5.1.

5.2.3 Learning Fair Rankings: Challenges

When interpreted as constraints of the form (5.19), fairness properties can be explicitly imposed to problem (P), resulting in a constrained empirical risk problem, formalized as follows:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \ \frac{1}{N}\sum_{q=1}^{N} U(\mathcal{M}_\theta(\boldsymbol{x}_q), \boldsymbol{y}_q) \tag{5.20a}$$

$$\texttt{s.t.} \ \ \left|\nu(\mathcal{M}_\theta(\boldsymbol{x}_q), g)\right| \leq \delta \ \ \forall q \in [N], g \in \mathcal{G}. \tag{5.20b}$$

Solving this new problem, however, becomes challenging due to the presence of constraints. Rather than enforcing constraints (5.20b) exactly, state of the art approaches in fair LTR (e.g., [134, 160]) rely on augmenting the loss function (5.20a) with a term that penalizes the constraint violations ν weighted by a multiplier λ. This approach, however, has several undesirable properties:

1. Because the constraint violation term is applied at the level of the loss function, it applies only on average over the samples encountered during training. Because the sign ($\pm$) of a fairness violation depends on which group is favored, disparities in favor of one group can cancel out those in favor of another group for different queries. This can lead to models which predict individual policies that are far from satisfying fairness in expectation, as desired. These effects will be shown in Section 5.2.6.
2. The multiplier λ must be treated as a hyperparameter, increasing the computational effort required to find desirable solutions. This is already challenging for binary groups and the task becomes (exponentially) more demanding with the increasing of the number of protected groups.
3. When a tradeoff between fairness and utility is desired, it cannot be controlled by specifying an allowable magnitude for fairness violation. This is due to the lack of a reliable relationship between the hyperparameter λ and the resulting constraint violations. In particular, choosing λ to satisfy Definition 3 for a given δ is near-impossible due to the sensitivity and unreliability of the relationship between these two values.

The proposed approach avoids these difficulties by providing an end-to-end integration of predictions and optimization into a single machine-learning pipeline, where (1) fair policies are obtained by an optimization model using the predicted relevance scores and (2) the utility metrics are back-propagated from the loss function to the inputs, through the optimization model and the predictive models. This also ensures that the fairness constraints are satisfied on *each* predicted ranking policy.

5.2.4 SPOFR

Overview. The underlying idea behind SPOFR relies on the realization that constructing an optimal ranking policy Π_q associated with a query q can be cast as a linear program (as detailed in the next section) which relies only on the relevance scores $\boldsymbol{y}_q$. The cost vector of the objective function of this program is however not observed, but can be predicted from the feature vectors x_q^i ($i \in [n]$) associated with the item list $\boldsymbol{x}_q$ to rank. The resulting framework thus operates into three steps:

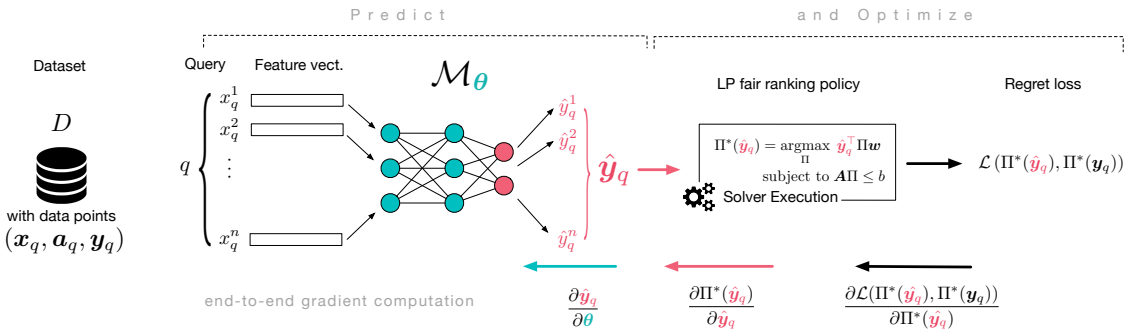


Figure 5.4: SPOFR. A single neural network learns to predict item scores from individual feature vectors, which are used to construct a linear objective function for the constrained program that produces a ranking policy.

1. First, for a given query q and its associated item list $\boldsymbol{x}_q$, a neural network model $\mathcal{M}_\theta$ is used to predict relevance scores $\hat{\boldsymbol{y}}_q = (\hat{y}_q^1, \ldots, \hat{y}_q^n)$;
2. Next, the predicted relevance scores are used to specify the objective function of a linear program whose solution will result in a *fair* optimal (with the respect to the predicted scores) ranking policy $\Pi^*(\hat{\boldsymbol{y}}_q)$;
3. Finally, a regret function, which measures the loss of optimality relative to the true optimal policy $\Pi^*(\boldsymbol{y}_q)$ is computed, and gradients are back-propagated along each step, including in the argmax operator adopted by the linear program, creating an end-to-end framework.

The overall scheme is illustrated in Figure 5.4. It is important to note that, rather than minimizing a standard error (such as a mean square loss) between the predicted quantities $\hat{\boldsymbol{y}}_q$ and the target scores $\boldsymbol{y}_q$, SPOFR minimizes directly a loss in optimality of the predicted ranking with respect to the optimal ones. Minimizing this loss is however challenging as ranking are discrete structures, which requires to back-propagate gradients through a linear program. These steps are examined in detail in the rest of this section.

While the proposed framework is general and can be applied to any linear utility metric U for rankings (see Problem (5.17)), this section grounds the presentation on a widely adopted utility metric, the *Discounted Cumulative Gain (DCG)*:

$$DCG(\sigma, \boldsymbol{y}_q) = \sum_{i=1}^{n} y_q^i w_{\sigma_i}, \tag{5.21}$$

where σ is a permutation over $[n]$, $\boldsymbol{y}_q$ are the true relevance scores, and $\boldsymbol{w}$ is an arbitrary weighting vector over ranking positions, capturing the concept of *position discount*. Commonly, and throughout this section, $w_i = 1/\log_2(1+i)$. Note that following [160, 135], these discount factors are considered distinct from the position bias factors $\boldsymbol{v}$ used in the calculation of group exposure.

Predict: Relevance Scores

Given a query q with a list of items $\boldsymbol{x}_q = (x_q^1, \ldots, x_q^n)$ to be ranked, the *predict* step uses a single fully connected ReLU neural network $\mathcal{M}_\theta$ acting on each individual item x_q^i to predict a score $\hat{y}_q^i$ ($i = 1, \ldots, n$). Combined, the predicted scores for query q are denoted with $\hat{\boldsymbol{y}}_q$ and serve as the cost vector associated with the optimization problem solved in the next phase.

Model 1: LP Computing the Fair Ranking Policy

$$\Pi^*(\hat{\boldsymbol{y}}_q) = \operatorname{argmax}_{\boldsymbol{\Pi}} \ \hat{\boldsymbol{y}}_q^\top \Pi \boldsymbol{w} \tag{5.22a}$$

$$\text{subject to:} \ \sum_j \Pi_{ij} = 1 \quad \forall i \in [n] \tag{5.22b}$$

$$\sum_i \Pi_{ij} = 1 \quad \forall j \in [n] \tag{5.22c}$$

$$0 \leq \Pi_{ij} \leq 1 \quad \forall i, j \in [n] \tag{5.22d}$$

$$|\nu(\Pi, g)| \leq \delta \quad \forall g \in \mathcal{G} \tag{5.22e}$$

Optimize: Fair Ranking Policies

The predicted relevance scores $\hat{\boldsymbol{y}}_q$, combined with the constant position discount values $\boldsymbol{w}$, can be used to form a linear function that estimates the utility metric (*DCG*) of a ranking policy. Expressing the utility metric as a linear function makes it possible to form the LTR model as an end-to-end continuous function.

Linearity of the Utility Function The following description omits subscripts "$_q$" for readability. The references below to ranking policy Π and relevance scores $\boldsymbol{y}$ are to be interpreted in relation to an underlying query q.

Using the Birkhoff–von Neumann decomposition [22], any $n \times n$ doubly stochastic matrix Π can be decomposed into a convex combination of at most $(n-1)^2 + 1$ permutation matrices $P^{(i)}$, each associated with a coefficient $\mu_i \leq 0$, which can then represent rankings $\sigma^{(i)}$ under the interpretation $\boldsymbol{w}_{\sigma^{(i)}} = P^{(i)}\boldsymbol{w}$. A ranking policy is inferred from the set of resulting convex coefficients μ_i, which sum to one, forming a discrete probability distribution: each permutation has likelihood equal to its respective coefficient

$$\Pi = \sum_{i=1}^{(n-1)^2+1} \mu_i P^{(i)}. \tag{5.23}$$

Next, note that any linear function on rankings can be formulated as a linear function on their permutation matrices, which can then be applied to any square matrix. In particular, applying the DCG operator to a doubly stochastic matrix Π results in the expected DCG over rankings sampled from its inferred policy. Given item relevance scores $\boldsymbol{y}$:

$$\begin{aligned}\mathbb{E}_{\sigma\sim\Pi} DCG(\sigma, \boldsymbol{y}) &= \sum_{i=1}^{(n-1)^2+1} \mu_i \, \boldsymbol{y}^\top P^{(i)} \, \boldsymbol{w} \\ &= \boldsymbol{y}^\top \left(\sum_{i=1}^{(n-1)^2+1} \mu_i \, P^{(i)} \right) \boldsymbol{w} = \boldsymbol{y}^\top \Pi \, \boldsymbol{w}. \qquad \text{(by Eq. (5.23))}\end{aligned}$$

The expected DCG of a ranking sampled from a ranking policy Π can thus be represented as a linear function on Π, which serves as the objective function for Model 1 (see Equation (5.22a)). This analytical evaluation of expected utility is key to optimizing fairness-constrained ranking policies in an end-to-end manner.

Importantly, and in contrast to state-of-the art methods, this approach does not require sampling from ranking policies during training in order to evaluate ranking utilities. Sampling is only required during deployment of the ranking model.

Ranking Policy Constraints Note that, with respect to *any* such linear objective function, the optimal fair ranking policy Π^* can be found by solving a linear program (LP). The linear programming model for optimizing fair ranking *DCG* functions is presented in Model 1, which follows the formulations presented in [134].

The ranking policy predicted by the SPOFR model takes the form of a doubly stochastic $n \times n$ matrix Π, in which Π_{ij} represents the marginal probability that item i takes position j within the ranking. The doubly stochastic form is enforced by equality constraints which require each row and column of Π to sum to 1. With respect to row i, these constraints express that the likelihood of item i taking any of n possible positions must be equal to 1 (Constraints (5.22b)). Likewise, the constraint on column j says that the total probability of some item occupying position j must also be 1 (Constraints (5.22c)).

For the policy implied by Π to be fair, additional *fairness constraints* must be introduced.

Fairness Constraints. Enforcing fairness requires only one additional set of constraints, which ensures that the exposures are allocated fairly among the distinct groups. The expected exposure of item i in rankings σ derived from a policy matrix Π can be expressed in terms of position bias factors $\boldsymbol{v}$ as $\mathbb{E}_{\sigma\sim\Pi}\mathcal{E}(i,\sigma) = \sum_{j=1}^{n} \Pi_{ij} v_j$. The δ-fairness of exposure constraints associated with predicted ranking policy Π and group $g \in \mathcal{G}$ becomes:

$$\left| \left(\frac{1}{|G_q^g|} \mathbb{1}_g - \frac{1}{n} \mathbb{1} \right)^\top \Pi \boldsymbol{v} \right| \leq \delta, \tag{5.24}$$

where, $G_q^g = \{i : (a_q^i) = g\}$, $\mathbb{1}$ is the vector of all ones, and $\mathbb{1}_g$ is a vector whose values equal to 1 if the corresponding item to be ranked is in G_q^g, and 0 otherwise. This definition is consistent with that of Equation (5.18). It is also natural to consider a notion of weighted fairness of exposure:

$$\left| \left(\frac{\mu}{|G_q^g|} \mathbb{1}_g - \frac{\mu_g}{n} \mathbb{1} \right)^\top \Pi \boldsymbol{v} \right| \leq \delta, \tag{5.25}$$

which specifies that group g receive exposure in proportion to the weight μ_g. In this section, where applicable and for a notion of *merit-weighted* fairness of exposure, μ_g is chosen to be the average relevance score of items in group g, while μ is the average over all items. Note that, while the above are natural choices for fairness in ranking systems, any linear constraint can be used instead.

Regret Loss and SPO Training

The training of the end-to-end fair ranking model uses a loss function that minimizes the *regret* between the exact and approximate policies, i.e.,

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \boldsymbol{y}^\top \Pi^*(\boldsymbol{y})\, \boldsymbol{w} - \boldsymbol{y}^\top \Pi^*(\hat{\boldsymbol{y}})\, \boldsymbol{w}. \tag{5.26}$$

To train the model with stochastic gradient descent, the main challenge is the back-propagation through Model (1), i.e., the final operation of our learnable ranking function. It is well-known that a parametric linear program with fixed constraints is a nonsmooth mapping from objective coefficients to optimal solutions. Nonetheless, effective approximations for the gradients of this mapping can be found [52] (see also [89] for a review on the topic). Consider the optimal solution to a linear

Algorithm 3: *Training the Fair Ranking Function*

input : $D, \alpha, \boldsymbol{w}$: Training Data, Learning Rate, Position Discount.

for *epoch* $k = 0, 1, \ldots$ **do**
  foreach $(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{y}) \leftarrow D$ **do**
    $\hat{\boldsymbol{y}} \leftarrow \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x})$
    $\Pi_1 \leftarrow \Pi^*(\boldsymbol{y}^\top \boldsymbol{w})$ by Model 1
    $\Pi_2 \leftarrow \Pi^*(2\hat{\boldsymbol{y}}^\top \boldsymbol{w} - \boldsymbol{y}^\top \boldsymbol{w})$ by Model 1
    $\nabla \mathcal{L}(\boldsymbol{y}^\top \boldsymbol{w}, \hat{\boldsymbol{y}}^\top \boldsymbol{w}) \leftarrow \Pi_2 - \Pi_1$
    $\theta \leftarrow \theta - \alpha \nabla \mathcal{L}(\boldsymbol{y}^\top \boldsymbol{w}, \hat{\boldsymbol{y}}^\top \boldsymbol{y}) \frac{\partial \hat{\boldsymbol{y}}^\top \boldsymbol{w}}{\partial \theta}$

programming problem with fixed constraints, as a function of its cost vector $\hat{\boldsymbol{y}}$:

$$\Pi^*(\hat{\boldsymbol{y}}) = \underset{\Pi}{\operatorname{argmax}} \ \hat{\boldsymbol{y}}^\top \Pi \quad \texttt{s.t.} \ \ \boldsymbol{A}\Pi \leq b,$$

with A and b being an arbitrary matrix and vector, respectively. Given candidate costs $\hat{\boldsymbol{y}}$, the resulting optimal solution $\Pi^*(\hat{\boldsymbol{y}})$ can be evaluated relative to a known cost vector $\boldsymbol{y}$. Further, the resulting objective value can be compared to that of the optimal objective under the known cost vector using the regret metric $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$.

The regret measures the loss in objective value, relative to the true cost function, induced by the predicted cost. It is used as a loss function by which the predicted linear program costs vectors can be supervised by ground-truth values. However, the regret function is discontinuous with respect to $\hat{\boldsymbol{y}}$ for fixed $\boldsymbol{y}$. Following the approach pioneered in [52], we use a convex surrogate loss function, called the SPO+ loss, which forms a convex upper-bounding function over $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$. Its gradient is computed as follows:

$$\frac{\partial}{\partial \boldsymbol{y}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) \approx \frac{\partial}{\partial \boldsymbol{y}} \mathcal{L}_{\text{SPO+}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \Pi^*(2\hat{\boldsymbol{y}} - \boldsymbol{y}) - \Pi^*(\boldsymbol{y}). \tag{5.27}$$

Remarkably, risk bounds about the SPO+ loss relative to the SPO loss can be derived [98], and the empirical minimizer of the SPO+ loss is shown to achieve low excess true risk with high probability. Note that, by definition, $\boldsymbol{y}^\top \Pi^*(\boldsymbol{y}) \geq \boldsymbol{y}^\top \Pi^*(\hat{\boldsymbol{y}})$ and therefore $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) \geq 0$. Hence, finding the $\hat{\boldsymbol{y}}$ minimizing $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is equivalent to finding the $\hat{\boldsymbol{y}}$ maximizing $\boldsymbol{y}^\top \Pi^*(\hat{\boldsymbol{y}})$, since $\boldsymbol{y}^\top \Pi^*(\boldsymbol{y})$ is a constant value.

In the context of fair learning to rank, the goal is to predict the cost coefficients $\hat{\boldsymbol{y}}$ for Model 1 which maximize the empirical DCG, equal to $\boldsymbol{y}^\top \Pi^*(\hat{\boldsymbol{y}})\, \boldsymbol{w}$ for ground-truth relevance scores $\boldsymbol{y}$. A vectorized form can be written:

$$\boldsymbol{y}^\top \Pi \boldsymbol{w} = \overrightarrow{(\boldsymbol{y}^\top \boldsymbol{w})} \cdot \overrightarrow{\Pi}, \tag{5.28}$$

where $\overrightarrow{A}$ represents the row-major-order vectorization of a matrix A. Hence, the regret induced by prediction of cost coefficients $\hat{\boldsymbol{y}}$ is

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \overrightarrow{(\boldsymbol{y}^\top \boldsymbol{w})} \cdot \overrightarrow{\Pi^*(\boldsymbol{y})} - \overrightarrow{(\boldsymbol{y}^\top \boldsymbol{w})} \cdot \overrightarrow{\Pi^*(\hat{\boldsymbol{y}})}. \tag{5.29}$$

Note that while the cost coefficients $\boldsymbol{y}$ can be predicted generically (i.e., predicting an n^2-sized matrix), the modeling approach taken in this work is to predict item scores independently from individual feature vectors (resulting in an n-sized vector). These values combine naturally with the

known position bias values $\boldsymbol{v}$, to estimate DCG in the absence of true item scores. This simplification allows for learning independently over individual feature vectors, and was found in practice to outperform frameworks which use larger networks which take as input the entire feature vector lists.

Algorithm 3 maximizes the expected DCG of a learned ranking function by minimizing this regret. Its gradient is approximated as

$$\overrightarrow{\Pi^*(2\hat{\boldsymbol{y}}^\top \boldsymbol{w} - \boldsymbol{y}^\top \boldsymbol{w})} - \overrightarrow{\Pi^*(\boldsymbol{y}^\top \boldsymbol{w})}, \tag{5.30}$$

with $\hat{\boldsymbol{y}}$ predicted as described in Section 5.2.4. To complete the calculation of gradients for the fair ranking model, the remaining chain rule factor of line 19 is completed using the typical automatic differentiation.

5.2.5 Multigroup Fairness

SPOFR generalizes naturally to more than two groups. In contrast, multi-group fairness raises challenges for existing approaches that rely on penalty terms in the loss function [160, 135, 163]. Reference [160] proposes to formulate multi-group fairness using the single constraint

$$\sum_{g \neq g'} \left| \left(\frac{1}{|G_q^g|} \mathbb{1}_{G_q^g} - \frac{1}{|G_q^{g'}|} \mathbb{1}_{G_q^{g'}} \right)^\top \Pi \, \boldsymbol{v} \right| \leq \delta \tag{5.31}$$

where g and g' are groups indicators in $\mathcal{G}$, so that the average pairwise disparity between groups is constrained. However, this formulation suffers when $\delta \geq 0$, because the allowed fairness gap can be occupied by disparities associated with a single group in the worst case. Multiple constraints are required to provide true multi-group fairness guarantees and allow a controllable tradeoff between muti-group fairness and utility. Furthermore, the constraints (5.24) ensure satisfaction of (5.31) for appropriately chosen δ and are thus a generalization of (5.31). If unequal group disparities are desired, δ may naturally be chosen differently for each group in the equations (5.24).

5.2.6 Experiments

This section evaluates the performance of SPOFR against the prior approaches of [160] and [163], the current state-of-the-art methods for fair learning rank, which are denoted by FULTR and DELTR respectively. The experimental evaluation follows the more realistic *Partial Information setting* described in [160].

Datasets Two full-information datasets were used in [160] to generate partial-information counterparts using click simulation:

- *German Credit Data* is a dataset commonly used for studying algorithmic fairness. It contains information about 1000 loan applicants, each described by a set of attributes and labeled as creditworthy or non-creditworthy. Two groups are defined by the binary feature A43, indicating the purpose of the loan applicant. The ratio of applicants between the two groups is around $8:2$.
- *Microsoft Learn to Rank (MSLR)* is a standard benchmark dataset for LTR, containing a large number of queries from Bing with manually-judged relevance labels for retrieved web pages. The QualityScore attribute (feature id 133) is used to define binary protected groups using the 40^{th} percentile as threshold as in [160]. For evaluating fairness between $k > 2$ groups (multi-group fairness), k evenly-spaced quantiles define the thresholds.

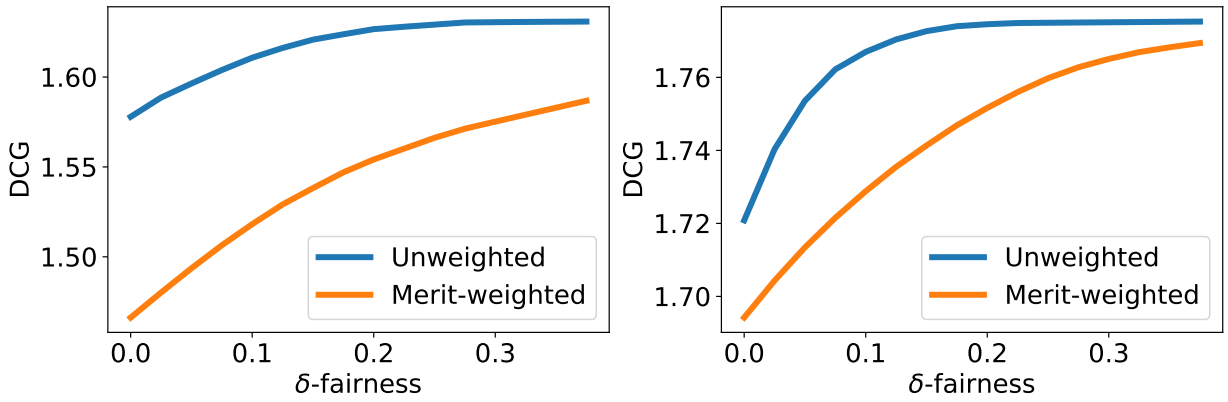


Figure 5.5: Fairness-Utility tradeoff for unweighted and merit-weighted fairness on credit (left) and MSLR (right) datasets.

Following the experimental settings of [160, 135], all datasets are constructed to contain item lists of size 20 for each query. The German Credit and MSLR training sets consist of 100k and 120k sample queries, respectively while full-information test sets consist of 1500 and 4000 samples.

The reader is referred to [160] for details of the click simulation used to produce the training and validation sets.

Models and Hyperparameters The prediction of item scores is the same for each model, with a single neural network which acts at the level of individual feature vectors as described in Section 5.2.4. The size of each layer is half that of the previous, and the output is a scalar value representing an item score.

The special *fairness parameters*, while also hyperparameters, are treated differently. Recall that fair LTR systems often aim to offer a tradeoff between utility and group fairness, so that fairness can be reduced by an acceptable tolerance in exchange for increased utility. For the baseline methods FULTR and DELTR, this tradeoff is controlled indirectly through the constraint violation penalty term denoted λ, as described in Section 5.2.4. Higher values of λ correspond to a preference for stronger adherence to fairness. In order to achieve δ-fairness for some specified δ, many values of λ must be searched until a trained model satisfying δ-fairness is found. As described in Section 5.2.4, this approach is unwieldly. In the case of SPOFR, the acceptable violation magnitude δ can be directly specified as in Definition (3).

The performance of each method is reported on the full-information test set for which all relevance labels are known. Ranking utility and fairness are measured with average DCG (Equation (5.17)) and fairness violation (Equation (5.19)), where each metric is computed on average over the entire dataset. The position bias power $p = 1$, so that $v_j = 1/(1+j)$ when computing fairness disparity.

