

# Backpropagation of Unrolled Solvers with Folded Optimization

James Kotary<sup>1</sup>, My H Dinh<sup>1</sup> and Ferdinando Fioretto<sup>1</sup>

<sup>1</sup> University of Virginia

{jkotary, mydinh}@syr.edu, fioretto@virginia.edu

## Abstract

The integration of constrained optimization models as components in deep networks has led to promising advances on many specialized learning tasks. A central challenge in this setting is backpropagation through the solution of an optimization problem, which typically lacks a closed form. One typical strategy is algorithm unrolling, which relies on automatic differentiation through the operations of an iterative solver. While flexible and general, unrolling can encounter accuracy and efficiency issues in practice. These issues can be avoided by analytical differentiation of the optimization, but current frameworks impose rigid requirements on the optimization problem’s form. This paper provides theoretical insights into the backward pass of unrolled optimization, leading to a system for generating efficiently solvable analytical models of backpropagation. Additionally, it proposes a unifying view of unrolling and analytical differentiation through optimization mappings. Experiments over various model-based learning tasks demonstrate the advantages of the approach both computationally and in terms of enhanced expressiveness.

## 1 Introduction

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, which is treated as a layer of a neural network. Outputs of the layer are then guaranteed to obey the problem’s constraints, which may be predefined or learned [Kotary *et al.*, 2021].

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 [Martins and Astudillo, 2016], learning to rank based on optimal matching [Adams and Zemel, 2011; ?], accurate model selection protocols [Kotary *et al.*, 2023a], and enhanced models for optimal decision-making under uncertainty [Wilder *et al.*, 2019].

While constrained optimization mappings can be used as components in neural networks in a similar manner to linear layers or activation functions [Amos and Kolter, 2017], a prerequisite is their differentiation, for 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 [Monga *et al.*, 2021]. *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 [Agrawal *et al.*, 2019]. 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.<sup>1</sup>

**Contributions.** To address these limitations, this paper 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, the paper makes 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 `fold-opt`, a Python library

<sup>1</sup>A discussion of related work on differentiable optimization and decision-focused learning is provided in Appendix A.

to facilitate automation. (3) Its efficiency and modeling advantages are demonstrated on a set of end-to-end optimization and learning tasks, including the first demonstration of decision-focused learning with *nonconvex* decision models.

**Supplemental material.** Please refer to [Kotary *et al.*, 2023b] for an extended version of this paper including Appendix additional related work, implementation details, analysis, and experiments. All references to the Appendix in this paper refer to that of the extended version.

## 2 Setting and Goals

In this paper, the goal is to differentiate mappings that are defined as the solution to an optimization problem. Consider the parameterized problem (1) which defines a function from a vector of parameters  $\mathbf{c} \in \mathbb{R}^p$  to its associated optimal solution  $\mathbf{x}^*(\mathbf{c}) \in \mathbb{R}^n$ :

$$\mathbf{x}^*(\mathbf{c}) = \underset{\mathbf{x}}{\operatorname{argmin}} f(\mathbf{x}, \mathbf{c}) \quad (1a)$$

$$\text{subject to: } g(\mathbf{x}, \mathbf{c}) \leq \mathbf{0}, \quad (1b)$$

$$h(\mathbf{x}, \mathbf{c}) = \mathbf{0}, \quad (1c)$$

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  $\mathbf{x}^*(\mathbf{c})$  can be found by conventional 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  $\frac{\partial \mathbf{x}^*(\mathbf{c})}{\partial \mathbf{c}}$ .

**Backpropagation.** Given a downstream task loss  $\mathcal{L}$ , backpropagation through  $\mathbf{x}^*(\mathbf{c})$  amounts to computing  $\frac{\partial \mathcal{L}}{\partial \mathbf{c}}$  given  $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^*}$ . In deep learning, backpropagation through a layer is typically accomplished 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  $\mathbf{x}^*(\mathbf{c}) : \mathbb{R}^p \rightarrow \mathbb{R}^n$  is a matrix  $\frac{\partial \mathbf{x}^*}{\partial \mathbf{c}}$  in  $\mathbb{R}^{n \times p}$ , whose elements at  $(i, j)$  are the partial derivatives  $\frac{\partial x_i^*(\mathbf{c})}{\partial c_j}$ . When the Jacobian is known, backpropagation through  $\mathbf{x}^*(\mathbf{c})$  can be performed by computing the product

