

Analyzing and Enhancing the Backward-Pass Convergence of Unrolled Optimization

James Kotary, Jacob K Christopher, My H Dinh, Ferdinando Fioretto

University of Virginia, Charlottesville, Virginia, USA

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 often lacks a closed form. One typical strategy is algorithm unrolling, which relies on automatic differentiation through the entire chain of operations executed by an iterative optimization solver. This paper provides theoretical insights into the backward pass of unrolled optimization, showing that it is asymptotically equivalent to the solution of a linear system by a particular iterative method. Several practical pitfalls of unrolling are demonstrated in light of these insights, and a system called Folded Optimization is proposed to construct more efficient backpropagation rules from unrolled solver implementations. Experiments over various end-to-end optimization and learning tasks demonstrate the advantages of this system both computationally, and in terms of flexibility over various optimization problem forms.

Code available at: <https://fold-opt.github.io>

Keywords: Folded optimization, deep unrolling, decision focused learning, differentiable optimization

Email addresses: jk4pn@virginia.edu (James Kotary), csk4sr@virginia.edu (Jacob K Christopher), fqw2tz@virginia.edu (My H Dinh), fioretto@virginia.edu (Ferdinando Fioretto)

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. By allowing neural networks to learn over the space of parametrized optimal solutions, optimization as a layer can offer enhanced accuracy and efficiency on specialized learning tasks, by imparting predefined task-specific structure to their representations [1].

For example, optimization layers can generalize the functionality of earlier structured prediction mechanisms for tasks such as multilabel classification [2] and learning to rank [3, 4] using simple optimization models. The integration of operational decision problems as components in neural networks has shown promise in enhancing the effectiveness of data-driven decision models [5]. Some work has even shown that optimization components with learnable constraints can be used as general-purpose layers, capable of greater expressiveness than conventional linear layers [6].

While these mechanisms can be used in much the same way as linear layers and activation functions, the resulting end-to-end models require training by stochastic gradient descent, which in turn requires differentiation of the optimization mappings for backpropagation of gradients. 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 [7]. *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, this requires construction of the problem-specific implicit derivative models, and the most popular current framework for its automation puts rigid requirements on the form of the optimization problems, relying on transformations to canonical convex cone programs before applying a standardized procedure for their solution and differentiation [8]. This precludes the use of specialized solvers that are best-suited to handle various optimization problems, and inherently restricts itself only to convex problems.

Contributions. This paper presents 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 reveals an equivalence between unrolling and analytical differentiation, and allows for convergence of the backward pass to be analyzed. Practically, it allows for the forward and backward passes of unrolled optimization to be disentangled and solved separately, using generic implementations of specialized algorithms. More specifically, this paper makes the following novel contributions¹:

1. A theoretical analysis of unrolling, which shows that its backward pass is asymptotically equivalent to the solution of a linear system of equations by a particular iterative method, and allows for its convergence properties to be quantified.
2. An empirical evaluation of the backward-pass convergence behavior of unrolled optimization, which corroborates several aspects of the aforementioned theory and indicates several potential pitfalls that arise as a result of its naive implementation.
3. Building on this analysis, the paper proposes a system for generating analytically differentiable optimization solvers from unrolled implementations called *folded optimization*, accompanied by a Python library called `fold-opt` to facilitate automation, available at: <https://fold-opt.github.io>.
4. The efficiency and modeling advantages of folded optimization are demonstrated on a diverse set of end-to-end optimization and learning tasks, which include end-to-end learning with difficult *nonconvex* optimization.

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

¹This paper is an extended version of IJCAI-23 paper [9]. It expands on the conference version substantially, with new material detailing extensions to the folded optimization library, and a new collection of experiments which study the backward-pass convergence of unrolled optimization and empirically illustrate the improvements therein which are made possible by the paper’s proposed framework.