Fairness-Utility Tradeoff for Two Groups The analysis first focuses on experiments involving two protected groups. Figure 5.5 shows the average test DCG attained by SPOFR on both the German Credit and MSLR datasets, for each level of both unweighted and merit-weighted δ-fairness as input to the model. Recall that each value of δ (defined as in Definition 3) on the x-axis is guaranteed to bound the ranking policy's expected fairness violation in response to *each* query. Note

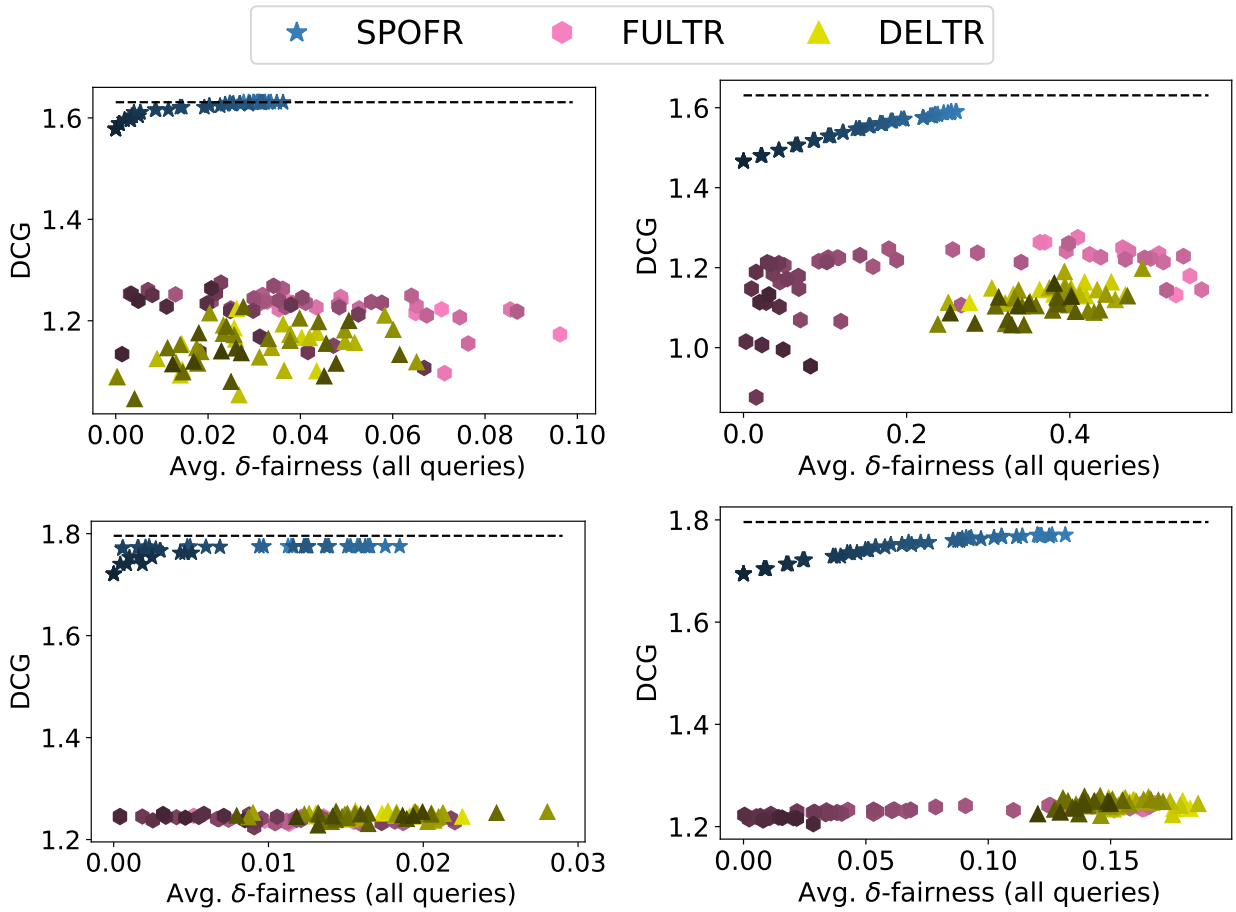


Figure 5.6: Fairness-Utility tradeoff for unweighted (left) and merit-weighted (right) fairness on credit (top) and MSLR (bottom) datasets.

the clear trend showing an increase in utility with the relaxation of the fairness bound δ, for all metrics and datasets. Note also that, in the datasets studied here, average merit favors the majority group. Merit-weighted group fairness can thus constrain the ranking positions of the minority group items further down than in unweighted fairness, regardless of their individual relevance scores, leading to more restricted (thus with lower utility) policies than in the unweighted case.

Fairness Parameter Search Figure 5.6 shows the average DCG vs average fairness disparity over the *test set* due to SPOFR, and compares it with those attained by FULTR and DELTR. Each point represents the performance of a single trained model, taken from a grid-search over *fairness parameters* δ (for SPOFR) and λ (for FULTR and DELTR). Darker colors represent more restrictive fairness parameters in each case.

Note that points on the grid which are lower on the x-axis and higher on the y-axis represent results which are strictly superior relative to others, as they represent a larger utility for smaller fairness violations. Dashed lines represent the maximum utility attainable in each case, computed by averaging over the test set the maximum possible DCG associated to each relevance vector.

Observe that the expected fairness violations due to SPOFR are much lower than the fairness levels guaranteed by the listed fairness parameters δ. This is because δ is a bound on the worst-case

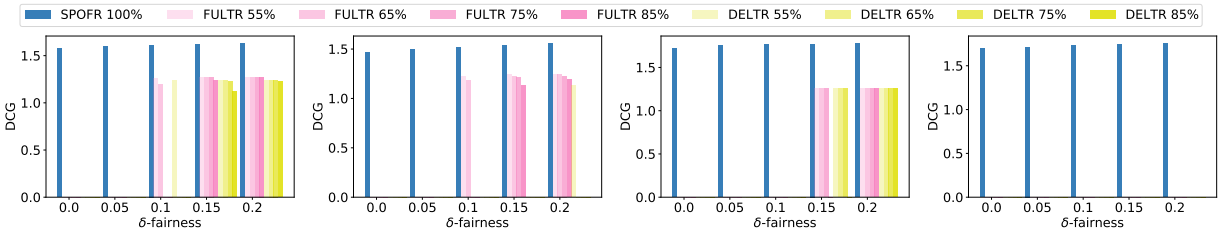


Figure 5.7: Query level guarantees: German credit unweighted (1st column) and merit-weighted fairness (2nd column); MSLR unweighted (3rd column) and merit-weighted fairness (4th column).

violation of fairness associated with any query, but actual resulting fairness disparities are typically much lower on average.

Second, the figure shows that SPOFR attains a substantial improvement in utility over the baselines, while exhibiting more consistent results across independently trained models. Note the dashed line represents the theoretical maximum attainable utility; remarkably, as the fairness parameter is relaxed, the DCG attained by SPOFR converges very close to this value. Section 5.2.7 provides theoretical motivation to explain these marked improvements.

Finally, notice that for FULTR and DELTR, large λ values (darker colors) should be associated with smaller fairness violations, compared to models trained with smaller λ values (lighter colors). However, this trend is not consistently observable: These results show the challenge to attain a meaningful relationship between the fairness penalizers λ and the fairness violations in these fair LTR methods. A similar observation also pertains to utility; It is expected that more permissive models in terms of fairness would attain larger utilities; this trend is not consistent in the FULTR and DELTR models. In contrast, the ability of the models learned by SPOFR to *guarantee* satisfying the desired fairness violation equips the resulting LTR models with much more interpretable and consistent outcomes.

Query-level Guarantees As discussed in Section 5.2.4, current fair LTR methods apply a fairness violation term on average over all training samples. Thus, disparities in favor of one group can cancel out those in favor of another group leading to *individual* policies that may not satisfy a desired fairness level. This section illustrates on these behaviors and analyzes the fairness guarantees attained by each model compared at the level of each individual query.

The results are summarized in Figure 5.7 which compares, SPOFR with FULTR (top) and SPOFR with DELTR (bottom). Each bar represents the maximum expected DCG attained while guaranteeing δ-fairness at the query level for δ as shown on the x-axis. Since neither baseline method can satisfy δ-fairness for every query, confidence levels are shown which correspond to the percentage of queries within the test set that resulted in ranking policies that satisfy *delta*-fairness. If no bar is shown at some fairness level, it was satisfied by no model at the given confidence level.

Notably, SPOFR satisfies δ-fairness with 100 percent confidence while also surpassing the baseline methods in terms of expected utility. This is remarkable and is due partly to the fact that the baseline methods can only be specified to optimize for fairness *on average* over all queries, which accomodates large query-level fairness disparities when they are balanced in opposite directions; i.e., in favor of opposite groups. In contrast SPOFR guarantees the specified fairness violation to be attained for ranking policies associated with each individual query.

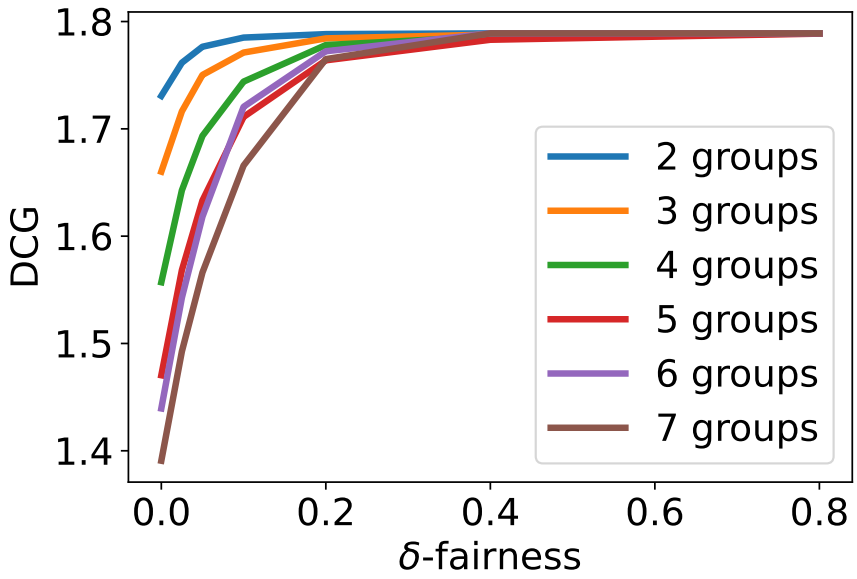


Figure 5.8: Multigroup Fairness on MSLR 120k.

Multi-group Fairness Finally, Figure 5.8 shows the fairness-utility tradeoff curves attained by SPOFR for each number of groups between 2 and 7 on the MSLR dataset. Note the decrease in expected DCG as the number of groups increases. This is not necessarily due to a degradation in predictive capability from SPOFR; the expected utility of any ranking policy necessarily decreases as fairness constraints are added. In fact, the expected utility converges for each multi-group model as the allowed fairness gap increases. Because this strict notion of multi-group fairness in LTR is uniquely possible using SPOFR, no results from prior approaches are available for direct comparison.

5.2.7 Discussion

Theoretical Remarks This section provides theoretical intuitions to explain the strong performance of SPOFR. As direct outputs of a linear programming solver, the ranking policy returned by SPOFR are subject to the properties of LP optimal solutions. This allows for certain insights on the representative capacity of the ranking model and on the properties of its resulting ranking policies. Let predicted scores be said to be *regret-optimal* if their resulting policy induces zero regret, i.e, $\boldsymbol{y}^\top \Pi^*(\boldsymbol{y})\ \boldsymbol{v} - \boldsymbol{y}^\top \Pi^*(\hat{\boldsymbol{y}})\ \boldsymbol{w} = 0$. That is equivalent to the maximization of the empirical utility.

Theorem 6 (Optimal Policy Prediction)**.** *For any given ground-truth relevance scores $\boldsymbol{y}$, there exist predicted item scores which maximize the empirical utility relative to $\boldsymbol{y}$.*

Proof. It suffices to predict $\hat{\boldsymbol{y}} = \boldsymbol{y}$. These scores are regret-optimal by definition, thus maximizing empirical utility. □

Note that the above property is due to the alignment between the structured policy prediction of SPOFR and the evaluation metrics, and is not shared by prior fair learning to rank frameworks.

Next, recall that any linear programming problem has a finite number of feasible solutions whose objective values are distinct [14]. There is thus not only a single point but a region of $\hat{\boldsymbol{y}}$

which are regret-optimal under $\boldsymbol{y}$, with respect to any instance of Model 1. This important property eases the difficulty in finding model parameters which maximize the empirical utility for any input sample, as item scores do not need to be predicted precisely in order to do so.

Finally, the set of $\hat{\boldsymbol{y}}$ which minimize the regret with respect to any instance of Model 1 overlaps (has nonempty intersection) with the set of $\hat{\boldsymbol{y}}$ which minimize the regret with respect to any other instance of the model, regardless of fairness constraints, under the same ground-truth $\boldsymbol{y}$. To show this, it suffices to exhibit a value of $\hat{\boldsymbol{y}}$ which minimizes the respective regret in every possible instance of Model 1, namely $\boldsymbol{y}$:

$$\boldsymbol{y}^\top \Pi^*_{f_1}(\boldsymbol{y})\boldsymbol{w} - \boldsymbol{y}^\top \Pi^*_{f_1}(\boldsymbol{y})\boldsymbol{w} = 0 = \boldsymbol{y}^\top \Pi^*_{f_2}(\boldsymbol{y})\boldsymbol{w} - \boldsymbol{y}^\top \Pi^*_{f_2}(\boldsymbol{y})\boldsymbol{w}, \tag{5.32}$$

where $\Pi^*_{f_1}(y)$ and $\Pi^*_{f_2}(y)$ are the optimal policies subject to distinct fairness constraints f_1 and f_2. This implies that a model which learns to rank fairly under this framework need not account for the group composition of item lists in order to maximize empircal utility; It suffices to learn item scores from independent feature vectors, rather than learn the higher-level semantics of feature vector lists required to enforce fairness. This is because group fairness is ensured automatically by the embedded optimization model.

The empirical results presented, additionally, show that the utility attained by SPOFR is close to optimal on the test cases analyzed. Together with the theoretical observations above, this suggests that, on the test cases analyzed, fairness does not change drastically the objective of the optimal ranking policy. This may be an artifact of the LTR tasks, obtained from [160], being relatively easy predict given a sufficiently powerful model. These observation may signal a need for the study and curation of more challenging benchmark datasets for future research on fairness in LTR.

SPOFR Limitations The primary disadvantage of SPOFR is that it cannot be expected to learn to rank lists of arbitrary size, as runtime increases with the size of the lists to be ranked. In contrast to penalty-based methods, which require a single linear pass to the neural network to derive a ranking policy for a given query, SPOFR requires solving a linear programming problem to attain an optimal ranking policy. While this is inevitably computationally more expensive, solving the LP of Model 1 requires low degree polynomial time in the number of items to rank [143], due to the sparsity of its constraints. Fortunately, this issue can be vastly alleviated with the application of hot-starting schemes [103], since the SPO framework relies on iteratively updating a stored solution to each LP instance for slightly different objective coefficients as model weights are updated. Thus, each instance of Model 1 need not be solved from scratch.

5.2.8 Conclusions

This section has described a framework for learning fair ranking functions by integrating constrained optimization with deep learning. By enforcing fairness constraints on its ranking policies at the level of each prediction, this approach provides direct control over the allowed disparity between groups, which is guaranteed to hold for every user query. Since the framework naturally accommodates the imposition of many constraints, it generalizes to multigroup fair LTR settings without substantial degradation in performance, while allowing for stronger notions of multigroup fairness than previously possible. Further, it has been shown to outperform previous approaches in terms of both the expected fairness and utility of its learned ranking policies. By integrating constrained optimization algorithms into its fair ranking function, SPOFR allows for analytical

representation of expected utility metrics and end-to-end training for their optimization, along with theoretical insights into properties of its learned representations. These advantages may highlight the integration of constrained optimization and machine learning techniques as a promising avenue to address further modeling challenges in future research on learning to rank.

Chapter 6

Future Directions: End-to-End Integration of Learned Optimizers

The previous chapters have studied original and prior works as they relate to the three main categories of interest in this thesis: Learning to Optimize (Chapter 3), Predict-Then-Optimize (Chapter 5), and Differentiable Programming (Chapter 4). Several areas of overlap between these domains have been apparent: for example, we have seen how both of the former problem settings can benefit from the use of differentiable programming solvers.

This chapter looks closer at the intersections between these topics, to consider ideas that may have potential for future research directions. In particular, it proposes two frameworks in which learned optimization models, in the Learning-to-Optimize sense, may be integrated end-to-end with predictive models or even other learned optimizers. One motivating intuition shared by these proposals, is that learned optimization models are differentiable by construction in addition to be efficiently callable, giving them potential for integration into end-to-end trainable pipelines.

Section 6.1 shows how problems in the Predict-Then-Optimize setting can be treated purely with Learning to Optimize methods, by learning the prediction and optimization steps in a single joint model. It is based on published work [95]. Section 6.2, based on yet unpublished work, then shows how pretrained LtO models can function as efficient surrogates for differentiable programming solvers. We leverage their speed advantages and automatic differentiability to enable fast gradient-based constraint correction schemes, and use them to learn bilevel programming with applications to optimal control. It is hoped that these works will be part of a larger future effort to harness the benefits of learned optimization models, in scientific and engineering applications beyond their originally intended purposes.

6.1 Learning Joint Models of Prediction and Optimization

The *Predict-Then-Optimize* (PtO) framework models decision-making processes as optimization problems whose parameters are only partially known while the remaining, unknown, parameters must be estimated by a machine learning (ML) model. The predicted parameters complete the specification of an optimization problem which is then solved to produce a final decision. The

problem is posed as estimating the solution $\boldsymbol{x}^\star(\zeta) \in \mathcal{X} \subseteq \mathbb{R}^n$ of a *parametric* optimization problem:

$$\boldsymbol{x}^\star(\boldsymbol{\zeta}) = \underset{\boldsymbol{x}}{\operatorname{argmin}} \;\; f(\boldsymbol{x}, \boldsymbol{\zeta}) \qquad (6.1)$$
$$\textit{such that:} \;\; \boldsymbol{g}(\boldsymbol{x}) \leq 0, \;\; \boldsymbol{h}(\boldsymbol{x}) = 0,$$

given that parameters $\boldsymbol{\zeta} \in C \subseteq \mathbb{R}^p$ are unknown, but that a correlated set of observable values $\boldsymbol{z} \in \mathcal{Z}$ are available. Here f is an objective function, and $\boldsymbol{g}$ and $\boldsymbol{h}$ define the set of the problem's inequality and equality constraints. The combined prediction and optimization model is evaluated on the basis of the optimality of its downstream decisions, with respect to f under its ground-truth problem parameters [52]. This setting is ubiquitous to many real-world applications confronting the task of decision-making under uncertainty, such as planning the shortest route in a city, determining optimal power generation schedules, or managing investment portfolios. For example, a vehicle routing system may aim to minimize a rider's total commute time by solving a shortest-path optimization model (6.1) given knowledge of the transit times $\boldsymbol{\zeta}$ over each individual city block. In absence of that knowledge, it may be estimated by models trained to predict local transit times based on exogenous data $\boldsymbol{z}$, such as weather and traffic conditions. In this context, more accurately predicted transit times $\hat{\boldsymbol{\zeta}}$ tend to produce routing plans $\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}})$ with shorter overall commutes, with respect to the true city-block transit times $\boldsymbol{\zeta}$.

However, direct training of predictions from observable features to problem parameters tends to generalize poorly with respect to the ground-truth optimality achieved by a subsequent decision model [104, 89]. To address this challenge, *End-to-end Predict-Then-Optimize* (EPO) [52] has emerged as a transformative paradigm in data-driven decision making in which predictive models are trained by directly minimizing loss functions defined on the downstream optimal solutions $\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}})$.

On the other hand, EPO implementations require backpropagation through the solution of the optimization problem (6.1) as a function of its parameters for end-to-end training. The required back-propagation rules are highly dependent on the form of the optimization model and are typically derived by hand analytically for limited classes of models [6, 4]. Furthermore, difficult decision models involving nonconvex or discrete optimization may not admit well-defined backpropagation rules.

To address these challenges, this paper outlines a framework for training Predict-Then-Optimize models by techniques adapted from a separate but related area of work that combines constrained optimization end-to-end with machine learning. Such paradigm, called *Learn-to-Optimize* (LtO), learns a mapping between the parameters of an optimization problem and its corresponding optimal so-

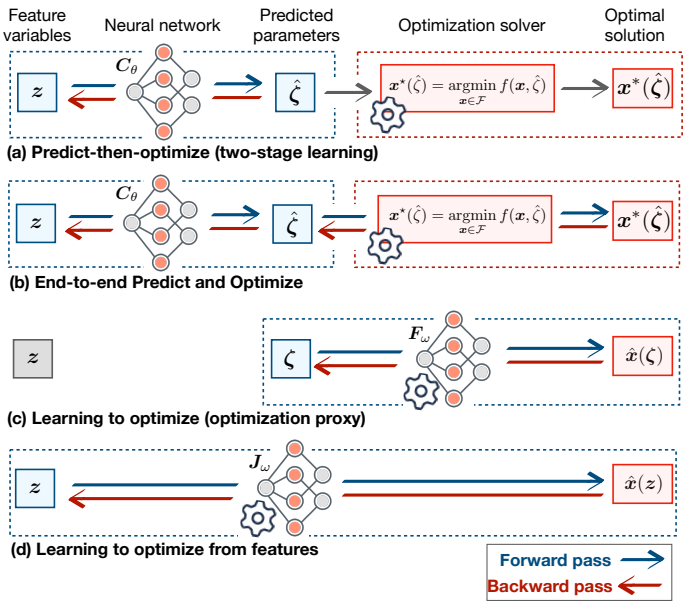


Figure 6.1: Illustration of Learning to Optimize from Features, in relation to other learning paradigms.

lutions using a deep neural network (DNN),
as illustrated in Figure 6.1(c). The resulting DNN mapping is then treated as an *optimization proxy* whose role is to repeatedly solve difficult, but related optimization problems in real time [147, 54]. Several LtO methods specialize in training proxies to solve difficult problem forms, especially those involving nonconvex optimization.

The proposed methodology of this paper, called *Learning to Optimize from Features* (LtOF), recognizes that existing Learn-to-Optimize methods can provide an array of implementations for producing learned optimization proxies, which can handle hard optimization problem forms, have fast execution speeds, and are differentiable by construction. As such, they can be adapted to the Predict-Then-Optimize setting, offering an alternative to hard optimization solvers with handcrafted backpropagation rules. However, directly transferring a pretrained optimization proxy into the training loop of an EPO model leads to poor accuracy, as shown in Section 6.1.2, due to the inability of LtO proxies to generalize outside their training distribution. To circumvent this distributional shift issue, this paper shows how to adapt the LtO methodology to learn optimal solutions directly from features.

Contributions. In summary, this paper makes the following novel contributions: **(1)** It investigates the use of pretrained LtO proxy models as a means to approximate the decision-making component of the PtO pipeline, and demonstrates a distributional shift effect between prediction and optimization models that leads to loss of accuracy in end-to-end training. **(2)** It proposes Learning to Optimize from Features (LtOF), in which existing LtO methods are adapted to learn solutions to optimization problems directly from observable features, circumventing the distribution shift effect over the problem parameters. **(3)** The generic LtOF framework is evaluated by adapting several well-known LtO methods to solve Predict-then-Optimize problems with difficult optimization components, under complex feature-to-parameter mappings. Besides the performance improvement over two-stage approaches, *the results show that difficult nonconvex optimization components can be incorporated into PtO pipelines naturally*, extending the flexibility and expressivity of PtO models.