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^*} \cdot \frac{\partial \mathbf{x}^*(\mathbf{c})}{\partial \mathbf{c}}. \quad (2)$$

**Folded Optimization: Overview.** The problem (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

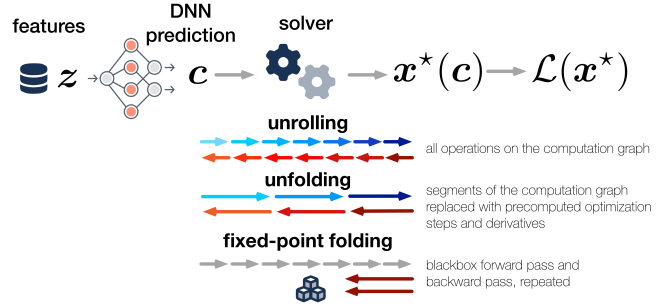


Figure 1: Compared to unrolling, unfolding requires fewer operations on the computational graph by replacing inner loops with Jacobian-gradient products. Fixed-point folding models the unfolding analytically, allowing for blackbox implementations.

$\mathbf{x} \in \mathbb{R}^n$ , the *update function* is a vector-valued function  $\mathcal{U} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ :

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

The iterations (U) *converge* if  $\mathbf{x}_k(\mathbf{c}) \rightarrow \mathbf{x}^*(\mathbf{c})$  as  $k \rightarrow \infty$ . When *unrolling*, the iterations (U) are computed and recorded on the computational graph, and the function  $\mathbf{x}^*(\mathbf{c})$  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 [Monga *et al.*, 2021]. In the following sections, we show how the backward pass of unrolling can be analyzed to yield equivalent analytical models for the Jacobian of  $\mathbf{x}^*(\mathbf{c})$ . We recognize 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 3 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 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 entangled across unrolled iterations, greatly simplifying the backward pass model and allowing for blackbox implementations of both passes.

Section 5 uses these concepts to show that the backward pass of unfolding (U) follows exactly the solution of the linear system for  $\frac{\partial \mathbf{x}^*(\mathbf{c})}{\partial \mathbf{c}}$  which arises by differentiating the fixed-point conditions of (U). Section 6 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 5. The main differences between unrolling, unfolding, and fixed-point folding are illustrated in Figure 1.

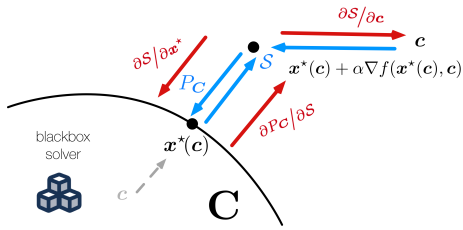


Figure 2: Unfolding Projected Gradient Descent at  $\mathbf{x}^*$  consists of alternating gradient step  $\mathcal{S}$  with projection  $\mathcal{P}_C$ . Each function’s forward and backward pass are in blue and red, respectively.

### 3 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,

$$\mathcal{U}(\mathbf{x}_k) := \mathcal{T}(\mathcal{O}(\mathcal{S}(\mathbf{x}_k)), \mathbf{x}_k), \quad (\text{O})$$

wherein the differentiation of  $\mathcal{U}$  is complicated by an *inner optimization* sub-routine  $\mathcal{O} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ . Here,  $\mathcal{S}$  and  $\mathcal{T}$  represent any steps preceding or following the inner optimization (such as gradient steps), viewed as closed-form 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 in this framework is handled naturally 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  $\mathbf{x}^*(c)$ .

**Definition 1** (Unfolding). *An unfolded 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 is distinguished from more general unrolling by the presence of only a single unrolled loop. This definition sets the stage for Section 5, which shows how 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, the definitions unrolling and unfolding coincide. When  $\mathcal{O}$  is nontrivial to solve but has known Jacobians, they can be used to produce an unfolding of (U). Such is the case when  $\mathcal{O}$  is a Quadratic Program (QP); a JgP-based differentiable QP solver called `qpth` is provided by [Amos and Kolter, 2017]. Alternatively, the replacement of unrolled loops by JgP’s proposed in Section 5 can be applied recursively  $\mathcal{O}$ .

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 folded optimization mappings for a variety of learning tasks in Section 6.