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

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

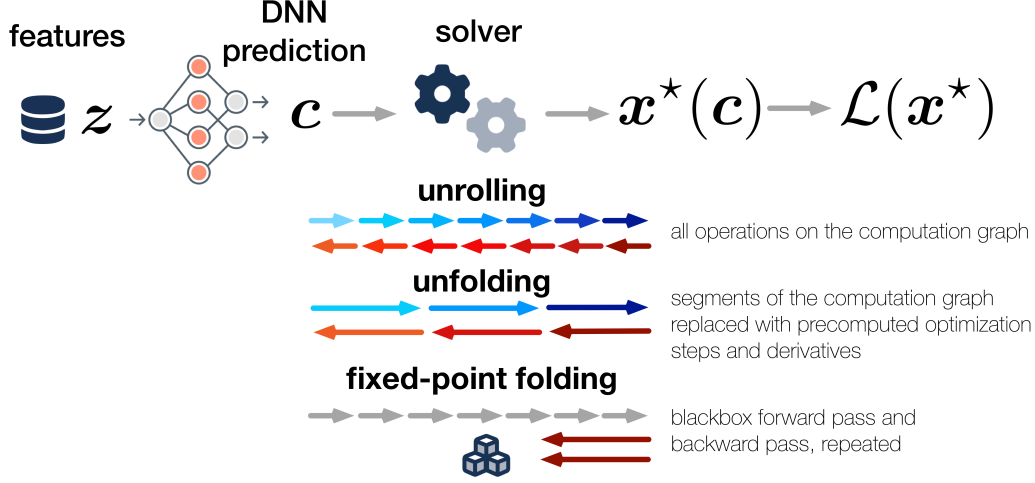


Figure 1: Compared to unrolling, unfolding requires fewer operations on the computational graph by replacing inner loops with Jacobian-gradient products. Fixed-point folding models the unfolding analytically, allowing for generic implementations.

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

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

The iterations (U) converge if $\mathbf{x}_k(\mathbf{c}) \rightarrow \mathbf{x}^*(\mathbf{c})$ as $k \rightarrow \infty$; in the present paper, this is referred to as *forward-pass convergence*. When the iterations (U) are *unrolled*, they are computed and backpropagated on the computational graph, and the overall function $\mathbf{x}^*(\mathbf{c})$ is thereby backpropagated by AD without explicitly representing its Jacobian matrix $\frac{\partial \mathbf{x}^*(\mathbf{c})}{\partial \mathbf{c}}$. The backpropagation of the unrolled solution process is also an iterative procedure, and we aim to analyze its convergence. To this end, we define convergence of the backward pass in unrolling as follows:

Definition 1. Suppose that an unrolled iteration (U) produces a convergent sequence of solution iterates $\lim_{k \rightarrow \infty} \mathbf{x}_k = \mathbf{x}^*$ in its forward pass. Then con-

vergence of the backward pass is defined as the condition

$$\lim_{k \rightarrow \infty} \frac{\partial \mathbf{x}_k}{\partial \mathbf{c}}(\mathbf{c}) = \frac{\partial \mathbf{x}^*}{\partial \mathbf{c}}(\mathbf{c}), \quad (3)$$

assuming that all requisite derivatives exist.

Unrolling (U) over many iterations often faces time and space inefficiency issues due to the need for storage and traversal of the optimization procedure’s entire computational graph [7]. The following sections analyze the backward pass of unrolled optimization to yield equivalent analytical models for the Jacobian $\frac{\partial \mathbf{x}^*}{\partial \mathbf{c}}(\mathbf{c})$, and show how to efficiently solve those models by leveraging the backward pass of (U). To do so, 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 5 introduces **unfolding** as a variant of unrolling, in which the unrolling of such inner loops is circumvented by analytical differentiation of the subproblem, allowing its 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 6 shows that the backward pass converges correctly even when the forward-pass iterations are initialized at the 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 generic implementations of both passes.

Section 6 uses these concepts to show that the backward pass of unfolding (U) follows the solution, by fixed-point iteration, of the linear system for $\frac{\partial \mathbf{x}^*(\mathbf{c})}{\partial \mathbf{c}}$ which arises by differentiating the fixed-point conditions of (U). This allows for the conditions and rate of its convergence to be analyzed. Section 8 then outlines **folded optimization**, a system for generating Jacobian-gradient products through optimization based on efficient solution of the models proposed in Section 6. The main differences between unrolling, unfolding, and folded optimization are illustrated in Figure 1.