6.1.1 Problem Setting and Background

In the Predict-then-Optimize (PtO) setting, a (DNN) prediction model $\boldsymbol{C}_\theta : \mathcal{Z} \to C \subseteq \mathbb{R}^k$ first takes as input a feature vector $\boldsymbol{z} \in \mathcal{Z}$ to produce predictions $\hat{\boldsymbol{\zeta}} = \boldsymbol{C}_\theta(\boldsymbol{z})$. The model $\boldsymbol{C}$ is itself parametrized by learnable weights θ. The predictions $\hat{\boldsymbol{\zeta}}$ are used to parametrize an optimization model of the form (6.1), which is then solved to produce optimal decisions $\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}}) \in \mathcal{X}$. We call these two components, respectively, the *first* and *second* stage models. Combined, their goal is to produce decisions $\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}})$ which minimize the ground-truth objective value $f(\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}}), \boldsymbol{\zeta})$ given an observation of $\boldsymbol{z} \in \mathcal{Z}$. Concretely, assuming a dataset of samples $(\boldsymbol{z}, \boldsymbol{\zeta})$ drawn from a joint distribution Ω, the goal is to learn a model $\boldsymbol{C}_\theta : \mathcal{Z} \to C$ producing predictions $\hat{\boldsymbol{\zeta}} = \boldsymbol{C}_\theta(\boldsymbol{z})$ which achieves

$$\underset{\theta}{\text{minimize}} \;\; \mathbb{E}_{(\boldsymbol{z},\boldsymbol{\zeta})\sim\Omega}\left[f\left(\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}}), \boldsymbol{\zeta}\right)\right]. \tag{6.2}$$

This optimization is equivalent to minimizing expected *regret*, defined as the magnitude of suboptimality of $\boldsymbol{x}^\star(\hat{\zeta})$ with respect to the ground-truth parameters:

$$\mathit{regret}(\boldsymbol{x}^\star(\hat{\zeta}), \zeta) = f(\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}}), \boldsymbol{\zeta}) - f(\boldsymbol{x}^\star(\boldsymbol{\zeta}), \boldsymbol{\zeta}). \tag{6.3}$$

Two-stage Method. A common approach to training the prediction model $\hat{\boldsymbol{\zeta}} = \boldsymbol{C}_\theta(\boldsymbol{z})$ is the *two-stage* method, which trains to minimize the mean squared error loss $\ell(\hat{\boldsymbol{\zeta}}, \boldsymbol{\zeta}) = \|\hat{\boldsymbol{\zeta}} - \boldsymbol{\zeta}\|_2^2$, without

taking into account the second stage optimization. While directly minimizing the prediction errors is confluent with the task of optimizing ground-truth objective $f(\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}}), \boldsymbol{\zeta})$, the separation of the two stages in training leads to error propagation with respect to the optimality of downstream decisions, due to misalignment of the training loss with the true objective [52].

End-to-End Predict-Then-Optimize. Improving on the two-stage method, the End-to-end Predict-end-Optimize (EPO) approach trains directly to optimize the objective $f(\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}}), \boldsymbol{\zeta})$ by gradient descent, which is enabled by finding or approximating the derivatives through $\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}})$. This allows for end-to-end training of the PtO goal (6.2) directly as a loss function, which consistently outperforms two-stage methods with respect to the evaluation metric (6.2), especially when the mapping $\boldsymbol{z} \to \boldsymbol{\zeta}$ is *difficult to learn* and subject to significant prediction error. Such an integrated training of prediction and optimization is referred to as *Smart Predict-Then-Optimize* [52], *Decision-Focused Learning* [153], or End-to-End Predict-Then-Optimize (EPO) [139]. This paper adopts the latter term throughout, for consistency. Various implementations of this idea have shown significant gains in downstream decision quality over the conventional two-stage method. See Figure 6.1 (a) and (b) for an illustrative comparison, where the constraint set is denoted with $\mathcal{F}$.

Challenges in End-to-End Predict-Then-Optimize Despite their advantages over the two-stage, EPO methods face two key challenges: **(1) Differentiability**: the need for handcrafted backpropagation rules through $\boldsymbol{x}^\star(\boldsymbol{\zeta})$, which are highly dependent on the form of problem (6.1), and rely on the assumption of derivatives $\frac{\partial \boldsymbol{x}^\star}{\partial \boldsymbol{\zeta}}$ which may not exist or provide useful descent directions, and require that the mapping (6.1) is unique, producing a well-defined function; **(2) Efficiency**: the need to solve the optimization (6.1) to produce $\boldsymbol{x}^\star(\boldsymbol{\zeta})$ for each sample, at each iteration of training, which is often inefficient even for simple optimization problems.

This paper is motivated by a need to address these disadvantages. To do so, it recognizes a body of work on training DNNs as *learned optimization proxies* which have fast execution, are automatically differentiable by design, and specialize in learning mappings $\boldsymbol{\zeta} \to \boldsymbol{x}^\star(\boldsymbol{\zeta})$ of hard optimization problems. While the next section discusses why the direct application of learned proxies as differentiable optimization solvers in an EPO approach tends to fail, Section 6.1.3 presents a successful adaptation of the approach in which optimal solutions are learned end-to-end from the observable features $\boldsymbol{z}$.

6.1.2 EPO with Optimization Proxies

The Learning-to-Optimize problem setting encompasses a variety of distinct methodologies with the common goal of learning to solve optimization problems. This section characterizes that setting, before proceeding to describe an adaptation of LtO methods to the Predict-Then-Optimize setting.

Learning to Optimize. The idea of training DNN models to emulate optimization solvers is referred to as *Learning-to-Optimize (LtO)* [89]. Here the goal is to learn a mapping $\mathbf{F}_\omega : C \to \mathcal{X}$ from the parameters $\boldsymbol{\zeta}$ of an optimization problem (6.1) to its corresponding optimal solution $\boldsymbol{x}^\star(\boldsymbol{\zeta})$ (see Figure 6.1 (c)). The resulting *proxy* optimization model has as its learnable component a DNN denoted $\hat{\boldsymbol{F}}_\omega$, which may be augmented with further operations $\boldsymbol{\mathcal{S}}$ such as constraint corrections or unrolled solver steps, so that $\mathbf{F}_\omega = \boldsymbol{\mathcal{S}} \circ \hat{\mathbf{F}}_\omega$. While training such a lightweight model to emulate optimization solvers is in general difficult, it is made tractable by restricting the task over a *limited distribution* Ω^F of problem parameters $\boldsymbol{\zeta}$.

A variety of LtO methods have been proposed, many of which specialize in learning to solve problems of a specific form. Some are based on supervised learning, in which case precomputed solutions $\boldsymbol{x}^\star(\boldsymbol{\zeta})$ are required as target data in addition to parameters $\boldsymbol{\zeta}$ for each sample. Others are *self-supervised*, requiring only knowledge of the problem form (6.1) along with instances of the parameters $\boldsymbol{\zeta}$ for supervision in training. LtO methods employ special learning objectives to train the proxy model $\boldsymbol{F}_\omega$:

$$\underset{\omega}{\text{minimize}} \ \mathbb{E}_{\boldsymbol{\zeta}\sim\Omega^F}\left[\ell^{\text{LtO}}\Big(\boldsymbol{F}_\omega(\boldsymbol{\zeta}),\boldsymbol{\zeta}\Big)\right], \tag{6.4}$$

where ℓ^{LtO} represents a loss that is specific to the LtO method employed. A primary challenge in LtO is ensuring the satisfaction of constraints $\boldsymbol{g}(\hat{\boldsymbol{x}}) \leq 0$ and $\boldsymbol{h}(\hat{\boldsymbol{x}}) = 0$ by the solutions $\hat{\boldsymbol{x}}$ of the proxy model $\boldsymbol{F}_\omega$. This can be achieved, exactly or approximately, by a variety of methods, for example iteratively retraining Equation (6.4) while applying dual optimization steps to a Lagrangian loss function [54, 119], or designing $\boldsymbol{\mathcal{S}}$ to restore feasibility [48], as reviewed in Appendix 6.1.4. In cases where small constraint violations remain in the solutions $\hat{\mathbf{x}}$ at inference time, they can be removed by post-processing with efficient projection or correction methods as deemed suitable for the particular application [89].

EPO with Pretrained Optimization Proxies Viewed from the Predict-then-Optimize lens, learned optimization proxies have two beneficial features by design: **(1)** they enable very fast solving times compared to conventional solvers, and **(2)** are differentiable by virtue of being trained end-to-end. Thus, a natural question is whether it is possible to use a pre-trained optimization proxy to substitute the differentiable optimization component of an EPO pipeline. Such an approach modifies the EPO objective (6.2) as:

$$\underset{\theta}{\text{minimize}} \ \mathbb{E}_{(\boldsymbol{z},\boldsymbol{\zeta})\sim\Omega}\Big[f\Big(\overbrace{\boldsymbol{F}_\omega(\underbrace{\boldsymbol{C}_\theta(\boldsymbol{z})}_{\hat{\boldsymbol{\zeta}}})}^{\hat{\boldsymbol{x}}},\boldsymbol{\zeta}\Big)\Big], \tag{6.5}$$

in which the solver output $\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}})$ of problem (6.2) is replaced with the prediction $\hat{\boldsymbol{x}}$ obtained by LtO model $\boldsymbol{F}_\omega$ on input $\hat{\boldsymbol{\zeta}}$ (gray color highlights that the model is pretrained, before freezing its weights ω).

However, a fundamental challenge in LtO lies in the inherent limitation that ML models act as reliable optimization proxies *only within the distribution of inputs they are trained on*. This challenges the implementation of the idea of using pretrained LtOs as components of an end-to-end Predict-Then-Optimize model as the weights θ update during training, leading to continuously evolving inputs $\boldsymbol{C}_\theta(\boldsymbol{z})$ to the pretrained optimizer $\boldsymbol{F}_\omega$. Thus, to ensure robust performance, $\boldsymbol{F}_\omega$ must generalize well across virtually any input during training. However, due to the dynamic nature of θ, there is an inevitable *distribution shift* in the inputs to $\boldsymbol{F}_\omega$, destabilizing the EPO training.

Figures 6.2 and 6.3 illustrate this issue. The former highlights how the input distribution to a pretrained proxy drifts during EPO training, adversely affecting both output and backpropagation. The latter quantifies this behavior, exemplified on a simple two-dimensional problem (described in Appendix D.1), showing rapid increase in proxy regret as $\hat{\boldsymbol{\zeta}}$ diverges from the initial training distribution $\boldsymbol{\zeta} \sim \Omega^F$ (shown in black). The experimental results presented in Tables 6.2,6.3, and 6.4 reinforce these observations. While each proxy solver performs well within its training distribution, their effectiveness deteriorates sharply when utilized as described in (6.5). This degradation is observed irrespective of any normalization applied to the proxy's input parameters during EPO

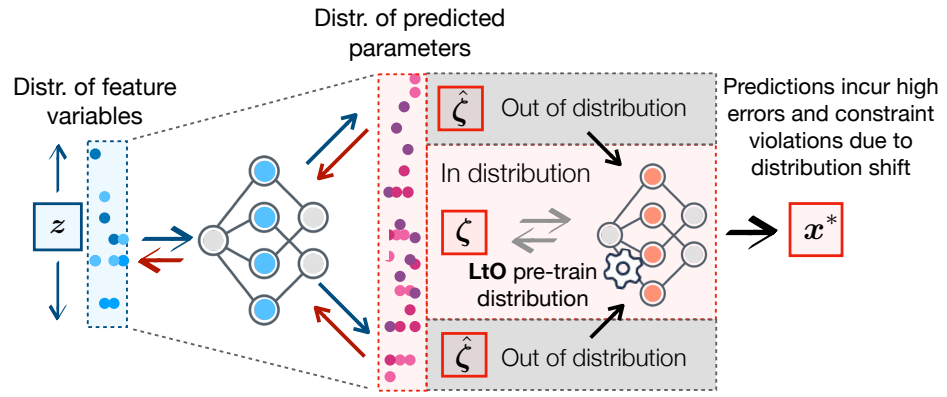


Figure 6.2: A distribution shift between the training distribution of a LtO proxy and the parameter predictions during training leads to inaccuracies in the proxy solver.

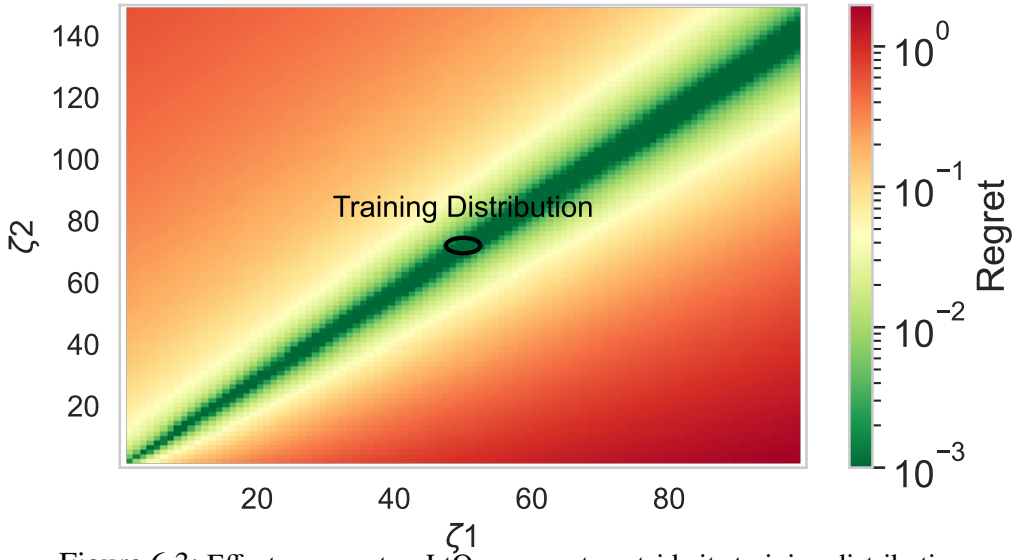


Figure 6.3: Effect on regret as LtO proxy acts outside its training distribution.

training.

A step toward resolving this distribution shift issue allows the weights of $\boldsymbol{F}_\omega$ to adapt to its changing inputs, by *jointly* training the prediction and optimization models:

$$\underset{\theta,\omega}{\text{minimize}} \;\; \mathbb{E}_{(\boldsymbol{z},\zeta)\sim\Omega}\Big[f\big(\overbrace{\boldsymbol{F}_\omega(\underbrace{\boldsymbol{C}_\theta(\boldsymbol{z})}_{\hat{\zeta}})}^{\hat{\boldsymbol{x}}},\zeta\big)\Big]. \tag{6.6}$$

The predictive model $\boldsymbol{C}_\theta$ is then effectively absorbed into the predictive component of $\boldsymbol{F}_\omega$, resulting in a *joint* prediction and optimization proxy model $\boldsymbol{J}_\phi = \boldsymbol{F}_\omega \circ \boldsymbol{C}_\theta$, where $\phi = (\omega, \theta)$. Given the requirement for feasible solutions, the training objective (6.6) must be replaced with an LtO procedure that enforces the constraints on its outputs. This leads us to the framework presented next.

6.1.3 Learning to Optimize from Features

The distribution shift effect described above arises due to the disconnect in training between the first-stage prediction network $\boldsymbol{C}_\theta : \mathcal{Z} \to C$ and the second-stage optimization proxy $\boldsymbol{F}_\omega : C \to \mathcal{X}$. However, the Predict-Then-Optimize setting (see Section 6.1.1) ultimately only requires the combined model to produce a candidate optimal solution $\hat{\boldsymbol{x}} \in \mathcal{X}$ given an observation of features $\boldsymbol{z} \in \mathcal{Z}$. Thus, the intermediate prediction $\hat{\boldsymbol{\zeta}} = \boldsymbol{C}_\theta(\boldsymbol{z})$ in Equation (6.6) is, in principle, not needed. This motivates the choice to learn direct mappings from features to optimal solutions of the second-stage decision problem. The joint model $\boldsymbol{J}_\phi : \mathcal{Z} \to \mathcal{X}$ is trained by Learning-to-Optimize procedures, employing

$$\underset{\phi}{\text{minimize}} \ \mathbb{E}_{(\boldsymbol{z},\boldsymbol{\zeta})\sim\Omega}\Big[\ell^{\text{LtO}}\Big(\boldsymbol{J}_\phi(\boldsymbol{z}), \boldsymbol{\zeta}\Big)\Big]. \tag{6.7}$$

This method can be seen as a direct adaptation of the Learn-to-Optimize framework to the Predict-then-Optimize setting. The key difference from the typical LtO setting, described in Section 6.1.2, is that problem parameters $\boldsymbol{\zeta} \in C$ are not known as inputs to the model, but the correlated features $\boldsymbol{z} \in \mathcal{Z}$ are known instead. Therefore, estimated optimal solutions now take the form $\hat{\boldsymbol{x}} = \boldsymbol{J}_\phi(\boldsymbol{z})$ rather than $\hat{\boldsymbol{x}} = \boldsymbol{F}_\omega(\boldsymbol{\zeta})$. Notably, this causes the self-supervised LtO methods to become *supervised*, since the ground-truth parameters $\boldsymbol{\zeta} \in C$ now act only as target data while the separate feature variable $\boldsymbol{z}$ takes the role of input data.

We refer to this approach as *Learning to Optimize from Features (LtOF)*. Figure 6.1 illustrates the key distinctions of LtOF relative to the other learning paradigms studied in the paper. Figures (6.1c) and (6.1d) distinguish LtO from LtoF by a change in model's input space, from $\boldsymbol{\zeta} \in C$ to $\boldsymbol{z} \in \mathcal{Z}$. This brings the framework into the same problem setting as that of the two-stage and end-to-end PtO approaches, illustrated in Figures (6.1a) and (6.1b). The key difference from the PtO approaches is that they produce an estimated optimal solution $\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}})$ by using a true optimization solver, but applied to an imperfect parametric prediction $\hat{\boldsymbol{\zeta}} = \boldsymbol{C}_\theta(\boldsymbol{z})$. In contrast, LtOF directly estimates optimal solution $\hat{\boldsymbol{x}}(\boldsymbol{z}) = \boldsymbol{J}_\phi(\boldsymbol{z})$ from features $\boldsymbol{z}$, circumventing the need to represent an estimate of $\boldsymbol{\zeta}$.

Sources of Error Both the PtO and LtOF methods yield solutions subject to *regret*, which measures suboptimality relative to the true parameters $\boldsymbol{\zeta}$, as defined in Equation 6.3. However, while in end-to-end and, especially, in the two-stage PtO approaches, the regret in $\boldsymbol{x}^\star(\hat{\boldsymbol{\zeta}})$ arises from imprecise parameter predictions $\hat{\boldsymbol{\zeta}} = \boldsymbol{C}_\theta(\boldsymbol{z})$ [104], in LtOF, the regret in the inferred solutions $\hat{\boldsymbol{x}}(\boldsymbol{z}) = \boldsymbol{J}_\phi(\boldsymbol{z})$ arises due to imperfect learning of the proxy optimization. This error is inherent to the LtO methodology used to train the joint prediction and optimization model $\boldsymbol{J}_\phi$, and persists even in typical LtO, in which $\boldsymbol{\zeta}$ are precisely known. In principle, a secondary source of error can arise from imperfect learning of the implicit feature-to-parameter mapping $\boldsymbol{z} \to \boldsymbol{\zeta}$ within the joint model $\boldsymbol{J}_\phi$. However, these two sources of error are indistinguishable, as the prediction and optimization steps are learned jointly. Finally, depending on the specific LtO procedure adopted, a further source of error arises when small violations to the constraints occur in $\hat{\boldsymbol{x}}(\boldsymbol{z})$. In such cases, restoring feasibility (e.g, through projection or heuristics steps) often induces slight increases in regret [54].

Despite being prone to optimization error, Section 6.1.4 shows that Learning to Optimize from Features greatly outperforms two-stage methods, and is competitive with EPO training based on exact differentiation through $\boldsymbol{x}^\star(\boldsymbol{\zeta})$, when the feature-to-parameter mapping $\boldsymbol{z} \to \boldsymbol{\zeta}$ is complex. This is achieved *without* any access to exact optimization solvers, nor models of their derivatives.

This feat can be explained by the fact that by learning optimal solutions end-to-end directly from features, LtOF does not directly depend on learning an accurate representation of the underlying mapping from z to ζ.

Efficiency Benefits Because the primary goal of the Learn-to-Optimize methodology is to achieve *fast solving times*, the LtOF approach broadly inherits this advantage. While these benefits in speed may be diminished when constraint violations are present and complex feasibility restoration are required, efficient feasibility restoration is possible for many classes of optimization models [15]. This enables the design of *accelerated* PtO models within the LtOF framework, as shown in Section 6.1.4.

Modeling Benefits While EPO approaches require the implementation of problem-specific back-propagation rules, the LtOF framework allows for the utilization of existing LtO methodologies in the PtO setting, on a problem-specific basis. A variety of existing LtO methods specialize in learning to solve convex and nonconvex optimization [54, 119, 48], combinatorial optimization [16, 86], and other more specialized problem forms [158]. The experiments of this paper focus on the scope of continuous optimization problems, whose LtO approaches share a common set of solution strategies.

6.1.4 Experiments

This section evaluates three distinct LtO methods adapted to the LtOF setting, on three different Predict-Then-Optimize tasks, where each task involves a distinct second stage optimization component $x^\star : C \to \mathcal{X}$, as in (6.1). These include a convex quadratic program (QP), a nonconvex quadratic programming variant, and a nonconvex AC-Optimal Power Flow problem, to demonstrate the general utility of the framework. First, the section's three LtOF methods are briefly described.

Learning to Optimize Methods This section reviews in more depth those LtO methods which are adapted to solve PtO problems in Section 6.1.4 of this paper. Each description below assumes a DNN model $\hat{F}_\omega$, which acts on parameters ζ_i specifying an instance of problem (6.1), to produce an estimate of the optimal solution $\hat{x} := F_\omega(\zeta)$, so that $\hat{x} \approx x^\star(\zeta)$.

Lagrangian Dual Learning (LD) Fioretto et al. [54] uses the following modified Lagrangian loss function for training $\hat{x} = F_\omega(\zeta)$:

$$\mathcal{L}_{\mathbf{LD}}(\hat{x}, \zeta) = \|\hat{x} - x^\star(\zeta)\|_2^2 + \lambda^T [g(\hat{x}, \zeta)]_+ + \mu^T h(\hat{x}, \zeta). \tag{6.8}$$

At each iteration of LD training, the model F_ω is trained to minimize the loss $\mathcal{L}_{\mathbf{LD}}$. Then, updates to the multiplier vectors λ and μ are calculated based on the average constraint violations incurred by the predictions $\hat{x}$, mimicking a dual ascent method [25]. In this way, the method minimizes a balance of constraint violations and proximity to the precomputed target optima $x^\star(\zeta)$.

Self-Supervised Primal-Dual Learning (PDL) Park and Van Hentenryck [119] use an augmented

Lagrangian loss for self-supervised learning:

$$\mathcal{L}_{\mathbf{PDL}}(\hat{\boldsymbol{x}}, \boldsymbol{\zeta}) = f(\hat{\boldsymbol{x}}, \boldsymbol{\zeta}) + \hat{\boldsymbol{\lambda}}^T \boldsymbol{g}(\hat{\boldsymbol{x}}, \boldsymbol{\zeta}) + \hat{\boldsymbol{\mu}}^T \boldsymbol{h}(\hat{\boldsymbol{x}}, \boldsymbol{\zeta}) + \frac{\rho}{2}\left(\sum_j \nu(g_j(\hat{\boldsymbol{x}})) + \sum_j \nu(h_j(\hat{\boldsymbol{x}}))\right), \tag{6.9}$$

where ν measures the constraint violation. At each iteration of PDL training, a separate estimate of the Lagrange multipliers is stored for each problem instance in training, and updated by an augmented Lagrangian method [25] after training $\hat{\boldsymbol{x}} = \boldsymbol{F}_\omega(\boldsymbol{\zeta})$ to minimize (6.9). In addition to the primal network $\boldsymbol{F}_\omega$, a dual network $\mathcal{D}_{\omega'}$ learns to store updates of the multipliers for each instance, and predict them as $(\hat{\boldsymbol{\lambda}}, \hat{\boldsymbol{\mu}}) = \mathcal{D}_{\omega'}(\boldsymbol{\zeta})$ to the next iteration.