**Projected gradient descent.** Given a problem

$$\min_{\mathbf{x} \in \mathbf{C}} f(\mathbf{x}) \quad (3)$$

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

$$\mathbf{x}_{k+1} = \mathcal{P}_C(\mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)), \quad (4)$$

where  $\mathcal{O} = \mathcal{P}_C$  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) (see [Beck, 2017]). Further, when  $\mathbf{C}$  is linear,  $\mathcal{P}_C$  is a quadratic programming (QP) problem for which a differentiable solver `qpth` is available from [Amos and Kolter, 2017].

Figure 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

$$\min_{\mathbf{x}} f(\mathbf{x}) + g(\mathbf{x}) \quad (5)$$

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

$$\mathbf{x}_{k+1} = \text{Prox}_{\alpha_k g}(\mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)). \quad (6)$$

Here  $\mathcal{O}$  is the proximal operator, defined as

$$\text{Prox}_g(\mathbf{x}) = \underset{\mathbf{y}}{\text{argmin}} \left\{ g(\mathbf{y}) + \frac{1}{2} \|\mathbf{y} - \mathbf{x}\|^2 \right\}, \quad (7)$$

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, whose closed-form formula and derivative are given in Appendix C.

**Sequential quadratic programming.** Sequential Quadratic Programming (SQP) solves the general optimization problem (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 implementing its QP step (shown in Appendix C with the `qpth` differentiable QP solver).

**Quadratic programming by ADMM.** The QP solver of [Boyd *et al.*, 2011], based on the alternating direction of multipliers, is specified in Appendix C. Its inner optimization step  $\mathcal{O}$  is a simpler 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 5. The resulting differentiable QP solver can then

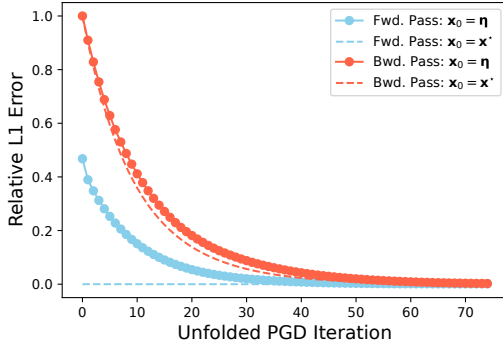


Figure 3: Forward and backward pass error in unfolding PGD

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 6.

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 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  $\mathbf{x}_k \rightarrow \mathbf{x}^*$  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  $\lim_{k \rightarrow \infty} \mathbf{x}_k = \mathbf{x}^*$  in its forward pass. Then convergence of the backward pass is*

$$\lim_{k \rightarrow \infty} \frac{\partial \mathbf{x}_k}{\partial \mathbf{c}}(\mathbf{c}) = \frac{\partial \mathbf{x}^*}{\partial \mathbf{c}}(\mathbf{c}). \quad (8)$$

**Effect of the starting point on backpropagation.** Consider the optimization mapping (19) which maps feature embeddings to smooth top- $k$  class predictions, and will be used to learn multilabel classification later in Section 6. 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 pre-computed optimal solution  $\mathbf{x}_0^a = \mathbf{x}^*$ , and a uniform random vector  $\mathbf{x}_0^b = \boldsymbol{\eta} \sim \mathbf{U}(0, 1)$ . The former case is illustrated in Figure 2, in which  $\mathbf{x}_k$  remains stationary at each step.

Figure 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  $\mathbf{x}^*(\mathbf{c}) = \mathcal{U}(\mathbf{x}^*(\mathbf{c}), \mathbf{c})$  is a *fixed point* of (U). Interestingly though, the backward pass also converges, but at a slightly faster rate than when starting from the random  $\mathbf{x}_0^b$ .

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 (1) [Beck, 2017]. 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  $\mathbf{x}_0 = \mathbf{x}^*$  is a fixed point of the iteration (U), we have  $\mathcal{U}(\mathbf{x}_k) = \mathbf{x}_k = \mathbf{x}^*$ ,  $\forall k$ . Hence, *there is no need to evaluate the forward pass* of  $\mathcal{U}$  in each unfolded iteration of (U) at  $\mathbf{x}^*$ .