4. Related Work

We end-to-end optimization and learning approaches into those based on *unrolling*, and *analytical* differentiation. Since this paper focuses on converting unrolled implementations into analytical ones, each category is reviewed first below.

Unrolling optimization algorithms. Automatic Differentiation (AD) is the primary method of backpropagating gradients in deep learning models for training with stochastic gradient descent. Modern machine learning frameworks such as PyTorch have natively implemented differentiation rules for a variety of functions that are commonly used in deep models, as well as interfaces to define custom differentiation rules for new functions [10]. As a mainstay of deep learning, AD is also a natural tool for backpropagating through constrained optimization mappings. *Unrolling* refers to the execution of an optimization algorithm, entirely on the computational graph, for backpropagation by AD from the resulting optimal solution to its input parameters. Such approaches are general and apply to a broad range of optimization models. They can be performed simply by implementing a solution algorithm within an AD framework, without the need for analytical modeling of an optimization mapping’s derivatives [11]. However, unrolling over many iterations has been shown to encounter issues of time and memory inefficiency due to the size of its computational graph [6]. Further issues encountered in unrolling, such as vanishing and exploding gradients, are reminiscent of recurrent neural networks [7]. On the other hand, unrolling may offer some unique practical advantages, like the ability to learn optimization parameters such as stepsizes to accelerate the solution of each optimization during training [12].

Analytical differentiation of optimization models. Differentiation through constrained argmin problems in the context of machine learning was discussed as early as [13], who proposed first to implicitly differentiate the argmin of a smooth, unconstrained convex function by its first-order optimality conditions, defined when the gradient of the objective function equals zero. This technique is then extended to find approximate derivatives for constrained problems, by applying it to their unconstrained log-barrier approximations. Subsequent approaches applied implicit differentiation to the KKT optimality conditions of constrained problems directly [6, 14], but only on special problem classes such as Quadratic Programs. [15] extend

the method of [6], by modeling second-order derivatives of the optimization for training with gradient boosting methods. [16] uses the differentiable quadratic programming solver of [6] to approximately differentiate general convex programs through quadratic surrogate problems. Other problem-specific approaches to analytical differentiation models include ones for sorting and ranking [17], linear programming [18], and convex cone programming [19].

The first general-purpose differentiable optimization solver was proposed in [8], which leverages the fact that any convex program can be converted to a convex cone program [20]. The equivalent cone program is subsequently solved and differentiated following [19], which implicitly differentiates a zero-residual condition representing optimality [21]. A differentiable solver library `cvxpy` is based on this approach, which converts convex programs to convex cone programs by way of their graph implementations as described in [22]. The main advantage of the system is that it applies to any convex program and has a simple symbolic interface. A major disadvantage is its restriction to solving problems only in a standard convex cone form with an ADMM-based conic programming solver, which performs poorly on some problem classes, as seen in Section 8.

A related line of work concerns end-to-end learning with *discrete* optimization problems, which includes linear programs, mixed-integer programs and constraint programs. These problem classes often define discontinuous mappings with respect to their input parameters, making their true gradients unhelpful as descent directions in optimization. Accurate end-to-end training can be achieved by *smoothing* the optimization mappings, to produce approximations which yield more useful gradients. A common approach is to augment the objective function with smooth regularizing terms such as euclidean norm or entropy functions [5, 23, 18]. Others show that similar effects can be produced by applying random noise to the objective [24, 25], or through finite difference approximations [26, 27]. This enables end-to-end learning with discrete structures such as constrained ranking policies [4], shortest paths in graphs [28], and various decision models [5].

5. 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* subroutine $\mathcal{O} : \mathbb{R}^n \rightarrow \mathbb{R}^n$. 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 $\mathbf{x}^*(\mathbf{c})$.

Definition 2 (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 7, 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 [6]. Alternatively, the replacement of unrolled loops by JgP’s proposed in Section 7 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 8.