Deep Constraint Completion and Correction (DC3) Donti et al. [48] use the loss function

$$\mathcal{L}_{\mathbf{DC3}}(\hat{\boldsymbol{x}}, \boldsymbol{\zeta}) = f(\hat{\boldsymbol{x}}, \boldsymbol{\zeta}) + \lambda \| [\boldsymbol{g}(\hat{\boldsymbol{x}}, \boldsymbol{\zeta})]_+ \|_2^2 + \mu \|\boldsymbol{h}(\hat{\boldsymbol{x}}, \boldsymbol{\zeta})\|_2^2 \tag{6.10}$$

which combines a problem's objective value with two additional terms which aggregate the total violations of its equality and inequality constraints. The scalar multipliers λ and μ are not adjusted during training. However, feasibility of predicted solutions is enforced by treating $\hat{\boldsymbol{x}} = \hat{\boldsymbol{F}}_\omega(\boldsymbol{\zeta})$ as an estimate for only a subset of optimization variables. The remaining variables are completed by solving the underdetermined equality constraints $\boldsymbol{h}(\hat{\boldsymbol{x}}) = \boldsymbol{0}$ as a system of equations. Inequality violations are corrected by gradient descent on the their aggregated values $\| [\boldsymbol{g}(\hat{\boldsymbol{x}}, \boldsymbol{\zeta})]_+ \|^2$. These completion and correction steps form the function $\boldsymbol{S}$, where $\boldsymbol{F}_\omega(\boldsymbol{\zeta}) = \boldsymbol{S} \circ \hat{\boldsymbol{F}}_\omega(\boldsymbol{\zeta})$.

While several other Learning-to-Optimize methods have been proposed in the literature, the above-described collection represents diverse subset which is used to demonstrate the potential of adapting the end-to-end LtO methodology as a whole to the Predict-Then-Optimize setting.

Experimental Settings **Feature generation**. End-to-End Predict-Then-Optimize methods integrate learning and optimization to minimize the propagation of prediction errors–specifically, from feature mappings $\boldsymbol{z} \to \boldsymbol{\zeta}$ to the resulting decisions $\boldsymbol{x}^\star(\boldsymbol{\zeta})$ (regret). It's crucial to recognize that *even methods with high error propagation* can yield low regret *if the prediction errors are low*. To account for this, EPO studies often employ synthetically generated feature mappings to control prediction task difficulty [52, 104]. Accordingly, for each experiment, we generate feature datasets $(\boldsymbol{z}_1, \ldots \boldsymbol{z}_N) \in \mathcal{Z}$ from ground-truth parameter sets $(\boldsymbol{\zeta}_1, \ldots \boldsymbol{\zeta}_N) \in C$ using random mappings of increasing complexity. A feedforward neural network, $\boldsymbol{G}^k$, initialized uniformly at random with k layers, serves as the feature generator $\boldsymbol{z} = \boldsymbol{G}^k(\zeta)$. Evaluation is then carried out for each PtO task on feature datasets generated with $k \in \{1, 2, 4, 8\}$, keeping target parameters $\boldsymbol{\zeta}$ constant.

Baselines. In our experiments, LtOF models use feedforward networks with k hidden layers. For comparison, we also evaluate two-stage and, where applicable, EPO models, using architectures with k hidden layers where $k \in \{1, 2, 4, 8\}$. Further training specifics are provided in Appendix D.2.

Comparison to LtO setting. It is natural to ask how solution quality varies when transitioning from LtO to LtOF in a PtO setting, where solutions are learned directly from features. To address this question, each PtO experiment includes results from its analogous Learning to Optimize setting, where a DNN $\mathbf{F}_\omega : C \to \mathcal{X}$ learns a mapping from the parameters $\boldsymbol{\zeta}$ of an optimization problem to its corresponding solution $\boldsymbol{x}^\star(\boldsymbol{\zeta})$. This is denoted $k=0$ (LtO), indicating the absence of any feature mapping. All figures report the regret obtained by LtO methods for reference, although they are not directly comparable to the Predict-then-Optimize setting.

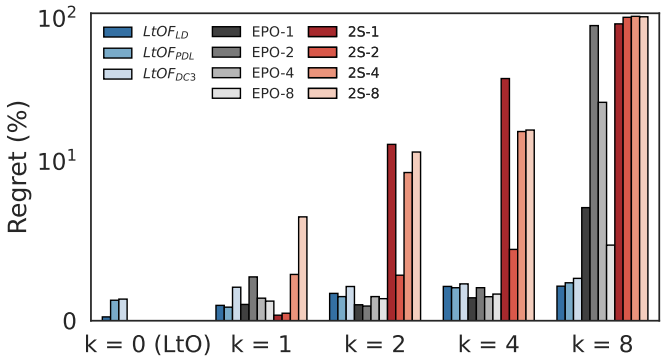


Figure 6.4: Comparison between **LtO** ($k = 0$), **LtOF**, Two-stage (**2S**) and **EPO** ($k > 1$) on the portfolio optimization. 2S(EPO)-m indicates that the prediction model of the respective PtO method is an m layer ReLU neural network.

	Method	Portfolio	N/conv. QP	AC-OPF
LtOF	LD it	**0.0003**	0.0000	**0.0004**
	LD fct	0.0000	0.0045	0.1573
	PDL it	**0.0003**	0.0000	0.0006
	PDL fct	0.0000	0.0045	0.1513
	DC3 it	0.0011	0.0001	-
	DC3 fct	0.0003	0.0000	-
PtO	PtO-1 et	0.0054	0.0122	0.1729
	PtO-2 et	0.0059	0.0104	0.1645
	PtO-4 et	0.0062	0.0123	0.1777
	PtO-8 et	0.0067	0.0133	0.1651

Table 6.1: Execution (*et*), inference (*it*), and feasibility correction (*fct*) times for **LtOF** and **PtO** (in seconds) for each sample. **Two-stage** methods execution times are comparable to PtO's ones.

Comparison to EPO with Pretrained Proxy. The end-to-end LtOF implementations are also compared against EPO models with pre-trained optimization proxies as a baseline, as described in Section 6.1.2.

All reported results are averages across 20 random seeds and the reader is referred to Appendix D.2 for additional details regarding experimental settings, architectures, and hyperparamaters adopted.

Convex Quadratic Portfolio Optimization A well-known problem combining prediction and optimization is the Markowitz Portfolio Optimization [129]. This task has as its optimization component a convex Quadratic Program:

$$\boldsymbol{x}^\star(\boldsymbol{\zeta}) = \underset{\boldsymbol{x}\geq \mathbf{0}}{\operatorname{argmax}} \ \boldsymbol{\zeta}^T\boldsymbol{x} - \lambda \boldsymbol{x}^T\boldsymbol{\Sigma}\boldsymbol{x}, \quad \text{s.t. } \mathbf{1}^T\boldsymbol{x} = 1 \tag{6.11}$$

in which parameters $\boldsymbol{\zeta} \in \mathbb{R}^D$ represent future asset prices, and decisions $\boldsymbol{x} \in \mathbb{R}^D$ represent their fractional allocations within a portfolio. The objective is to maximize a balance of risk, as measured by the quadratic form covariance matrix Σ, and total return $\boldsymbol{\zeta}^T\boldsymbol{x}$. Historical prices of $D = 50$ assets are obtained from the Nasdaq online database [112] and used to form price vectors $\boldsymbol{\zeta}_i$, $1 \leq i \leq N$, with $N = 12,000$ individual samples collected from 2015-2019. In the outputs $\hat{\boldsymbol{x}}$ of each LtOF method, possible feasibility violations are restored, at *low computational cost*, by first clipping $[\hat{\boldsymbol{x}}]_+$ to satisfy $\boldsymbol{x} \geq \mathbf{0}$, then dividing by its sum to satisfy $\mathbf{1}^T\boldsymbol{x} = 1$. The convex solver `cvxpy` [40] is used as the optimization component in each PtO method.

Results. Figure 6.4 shows the percentage regret due to LtOF implementations based on *LD*, *PDL* and *DC3*. Two-stage and EPO models are evaluated for comparison, with predictive components given various numbers of layers. For feature complexity $k > 1$, each LtOF model outperforms the best two-stage model, increasingly with k and up to nearly *two orders of magnitude* when $k = 8$. The EPO model, trained using exact derivatives through (6.11) as provided by the differentiable solver in `cvxpylayers` [4] is competitive with LtOF until $k = 4$, after which point its best variant is outperformed by each LtOF variant. This result showcases the ability of LtOF models to reach high accuracy under complex feature mappings *without* access to optimization problem solvers *or*

	Method	$k = 0$ (*LtO*)	$k = 1$	$k = 2$	$k = 4$	$k = 8$
LtOF	LD Regret	**1.2785**	0.9640	1.7170	2.1540	**2.1700**
	LD Regret (*)	1.1243	1.0028	1.5739	2.0903	2.1386
	LD Violation (*)	0.0037	0.0023	0.0010	0.0091	0.0044
	PDL Regret	1.2870	0.8520	1.5150	2.0720	2.3830
	PDL Regret (*)	1.2954	0.9823	1.4123	1.9372	2.0435
	PDL Violation (*)	0.0018	0.0097	0.0001	0.0003	0.0003
	DC3 Regret	1.3580	2.1040	2.1490	2.3140	2.6600
	DC3 Regret (*)	1.2138	1.8656	2.0512	1.9584	2.3465
	DC3 Violation (*)	0.0000	0.0000	0.0000	0.0000	0.0000
	Two-Stage Regret (Best)	-	**0.3480**	2.8590	4.4790	91.3260
	EPO Regret (Best)	-	1.0234	**0.9220**	**1.4393**	4.7495
	EPO Proxy Regret (Best)	-	136.4341	154.3960	119.3082	114.6953

Table 6.2: Regret and Constraint Violations for Portfolio Experiment. (*) denotes "Before Restoration".

their derivatives, in training or inference, in contrast to conventional PtO frameworks. Full accuracy results are reported in Table 6.2, which includes constraint violation and regret of the inferred solutions before feasibility restoration.

Table 6.1 presents LtOF inference times (*it*) and feasibility correction times (*fct*), which are compared with the per-sample execution times (*et*) for PtO methods. Run times for two-stage methods are closely aligned with those of EPO, and thus obmitted. Notice how the LtOF methods are at least an order of magnitude faster than PtO methods. This efficiency has two key implications: firstly, the per-sample speedup can significantly accelerate training for PtO problems. Secondly, it is especially advantageous during inference, particularly if data-driven decisions are needed in real-time.

Nonconvex QP Variant As a step in difficulty beyond convex QPs, this experiment considers a generic QP problem augmented with an additional oscillating objective term, resulting in a *nonconvex* optimization component:

$$\begin{aligned} \mathbf{x}^\star(\boldsymbol{\zeta}) = \underset{\boldsymbol{x}}{\operatorname{argmin}} \ & \frac{1}{2}\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} + \boldsymbol{\zeta}^T \sin(\boldsymbol{x}) \\ \texttt{s.t.} \ & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},\ \boldsymbol{G}\boldsymbol{x} \leq \boldsymbol{h}, \end{aligned}$$

in which the sin function is applied elementwise. This formulation was used to evaluate the LtO methods proposed both in [48] and in [119]. Following those works, $\mathbf{0} \preccurlyeq \boldsymbol{Q} \in \mathbb{R}^{n \times n}$, $\boldsymbol{A} \in \mathbb{R}^{n_{\text{eq}} \times n}$, $\boldsymbol{b} \in \mathbb{R}^{n_{\text{eq}}}$, $\boldsymbol{G} \in \mathbb{R}^{n_{\text{ineq}} \times n}$ and $\boldsymbol{h} \in \mathbb{R}^{n_{\text{ineq}}}$ have elements drawn uniformly at random. Here it is evaluated as part of a Predict-Then-Optimize pipeline in which predicted coefficients occupy the nonconvex term. Feasibility is restored by a projection onto the feasible set, which is calculated by a more efficiently solvable *convex* QP. The problem dimensions are $n = 50$ $n_{\text{eq}} = 25$, and $n_{\text{ineq}} = 25$.

Results. Figure 6.5 (left) shows regret due to LtOF models based on *LD*, *PDL* and *DC3*, along with two-stage baseline PtO methods. No EPO baselines are available due to the optimization

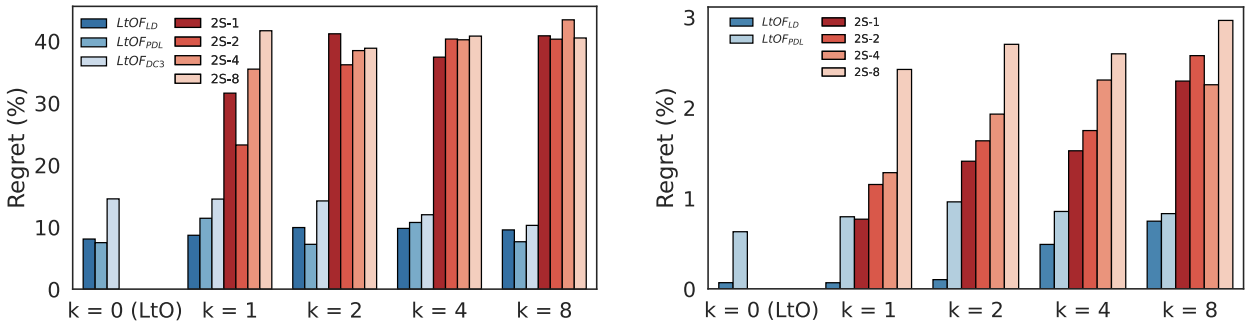


Figure 6.5: Comparison between LtO ($k = 0$), LtOF, and Two Stage Method (2S) on the nonconvex QP (**left**) and AC-OPF case (**right**). Right plot y-axis is in log-scale.

	Method	$k = 0$ (*LtO*)	$k = 1$	$k = 2$	$k = 4$	$k = 8$
LtOF	LD Regret	8.0757	**8.6826**	9.9279	**9.7879**	9.5473
	LD Regret (*)	8.1120	8.7416	9.9250	9.8211	9.5556
	LD Violation (*)	0.0753	0.0375	0.0148	0.0162	0.0195
	PDL Regret	**7.4936**	11.424	**7.2699**	10.7474	**7.6399**
	PDL Regret (*)	7.7985	11.429	7.2735	10.749	7.6394
	PDL Violation (*)	0.0047	0.0032	0.0028	0.0013	0.0015
	DC3 Regret	13.946	14.623	14.271	11.028	10.666
	DC3 Regret (*)	14.551	14.517	13.779	11.755	10.849
	DC3 Violation (*)	1.4196	0.8259	0.5158	0.5113	0.5192
	Two-Stage Regret (Best)	-	23.2417	36.1684	37.3995	38.2973
	EPO Proxy Regret (Best)	-	793.2369	812.7521	804.2640	789.5043

Table 6.3: Regret and Constraint Violations for Nonconvex QP Experiment. (*) denotes "Before Restoration".

component's nonconvexity. The best two-stage models perform poorly for most values of k, implying that the regret is particularly sensitive to prediction errors in the oscillating term. Thus its increasing trend with k is less pronounced than in other experiments. The best LtOF models achieve over 4 times lower regret than the best baselines, suggesting strong potential for this approach in contexts which require predicting parameters of non-linear objective functions. Additionally, the fastest LtOF method achieves up to three order magnitude speedup over the two-stage, after restoring feasibility.

Nonconvex AC-Optimal Power Flow Given a vector of marginal costs ζ for each power generator in an electrical grid, the AC-Optimal Power Flow problem optimizes the generation and dispatch of electrical power from generators to nodes with predefined demands. The objective is to minimize cost, while meeting demand exactly. The full optimization problem and more details are specified in Appendix D.1, where a quadratic cost objective is minimized subject to nonconvex physical and engineering power systems constraints. This experiment simulates a energy market situation in which generation costs are as-yet unknown to the power system planners, and must be estimated based on correlated data. The overall goal is to predict costs so as to minimize cost-regret over an

	Method	$k = 0$ (*LtO*)	$k = 1$	$k = 2$	$k = 4$	$k = 8$
LtOF	LD Regret	0.0680	**0.0673**	**0.1016**	**0.4904**	**0.7470**
	LD Regret (*)	0.0009	0.0009	0.0013	0.0071	0.0195
	LD Violation (*)	0.0035	0.0017	0.0020	0.0037	0.0042
	PDL Regret	0.6305	0.7958	0.9603	0.8543	0.8304
	PDL Regret (*)	0.0210	0.0242	0.0260	0.0243	0.0242
	PDL Violation (*)	0.0001	0.0002	0.0000	0.0002	0.0002
	Two-Stage Regret (Best)	-	0.7620	1.4090	1.5280	2.4740
	EPO Proxy Regret (Best)	-	431.7664	389.0421	413.8941	404.7452

Table 6.4: Regret and Constraint Violations for AC-OPF Experiment. (*) denotes "Before Restoration".

example network with 54 generators, 99 demand loads, and 118 buses taken from the well-known NESTA energy system test case archive [33]. Feasibility is restored for each LtOF model by a projection onto the nonconvex feasible set. Optimal solutions to the AC-OPF problem, along with such projections, are obtained using state-of-the-art Interior Point OPTimizer IPOPT [150].

6.1.5 Limitations, Discussion, and Conclusions

The primary *advantage* of the Learning to Optimize from Features approach to PtO settings is its generic framework, which enables it to leverage a variety of existing techniques and methods from the LtO literature. On the other hand, as such, a particular implementation of LtOF may inherit any limitations of the specific LtO method that it adopts. For example, when the LtO method does not ensure feasibility, the ability to restore feasibility may be need as part of a PtO pipeline. Future work should focus on understanding to what extent a broader variety of LtO methods can be applied to PtO settings; given the large variety of existing works in the area, such a task is beyond the scope of this paper. In particular, this paper does not investigate of the use of *combinatorial* optimization proxies in learning to optimize from features. Such methods tend to use a distinct set of approaches from those studied in this paper, often relying on training by reinforcement learning [16, 86, 105], and are not suited for capturing broad classes of optimization problems. As such, this direction is left to future work.

The main *disadvantage* inherent to any LtOF implementation, compared to end-to-end PtO, is the inability to recover parameter estimations from the predictive model, since optimal solutions are predicted end-to-end from features. Although it is not required in the canonical PtO problem setting, this may present a complication in situations where transferring the parameter estimations to external solvers is desirable. This presents an interesting direction for future work.

By showing that effective Predict-Then-Optimize models can be composed purely of Learning-to-Optimize methods, this paper has aimed to provide a unifying perspective on these as-yet distinct problem settings. The flexibility of its approach has been demonstrated by showing superior performance over PtO baselines with diverse problem forms. As the advantages of LtO are often best realized in combination with application-specific techniques, it is hoped that future work can build on these findings to maximize the practical benefits offered by Learning to Optimize in settings

that require data-driven decision-making.

6.2 Learning Bilevel Optimization for Optimal Control

Bilevel optimization problems arise in a wide range of control system applications, enabling the determination of system parameters that satisfy predefined objectives under optimal control policies. These problems have been instrumental in achieving state-of-the-art results in both the design and analysis of control systems. For example, in *inverse system design*, bilevel optimization is employed to infer the design parameters of a system by observing its optimal control sequences. Similarly, in optimal control co-design, the goal is to develop new system designs whose performance under optimal control meets additional economic and performance criteria. Though they provide a powerful analytical tool, bilevel optimization problems are notoriously difficult and computationally expensive to solve, especially when dealing with nonlinear systems, which is the case for many real settings. This complexity thus hinders their use in scenarios requiring real-time or repeated solutions.

To address these challenges, this paper proposes *Neural Control Codesign*, a novel method for learning to solve parametric bilevel optimization problems over optimal control. Building upon prior work in neural control policy learning, we demonstrate that these problems can be treated through unsupervised training of neural networks. The trained models effectively map the parameters of a bilevel problem to both its upper-level and lower-level solutions.

The approach of this paper is based on an assumed capability of solving and differentiating through the lower-level optimal control problem, as a function of its system design parameters for each iteration of training. This allows for a predictor of system parameters to be trained by a loss function on its resulting behavior, in terms of state trajectories and control sequences. However, such differentiable optimal control is difficult to implement in general, especially when the underlying control model is nonlinear. Additionally, solving the associated *nonconvex* programs during each forward pass of training can be prohibitively inefficient.

To overcome these challenges, we propose an alternative approach to differentiable optimal control within our end-to-end learning framework. Our proposal builds upon recent work demonstrating that recurrent neural networks can be trained as closed-loop optimal control policies, offering substantial runtime efficiency improvements over conventional model predictive control. In this paper, we recognize that such neural control models are *automatically differentiable* by construction while possessing the efficiency to also render the training of an upper-level prediction model highly efficient.

6.2.1 Problem Setting

We consider optimal control problems, which call for the control inputs to a dynamical system that optimize a given objective. The solution to such a problem consists of time-varying functions $\boldsymbol{u} : [0, 1] \to \mathbb{R}^{N_u}$ and $\boldsymbol{x} : [0, 1] \to \mathbb{R}^{N_x}$, which represent *control inputs* and their resulting system *state trajectories*, respectively. We view the problem parametrically, with dependence on parameters $\boldsymbol{p} \in \mathbb{R}^{N_p}$ which may represent initial and boundary conditions along with reference trajectories and objective function weights. Since we are interested in optimizing the *design* of a control system, we

also make explicit the dependence on a set of free *design variables* $\boldsymbol{c} \in \mathbb{R}^{N_c}$:

$$
\begin{aligned}
O(\boldsymbol{p}, \boldsymbol{c}) \coloneqq \underset{\boldsymbol{x}(t),\, \boldsymbol{u}(t)}{\operatorname{argmin}} \quad & l(\, \boldsymbol{x}(t),\ \boldsymbol{u}(t),\ \boldsymbol{p}\,) && && (6.13a)\\
s.t. \quad & \dot{\boldsymbol{x}}(t) = \boldsymbol{f}_{\boldsymbol{c}}(\, \boldsymbol{x}(t),\ u(t)\,) && && (6.13b)\\
& \boldsymbol{h}(\boldsymbol{x}(t),\ \boldsymbol{p}) = 0 && \forall t \in [0,1] && (6.13c)\\
& \boldsymbol{g}(\boldsymbol{u}(t),\ \boldsymbol{p}) \leq 0 && \forall t \in [0,1] && (6.13d)\\
& \boldsymbol{e}(\, \boldsymbol{x}(0), \boldsymbol{x}(1)\ , \boldsymbol{p}) = 0. && && (6.13e)
\end{aligned}
$$

The relationship between $x(t)$ and $u(t)$ is defined by the *dynamics* (6.13b), which take the form of an ordinary differential equation. Constraint (6.13d) represents physical limits on the control inputs which hold for all time. Similarly, (6.13c) constrains the state of the system, to reflect its physical limits as well as obstacles to be avoided. Additionally, (6.13e) indicates that additional constraints may be used to specify initial (at $t = 0$) and final (at $t = 1$) conditions of the system's state. Optimal control inputs $\boldsymbol{u}(t)$ and their resulting state trajectory $\boldsymbol{x}(t)$ are determined by optimizing the control objective l in (6.13a). For example, (6.13a) may call for the $\boldsymbol{u}(t)$ which causes $\boldsymbol{x}(t)$ to have minimal deviation from a desired reference trajectory. Design variables $\boldsymbol{c}$ primarily affect the system's dynamics, via $\boldsymbol{f}_{\boldsymbol{c}}$.