This enables the use of any specialized method to compute the forward pass optimization (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 [Gurobi Optimization, LLC, 2023], 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 [Agrawal *et al.*, 2019], 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 3 illustrate that its backward pass must still be iterated until convergence. However, since  $\mathbf{x}_k = \mathbf{x}^*$ , we also have  $\frac{\partial \mathcal{U}(\mathbf{x}_k)}{\partial \mathbf{x}_k} = \frac{\partial \mathcal{U}(\mathbf{x}^*)}{\partial \mathbf{x}^*}$  at each iteration. Therefore the backward pass of  $\mathcal{U}$  need only be computed *once*, and iterated until backpropagation of the full optimization mapping (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 [Quarteroni *et al.*, 2010].

**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*

$$\mathbf{z}_{k+1} = \mathbf{B}\mathbf{z}_k + \mathbf{b} \quad (\text{LFPI})$$

*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  $\mathbf{z}_k \rightarrow \mathbf{z}$  is*

$$-\log \rho(\mathbf{B}). \quad (9)$$

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  $\frac{\partial \mathbf{x}^*}{\partial \mathbf{c}}(\mathbf{c})$  by unfolding at a fixed point. Consider the fixed-point conditions of the iteration (U):

$$\mathbf{x}^*(\mathbf{c}) = \mathcal{U}(\mathbf{x}^*(\mathbf{c}), \mathbf{c}) \quad (\text{FP})$$

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

$$\frac{\partial \mathbf{x}^*}{\partial \mathbf{c}}(\mathbf{c}) = \underbrace{\frac{\partial \mathcal{U}}{\partial \mathbf{x}^*}(\mathbf{x}^*(\mathbf{c}), \mathbf{c})}_{\Phi} \cdot \frac{\partial \mathbf{x}^*}{\partial \mathbf{c}}(\mathbf{c}) + \underbrace{\frac{\partial \mathcal{U}}{\partial \mathbf{c}}(\mathbf{x}^*(\mathbf{c}), \mathbf{c})}_{\Psi} \quad (10)$$

by the chain rule and recognizing the implicit and explicit dependence of  $\mathcal{U}$  on the independent parameters  $\mathbf{c}$ . Equation (10) will be called the *differential fixed-point conditions*. Rearranging (10), the desired  $\frac{\partial \mathbf{x}^*}{\partial \mathbf{c}}(\mathbf{c})$  can be found in terms of  $\Phi$  and  $\Psi$  as defined above, to yield the system (DFP) below.

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

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

$$(\mathbf{I} - \Phi) \frac{\partial \mathbf{x}^*}{\partial \mathbf{c}} = \Psi. \quad (\text{DFP})$$

The result follows from the Implicit Function theorem [Munkres, 2018]. It implies that the Jacobian  $\frac{\partial \mathbf{x}^*}{\partial \mathbf{c}}$  can be found as the solution to a linear system once the prerequisite Jacobians  $\Phi$  and  $\Psi$  are found; these correspond to backpropagation of the update function  $\mathcal{U}$  at  $\mathbf{x}^*(\mathbf{c})$ .

## 5 Folded Optimization

We are now ready to discuss the central result of the paper. Informally, it states that the backward pass of an iterative solver (U), unfolded at a precomputed optimal solution  $\mathbf{x}^*(\mathbf{c})$ , 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 shows 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  $\mathbf{x}^*$  converges for certain parameters  $\mathbf{c}$  through a sequence of iterates  $\mathbf{x}_k(\mathbf{c}) \rightarrow \mathbf{x}^*(\mathbf{c})$  using algorithm (U), and that  $\Phi$  is nonsingular with a spectral radius  $\rho(\Phi) < 1$ .

**Theorem 1.** *The backward pass of an unfolding of algorithm (U), starting at the point  $\mathbf{x}_k = \mathbf{x}^*$ , is equivalent to linear fixed-point iteration on the linear system (DFP), and will converge to its unique solution at an asymptotic rate of*

$$-\log \rho(\Phi). \quad (11)$$

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

$$(\mathbf{I} - \Phi) \frac{\partial \mathbf{x}^*}{\partial \mathbf{c}} = \Psi \quad (12)$$

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

$$\frac{\partial}{\partial \mathbf{c}} [\mathbf{x}_{k+1}(\mathbf{c})] = \frac{\partial}{\partial \mathbf{c}} [\mathcal{U}(\mathbf{x}_k(\mathbf{c}), \mathbf{c})] \quad (13a)$$

$$\frac{\partial \mathbf{x}_{k+1}}{\partial \mathbf{c}}(\mathbf{c}) = \frac{\partial \mathcal{U}}{\partial \mathbf{x}_k} \frac{\partial \mathbf{x}_k}{\partial \mathbf{c}} + \frac{\partial \mathcal{U}}{\partial \mathbf{c}}, \quad (13b)$$