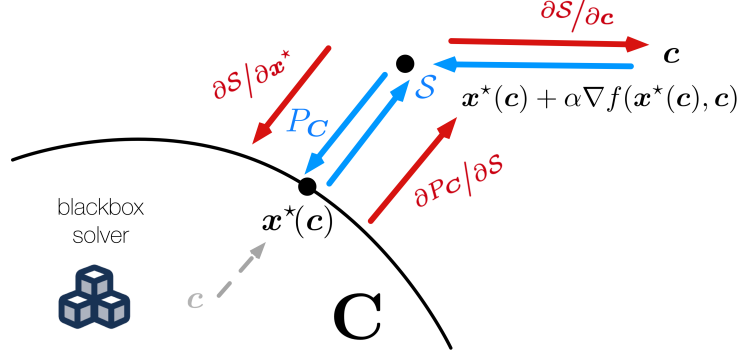


Figure 2: Unfolding Projected Gradient Descent at $\mathbf{x}^*$ consists of alternating gradient step $\mathcal{S}$ with projection $\mathcal{P}_{\mathbf{C}}$. Section 6 shows that the resulting chain of JgP operations in backpropagation is equivalent to solving the differential fixed-point conditions (DFP) by linear fixed-point iteration. Each function’s forward and backward pass are illustrated in blue and red, respectively.

Projected gradient descent. Given a problem

$$\min_{\mathbf{x} \in \mathbf{C}} f(\mathbf{x}) \quad (4)$$

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

$$\mathbf{x}_{k+1} = \mathcal{P}_{\mathbf{C}}(\mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)), \quad (5)$$

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 (5) (see [29]). 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 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

$$\min_{\mathbf{x}} f(\mathbf{x}) + g(\mathbf{x}) \quad (6)$$

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

$$\mathbf{x}_{k+1} = \text{Prox}_{\alpha_k g}(\mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)). \quad (7)$$

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

$$\text{Prox}_g(\mathbf{x}) = \underset{\mathbf{y}}{\text{argmin}} \left\{ g(\mathbf{y}) + \frac{1}{2} \|\mathbf{y} - \mathbf{x}\|^2 \right\}, \quad (8)$$

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 A](#).

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 A](#)) with the `qpth` differentiable QP solver.

Quadratic programming by ADMM. The QP solver of [\[30\]](#), based on the alternating direction of multipliers, is specified in [Appendix A](#). 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 7](#). 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 8](#).

From the viewpoint of unfolding, the analysis of backpropagation in unrolled solvers can be simplified by accounting for only a single unrolled loop at a time. The next section identifies a further simplification: *that the back-*

propagation of an unfolded solver can be completely characterized by its action at a fixed point of the solution’s algorithm.

6. Unfolding at a Fixed Point

Optimization methods of the form (U) require a starting point $\mathbf{x}_0$, which is often chosen arbitrarily, since (forward-pass) convergence $\mathbf{x}_k \rightarrow \mathbf{x}^*$ is typically ensured regardless of $\mathbf{x}_0$. In unfolded optimization, it is natural to also ask how the choice of $\mathbf{x}_0$ affects the backward-pass convergence. Here, the special case when $\mathbf{x}_0 = \mathbf{x}^*$ is of particular interest. In this case, the forward pass of an unrolled optimization is equivalent to an identity function at each iteration, since $\mathbf{x}^*$ is a fixed point of $\mathcal{U}$. Therefore if the backward pass converges in this case (as per Definition 1), it can be considered as a standalone procedure, independent of the optimization method’s forward pass. This separation of the forward and backward passes is key to an enhanced system of backpropagation, called *folded optimization*, introduced in Section 7.

In this section, we first demonstrate empirically that the backward pass of unfolded optimization does in fact converge, resulting in correct gradients when the starting point $\mathbf{x}_0 \rightarrow \mathbf{x}^*$ is chosen. Then a theoretical analysis is presented, which shows that the principle holds in general, and allows the backward-pass convergence to be analyzed. A more thorough empirical study is then discussed, which corroborates the main theoretical results in practice, while illustrating the disadvantages and potential pitfalls inherent to naive unfolding implementations. Section 7 then shows how the results of this Section form the basis of folded optimization, a more efficient and reliable system for backpropagation which builds on the idea of unfolding at a precomputed fixed point.