Discretization of the control problem Our approach to solving (6.13) will depend on discretization of the variable t into N_t time frames. Let $\boldsymbol{x}_k \in \mathbb{R}^{N_x}$ and $\boldsymbol{u}_k \in \mathbb{R}^{N_u}$ be the control and state variables at step k, as well as the k^{th} columns of matrices $\boldsymbol{x} \in \mathbb{R}^{N_x \times N_t}$, $\boldsymbol{u} \in \mathbb{R}^{N_u \times N_t}$. The continuous dynamics (6.13b) are then converted to discrete-time dynamics via a suitable integration method represented by $\bar{f}$; similarly we assume approximation of l by a discrete-time objective $\bar{l}$:

$$
\begin{aligned}
\bar{O}(\boldsymbol{p}, \boldsymbol{c}) \coloneqq \underset{\boldsymbol{x},\, \boldsymbol{u}}{\operatorname{argmin}} \quad & \sum_k \bar{l}(\, \boldsymbol{x}_k,\ \boldsymbol{u}_k,\ \boldsymbol{p}\,) && && (6.14a)\\
s.t. \quad & \boldsymbol{x}_{k+1} = \bar{\boldsymbol{f}}_{\boldsymbol{c}}(\, \boldsymbol{x}_k,\ \boldsymbol{u}_k\,) && \forall k\ \ 1 \leq k \leq N_t - 1 && (6.14b)\\
& \boldsymbol{h}(\, \boldsymbol{x}_k, \boldsymbol{p}\,) = \boldsymbol{0} && \forall k\ \ 1 \leq k \leq N_t && (6.14c)\\
& \boldsymbol{g}(\, \boldsymbol{u}_k,\ \boldsymbol{p}) \leq \boldsymbol{0} && \forall k\ \ 1 \leq k \leq N_t && (6.14d)\\
& \boldsymbol{e}_{\boldsymbol{c}}(\, \boldsymbol{x}_0, \boldsymbol{x}_{N_t}\,) = \boldsymbol{0}. && && (6.14e)
\end{aligned}
$$

Bilevel Optimization Setting

The goal of this paper is to learn to solve bilevel programs, whose lower level has the form (6.14). The *upper-level problem* (6.15) will be primarily concerned with optimizing the design variables $\boldsymbol{c}$ over a feasible set $\mathbb{C}$. The upper-level objective is modeled by a function $\mathcal{L}$ of $\boldsymbol{c}$ as well as its resulting states and optimal controls $(\, \boldsymbol{x}, \boldsymbol{u}\,)$. The constraint (6.15c) indicates how trajectories $(\, \boldsymbol{x}, \boldsymbol{u}\,)$ are related to design variables $\boldsymbol{c}$ via the lower-level problem (6.13):

$$
\begin{aligned}
\mathcal{B}(\boldsymbol{p}) = \operatorname{argmin}_{\boldsymbol{c}} \quad & \mathcal{L}_{\boldsymbol{p}}(\, \boldsymbol{c}, \boldsymbol{x}, \boldsymbol{u}\,) && (6.15a)\\
s.t. \quad & \boldsymbol{c} \in C && (6.15b)\\
& (\, \boldsymbol{x}, \boldsymbol{u}\,) \in \bar{O}(\boldsymbol{p}, \boldsymbol{c}). && (6.15c)
\end{aligned}
$$

Problem (6.15) above is a parametric bilevel optimization, whose solution can be viewed as a function of the parameters $\boldsymbol{p}$. Bilevel problems (6.15) are notoriously difficult to solve, even when $\mathcal{L}$, $\mathbb{C}$ and $\bar{O}$ are simple, due to the constraint (6.15c) which is itself defined by an optimization

problem (6.13). Further difficulty arises when $\mathcal{L}$ is nondifferentiable at the upper level, or when the lower-level problem defining (6.15c) is nonconvex, as occurs in our experiments (see Section 6.2.4). The goal of this paper is to develop a fast approximator, for settings which call for the problem (6.15) to be solved repeatly under various $\boldsymbol{p}$. Our approach, described in the next section, will be to *learn the mapping* $\boldsymbol{p} \to \mathcal{B}(\boldsymbol{p})$ by unsupervised training of a neural network.

6.2.2 Learning Bilevel Optimization for Optimal Control

This section proposes an unsupervised method for learning to solve the parametric bilevel optimization problems described in Section 6.2.1. In particular, it trains a neural network to approximate the mapping (6.15), from problem parameters $\boldsymbol{p}$ to *upper-level* solutions $\mathcal{B}(\boldsymbol{p})$. The general framework is based on employing a differentiable solver of (6.13) within the training loop, in order to ensure satisfaction of constraint (6.15c). However, this concept becomes especially difficult to implement when the optimal control problem (6.13) is nonlinear or otherwise difficult to solve. Solution of (6.13) even at each forward-pass iteration can be *cost-prohibitive* in many cases.

The *key insight* of this paper is that we can overcome those challenges by incorporating existing techniques in the realm of learning to solve the lower-level problem (6.13). In particular, we may train a recurrent neural network as a closed-loop control policy to solve (6.15), as a more efficient alternative to model-predictive control (MPC). In addition to their runtime efficiency, we show for the first time how such learned neural control policies can be leveraged for their *differentiability* via automatic differentiation. This insight enables an efficient framework for learning to solve parametric bilevel problems (6.15,6.14) which consists purely of unsupervised neural network training, *without* the need for MPC solvers. We first describe a generic framework for learning the upper-level mapping (6.15) based on differentiable MPC, and then its modification via a learned model of the lower-level mapping (6.13).

Learning the Upper-Level Solution Mapping

We first describe a generic method for learning the upper-level solutions $\boldsymbol{c} = \mathcal{B}(\boldsymbol{p})$ as a function of problem parameters $\boldsymbol{p}$. We model the function $\mathcal{B}$ using a neural network $\hat{\mathcal{B}}_\theta$ with weights θ, and pair it with differentiable components that maintain feasibility of predicted solutions throughout training. First, we assume access to a differentiable operation $\boldsymbol{\Pi}_{\mathbb{C}}$ which maps points in $\mathbb{R}^{N_c}$ to $\mathbb{C}$. Its implementation depends on the form of $\mathbb{C}$, and may for example consist of **(1)** a sigmoid function when $\mathbb{C}$ is a bounded box or interval, or **(2)** a differentiable projection onto $\mathbb{C}$. Second, a differentiable MPC solver implementing $\bar{O}$ is required to map candidate design solutions to their resulting trajectories.

Algorithm 4: *Neural Bilevel Optimal Control*

input : $\{p_{(i)}\}_{i=1}^{N}$: Input parameters, $\hat{\mathcal{B}}_{\boldsymbol{\theta}}$: a neural network with weights θ, α: the learning rate

for $i = 1$ *to* M **do**
- $\hat{\boldsymbol{c}} \leftarrow \hat{\mathcal{B}}_{\boldsymbol{\theta}}(\boldsymbol{p}_{(i)})$
- $\hat{\boldsymbol{c}} \leftarrow \boldsymbol{\Pi}_{\mathbb{C}}(\hat{\boldsymbol{c}})$
- $(\hat{\boldsymbol{x}}, \hat{\boldsymbol{u}}) \leftarrow \bar{O}(\boldsymbol{p}_{(i)}, \hat{\boldsymbol{c}})$ by solving (6.13)
- $\boldsymbol{g} \leftarrow \nabla_\theta \mathcal{L}_{\boldsymbol{p}_{(i)}}(\, \hat{\boldsymbol{c}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{u}}\,)$ by backpropagation through $\bar{O}$ and $\boldsymbol{\Pi}_{\mathbb{C}}$
- $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \boldsymbol{g}$

return $\hat{\mathcal{B}}_{\boldsymbol{\theta}}$

Algorithm 6 outlines the overall procedure. Given a prediction $\hat{\boldsymbol{c}} = \hat{\mathcal{B}}_\theta(\boldsymbol{p})$ at Line 2, Line 3 maps the prediction to a feasible point in $\mathbb{C}$, ensuring constraint (6.15b). Line 4 then computes resulting trajectories $(\hat{\boldsymbol{x}}, \hat{\boldsymbol{u}}) = \bar{O}(\boldsymbol{p}, \hat{\boldsymbol{c}})$ by solving problem (6.13), so that (6.15c) is also satisfied. Ultimately, Line 6 calls for a gradient descent step on upper-level objective value $F(\hat{\boldsymbol{c}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{u}})$, with respect to the weights of $\hat{\mathcal{B}}_\theta$. The prerequisite backpropagation of gradients at Line 5 can be carried out automatically on the overall model's computational graph, given differentiability of each preceding step.

This proposed framework works by minimizing the empirical objective value of the upper-level problem, while maintaining feasibility of the predicted solutions to (6.15b) and (6.15c). Importantly however, the bilevel problem (6.15, 6.14) must satisfy two main conditions for the method to be viable: **(1)** The lower-level problem (6.13) has a unique optimal solution for any choice of $(\boldsymbol{c}, \boldsymbol{p})$. Indeed, this is required for the mapping $\bar{O}$ to be a function, and uniqueness can generally be ensured by sufficiently specifying $\bar{l}$ in (6.13). **(2)** The upper-level problem is free of *coupling constraints* which jointly constrain the variables $\boldsymbol{c}$ with $\boldsymbol{x}$ and $\boldsymbol{u}$. Section 6.2.3 extends this approach to handle coupling constraints. As alluded earlier, the main drawback of Algorithm 6 is its reliance on solving and differentiating the problem (6.14) at each training iteration. This quickly becomes cost-prohibitive when problem (6.14) is nontrivial to solve. Next we discuss the incorporation of neural network models to replace $\bar{O}$ at the lower level in Line 4 to overcome this challenge.

Integrating Lower-Level Neural Control

We propose to enhance the computational efficiency of Algorithm 6, by implementing the differentiable optimal control solver $\bar{\boldsymbol{O}}$ at line 4 with a lightweight neural network. Recent work on learning optimal control policies has demonstrated the potential of training recurrent neural networks (RNNs) as closed-loop policies which execute much faster than traditional MPC, often with comparable accuracy. In this section, we recognize that such RNN policies are *automatically differentiable* by construction, and thus offer an *efficient alternative* to differentiable optimization. This work is the first to propose the integration of such models into end-to-end trainable pipelines, by leveraging their automatic differentiability with respect to their design variables. To do so, we must train the RNN policy over a joint distribution of lower-level problem parameters as well as upper-level design variables.

We introduce a recurrent neural network $\boldsymbol{\pi}_\phi$, which learns a closed-loop control policy by modeling control actions $\boldsymbol{u}_{k+1}$ as a function of the current state $\boldsymbol{x}_k$ along with sequential input data such as a reference trajectory $\{\boldsymbol{r}_k \in \mathbb{R}^{N_x} : 1 \leq k \leq N_t\}$. It is proposed in previous work to train it over a joint distribution of parameters such as initial and reference states, which equate to $\boldsymbol{p}$ in problem (6.13). Here we propose to train it *also* over a distribution of design parameters C which are uniformly sampled from their feasible space $\mathbb{C}$. This allows the trained control policy to act as a differentiable function of the design parameters learned by the upper-level prediction model.

The training objective for the control policy $\boldsymbol{\pi}_\phi$ is as follows:

$$\min_{\phi} \quad \mathbb{E}_{\boldsymbol{p}\sim\mathcal{P},\boldsymbol{c}\sim\mathcal{C}} \sum_{k} \bar{l}(\, \boldsymbol{x_k},\, \boldsymbol{u_k},\, \boldsymbol{p}\,) \tag{6.16a}$$

$$\text{s.t.} \quad \boldsymbol{x}_{k+1} = \bar{\boldsymbol{f}}_{\boldsymbol{c}}(\, \boldsymbol{x}_k,\, \boldsymbol{u}_k\,) \qquad \forall k \;\; 1 \le k \le N_t - 1 \tag{6.16b}$$

$$\boldsymbol{u}_{k+1} = \boldsymbol{\pi}_{\phi}(\, \boldsymbol{x}_k,\, \boldsymbol{p},\, \boldsymbol{c}\,) \qquad \forall k \;\; 1 \le k \le N_t - 1 \tag{6.16c}$$

$$\boldsymbol{h}(\boldsymbol{x}_k,\, \boldsymbol{p}) \le \boldsymbol{0} \qquad \forall k \;\; 1 \le k \le N_t \tag{6.16d}$$

$$\boldsymbol{g}(\boldsymbol{u}_k,\, \boldsymbol{p}) \le \boldsymbol{0} \qquad \forall k \;\; 1 \le k \le N_t \tag{6.16e}$$

$$\boldsymbol{e}(\, \boldsymbol{x}(0), \boldsymbol{x}(1)\, , \boldsymbol{p}) = 0, \tag{6.16f}$$

With a policy model $\boldsymbol{\pi}_{\phi}$ trained according to (6.16), we can estimate solutions to the lower-level problem (6.15) by rolling out the operations (6.16c,6.16b) from any initial state $\boldsymbol{x}_0$, ensuring satisfaction of the constraints (6.16b) and (6.16c). Constraints (6.16e) typically arise as bounds on the control inputs, enforcible with sigmoid activations within $\boldsymbol{\pi}_{\theta}$. Finally, (6.16d) is often enforced using a penalty on its violation, incorporated in $\bar{l}$. The rollout of $\boldsymbol{\pi}_{\phi}$ and $\bar{\boldsymbol{f}}_{\boldsymbol{c}}$ yields a mapping $\hat{\boldsymbol{O}}$, which can be used as a differentiable proxy for $\bar{\boldsymbol{O}}$ in Algorithm 6.

6.2.3 Incorporating Coupling Constraints

So far we have described a system for learning to solve bilevel problems (6.15) by leveraging neural networks as learned proxies of the lower-level control problem (6.14). Compared to other proposals for differentiable optimal control, this approach primarily offers the benefit of speed. In this section, we show how that speed advantage can be leveraged into a *modeling* advantage by building an iterative correction mechanism for *coupling constraints* on top of the learned proxies.

Our parametric bilevel problem can be restated with coupling constraints defined by the function $\mathcal{U}$:

$$\mathcal{B}(\boldsymbol{p}) = \text{argmin}_{\boldsymbol{c}} \quad \mathcal{L}_{\boldsymbol{p}}(\, \boldsymbol{c}, \boldsymbol{x}, \boldsymbol{u}\,) \tag{6.17a}$$

$$\text{s.t.} \quad \boldsymbol{c} \in \mathcal{C} \tag{6.17b}$$

$$(\, \boldsymbol{x}, \boldsymbol{u}\,) \in \bar{O}(\boldsymbol{p}, \boldsymbol{c}) \tag{6.17c}$$

$$U(\boldsymbol{x}, \boldsymbol{c}) \le \boldsymbol{0} \tag{6.17d}$$

Our efficient proxy for solving the lower-level problem *barO* allows the problem to be repeatedly solved and differentiated - this allows us to propose a graidient-based strategy for minimizing the violation of 6.17d with an internal gradient based optimization. Similar to the popular DC3 method for learning to optimize, we propose to iteratively satisfy 6.17d while maintaining feasibility to 6.17c and 6.17b by performing a projected gradient descent method on the coupling constraint violation, over the predicted design variables c. Algorithm 6 describes the resulting *Coupling Constraint Correction*.

Algorithm 5: *Coupling Constraint Correction*

input : $\hat{c}$: predicted variables, γ: stepsize, m number of steps

for $i = 1$ *to* m **do**
- $\hat{\boldsymbol{c}} \leftarrow \boldsymbol{\Pi}_{\mathbb{C}}(\hat{\boldsymbol{c}})$
- $(\hat{\boldsymbol{x}}, \hat{\boldsymbol{u}}) \leftarrow \bar{O}(\boldsymbol{p}_{(i)}, \hat{\boldsymbol{c}})$ by solving (6.13)
- $\boldsymbol{g} \leftarrow \nabla \| [U(\ \hat{\boldsymbol{c}}, \hat{\boldsymbol{x}}\ )]_+ \|^2$ by backpropagation through $\bar{O}$ and $\boldsymbol{\Pi}_{\mathbb{C}}$
- $\hat{\boldsymbol{c}} \leftarrow \hat{\boldsymbol{c}} + \gamma \cdot \boldsymbol{g}$

return $\hat{c}$

Using this mechanism as a means to ensure feasibility of any design to the full set of upper-level constraints, we may augment Algorithm 4 to incorporate coupling constraints. Since our aim is to solve optimal control co-design problems, Algorithm 6 is called Neural Control Codesign.

Algorithm 6: *Neural Control Codesign*

input : $\{p_{(i)}\}_{i=1}^{N}$: Input parameters, $\hat{\mathcal{B}}_{\boldsymbol{\theta}}$: a neural network with weights θ, α: the learning rate

for $i = 1$ *to* M **do**
- $\hat{\boldsymbol{c}} \leftarrow \hat{\mathcal{B}}_{\boldsymbol{\theta}}(\boldsymbol{p}_{(i)})$
- **for** $i = 1$ *to* m **do**
  - $\hat{\boldsymbol{c}} \leftarrow \boldsymbol{\Pi}_{\mathbb{C}}(\hat{\boldsymbol{c}})$
  - $(\hat{\boldsymbol{x}}, \hat{\boldsymbol{u}}) \leftarrow \bar{O}(\boldsymbol{p}_{(i)}, \hat{\boldsymbol{c}})$ by solving (6.13)
  - $\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{c}}} \| [U(\ \hat{\boldsymbol{c}}, \hat{\boldsymbol{x}}\ )]_+ \|^2$ by backpropagation through $\bar{O}$ and $\boldsymbol{\Pi}_{\mathbb{C}}$
  - $\hat{\boldsymbol{c}} \leftarrow \hat{\boldsymbol{c}} + \gamma \cdot \boldsymbol{g}$
- $(\hat{\boldsymbol{x}}, \hat{\boldsymbol{u}}) \leftarrow \bar{O}(\boldsymbol{p}_{(i)}, \hat{\boldsymbol{c}})$ by solving (6.13)
- $\boldsymbol{g} \leftarrow \nabla_{\theta} \mathcal{L}_{\boldsymbol{p}_{(i)}}(\ \hat{\boldsymbol{c}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{u}}\ )$ by backpropagation through the inner correction loop
- $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \boldsymbol{g}$

return $\hat{\mathcal{B}}_{\boldsymbol{\theta}}$

6.2.4 Experiments

We evaluate the ability of Neural Control Codesign to solve bilevel optimization problems over various control systems. The lower-level neural control models are all trained using the NeuroMANCER library [49]. All experimental models are implemented in PyTorch and trained using the Adam optimizer [166]. To our knowledge, there exists no solver package capable of solving the bilevel optimization problems being learned in these experiments, which include nonconvex lower-level problems and nondifferentiable objectives at the upper level (although they may be treated approximately with evolutionary algorithms. Therefore, objective values of the learned solutions are reported in terms of their nominal values, rather than more informative optimality gaps.

Design of a Closed-Loop System

We consider a canonical nonlinear control problem, in which two connected tanks are controlled by a single pump and a two-way valve. The system is a simplified model of a pumped-storage hydroelectricity, which is a form of energy storage used by electric power systems for load balancing. The system dynamics are described by the following nonlinear ODE's:

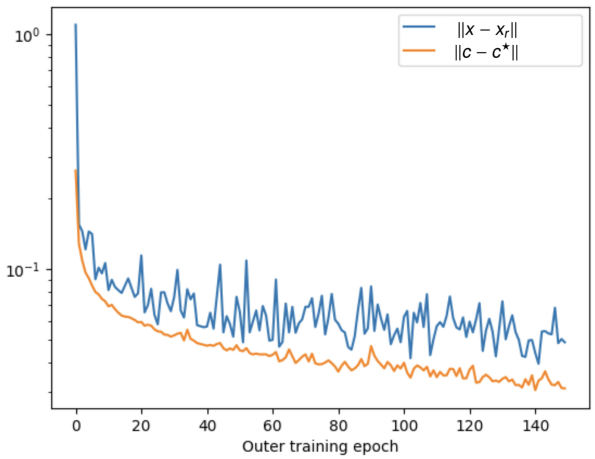


Figure 6.6: Training curves for the inverse design problem.

$$\dot{x_1} = c_1(1 - v)p - c_2\sqrt{x_1} \tag{6.18a}$$

$$\dot{x_2} = c_1 vp + c_2\sqrt{x_1} - c_2\sqrt{x_2} \tag{6.18b}$$

in which x_1, x_2 are the water levels in each tank. Control actions consist of p and v, which are the pump modulation and valve opening. The inlet and outlet valve coefficients c_1, c_2 comprise the system's free design parameters in this experiment.

Learning Inverse Design of a Closed-Loop System

We first evaluate the proposed method on an inverse design task: Assume we have knowledge of the system's control objective given a single observed state trajectory from a two-tank system. Given an observed state trajectory $\boldsymbol{x}$, predict its design parameters c_1 and c_2. The training loss for this task is the MSE residual from the target trajectory, and the optimal trajectory which results from a predicted design $\boldsymbol{c}$. We learn to solve the resulting bilevel problem with Algorithm 4.

Figure 6.6 shows that when learning over a distribution of observed state trajectories, the residual error in the state variable goes to zero as a loss function. As a result, the mis-specification of the underlying design variables also goes to zero, showing that design parameters are recovered to high accuracy on average. Figure 6.7 shows an example of the resulting learned trajectories, relative to the model's target trajectory.

An Optimal Control Co-Design Task

The controllability of the two-tank system is highly dependent on the relative values of c_1 and c_2. It happens that when c_1 is much smaller than c_2, the system cannot be reliably controlled to a reference state. We suppose that an economic cost is associated to the value of c_1. In this task, we wish to find the design with minimal sum $c_1 + c_2$ which allows the system to be controlled to a given reference state. This is expressed as a coupling constraint $\boldsymbol{x}_N - \boldsymbol{x}_r = 0$. The upper-level objective is to minimize $c_1 + c_2$.

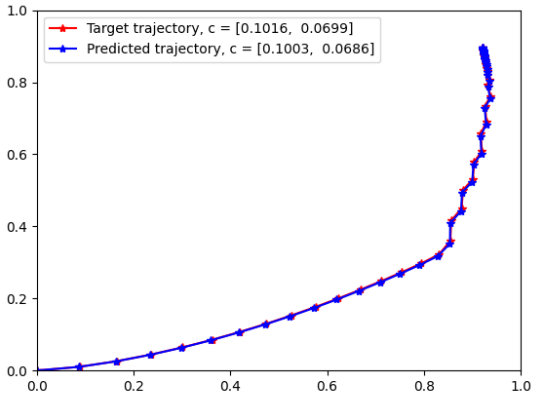


Figure 6.7: Example learned and target trajectories in the inverse design problem.

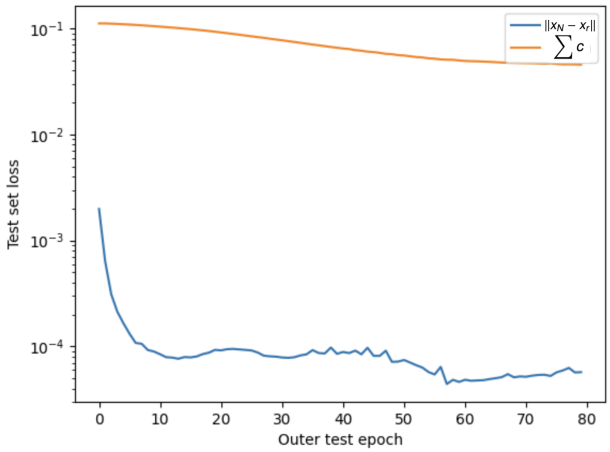


Figure 6.8: Training curves for the optimal control co-design problem.

Figure 6.8 shows the upper-level objective along with the coupling constraint residual, thoughout training via Algorithm 6. The coupling constraint suffers negligible violation, while the design objective is simultaneously driven down. Figure 6.9 shows an example of the resulting system behavior for a given target state, which the system just barely achieves under its control policy.

6.2.5 Conclusions

This section detailed designs and results of some preliminary work aimed at showing the promise of using learned optimizers as differentiable control policies within end-to-end trainable models. The inherent efficiency and differentiability of the those learned models allowed us to build efficient constraint correction mechanisms based on their repeated solution and backpropagation. This in turn allowed for the development of methods which learn to solve bilevel optimization problems subject to difficult coupling constraints. It is hoped that this effort can help encourage future work on integrating learned optimization models within larger end-to-end frameworks.

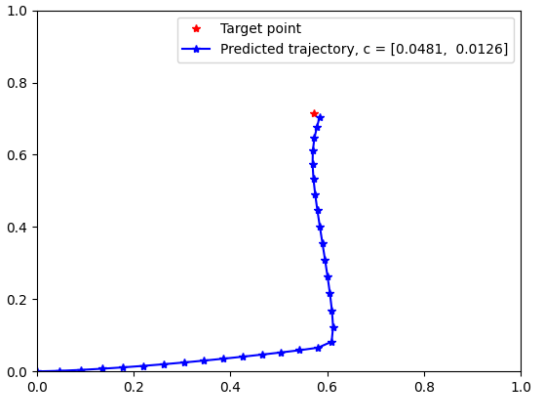


Figure 6.9: Example dynamics from the minimal system capable of reaching a target state.