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

$$\frac{\partial \mathbf{x}_1}{\partial \mathbf{c}}(\mathbf{c}) = \frac{\partial \mathcal{U}}{\partial \mathbf{c}}(\mathbf{x}_0, \mathbf{c}). \quad (14)$$

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

$$\mathbf{x}_k(\mathbf{c}) = \mathbf{x}^*(\mathbf{c}) \quad \forall k \geq 0, \quad (15)$$

which implies

$$\frac{\partial \mathcal{U}}{\partial \mathbf{x}_k}(\mathbf{x}_k(\mathbf{c}), \mathbf{c}) = \frac{\partial \mathcal{U}}{\partial \mathbf{x}^*}(\mathbf{x}^*(\mathbf{c}), \mathbf{c}) = \Phi, \quad \forall k \geq 0 \quad (16a)$$

$$\frac{\partial \mathcal{U}}{\partial \mathbf{c}}(\mathbf{x}_k(\mathbf{c}), \mathbf{c}) = \frac{\partial \mathcal{U}}{\partial \mathbf{c}}(\mathbf{x}^*(\mathbf{c}), \mathbf{c}) = \Psi, \quad \forall k \geq 0. \quad (16b)$$

Letting  $\mathbf{J}_k := \frac{\partial \mathbf{x}_k}{\partial \mathbf{c}}(\mathbf{c})$ , the rule (13b) for unfolding at a fixed-point  $\mathbf{x}^*$  becomes, along with initial conditions (14),

$$\mathbf{J}_0 = \Psi \quad (17a)$$

$$\mathbf{J}_{k+1} = \Phi \mathbf{J}_k + \Psi. \quad (17b)$$

The result then holds by direct application of Lemma 1 to (17), recognizing  $\mathbf{z}_k = \mathbf{J}_k$ ,  $\mathbf{B} = \Phi$  and  $\mathbf{z}_0 = \mathbf{b} = \Psi$ .  $\square$

The following is a direct result from the proof of Theorem 1.

**Corollary 1.** *Backpropagation of the fixed-point unfolding consists of the following rule:*

$$\mathbf{J}_0 = \Psi \quad (18a)$$

$$\mathbf{J}_{k+1} = \Phi \mathbf{J}_k + \Psi, \quad (18b)$$

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

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

**Corollary 2.** *When the parametric problem (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  $-\log \rho(\Phi)$ .*



The result above helps explain why the forward and backward pass in the experiment of Section 4 converge at different rates. If the forward pass converges faster than  $-\log \rho(\Phi)$ , the overall convergence rate of an unfolding is limited by that of the backward pass.

**Fixed-Point Folding.** To improve efficiency, and building on the above findings, we propose to replace unfolding at the fixed point  $\mathbf{x}^*$  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 *folded optimization* that eliminates unrolling entirely. 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. Its modularization of the forward and backward passes, which are typically intertwined in unrolling, also allows for efficient blackbox implementations of each pass.

It is important to note that as per Definition 1, the innermost optimization loop of a nested unrolling can be considered an unfolding and can be backpropagated by JgP with fixed-point folding. Subsequently, the next innermost loop can now be considered unfolded and the same process applied until all unrolled optimization loops are replaced with their analytical models. Figure 1 depicts fixed-point folding, where the gray arrows symbolize a blackbox forward pass and the long red arrows illustrate that a backpropagation is performed an iterative linear system solver. The procedure is also exemplified by *f-PGDb* (introduced in 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.

Note that although it is not used for forward pass convergence, a folded optimizer still typically requires selecting a constant stepsize, or similar parameter, to specify  $\mathcal{U}$  and the resulting Jacobian model (DFP). This can affect  $\rho(\Phi)$ , and hence the backward pass convergence and its rate by Theorem 1. A further discussion of the aspect is made in Appendix D.

## 6 Experiments

This section evaluates folded optimization on four end-to-end optimization and learning tasks. It is primarily evaluated against `cvxpy`, which is the preminent 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 (see Appendix A). 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  $\mathbf{x}^*(\mathbf{c})$ . In each experiment below, this is used to demonstrate a different advantage.