6.1. An illustrative example

The empirical results of this section are based on a representative example problem, in which the forward and backward pass errors are measured at each iteration of an unfolded solver. The optimization problem (36) maps feature embeddings $\mathbf{c}$ to smoothed top- k class indicators $\mathbf{x}^*$, and is used to learn multilabel classification later in Section 8. Unfolded projected gradient descent is used to differentially compute the mapping $\mathbf{c} \rightarrow \mathbf{x}^*(\mathbf{c})$, in which the projection onto linear constraints is computed and backpropagated using the differentiable `qpth` QP solver. A loss function $\mathcal{L}$ targets ground-truth top- k indicators, and the result of the backward pass is an estimate of the

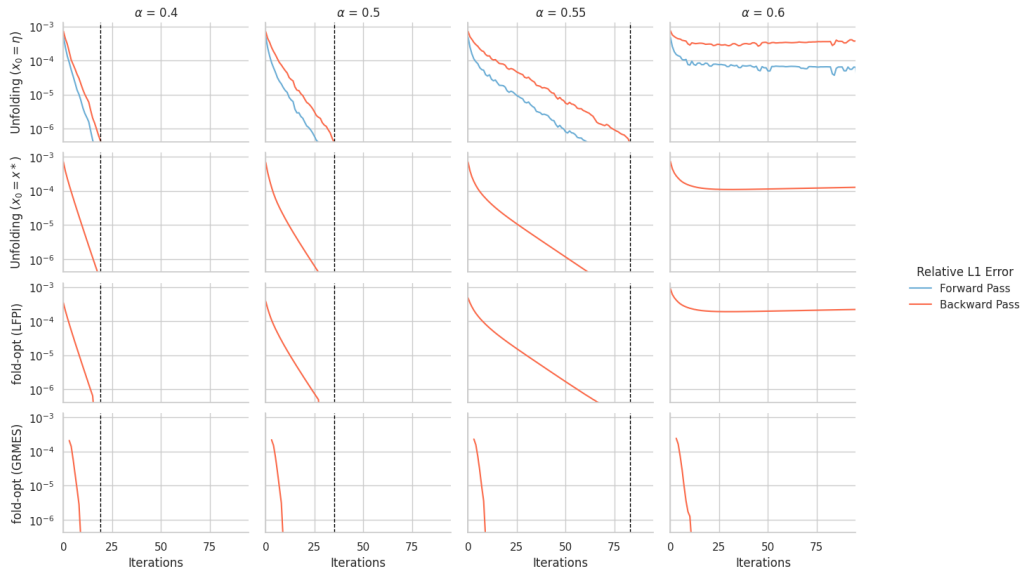


Figure 3: Forward and backward pass error per number of iterations, across different stepsizes on CIFAR100 Multilabel Classification. Error is measured on average over 100 samples. Each row represents a distinct differentiable solver implementation; the first two represent unfolded PGD and the latter two represent folded optimization counterparts. Columns correspond to PGD stepsize.

gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{c}}$. We evaluate the forward and backward-pass convergence of unfolded projected gradient descent, by measuring the relative L_1 errors of the forward and backward passes, relative to the true optimal solutions and corresponding loss gradients.

Two types of starting points are considered: the precomputed optimal solution $\mathbf{x}_0^a = \mathbf{x}^*$, and a uniform random vector $\mathbf{x}_0^b = \eta \sim \mathbf{U}(0, 1)$. In the latter case, the error is reported on average over 20 random starting points. The former case is illustrated in Figure 2 in which $\mathbf{x}_k$ remains stationary at each step of projected gradient descent in its forward pass. In addition, four different *fixed* gradient stepsizes $\alpha \in \{0.4, 0.5, 0.55, 0.6\}$ are considered. Figure 3 plots the relative L_1 errors of the forward pass (in blue) and backward pass (in red) for $0 \leq n \leq 100$ iterations of unfolded PGD under the two starting points and various stepsizes. The first two rows correspond to unfolding PGD in the cases $\mathbf{x}_0 = \eta$ and $\mathbf{x}_0 = \mathbf{x}^*$; the last two rows correspond to two folded optimization variants and are discussed later.