Chapter 7

Summary and Conclusion

This thesis has aimed to study the algorithmic foundations for the integration of machine learning models with constrained optimization methods, as well as its broad applications to enhancing performances on both machine learning and optimization tasks. We began by categorizing this topic into three distinct domains, named Learning to Optimize, Predict-Then-Optimize, and Differentiable Programming. Within each domain, we have described our methodological contributions as well as our original designs aimed at enhancing performance in various application areas.

As part of a commentary on future directions for the space, we ended with a chapter which explores the integration of these three major domains, and the extent to which techniques from one problem setting may benefit the other. By proposing algorithm designs which blur the lines between Learning to Optimize, Predict-Then-Optimize, and Differentiable Programming, we hope to make the case that the Integration of Optimization and Machine Learning merits a field of study in its own right, which should encompass all three of the major domains studied in this thesis.

As advances in machine learning continue to change the landscape of computational science, and the availability of data grows, informed decisions will increasing rely on a combination of predictive and prescriptive modeling. It is hoped that this thesis can provide a comprehensive view of these technologies in the current year, and how they can be not only combined but integrated together, to enable more than was possible by the sum of their individual parts.

Bibliography

[1] J. Abernethy, C. Lee, and A. Tewari. Perturbation techniques in online learning and optimization. *Perturbations, Optimization, and Statistics*, 233, 2016.

[2] R. P. Adams and R. S. Zemel. Ranking via sinkhorn propagation. *arXiv preprint arXiv:1106.1925*, 2011.

[3] A. Agarwal, A. Beygelzimer, M. Dudik, J. Langford, and H. Wallach. A reductions approach to fair classification. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[4] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and J. Z. Kolter. Differentiable convex optimization layers. *Advances in neural information processing systems*, 32, 2019.

[5] B. Amos and J. Z. Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *ICML*, pages 136–145. JMLR. org, 2017.

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

[7] B. Amos, V. Koltun, and J. Z. Kolter. The limited multi-label projection layer. *arXiv preprint arXiv:1906.08707*, 2019.

[8] B. Amos et al. Tutorial on amortized optimization. *Foundations and Trends® in Machine Learning*, 16(5):592–732, 2023.

[9] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee. Understanding deep neural networks with rectified linear units. *arXiv preprint arXiv:1611.01491*, 2016.

[10] H. Attouch, J. Bolte, and B. F. Svaiter. Convergence of descent methods for semi-algebraic and tame problems: proximal algorithms, forward–backward splitting, and regularized gauss–seidel methods. *Mathematical Programming*, 137(1):91–129, 2013.

[11] C. Audet, J. Brimberg, P. Hansen, S. L. Digabel, and N. Mladenović. Pooling problem: Alternate formulations and solution methods. *Management science*, 50(6):761–776, 2004.

[12] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik. Learning to branch. In *International conference on machine learning*, pages 344–353. PMLR, 2018.

[13] H. H. Bauschke and P. L. Combettes. *Convex Analysis and Monotone Operator Theory in Hilbert Spaces*. Springer, New York, New York, USA, 2019.

[14] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear programming and network flows*. John Wiley & Sons, 2008.

[15] A. Beck. *First-order methods in optimization*. SIAM, 2017.

[16] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv:1611.09940*, 2017.

[17] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

[18] M. Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182(2):418–477, 2002. ISSN 0021-9991. doi:https://doi.org/10.1006/jcph.2002.7176. URL https://www.sciencedirect.com/science/article/pii/S0021999102971767.

[19] L. Berrada, A. Zisserman, and M. P. Kumar. Smooth loss functions for deep top-k classification. *ArXiv*, abs/1802.07595, 2018.

[20] Q. Berthet, M. Blondel, O. Teboul, M. Cuturi, J.-P. Vert, and F. Bach. Learning with differentiable pertubed optimizers. *Advances in neural information processing systems*, 33: 9508–9519, 2020.

[21] D. Bertsimas and B. Stellato. The voice of optimization. *Machine Learning*, 110(2):249–277, 2021.

[22] G. Birkhoff. *Lattice theory*, volume 25. American Mathematical Soc., 1940.

[23] M. Blondel, O. Teboul, Q. Berthet, and J. Djolonga. Fast differentiable sorting and ranking. In *International Conference on Machine Learning*, pages 950–959. PMLR, 2020.

[24] S. Boyd, S. P. Boyd, and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[25] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.

[26] D. Cajas. Owa portfolio optimization: A disciplined convex programming framework. *Available at SSRN 3988927*, 2021.

[27] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136, 2007.

[28] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544*, 2021.

[29] L. E. Celis, D. Straszak, and N. K. Vishnoi. Ranking with fairness constraints. *arXiv preprint arXiv:1704.06840*, 2017.

[30] Y. Chen and A. Zhou. Multiobjective portfolio optimization via pareto front evolution. *Complex And Intelligent Systems*, 8:4301–4317, 2022. doi:10.1007/s40747-022-00715-8. URL https://doi.org/10.1007/s40747-022-00715-8.

[31] K. F. E. Chong. A closer look at the approximation capabilities of neural networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=rkevSgrtPr.

[32] C. Coffrin, D. Gordon, and P. Scott. NESTA, the NICTA energy system test case archive. *CoRR*, abs/1411.0359, 2014. URL http://arxiv.org/abs/1411.0359.

[33] C. Coffrin, D. Gordon, and P. Scott. Nesta, the nicta energy system test case archive. *arXiv preprint arXiv:1411.0359*, 2014.

[34] C. Coffrin, R. Bent, K. Sundar, Y. Ng, and M. Lubin. Powermodels.jl: An open-source framework for exploring power flow formulations. In *PSCC*, June 2018.

[35] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[36] G. B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity analysis of production and allocation*, 13:339–347, 1951.

[37] L. Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. doi:10.1109/MSP.2012.2211477.

[38] F. Detassis, M. Lombardi, and M. Milano. Teaching the old dog new tricks: supervised learning with constraints. In A. Saffiotti, L. Serafini, and P. Lukowicz, editors, *Proceedings of the First International Workshop on New Foundations for Human-Centered AI (NeHuAI)*, volume 2659 of *CEUR Workshop Proceedings*, pages 44–51, 2020.

[39] A. Deza and E. B. Khalil. Machine learning for cutting planes in integer programming: A survey. *arXiv preprint arXiv:2302.09166*, 2023.

[40] S. Diamond and S. Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[41] P. Diederik and B. Jimmy. Adam: A method for stochastic optimization. iclr. *arXiv preprint arXiv:1412.6980*, 2014.

[42] M. H. Dinh, J. Kotary, and F. Fioretto. Differentiable approximations of fair owa optimization. In *ICML 2024 Workshop on Differentiable Almost Everything: Differentiable Relaxations, Algorithms, Operators, and Simulators*.

[43] M. H. Dinh, J. Kotary, and F. Fioretto. End-to-end learning for fair multiobjective optimization under uncertainty. *arXiv preprint arXiv:2402.07772*, 2024.

[44] M. H. Dinh, J. Kotary, and F. Fioretto. Learning fair ranking policies via differentiable optimization of ordered weighted averages. In *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, pages 2508–2517, 2024.

[45] V. Do and N. Usunier. Optimizing generalized gini indices for fairness in rankings. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 737–747, 2022.

[46] J. Dona and P. Gallinari. Differentiable feature selection, a reparameterization approach. In *Machine Learning and Knowledge Discovery in Databases. Research Track: European Conference, ECML PKDD 2021, Bilbao, Spain, September 13–17, 2021, Proceedings, Part III 21*, pages 414–429. Springer, 2021.

[47] P. L. Donti, J. Z. Kolter, and B. Amos. Task-based end-to-end model learning in stochastic optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5484–5494, 2017.

[48] P. L. Donti, D. Rolnick, and J. Z. Kolter. Dc3: A learning method for optimization with hard constraints. *arXiv preprint arXiv:2104.12225*, 2021.

[49] J. Drgona, A. Tuor, J. Koch, M. Shapiro, and D. Vrabie. NeuroMANCER: Neural Modules with Adaptive Nonlinear Constraints and Efficient Regularizations. 2023. URL https://github.com/pnnl/neuromancer.

[50] J. Eckstein. *Splitting methods for monotone operators with applications to parallel optimization*. Phd thesis, MIT, 1989.

[51] S. Elbassuoni, S. Amer-Yahia, A. Ghizzawi, and C. Atie. Exploring fairness of ranking in online job marketplaces. In *22nd International Conference on Extending Database Technology (EDBT)*, 2019.

[52] A. N. Elmachtoub and P. Grigas. Smart "predict, then optimize". *Management Science*, 2021.

[53] A. Ferber, B. Wilder, B. Dilkina, and M. Tambe. Mipaal: Mixed integer program as a layer. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[54] F. Fioretto, P. V. Hentenryck, T. W. Mak, C. Tran, F. Baldo, and M. Lombardi. Lagrangian duality for constrained deep learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 118–135. Springer, 2020.

[55] F. Fioretto, P. V. Hentenryck, T. W. K. Mak, C. Tran, F. Baldo, and M. Lombardi. Lagrangian duality for constrained deep learning. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD*, volume 12461, pages 118–135. Springer, 2020.

[56] F. Fioretto, T. W. Mak, and P. Van Hentenryck. Predicting ac optimal power flows: Combining deep learning and lagrangian dual methods. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 630–637, 2020.

[57] F. Fioretto, T. W. Mak, and P. Van Hentenryck. Predicting ac optimal power flows: Combining deep learning and lagrangian dual methods. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 630–637, 2020.

[58] G. B. Folland. *Real analysis: modern techniques and their applications*, volume 40. John Wiley & Sons, 1999.

[59] J. Fu, H. Luo, J. Feng, K. H. Low, and T.-S. Chua. Drmad: Distilling reverse-mode automatic differentiation for optimizing hyperparameters of deep neural networks. *arXiv preprint arXiv:1601.00917*, 2016.

[60] D. Gabay. Applications of the method of multipliers to variational inequalities. In *Augmented Lagrangian Methods: Applications to the Solution of Boundary-Value Problems*. North Holland, Amsterdam, The Netherlands, 1983.

[61] M. Gagliolo and J. Schmidhuber. Learning restart strategies. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 792–797, 2007.

[62] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 15554–15566, 2019.

[63] E. Ghadimi, A. Teixeira, I. Shames, and M. Johansson. Optimal parameter selection for the alternating direction method of multipliers (admm): Quadratic problems. *IEEE Transactions on Automatic Control*, 60(3):644–658, 2015. doi:10.1109/TAC.2014.2354892.

[64] P. Giselsson and S. Boyd. Metric selection in fast dual forward–backward splitting. *Automatica*, 62:1–10, 2015.

[65] P. Giselsson and S. Boyd. Linear convergence and metric selection for douglas-rachford splitting and admm. *IEEE Transactions on Automatic Control*, 62(2):532–544, 2017. doi:10.1109/TAC.2016.2564160.

[66] E. J. Gumbel. *Statistical theory of extreme values and some practical applications: a series of lectures*, volume 33. US Government Printing Office, 1954.

[67] P. Gupta, M. Gasse, E. Khalil, P. Mudigonda, A. Lodi, and Y. Bengio. Hybrid models for learning to branch. *Advances in neural information processing systems*, 33:18087–18097, 2020.

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

[69] M. R. Hestenes. Multiplier and gradient methods. *Journal of optimization theory and applications*, 4(5):303–320, 1969.

[70] J. Hopfield and D. Tank. Neural computation of decisions in optimization problems. *Biological cybernetics*, 52:141–52, 02 1985. doi:10.1007/BF00339943.

[71] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[72] C. Huang. Relu networks are universal approximators via piecewise linear or constant functions. *Neural Computation*, 32(11):2249–2278, 2020.

[73] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[74] D. A. Iancu and N. Trichakis. Fairness and efficiency in multiportfolio optimization. *Operations Research*, 62(6):1285–1301, 2014. doi:10.1287/opre.2014.1310. URL https://doi.org/10.1287/opre.2014.1310.

[75] T. Joachims, A. Swaminathan, and T. Schnabel. Unbiased learning-to-rank with biased feedback. In *WSDM*, pages 781–789, 2017.

[76] C. Ju, A. Bibaut, and M. van der Laan. The relative performance of ensemble methods with deep convolutional neural networks for image classification. *Journal of Applied Statistics*, 45 (15):2800–2818, 2018.

[77] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, H. Eichner, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konecný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, H. Qi, D. Ramage, R. Raskar, M. Raykova, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021. ISSN 1935-8237. doi:10.1561/2200000083. URL http://dx.doi.org/10.1561/2200000083.

[78] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[79] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 30, 2016.

[80] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *NIPS*, pages 6348–6358, 2017.

[81] E. B. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6348–6358, 2017.

[82] E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao. Learning to run heuristics in tree search. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, (IJCAI)*, pages 659–666, 2017. doi:10.24963/ijcai.2017/92. URL https://doi.org/10.24963/ijcai.2017/92.

[83] E. King, J. Kotary, F. Fioretto, and J. Drgona. Metric learning to accelerate convergence of operator splitting methods for differentiable parametric programming. *IEEE Conference on Decision and Control*, 2024.

[84] A. V. Konstantinov and L. V. Utkin. A new computationally simple approach for implementing neural networks with output hard constraints. In *Doklady Mathematics*, pages 1–9. Springer, 2024.

[85] W. Kool, H. Van Hoof, and M. Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

[86] W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems! In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[87] M. M. Kostreva and W. Ogryczak. Linear optimization with multiple equitable criteria. *RAIRO-Operations Research-Recherche Opérationnelle*, 33(3):275–297, 1999.

[88] J. Kotary, F. Fioretto, and P. Van Hentenryck. Learning hard optimization problems: A data generation perspective. *Advances in Neural Information Processing Systems*, 34:24981–24992, 2021.

[89] J. Kotary, F. Fioretto, P. Van Hentenryck, and B. Wilder. End-to-end constrained optimization learning: A survey. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4475–4482, 2021. doi:10.24963/ijcai.2021/610. URL https://doi.org/10.24963/ijcai.2021/610.

[90] J. Kotary, F. Fioretto, and P. Van Hentenryck. Fast approximations for job shop scheduling: A lagrangian dual deep learning method. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 7239–7246, 2022.

[91] J. Kotary, F. Fioretto, P. Van Hentenryck, and Z. Zhu. End-to-end learning for fair ranking systems. In *Proceedings of the ACM Web Conference 2022*, pages 3520–3530, 2022.

[92] J. Kotary, F. Fioretto, P. Van Hentenryck, and Z. Zhu. End-to-end learning for fair ranking systems. In *Proceedings of the ACM Web Conference 2022*, pages 3520–3530, 2022.

[93] J. Kotary, F. Di Vito, and F. Fioretto. Differentiable model selection for ensemble learning. In *Proceedings of the Fifteen International Joint Conference on Artificial Intelligence, IJCAI-23*, 2023.

[94] J. Kotary, M. H. Dinh, and F. Fioretto. Backpropagation of unrolled solvers with folded optimization. *International Joint Conference on Artificial Intelligence*, 2023.

[95] J. Kotary, V. Di Vito, J. Christopher, P. Van Hentenryck, and F. Fioretto. Learning joint models of prediction and optimization. In *European Conference on Artificial Intelligence*, pages 2476–2483. IOS Press, 2024.

[96] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[97] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[98] H. Liu and P. Grigas. Risk bounds and calibration for a smart predict-then-optimize method. *arXiv preprint arXiv:2108.08887*, 2021.

[99] K. Liu, M. Zhang, and Z. Pan. Facial expression recognition with cnn ensemble. In *2016 International Conference on Cyberworlds (CW)*, pages 163–166, 2016. doi:10.1109/CW.2016.34.

[100] A. Lodi and G. Zarpellon. On learning and branching: a survey. *Top*, 25(2):207–236, Jul 2017. ISSN 1863-8279. doi:10.1007/s11750-017-0451-6.

[101] A. Lodi and G. Zarpellon. On learning and branching: a survey. *Top*, 25:207–236, 2017.

[102] J. Mandi and T. Guns. Interior point solving for lp-based prediction+optimisation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

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

[104] J. Mandi, J. Kotary, S. Berden, M. Mulamba, V. Bucarey, T. Guns, and F. Fioretto. Decision-focused learning: Foundations, state of the art, benchmark and future opportunities. *Journal of Artificial Intelligence Research*, 2024.

[105] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*, pages 270–288, 2019.

[106] A. Martins and R. Astudillo. From softmax to sparsemax: A sparse model of attention and multi-label classification. In *International conference on machine learning*, pages 1614–1623. PMLR, 2016.

[107] I. D. Mienye and Y. Sun. A survey of ensemble learning: Concepts, algorithms, applications, and prospects. *IEEE Access*, 10:99129–99149, 2022.

[108] S. Misra, L. Roald, and Y. Ng. Learning for constrained optimization: Identifying optimal active constraint sets. *INFORMS Journal on Computing*, 34(1):463–480, 2022.

[109] V. Monga, Y. Li, and Y. C. Eldar. Algorithm unrolling: Interpretable, efficient deep learning for signal and image processing. *IEEE Signal Processing Magazine*, 38(2):18–44, 2021.

[110] J. R. Munkres. *Analysis on manifolds*. CRC Press, 2018.

[111] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

[112] Nasdaq. Nasdaq end of day us stock prices. https://data.nasdaq.com/databases/EOD/documentation, 2022. Accessed: 2023-08-15.

[113] E. Ndiaye, O. Fercoq, A. Gramfort, and J. Salmon. Gap safe screening rules for sparsity enforcing penalties. *The Journal of Machine Learning Research*, 18(1):4671–4703, 2017.

[114] Y. Ng, S. Misra, L. Roald, and S. Backhaus. Statistical learning for DC optimal power flow. In *Power Systems Computation Conference*, 2018.

[115] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[116] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna. Revised note on learning algorithms for quadratic assignment with graph neural networks, 2018.

[117] W. Ogryczak and T. Śliwiński. On solving linear programs with the ordered weighted averaging objective. *European Journal of Operational Research*, 148(1):80–91, 2003.

[118] W. Ogryczak, H. Luss, M. Pióro, D. Nace, and A. Tomaszewski. Fair Optimization and Networks: A Survey. *Journal of Applied Mathematics*, 2014(SI08):1 – 25, 2014. doi:10.1155/2014/612018. URL https://doi.org/10.1155/2014/612018.

[119] S. Park and P. Van Hentenryck. Self-supervised primal-dual learning for constrained optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4052–4060, 2023.

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

[121] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[122] M. B. Paulus, G. Zarpellon, A. Krause, L. Charlin, and C. Maddison. Learning to cut by looking ahead: Cutting plane selection via imitation learning. In *International conference on machine learning*, pages 17584–17600. PMLR, 2022.

[123] M. V. Pogančić, A. Paulus, V. Musil, G. Martius, and M. Rolinek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations (ICLR)*, 2020.

[124] M. J. Powell. A method for nonlinear constraints in minimization problems. *Optimization*, pages 283–298, 1969.

[125] E. Prat and S. Chatzivasileiadis. Learning active constraints to efficiently solve bilevel problems. *arXiv preprint arXiv:2010.06344*, 2020.

[126] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010.

[127] M. Ribeiro, K. Grolinger, and M. A. Capretz. Mlaas: Machine learning as a service. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 896–902, 2015. doi:10.1109/ICMLA.2015.152.

[128] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[129] M. Rubinstein. Markowitz's" portfolio selection": A fifty-year retrospective. *The Journal of finance*, 57(3):1041–1045, 2002.

[130] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[131] J. Salas and V. Yepes. Enhancing sustainability and resilience through multi-level infrastructure planning. *International Journal of Environmental Research and Public Health*, 17(3): 962, 2020. doi:10.3390/ijerph17030962.

[132] R. Sambharya, G. Hall, B. Amos, and B. Stellato. End-to-end learning to warm-start for real-time quadratic optimization. In *Learning for Dynamics and Control Conference*, pages 220–234. PMLR, 2023.

[133] R. Sheth and N. Fusi. Differentiable feature selection by discrete relaxation. In *International Conference on Artificial Intelligence and Statistics*, pages 1564–1572. PMLR, 2020.

[134] A. Singh and T. Joachims. Fairness of exposure in rankings. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2219–2228, 2018.

[135] A. Singh and T. Joachims. Policy learning for fairness in ranking. *arXiv preprint arXiv:1902.04056*, 2019.

[136] J. Song, r. lanka, Y. Yue, and B. Dilkina. A general large neighborhood search framework for solving integer linear programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 20012–20023, 2020.

[137] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[138] tamy0612. Jsplib: Benchmark instances for job-shop scheduling problem, Nov 2014. URL https://github.com/tamy0612/JSPLIB.

[139] B. Tang and E. B. Khalil. Pyepo: A pytorch-based end-to-end predict-then-optimize library with linear objective function. In *OPT 2022: Optimization for Machine Learning (NeurIPS 2022 Workshop)*, 2022.

[140] Y. Tang, S. Agrawal, and Y. Faenza. Reinforcement learning for integer programming: Learning to cut. In *International Conference on Machine Learning (ICML)*, pages 9367–9376. PMLR, 2020.

[141] T. Terlouw, T. AlSkaif, C. Bauer, and W. van Sark. Multi-objective optimization of energy arbitrage in community energy storage systems using different battery technologies. *Applied Energy*, 239:356–372, 2019. ISSN 0306-2619. doi:https://doi.org/10.1016/j.apenergy.2019.01.227. URL https://www.sciencedirect.com/science/article/pii/S0306261919302478.

[142] C. Tran, F. Fioretto, and P. V. Hentenryck. Differentially private and fair deep learning: A lagrangian dual approach. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[143] J. van den Brand. A deterministic linear program solver in current matrix multiplication time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 259–278. SIAM, 2020.

[144] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[145] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *International Conference on Learning Representations, (ICLR)*, 2018.

[146] A. Velloso and P. Van Hentenryck. Combining deep learning and optimization for security-constrained optimal power flow. *arXiv:2007:2007.07002*, 2020.

[147] N. Vesselinova, R. Steinert, D. F. Perez-Ramirez, and M. Boman. Learning combinatorial optimization on graphs: A survey with applications to networking. *IEEE Access*, 8:120388–120416, 2020.

[148] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems (NeurIPS*, pages 2692–2700, 2015.

[149] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1): 25–57, 2006.

[150] A. Wächter and C. D. Laird. *IPOPT: Interior Point Optimizer, Version 3.1.4*. IBM and Carnegie Mellon University, 2023. Available online at: https://github.com/coin-or/Ipopt.

[151] P.-W. Wang, W.-C. Chang, and J. Z. Kolter. The mixing method: low-rank coordinate descent for semidefinite programming with diagonal constraints. *arXiv:1706.00476*, 2018.

[152] P.-W. Wang, P. Donti, B. Wilder, and Z. Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning (ICML)*, pages 6545–6554. PMLR, 2019.

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

[154] B. Wilder, B. Dilkina, and M. Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *AAAI*, volume 33, pages 1658–1665, 2019.

[155] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[156] G. Wilson and G. Pawley. On the stability of the travelling salesman problem algorithm of hopfield and tank. *Biological Cybernetics*, 58(1):63–70, 1988.

[157] I. H. Witten, E. Frank, M. A. Hall, C. J. Pal, and M. DATA. Practical machine learning tools and techniques. In *Data Mining*, volume 2, 2005.

[158] D. Wu and A. Lisser. A deep learning approach for solving linear programming problems. *Neurocomputing*, 2022.

[159] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.

[160] H. Yadav, Z. Du, and T. Joachims. Policy-gradient training of fair and unbiased ranking functions. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1044–1053, 2021.

[161] R. R. Yager. On ordered weighted averaging aggregation operators in multicriteria decisionmaking. In D. Dubois, H. Prade, and R. R. Yager, editors, *Readings in Fuzzy Sets for Intelligent Systems*, pages 80–87. Morgan Kaufmann, 1993. ISBN 978-1-4832-1450-4. doi:https://doi.org/10.1016/B978-1-4832-1450-4.50011-0. URL https://www.sciencedirect.com/science/article/pii/B9781483214504500110.