A summary of results is provided below for each study, and a more complete specification is provided in Appendix E.

**Implementation details.** All the folded optimizers used in this section were produced using the accompanying Python library `fold-opt`, which supplies routines for constructing and solving the system (DFP), and integrating the resulting Jacobian-vector products into the computational graph of PyTorch. To do so, it requires a Pytorch implementation of an update function  $\mathcal{U}$  for an appropriately chosen optimization routine. The linear system (DFP) is solved by a user-specified blackbox linear solver, as is the forward-pass optimization solver, as discussed in Section 4. Implementation details of `fold-opt` can be found in Appendix B.

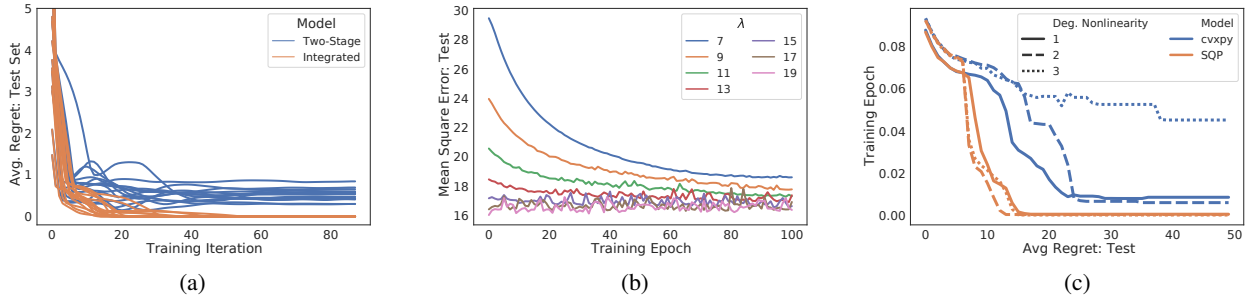
The experiments test four folded optimizers: (1) *f-PGDa* applies to optimization mappings with linear constraints, and is based on folding projected gradient descent steps, where each inner projection is a QP solved by the differentiable QP solver `qpth` [Amos and Kolter, 2017]. (2) *f-PGDb* is a variation on the former, in which the inner QP step is differentiated by fixed-point folding of the ADMM solver detailed in Appendix C. (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) shown in Appendix C. 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

$$\begin{aligned} \mathbf{x}^*(\mathbf{c}, \mathbf{d}) = \operatorname{argmax}_{\mathbf{0} \leq \mathbf{x}, \mathbf{y} \leq \mathbf{1}} \quad & \mathbf{c}^T \mathbf{x} + \mathbf{x}^T \mathbf{Q} \mathbf{y} + \mathbf{d}^T \mathbf{y} \\ \text{s. t.} \quad & \sum \mathbf{x} = p, \sum \mathbf{y} = q, \end{aligned}$$

in which two separable linear programs are confounded by a nonconvex quadratic objective term  $\mathbf{Q}$ . Costs  $\mathbf{c}$  and  $\mathbf{d}$  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 [Audet *et al.*, 2004]. 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 E, 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 [Attouch *et al.*, 2013], 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 (see Appendix F for additional details). The integrated *f-PGDb* model minimizes solution regret (i.e., suboptimality) directly. [Elmachtoub and Grigas, 2021]. Notice in Figure 4(a) how *f-PGDb* achieves much lower regret for each of the 15 nonconvex objectives.


 Figure 4: Bilinear decision focus (a), Enhanced Denoising with  $f$ -FDPG (b), and Portfolio optimization (c).

**Enhanced total variation denoising.** This experiment illustrates the efficiency benefit of incorporating problem-specific solvers. The optimization models a denoiser

$$\mathbf{x}^*(\mathbf{D}) = \operatorname{argmin}_{\mathbf{x}} \frac{1}{2} \|\mathbf{x} - \mathbf{d}\|^2 + \lambda \|\mathbf{D}\mathbf{x}\|_1,$$

which seeks to recover the true signal  $\mathbf{x}^*$  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(b) shows test MSE throughout training due to  $f$ -FDPG for various choice of  $\lambda$ . Appendix G shows comparable results from the framework of [Amos and Kolter, 2017], which converts the problem to a QP form (see Appendix C) 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  $\mathbf{x}^*(\mathbf{D})$  up to 40 times faster.