[162] R. R. Yager and J. Kacprzyk. *The Ordered Weighted Averaging Operators: Theory and Applications*. Springer Publishing Company, Incorporated, 2012. ISBN 1461378060.

[163] M. Zehlike and C. Castillo. Reducing disparate exposure in ranking: A learning to rank approach. In *Proceedings of The Web Conference 2020*, pages 2849–2855, 2020.

[164] M. Zehlike, F. Bonchi, C. Castillo, S. Hajian, M. Megahed, and R. Baeza-Yates. Fa*ir: A fair top-k ranking algorithm. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1569–1578, 2017.

[165] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[166] Z. Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th international symposium on quality of service (IWQoS)*, pages 1–2. Ieee, 2018.

[167] H. Q. Zhifei Zhang, Yang Song. Age progression/regression by conditional adversarial autoencoder. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2017.

Appendix A

Appendix for Chapter 3

A.1 Lagrangian Dual-based approach

In both case studies presented below, a constrained deep learning approach is used which encourages the satisfaction of constraints within predicted solutions by accounting for the violation of constraints in a *Lagrangian* loss function

$$f_{\boldsymbol{\lambda}}(y) = f(y) + \sum_{i=1}^{m} \lambda_i \max(0, g_i(y)), \tag{A.1}$$

where f is a standard loss function (i.e., *mean squared error*), λ_i are *Lagrange multipliers* and g_i represent the constraints of the optimization problem under the generic representation

$$\mathcal{P} = \underset{y}{\operatorname{argmin}}\, h(y) \quad \text{subject to} \quad g_i(y) \leq 0 \quad (\forall i \in [m]). \tag{A.2}$$

Training a neural network to minimize the Lagrangian loss for some value of λ is anologous to computing a Lagrangian Relaxation:

$$LR_{\boldsymbol{\lambda}} = \underset{y}{\operatorname{argmin}}\, f_{\boldsymbol{\lambda}}(y), \tag{A.3}$$

and the *Lagrangian Dual* problem maximizes the relaxation over all possible λ:

$$LD = \underset{\boldsymbol{\lambda} \geq 0}{\operatorname{argmax}}\, f(LR_{\boldsymbol{\lambda}}). \tag{A.4}$$

The Lagrangian deep learning model is trained by alternately carrying out gradient descent for each value of λ, and updating the λ_i based on the resulting magnitudes of constraint violation in its predicted solutions.

A.2 Job Shop Scheduling

The Job Shop Scheduling (JSS) problem can be viewed as an integer optimization program with linear objective function and linear, disjunctive constraints. For JSS problems with J jobs and T machines, a particular instance is fully determined by the processing times d_t^j, along with machine

assignments σ_t^j, and its solution consists of the resulting optimal task start times s_j^t. The full problem specification is shown below in the system (A.5). The constraints (A.5c) enforce precedence between tasks that must be scheduled in the specified order within their respective job. Constraints (A.5d) ensure that no two tasks overlap in time when assigned to the same machine.

A.2.1 Problem specification

$$\mathcal{P}(\boldsymbol{d}) = \underset{\boldsymbol{s}}{\operatorname{argmin}} \; u \tag{A.5a}$$

$$\text{subject to: } u \geq s_T^j \qquad \forall j \in [J] \tag{A.5b}$$

$$s_{t+1}^j \geq s_t^j + d_t^j \qquad \forall j \in [J-1], \forall t \in [T] \tag{A.5c}$$

$$s_t^j \geq s_{t'}^{j'} + d_{t'}^{j'} \;\vee\; s_{t'}^{j'} \geq s_t^j + d_t^j \qquad \forall j, j' \in [J], t, t' \in [T] \text{ with } \sigma_t^j = \sigma_{t'}^{j'} \tag{A.5d}$$

$$s_t^j \in \mathbb{N} \qquad \forall j \in [J], t \in [T] \tag{A.5e}$$

Given a predicted, possibly infeasible schedule $\hat{s}$, the degree of violation in each constraint must be measured in order to update the multipliers of the Lagrangian loss function. The violation of task-precedence constraints (A.5c) and no-overlap constraint (A.5d) are calculated as in (A.6a) and (A.6b), respectively. Note that the violation of the disjunctive no-overlap condition between two tasks is measured as the amount of time at which both tasks are scheduled simultaneously on some machine.

$$\nu_{10b}\left(\hat{s}_t^j, d_t^j\right) = \max\left(0, \hat{s}_t^j + d_t^j - \hat{s}_{t+1}^j\right) \tag{A.6a}$$

$$\nu_{10c}\left(\hat{s}_t^j, d_t^j, \hat{s}_{t'}^{j'}, d_{t'}^{j'}\right) = \min\left(\nu_{10c}^L\left(\hat{s}_t^j, d_t^j, \hat{s}_{t'}^{j'}, d_{t'}^{j'}\right), \nu_{10c}^R\left(\hat{s}_t^j, d_t^j, \hat{s}_{t'}^{j'}, d_{t'}^{j'}\right)\right), \tag{A.6b}$$

where

$$\nu_{10c}^L\left(\hat{s}_t^j, d_t^j, \hat{s}_{t'}^{j'}, d_{t'}^{j'}\right) = \max\left(0, \hat{s}_t^j + d_j^t - \hat{s}_{t'}^{j'}\right)$$
$$\nu_{10c}^R\left(\hat{s}_t^j, d_t^j, \hat{s}_{t'}^{j'}, d_{t'}^{j'}\right) = \max\left(0, \hat{s}_{t'}^{j'} + d_{j'}^{t'} - \hat{s}_t^j\right).$$

The Lagrangian-based deep learning model does not necessarily produce feasible schedules directly. An additional operation is required for the construction of feasible solutions, given the direct neural network outputs representing schedules. The model presented below is used to construct solutions that are integral, and feasible to the original problem constraints. Integrality follows from the total unimodularity of constraints (A.7a, A.7b), which converts the no-overlap condition of the problem (A.5) into addition task-precedence constraints following the order of predicted start times $\hat{s}$, denoted $\preceq_{\hat{s}}$. By minimizing the makespan as in (A.5), this procedure ensures optimality of the resulting schedules subject to the imposed ordering.

$$\Pi(\boldsymbol{s}) = \underset{\boldsymbol{s}}{\operatorname{argmin}} \; u$$
$$\text{subject to: (A.5b), (A.5c)}$$

$$s_t^j \geq s_{t'}^{j'} + d_{t'}^{j'} \qquad \forall j, j' \in [J], t, t' \in [T] \text{ s.t. } (j,t) \preceq_{\hat{s}} (j',t') \tag{A.7a}$$

$$s_t^j \geq 0 \qquad \forall j \in [J], t \in [T] \tag{A.7b}$$

A.2.2 Dataset Details

The experimental setting, as defined by the training and test data, simulates a situation in which some component of a manufacturing system 'slows down', causing processing times to extend on all tasks assigned to a particular machine. Each experimental dataset is generated beginning with a root problem instance taken from the JSPLIB benchmark library for JSS instances. Further instances are generated by increasing processing times on one machine, uniformly over 5000 new instances, to a maximum of 50 percent increase over the initial values. To accommodate these incremental perturbations in problem data while keeping all values integral, a large multiplicative scaling factor is applied to all processing times of the root instance. Targets for the supervised learning are generated by solving the individual instances according to the methodology proposed in Section 3.1.5. A baseline set of solutions is generated for comparison, by solving individual instances in parallel with a time limit per instance of 1800 seconds.

The results presented in Section 3.1.6 are taken from the best-performing models, with respect to optimality of the predicted solutions following application of the model (A.7), among the results of a hyperparameter search. The model training follows the selection of parameters presented in Table A.1.

Parameter	Value	Parameter	Value
Epochs	500	**Batch Size**	16
Learning rate	$[1.25e^{-4}, 2e^{-3}]$	**Batch Normalization**	False
Dual learning rate	$[1e^{-3}, 5e^{-2}]$	**Gradient Clipping**	False
Hidden layers	2	**Activation Function**	ReLU

Table A.1: JSS: Training Parameters

A.2.3 Network Architecture

The neural network architecture used to learn solutions to the JSS problem takes into account the structure of its constraints, organizing input data by individual job, and machine of the associated tasks. When $\mathcal{I}_k^{(j)}$ and $\mathcal{I}_k^{(m)}$ represent the input array indices corresponding to job k and machine k, the associated subarrays $d[\mathcal{I}_k^{(j)}]$ and $d[\mathcal{I}_k^{(m)}]$ are each passed from the input array to a series of respective *Job* and *Machine layers*. The resulting arrays, one for every job and machine, are concatenated to form a single array and passed to further *Shared Layers*. Each shared layer has size $2JT$ in the case of J jobs and T machines, and a final layer maps the output to an array of size JM, equal to the total number of tasks. This architecture improves accuracy significantly in practice, when compared with fully connected networks of comparable size.

A.3 AC Optimal Power Flow

A.3.1 Dataset Details

Table A.2 describes the power network benchmarks used, including the number of buses $|\mathcal{N}|$, and transmission lines/transformers $|\mathcal{E}|$. Additionally it presents a comparison of the total variation

Model 2: O_{OPF}: AC Optimal Power Flow

$$\textbf{variables:} \quad S_i^g, V_i \;\; \forall i \in N, \quad S_{ij}^f \;\; \forall (i,j) \in E \cup E^R$$

$$\textbf{minimize:} \quad O(\boldsymbol{S^d}) = \sum_{i \in N} c_{2i}(\Re(S_i^g))^2 + c_{1i}\Re(S_i^g) + c_{0i} \tag{A.8}$$

$$\textbf{subject to:} \quad \angle V_i = 0, \quad i \in N \tag{A.9}$$

$$v_i^l \leq |V_i| \leq v_i^u \;\; \forall i \in N \tag{A.10}$$

$$\theta_{ij}^l \leq \angle(V_i V_j^*) \leq \theta_{ij}^u \;\; \forall (i,j) \in E \tag{A.11}$$

$$S_i^{gl} \leq S_i^g \leq S_i^{gu} \;\; \forall i \in N \tag{A.12}$$

$$|S_{ij}^f| \leq s_{ij}^{fu} \;\; \forall (i,j) \in E \cup E^R \tag{A.13}$$

$$S_i^g - S_i^d = \textstyle\sum_{(i,j) \in E \cup E^R} S_{ij}^f \;\; \forall i \in N \tag{A.14}$$

$$S_{ij}^f = Y_{ij}^* |V_i|^2 - Y_{ij}^* V_i V_j^* \;\; \forall (i,j) \in E \cup E^R \tag{A.15}$$

Instance	Size		Total Variation	
	$\|N\|$	$\|E\|$	Standard Data	OD Data
30_ieee	30	82	2.56570	**0.00118**
57_ieee	57	160	11.5160	**0.00509**
89_pegase	89	420	20.9309	**0.02538**
118_ieee	118	372	40.2253	**0.01102**
300_ieee	300	822	213.075	**0.13527**

Table A.2: Standard vs OD training data: Total Variation.

resulting from the two datasets. Note that the OD datasets have total variation which is orders of magnitude lower than their Standard counterparts.

A.3.2 Network Architecture

The neural network architecture used to learn solutions to the OPF problem is a fully connected ReLU network composed of an input layer of size proportional to the number of loads in the power network. The architecture has 5 hidden layers, each of size double the number of loads in the power network, and a final layer of size proportional to the number of generators in the network. The details of the learning models are reported in Table A.3.

A.4 Additional Results

Table A.4 compares prediction errors and constraint violations for the OD and Standard approach to data generation for the Optimal Power Flow problems. As expressed in the main paper, the results show that the models trained on the OD datset present predictions that are closer to their optimal target solutions (error expressed in MegaWatt (MW)), reduce the constraint violations (expressed as

Parameter	Value	Parameter	Value
Epochs	20000	**Batch Size**	16
Learning rate	$[1e^{-5}, 1e^{-4}]$	**Batch Normalization**	True
Dual learning rate	$1e^{-4}$	**Gradient Clipping**	True
Hidden layers	5	**Activation Function**	LeakyReLU

Table A.3: OPF: Training Parameters

Instance	Size	Prediction Error		Constraint Violation		Optimality Gap (%)	
	No. buses	Standard	OD	Standard	OD	Standard	OD
IEEE-30	30	22.31	**0.11**	0.063	**0.00004**	6.28	**0.76**
IEEE-57	57	83.61	**0.58**	0.139	**0.0002**	1.04	**0.66**
Pegase-89	89	89.17	**2.78**	1.353	**0.003**	20.1	**0.83**
IEEE-118	118	36.55	**0.54**	1.330	**0.002**	3.80	**0.36**
IEEE-300	300	157.3	**2.27**	1.891	**0.009**	22.9	**0.12**

Table A.4: OPF – Standard vs OD training data: prediction errors, constraint violations, and optimality gap.

L_1-distance between the predictions and their projections), and improve the optimality gap, which is the relative difference in objectives between the predicted (feasible) solutions and the target ones.

Appendix B

Appendix for Chapter 4

B.1 Experimental Details

Additional details for each experiment of Section 4.1.5 are described in their respective subsections below. Note that in all cases, the machine learning models compared in Section 4.1.5 use identical settings within each study, with the exception of the optimization components being compared.

B.1.1 Nonconvex Bilinear Programming

Data generation Data is generated as follows for the nonconvex bilinear programming experiments. Input data consists of 1000 points $\in \mathbb{R}^{10}$ sampled uniformly in the interval $[-2, 2]$. To produce targets, inputs are fed into a randomly initialized 2-layer neural network with tanh activation, and gone through a nonlinear function $x \cos 2x + \frac{5}{2} \log \frac{x}{x+2} + x^2 \sin 4x$ to increase the nonlinearity of the mapping between inputs and targets. Train and test sets are split 90/10.

Settings A 5-layer NN with ReLU activation trained to predict cost **c** and **d**. We train model with Adam optimizer on learning rate of 10^{-2} and batch size 32 for 5 epochs.

Nonconvex objective coefficients Q are pre-generated randomly with 15 different seeds. Constraint parameters are chosen arbitrarily as $p = 1$ and $q = 2$. The average solving time in Gurobi is 0.8333s, and depends per instance on the predicted parameters **c** and **d**. However the average time tends to be dominated by a minority of samples which take up to ~ 3 min. This issue is mitigated by imposing a time limit in solving each instance. While the correct gradient is not guaranteed under early stopping, the overwhelming majority of samples are fully optimized under the time limit, mitigating any adverse effect on training. Differences in training curves under 10s and 120s timeouts are negligible due to this effect; the results reported use the 120s timeout.

B.1.2 Enhanced Denoising

Data generation The data generation follows [6], in which 10000 random $1D$ signals of length 100 are generated and treated as targets. Noisy input data is generated by adding random perturbations to each element of each signal, drawn from independent standard-normal distributions. A 90/10 train/test split is applied to the data.

Settings A learning rate of 10^{-3} and batch size 32 are used in each training run. Each denoising model is initialized to the classical total variation denoiser by setting the learned matrix of parameters $\mathbf{D} \in \mathbb{R}^{99\times100}$ to the differencing operator, for which $D_{i,i} = 1$ and $D_{i,i+1} = -1 \;\; \forall i$ with all other values 0.

B.1.3 Multilabel Classification

Dataset We follow the experimental settings and implementation provided by [19]. Each model is evaluated on the noisy top-5 CIFAR100 task. CIFAR-100 labels are organized into 20 "coarse" classes, each consisting of 5 "fine" labels. With some probability, random noise is added to each label by resampling from the set of "fine" labels. The 50k data samples are given a 90/10 training/testing split.

Settings The DenseNet 40-40 architecture is trained by SGD optimizer with learning rate 10^{-1} and batch size 64 for 30 epochs to minimize a cross-entropy loss function.

B.1.4 Portfolio Optimization

Data Generation The data generation follows exactly the prescription of Appendix D in [52]. Uniform random feature data are mapped through a random nonlinear function to create synthetic price data for training and evaluation. A random matrix is used as a linear mapping, to which nonlinearity is introduced by exponentiation of its elements to a chosen degree. The studies in Section 4.1.5 use degrees 1, 2 and 3.

Settings A five-layer ReLU network is trained to predict asset prices $\mathbf{c} \in \mathbb{R}^{20}$ using Adam optimizer with learning rate 10^{-2} and batch size 32.

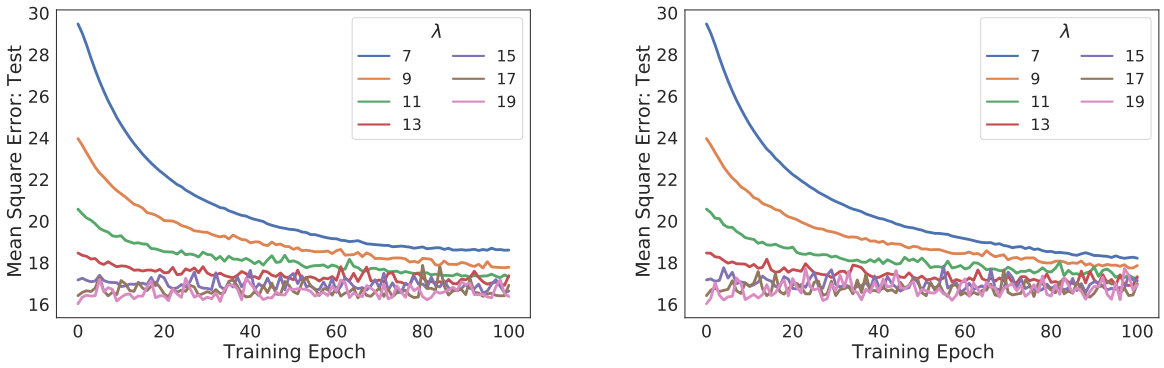
B.2 Additional Figures

B.2.1 Enhanced Denoising Experiment

Figure B.1 shows test loss curves, for a variety of λ, in learning enhanced denoisers with a baseline method which implements a denoising quadratic program in `qpth`. The results from *f-FDPG* are again shown alongside for comparison. Small differences between the results stem from the slightly different solutions found by their respective solvers at each training iteration, due to their differently-defined error tolerance thresholds.

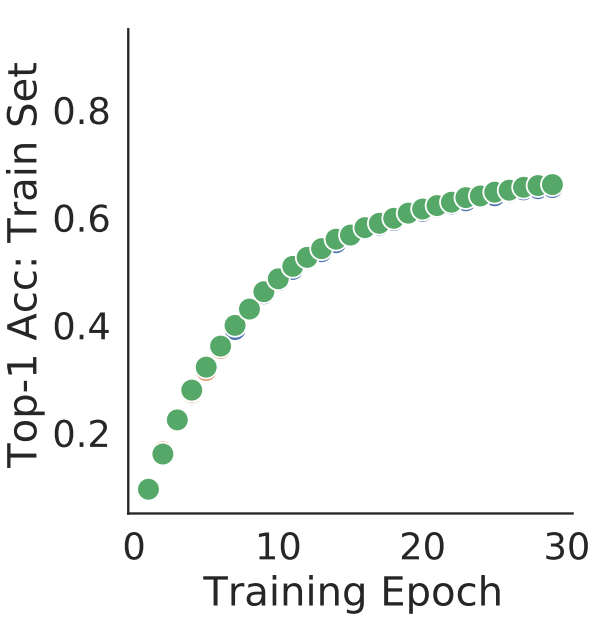
B.2.2 Multilabel Classification Experiment

Figure B.2 shows Top-1 and Top-k accuracy on both train and test sets where $k = 5$. Accuracy curves are indistinguishable on the training set even after 30 epochs. On the test set, generalization error manifests slightly differently for each model in the first few epochs.

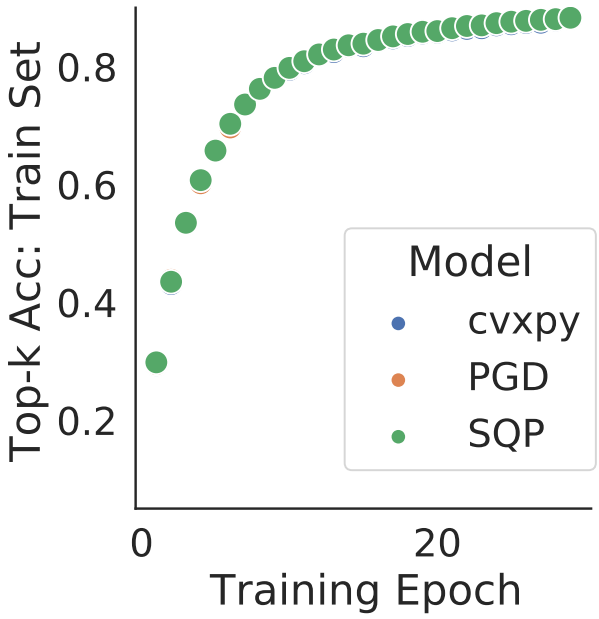


(a) *f-FDPG* (b) `qpth`

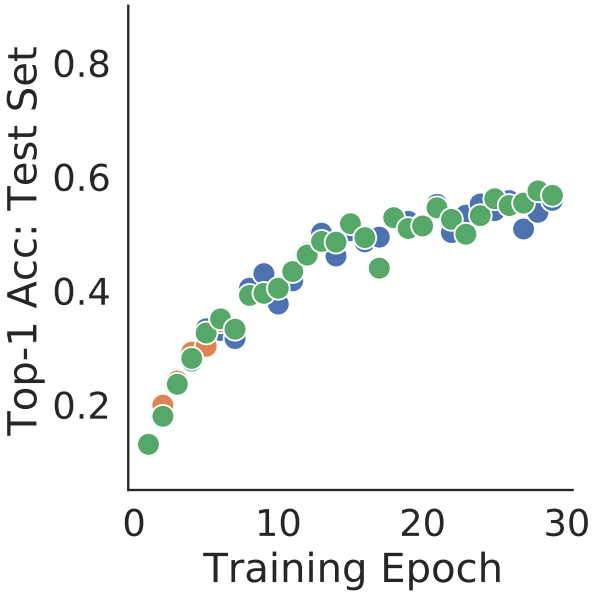
Figure B.1: Enhanced Denoiser Test Loss



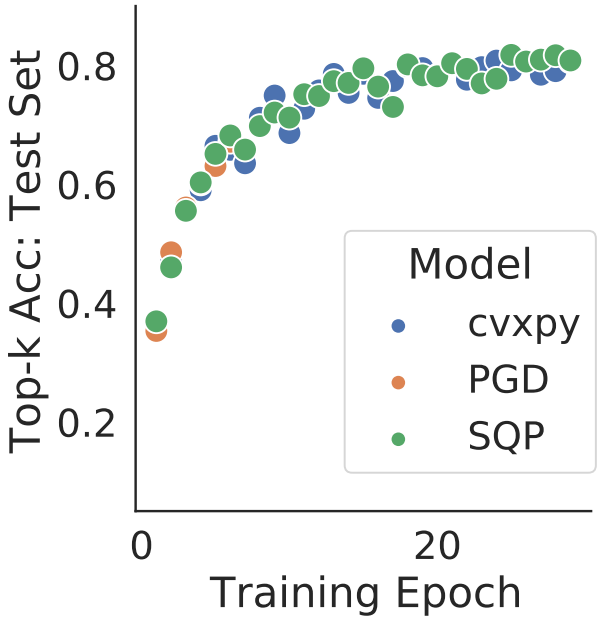
(a) Top-1 Accuracy: Train



(b) Top-k Accuracy: Train



(c) Top-1 Accuracy: Test



(d) Top-k Accuracy: Train

Figure B.2: Multilabel Classification Accuracy

Appendix C

Appendix for Chapter 5

C.1 Portfolio Optimization Experiment

The classic Markowitz portfolio problem is concerned with constructing an optimal investment portfolio, given future returns $\boldsymbol{c} \in \mathbb{R}^n$ on n assets, which are unknown and predicted from exogenous data. A common formulation maximizes future returns subject to a risk limit, modeled as a quadratic covariance constraint. Define the set of valid fractional allocations $\Delta_n = \{\boldsymbol{x} \in \mathbb{R}^n : \mathbf{1}^T\boldsymbol{x} = 1, \boldsymbol{x} \geq 0\}$, then :

$$\boldsymbol{x}^\star(\boldsymbol{c}) = \underset{\boldsymbol{x}\in\Delta_n}{\operatorname{argmax}} \quad \boldsymbol{c}^T\boldsymbol{x} \quad s.t.: \quad \boldsymbol{x}^T\boldsymbol{\Sigma}\boldsymbol{x} \leq \delta. \tag{C.1}$$

where $\boldsymbol{\Sigma} \in \mathbb{R}^{n\times n}$ are the price covariances over n assets. The optimal portfolio allocation (C.1) as a function of future returns $\boldsymbol{c} \in \mathbb{R}^n$ is differentiable using known methods [4], and is commonly used in evaluation of Predict-Then-Optimize methods [104].

Settings. Historical prices of $n = 50$ assets are obtained from the Nasdaq online database [112] years 2015-2019, and $N = 5000$ baseline asset price samples $\boldsymbol{c}_i$ are generated by adding Gaussian random noise to randomly drawn price vectors. Price scenarios are simulated as a matrix of multiplicative factors uniformly drawn as $\mathcal{U}(0.5, 1.5)^{m\times n}$, whose rows are multiplied elementwise with $\boldsymbol{c}_i$ to obtain $\boldsymbol{C}_i \in \mathbb{R}^{m\times n}$. While future asset prices can be predicted on the basis of various exogenous data including past prices or sentiment analysis, this experiment generates feature vectors $\boldsymbol{z}_i$ using a randomly generated nonlinear feature mapping. The experiment is replicated in three settings which assume $m = 3$, 5, and 7 scenarios.

Two sets of stocks were selected to generate two different datasets based on their average returns across observations. The first set consists of assets from the index with average returns within the 25th to 50th quantile range, while the second set includes assets from the 75th quantile.

The predictive model $\mathcal{M}_\theta$ is a feedforward neural network with three shared hidden layers followed by one separated hidden layer for each species that is trained using Adam Optimizer and with a batch size of 64. The size of each shared layer is halved, and the output dimension of the separated layer is equal to the number of assets. Hyperparameters were selected as the best-performing on average among those listed in Table C.1). Results for each hyperparameter setting are averaged over five random seeds. In the OWA-Moreau model, the forward pass is executed using projected gradient descent for 300, 500, and 750 iterations, respectively, for scenarios with 3, 5, and 7 inputs. The update step size is set to $\gamma = 0.02$.

Table C.1: Hyperparameters

Hyperparameter	Min	Max	Final Value					
			OWA-LP	Two-Stage	Sum-QP	OWA-QP	OWA-Moreau	Sur-C
learning rate	$1e^{-3}$	$1e^{-1}$	$\mathbf{1e^{-2}}$	$\mathbf{5e^{-3}}$	$\mathbf{1e^{-2}}$	$\mathbf{1e^{-2}}$	$\mathbf{1e^{-2}}$	$\mathbf{1e^{-2}}$
smoothing parameter ϵ	0.1	1.0	**N/A**	**N/A**	1.0	1.0	**N/A**	1.0
smoothing parameter β_0	0.005	10.0	**N/A**	**N/A**	**N/A**	**N/A**	0.05	**N/A**
MSE loss weight λ	0.1	0.5	0.4	**N/A**	0.3	0.4	0.1	0.3

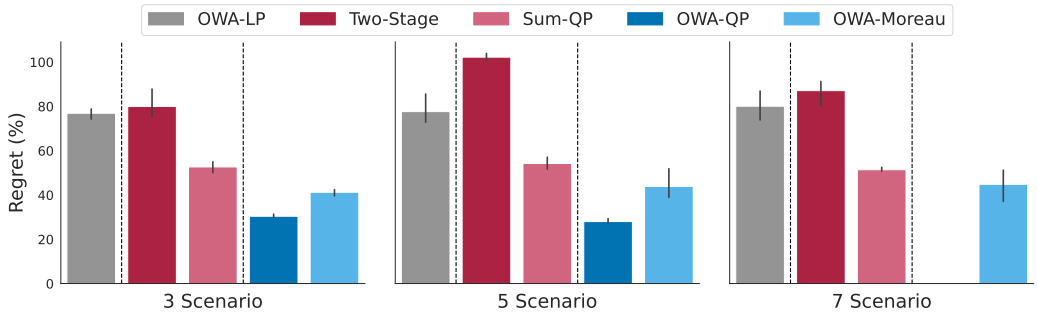


Figure C.1: Percentage OWA regret (lower is better) on test set, on robust portfolio problem over 3,5,7 scenarios.

At test time, $\mathcal{M}_\theta$ is evaluated over a test set for the distribution $(\boldsymbol{z}, \boldsymbol{C}) \in \Omega$, by passing its predictions to a projected subgradient solver of (5.16).

C.1.1 Additional Results

Figure C.1 and 5.2 display models' performance on datasets generated from assets with average returns in the 75th quantile and within the 25th-50th percentiles, respectively. The y-axis represents the percentage of regret based on optimal OWA values. A consistent trend is observed in both datasets: end-to-end approaches tend to outperform two-stage approaches. Additionally, our proposed frameworks (*OWA-QP* and *OWA-Moreau*) perform better than *Sum-QP*, with improvements ranging from 5-30%. *OWA-QP* performs better when the number of scenarios is small but struggles to scale beyond 6 scenarios.

C.1.2 Effect of adding MSE loss

Figure C.2 illustrates the impact of combining the Mean Squared Error loss $\mathcal{L}_{MSE}$ in a weighted combination with the decision quality loss $\mathcal{L}_{DQ}$. Except OWA-LP, which exhibited instability, and Two-Stage, already trained with MSE Loss, the addition of MSE resulted in slight enhancements to the regret performance.

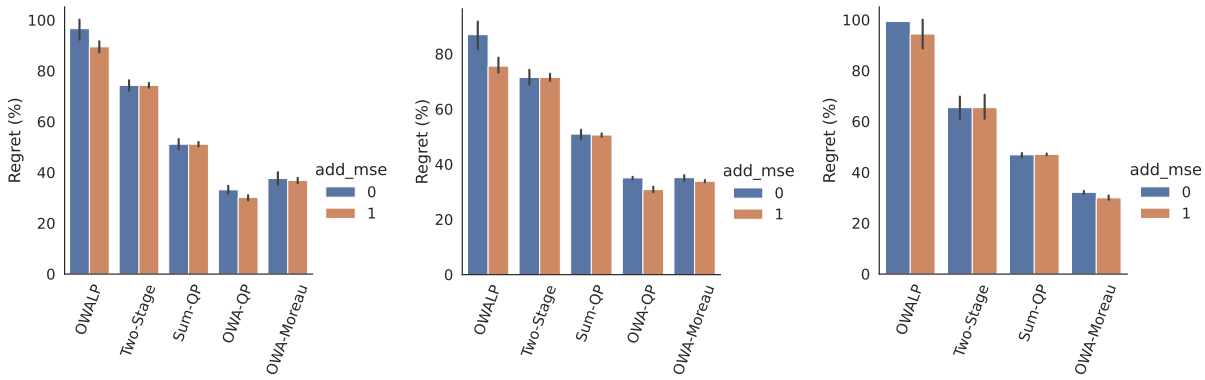


Figure C.2: Effect of MSE Loss on differentiable optimization models. From left to right: 3, 5, 7 scenarios

C.1.3 Solution Methods

The OWA portfolio optimization problem (5.16) is solved at test time, for each compared method, by projected subgradient descent using OWA subgradients (5.7) and an efficient projection onto the unit simplex Δ as in [106]:

$$\boldsymbol{x}^{k+1} = proj_{\Delta}\left(\boldsymbol{x}^{k} - \alpha \frac{\partial}{\partial \boldsymbol{x}} \mathrm{OWA}_{\boldsymbol{w}}(\boldsymbol{C}\boldsymbol{x})\right) \tag{C.2}$$

For the Moreau-envelope smoothed OWA optimization (5.11) proposed for end-to-end training, the main difference is that its objective function is differentiable (with gradients (5.12)), which allows solution by a more efficient Frank-Wolfe method [15], whose inner optimization over Δ reduces to the simple argmax function which returns a binary vector with unit value in the highest vector position and 0 elsewhere, which can be computed in linear time:

$$\boldsymbol{x}^{k+1} = \frac{k}{k+2}\boldsymbol{x}^{k} + \frac{2}{k+2}\,\mathrm{argmax}\left(\frac{\partial}{\partial \boldsymbol{x}} \mathrm{OWA}_{\boldsymbol{w}}(\boldsymbol{C}\boldsymbol{x}^{k})\right) \tag{C.3}$$

Appendix D

Appendix for Chapter 6

D.1 Optimization Problems

Illustrative $2D$ **example** Used for illustration purposes, the $2D$ optimization problem used to produce the results of Figure 6.3 takes the form

$$\begin{aligned} \mathbf{x}^\star(\boldsymbol{\zeta}) = \underset{\boldsymbol{x}}{\text{argmin}} \;\; & \zeta_1 x_1^2 + \zeta_2 x_2^2 \\ \text{s.t.} \;\; & x_1 + 2x_2 \leq 0.5, \\ & 2x_1 - x_2 \leq 0.2, \\ & x_1 + x_2 \leq 0.3 \end{aligned}$$

and its optimization proxy model is learned using *PDL* training.

AC-Optimal Power Flow Problem. The OPF determines the least-cost generator dispatch that meets the load (demand) in a power network. The OPF is defined in terms of complex numbers, i.e., *powers* of the form $S = (p + jq)$, where p and q denote active and reactive powers and j the imaginary unit, *admittances* of the form $Y = (g + jb)$, where g and b denote the conductance and susceptance, and *voltages* of the form $V = (v\angle\theta)$, with magnitude v and phase angle θ. A power network is viewed as a graph $(\mathcal{N}, \mathcal{E})$ where the nodes $\mathcal{N}$ represent the set of *buses* and the edges $\mathcal{E}$ represent the set of *transmission lines*. The OPF constraints include physical and engineering constraints, which are captured in the AC-OPF formulation of Figure D.1. The model uses p^g, and p^d to denote, respectively, the vectors of active power generation and load associated with each bus and p^f to describe the vector of active power flows associated with each transmission line. Similar notations are used to denote the vectors of reactive power q. Finally, the model uses v and θ to describe the vectors of voltage magnitude and angles associated with each bus. The OPF takes as inputs the loads $(\boldsymbol{p}^d, \boldsymbol{q}^d)$ and the admittance matrix $\boldsymbol{Y}$, with entries $\boldsymbol{g}_{ij}$ and $\boldsymbol{b}_{ij}$ for each line $(i\,j) \in \mathcal{E}$; It returns the active power vector p^g of the generators, as well the voltage magnitude v at the generator buses. The problem objective (2a) captures the cost of the generator dispatch and is typically expressed as a quadratic function. Constraints (2b) and (2$\bar{c}$) restrict the voltage magnitudes and the phase angle differences within their bounds. Constraints (2$\bar{d}$) and (2$\bar{e}$) enforce the generator active and reactive output limits. Constraints (2$\bar{f}$) enforce the line flow limits. Constraints (2$\bar{g}$) and (2$\bar{h}$) capture *Ohm's Law*. Finally, Constraint (2$\bar{i}$) and (2j) capture *Kirchhoff's Current Law* enforcing flow conservation at each bus.

$$
\begin{aligned}
\text{minimize}: \quad & \sum_{i \in \mathcal{N}} \text{cost}(p_i^g, \zeta_i) && \text{(2a)}\\
\text{s.t.} \quad & \boldsymbol{v}_i^{\min} \leq v_i \leq \boldsymbol{v}_i^{\max} \quad \forall i \in \mathcal{N} && \text{(2b)}\\
& -\boldsymbol{\theta}_{ij}^{\Delta} \leq \theta_i - \theta_j \leq \boldsymbol{\theta}_{ij}^{\Delta} \quad \forall (ij) \in \mathcal{E} && \text{(2}\bar{c}\text{)}\\
& \boldsymbol{p}_i^{g\,\min} \leq p_i^g \leq \boldsymbol{p}_i^{g\,\max} \quad \forall i \in \mathcal{N} && \text{(2}\bar{d}\text{)}\\
& \boldsymbol{q}_i^{g\,\min} \leq q_i^g \leq \boldsymbol{q}_i^{g\,\max} \quad \forall i \in \mathcal{N} && \text{(2}\bar{e}\text{)}\\
& (p_{ij}^f)^2 + (q_{ij}^f)^2 \leq \boldsymbol{S}_{ij}^{f\,\max} \quad \forall (ij) \in \mathcal{E} && \text{(2}\bar{f}\text{)}\\
& p_{ij}^f = \boldsymbol{g}_{ij} v_i^2 - v_i v_j (\boldsymbol{b}_{ij} \sin(\theta_i - \theta_j) + \\
& \qquad \boldsymbol{g}_{ij} \cos(\theta_i - \theta_j)) \quad \forall (ij) \in \mathcal{E} && \text{(2}\bar{g}\text{)}\\
& q_{ij}^f = -\boldsymbol{b}_{ij} v_i^2 - v_i v_j (\boldsymbol{g}_{ij} \sin(\theta_i - \theta_j) - \\
& \qquad \boldsymbol{b}_{ij} \cos(\theta_i - \theta_j)) \quad \forall (ij) \in \mathcal{E} && \text{(2}\bar{h}\text{)}\\
& p_i^g - \boldsymbol{p}_i^d = \textstyle\sum_{(ij) \in \mathcal{E}} p_{ij}^f \quad \forall i \in \mathcal{N} && \text{(2}\bar{i}\text{)}\\
& q_i^g - \boldsymbol{q}_i^d = \textstyle\sum_{(ij) \in \mathcal{E}} q_{ij}^f \quad \forall i \in \mathcal{N} && \text{(2j)}\\
\text{Output}: \quad & (p^g, v) \text{ – The system operational parameters}
\end{aligned}
$$

Figure D.1: AC Optimal Power Flow (AC-OPF).

$$
\Delta \mathbf{x}_n = -\mathbf{J}^{-1}(\mathbf{x}_n)\mathbf{f}(\mathbf{x}_n) \qquad \text{(D.2)}
$$

$$
\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta \mathbf{x}_n \qquad \text{(D.3)}
$$

Figure D.2: Newton's method.

Feasibility restoration (AC-Optimal Power Flow) Being an approximation, a LtO solution $(\hat{\boldsymbol{p}}^g, \hat{v})$ may not satisfy the original constraints. Feasibility can be restored by applying Netwon's method, which is reported in Figure D.2. It is an iterative method that produces better approximation to the root $\mathbf{x} \in \mathbb{R}^p$, of a function $f(x) \in \mathbb{R}^m$ by iteratively solving a non-linear system of equations. If solving for $\mathbf{x}_{n+1}$, given $\mathbf{x}_n$, the method requires to compute the inverse of the Jacobian $J(\mathbf{x}_n) \in \mathbb{R}^{m \times p}$. From Eq. D.2 and D.3, it can be noticed that $J(\mathbf{x}_n)\Delta\mathbf{x}_n = -f(\mathbf{x}_n)$, and so is possible to avoid computing the inverse of the Jacobian J of f, and solving a linear system of equation for the unknown $\Delta\mathbf{x}_n$. In the context of restoring feasibility of the LtO solution to the AC-Optimal Power Flow problem, f represents the set of inequality and equality constraint functions, from (2b) to (2$\bar{h}$), while $x = [v, \theta, p^g, q^g]^T$. Since the method requires each $f_i(x), i = 1, \ldots m$ to be an equality function, to construct a system of only equations, a $\mathbf{ReLU}(f(x)) = \max(0, f(x))$ is applied to each inequality function. For the AC-OPF experiment, the number of constraint function $m = 602$ while the number of variables $p = 472$; being $m > p$, the inverse J^{-1} of the Jacobian J is the generalized

inverse $J^+ = (J^T J)^{-1} J^T$, and $\Delta \mathbf{x}_n$ is the solution in the least square sense. The convergence of the method requires the starting point x_0 to be such that the 2-norm $\|f(x_0)\|_2 \ll 1$. In the experiments, we verified that such assumption holds as evidenced by the minimal Constraint Violation achieved by each LTO method adopted . We consider the method to have converged when the absolute value of each constraint function $|f_i(x_n)| < 1\text{e}{-6}$.

D.2 Experimental Details

D.2.1 Portfolio Optimization Dataset

The stock return dataset is prepared exactly as prescribed in [132]. The return parameters and asset prices are $\zeta = \alpha(\hat{\zeta}_t + \epsilon_t)$ where $\hat{\zeta}$ is the realized return at time t, ϵ_t is a normal random variable, $\epsilon_t \sim \mathcal{N}(0, \sigma_\epsilon I)$, and $\alpha = 0.24$ is selected to minimize $\mathbb{E}\|\hat{\zeta}_t - \zeta\|_2^2$. For each problem instance, the asset prices ζ are sampled by circularly iterating over the five year interval. In the experiments, see Prob. 6.11, $\lambda = 2.0$.

The covariance matrix $\mathbf{\Sigma}$ is constructed from historical price data and set as $\mathbf{\Sigma} = F\mathbf{\Sigma}_F F^T + D$, where $F \in \mathbb{R}^{n,l}$ is the factor-loading matrix, $\mathbf{\Sigma} \in \mathbb{S}_+^l$ estimates the factor returns and $D \in \mathbb{S}_+^l$, also called the idiosyncratic risk, is a diagonal matrix which takes into account for additional variance for each asset.

D.2.2 Nonconvex Optimization Dataset

The nonconvex optimization dataset has 2400 samples, divided into training, validation and test set, each consisting of 2000, 200 and 200 samples, respectively. The matrix $Q = \mu I$, where $\mu \in \mathbb{R}^n \sim \mathcal{U}(0, 1)$. The parameter $\zeta \sim \mathcal{U}(0, 5)$ and the matrix A and G are both drawn from the normal distribution $\mathcal{N}(0, 1)$. The right-hand side of the equality constraint $b \sim \mathcal{U}(-1, 1)$, while the right-hand side of the inequality constraint $h = \sum_{i=1}^n |M_{ij}|$, where $M = GA^+$ and $A^+ = (A^T A)^{-1} A^T$.

D.2.3 Nonconvex AC-OPF Dataset

The nonconvex optimization dataset has 10000 samples, divided into training, validation and test set, each consisting of 8334, 833 and 833 samples, respectively. The Nonconvex AC-OPF Dataset is constructed by applying random perturbations of the cost values found in NESTA benchmark case 118. More specifically, a perturbation $\mu \in \mathcal{U}(0, 100)$ is applied to each generator cost value ζ_i.

D.2.4 Nonconvex EPO Baselines

The nonconvex QP variant (6.12) of Section 4.1.5 admits derivatives for EPO training by differentiation of the fixed-point conditions of a locally convergent solution method. Projected Gradient Descent is known to be locally convergent in nonconvex optimization [10], and it is found empirically to converge locally on the problem (6.12).

On a problem of form

$$\boldsymbol{x}^\star(\boldsymbol{\zeta}) = \underset{\boldsymbol{x}}{\text{argmin}} \quad f(\boldsymbol{x}, \boldsymbol{\zeta})) \\ \text{s.t.} \quad \boldsymbol{x} \in \mathcal{S}$$

one step of the method follows

$$\boldsymbol{x}_{k+1} = proj_{\mathcal{S}}(\boldsymbol{x}_k - \alpha \nabla f(\boldsymbol{x}_k, \boldsymbol{\zeta})) \tag{D.5}$$

leading to the fixed-point conditions

$$\boldsymbol{x}^\star = proj_{\mathcal{S}}(\boldsymbol{x}^\star - \alpha \nabla f(\boldsymbol{x}^\star, \boldsymbol{\zeta})) \tag{D.6}$$

whose implicit differentiation results in a linear system which can be solved for $\frac{\partial \boldsymbol{x}^\star}{\partial \boldsymbol{\zeta}}$:

$$\frac{\partial \boldsymbol{x}^\star}{\partial \boldsymbol{\zeta}} = \frac{\partial}{\partial \boldsymbol{x}^\star} proj_{\mathcal{S}}(\boldsymbol{x}^\star - \alpha \nabla f(\boldsymbol{x}^\star, \boldsymbol{\zeta})) \cdot \frac{\partial \boldsymbol{x}^\star}{\partial \boldsymbol{\zeta}} \tag{D.7a}$$

$$+ \frac{\partial}{\partial \boldsymbol{\zeta}} proj_{\mathcal{S}}(\boldsymbol{x}^\star - \alpha \nabla f(\boldsymbol{x}^\star, \boldsymbol{\zeta})) \tag{D.7b}$$

Differentiation of the inner projection step is performed by `cvxpy` [4], while the system (D.7) is constructed and solved by `fold-opt` [94].

D.2.5 Hyperparameters

For all the experiments, the size of the mini-batch $\mathcal{B}$ of the training set is equal to 200. The optimizer used for the training of the optimization proxy's is Adam, and the learning rate is chosen as the best among $\{5e-2, 1e-2, 5e-3, 1e-3, 5e-4, 1e-4\}$. For each task, an early stopping criteria based on the evaluation of the test-set percentage regret after restoring feasibility, is adopted to all the LtO(F) the proxies, the predictive EPO (w/o) proxy model, and pre-trained predictive model; an early stopping criteria based on the evaluation of the mean squared error is adopted to all the Two-Stage predictive model.

For each optimization problem, the LtO proxies are 2-layers ReLU neural networks with dropout equal to 0.1 and batch normalization. All the LtOF proxies are $(k+1)$-layers ReLU neural networks with dropout equal to 0.1 and batch normalization, where k denotes the complexity of the feature mapping. For the LtOF, Two-Stage, EPO (w/o) Proxy algorithm, the feature size of the Convex Quadratic Optimization and Non Convex AC Optimal Power Flow $|z| = 30$, while for the Non Convex Quadratic Optimization $|z| = 50$. The hidden layer size of the feature generator model is equal to 50, and the hidden layer size of the LtO(F) proxies, and the 2Stage, EPO and EPO w/ proxy's predictive model is equal to 500.

A grid search method is adopted to tune the hyperparameters of each LtO(F) models. For each experiments, and for each LtO(F) methods, below is reported the list of the candidate hyperparameters for each k, with the chosen ones marked in bold. We refer to [54], [119] and [48] for a comprehensive description of the parameters of the LtO methods adopted in the proposed framework. In our result, two-stage methods report the *lowest regret* found in each experiment and each k across all hyperparameters adopted.

Convex Quadratic Optimization and Non Convex Quadratic Optimization

LD

Parameter	Values
$\lambda(0)$	$\mathbf{0.1}$, 0.5, 1.0, 5.0, 10.0, 50.0
$\mu(0)$	0.1, $\mathbf{0.5}$, 1.0, 5.0, 10.0, 50.0
Training epochs	50, 100, $\mathbf{200}$, 300, 500
LD step size	1.0, 0.1, 0.01, $\mathbf{0.001}$, 0.0001

PDL

Parameter	Values
τ	0.5, 0.6, 0.7, $\mathbf{0.8}$, 0.9
ρ	0.1, $\mathbf{0.5}$, 1, 10
$\rho_{\max}$	1000, $\mathbf{5000}$, 10000
α	1, 1.5, 2.5, $\mathbf{5}$, 10

DC3

Parameter	Values
$\lambda + \mu$	0.1, 1.0, $\mathbf{10.0}$, 50.0, 100.0
$\frac{\lambda}{\lambda+\mu}$	0.1, 0.5, $\mathbf{0.75}$, 1
t_{test}	1, 2, $\mathbf{5}$, 10, 100
t_{train}	1, 2, $\mathbf{5}$, 50, 100

Non Convex AC-Optimal Power Flow

LD

Parameter	Values
$\lambda(0)$	$\mathbf{0.1}$, 0.5, 1.0, 5.0, 10.0, 50.0
$\mu(0)$	$\mathbf{0.1}$, 0.5, 1.0, 5.0, 10.0, 50.0
Training epochs	50, 100, 200, $\mathbf{300}$, 500
LD step size	$\mathbf{1.0}$, 0.1, 0.01, 0.001, 0.0001

PDL

Parameter	Values
τ	0.5, 0.6, 0.7, **0.8**, 0.9
ρ	0.1, 0.5, **1**, 10
$\rho_{\max}$	1000, 5000, **10000**
α	1, 1.5, 2.5, 5, **10**