**Multilabel 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 [Berrada *et al.*, 2018], implements a smooth top-5 classification model on noisy CIFAR-100. The optimization below maps image feature embeddings  $\mathbf{c}$  from DenseNet 40-40 [Huang *et al.*, 2017], to smoothed top- $k$  binary class indicators (see Appendix E for more details):

$$\mathbf{x}^*(\mathbf{c}) = \operatorname{argmax}_{\mathbf{x}} \mathbf{c}^T \mathbf{x} + \sum_i x_i \log x_i \quad \text{s.t.} \quad \sum \mathbf{x} = k \quad (19)$$

Appendix G 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:

$$\mathbf{x}^*(\mathbf{c}) = \operatorname{argmax}_{\mathbf{0} \leq \mathbf{x}} \mathbf{c}^T \mathbf{x} \quad \text{s.t.} \quad \mathbf{x}^T \mathbf{V} \mathbf{x} \leq \gamma, \quad \sum \mathbf{x} = 1. \quad (20)$$

This model describes a risk-constrained portfolio optimization where  $\mathbf{V}$  is a covariance matrix, and the predicted cost coefficients  $\mathbf{c}$  represent assets prices [Elmachtoub and Grigas, 2021]. A 5-layer ReLU network is used to predict future prices  $\mathbf{c}$  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 (20). 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(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 (please see Appendix E for details). 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.*

## 7 Conclusions

This paper introduced folded optimization, a framework for generating analytically differentiable optimization solvers from 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. The paper 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.

## Acknowledgements

This research is partially supported by NSF grant 2232054 and NSF CAREER Award 2143706. Fioretto is also supported by an Amazon Research Award and a Google Research Scholar Award. Its views and conclusions are those of the authors only.

## References

- [Adams and Zemel, 2011] Ryan Prescott Adams and Richard S Zemel. Ranking via sinkhorn propagation. *arXiv preprint arXiv:1106.1925*, 2011.
- [Agrawal *et al.*, 2019] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J Zico Kolter. Differentiable convex optimization layers. *Advances in neural information processing systems*, 32, 2019.
- [Amos and Kolter, 2017] Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, pages 136–145. PMLR, 2017.
- [Attouch *et al.*, 2013] Hedy Attouch, Jérôme Bolte, and Benar Fux 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.
- [Audet *et al.*, 2004] Charles Audet, Jack Brimberg, Pierre Hansen, Sébastien Le Digabel, and Nenad Mladenović. Pooling problem: Alternate formulations and solution methods. *Management science*, 50(6):761–776, 2004.
- [Beck, 2017] Amir Beck. *First-order methods in optimization*. SIAM, 2017.
- [Berrada *et al.*, 2018] Leonard Berrada, Andrew Zisserman, and M. Pawan Kumar. Smooth loss functions for deep top-k classification. *ArXiv*, abs/1802.07595, 2018.
- [Boyd *et al.*, 2011] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan 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.
- [Elmachtoub and Grigas, 2021] Adam N Elmachtoub and Paul Grigas. Smart “predict, then optimize”. *Management Science*, 2021.
- [Gurobi Optimization, LLC, 2023] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual. url-<https://www.gurobi.com>, 2023.
- [Huang *et al.*, 2017] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.
- [Kotary *et al.*, 2021] James Kotary, Ferdinando Fioretto, Pascal Van Hentenryck, and Bryan 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.
- [Kotary *et al.*, 2023a] James Kotary, Francesco Di Vito, and Ferdinando Fioretto. Differentiable model selection for ensemble learning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI-23*, 2023.
- [Kotary *et al.*, 2023b] James Kotary, My H. Dinh, and Ferdinando Fioretto. Backpropagation of unrolled solvers with folded optimization. *arXiv preprint arXiv:2301.12047*, 2023.
- [Martins and Astudillo, 2016] Andre Martins and Ramon 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.
- [Monga *et al.*, 2021] Vishal Monga, Yuelong Li, and Yonina C Eldar. Algorithm unrolling: Interpretable, efficient deep learning for signal and image processing. *IEEE Signal Processing Magazine*, 38(2):18–44, 2021.
- [Munkres, 2018] James R Munkres. *Analysis on manifolds*. CRC Press, 2018.
- [Quarteroni *et al.*, 2010] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010.
- [Wilder *et al.*, 2019] Bryan Wilder, Bistra Dilkina, and Milind Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *AAAI*, volume 33, pages 1658–1665, 2019.