The absence of blue curves in the case $\mathbf{x}_0 = \mathbf{x}^*$ indicates that when starting the unfolding from the precomputed optimal solution, the forward pass error remains near zero. This behavior is expected, since $\mathbf{x}^*(\mathbf{c}) = \mathcal{U}(\mathbf{x}^*(\mathbf{c}), \mathbf{c})$ is a fixed point of $\mathcal{U}$. On the other hand, the figure’s red curves show that for each chosen stepsize α , the backward pass of unfolding converges whenever the forward pass converges, whether it is initialized at a fixed point or a random point. Further, the rate of backward-pass convergence is highly dependent on the chosen stepsize α , even when $\mathbf{x}_0 = \mathbf{x}^*$. The case $\alpha = 0.6$ indicates a critical point, beyond which the unfolding fails to converge.

We observe from Figure 3 two major trends in the convergence patterns of the backward pass: (1) The vertical dotted lines of Figure 3 aid comparison against time to convergence in the backward pass which results from the choice of $\mathbf{x}_0 = \eta$. Backward-pass convergence is always faster in the case $\mathbf{x}_0 = \mathbf{x}^*$, and this effect becomes more pronounced for the α which result in slower convergence. (2) The convergence of the backward pass always lags behind that of the forward pass in the case where $\mathbf{x}_0 = \eta$, and this effect also becomes more pronounced as the choice of α leads to slower convergence.

Unfolding in the case when $\mathbf{x}_0 = \mathbf{x}^*$ is referred to as *fixed-point unfolding*. While its backward pass tends to converge faster than that of general unfolding, its requirement of both the precomputed solution $\mathbf{x}^*$ and the unfolded iterations $\mathcal{U}$ make it impractical in terms of efficiency. However, as shown next, fixed-point unfolding forms an important conceptual starting point for understanding the backward-pass convergence of unfolded optimization in

general, and for the more efficient *folded* optimization system introduced in Section 7.

6.2. Backward Convergence of Fixed-Point Unfolding

Next, it will be shown that backpropagation of unfolded optimization at a fixed point is equivalent to solving a linear system of equations for the backpropagated gradients, using a particular iterative method for linear systems. In order to prove this, the following two Lemmas respectively identify the iterative solution method, and the linear system it solves. The following textbook result can be found, e.g., in [31].

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 $\mathbf{c}$, we define the Jacobians Φ and Ψ :

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

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 Φ and Ψ 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 Φ nonsingular,*

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

The result follows from the Implicit Function theorem [32]. 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 Φ and Ψ are found; these Jacobians correspond to backpropagation through the update function $\mathcal{U}$ at $\mathbf{x}^*(\mathbf{c})$, with respect to $\mathbf{x}^*$ and $\mathbf{c}$.

Using the above two Lemmas, the central result of the paper can be proved. 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 perspective allows 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 Φ 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 Φ 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 above results can help to explain the empirical convergence patterns of the illustrative example in the beginning of this Section. First, they help explain the difference in forward and backward-pass convergence rates due to unfolded PGD as shown in row 1 of Figure 3. Regardless of the convergence rate of its forward pass solution, the overall convergence rate of an unfolded optimization is limited by that of the LFPI implicitity applied in its backward pass. It is also clear why the backward pass of unfolding converges faster when its forward pass is initialized at the optimal solution $\mathbf{x}_0 = \mathbf{x}^*$: the correct Φ and Ψ are exactly known at every iteration in this case, and backpropagation follows the rule (18). In the typical case when $\mathbf{x}_0$ is chosen randomly, Φ and Ψ are "moving targets" with respect to the LFPI iterations (18), so the convergence of LFPI in this case is bound to lag behind the convergence of $\mathbf{x}_k$ to $\mathbf{x}^*$.

7. Folded Optimization

As noted in Section 6.1, the fixed-point unfolding approach to backpropagation is inefficient because it requires precomputation of $\mathbf{x}^*$ along with additional unfolded iterations (U). In principle, this inefficiency can be addressed by noting that the forward pass of each unfolded iteration (U) need

not be recomputed at the fixed point $\mathbf{x}^*$, since $\mathbf{x}_k(\mathbf{c}) = \mathbf{x}^*(\mathbf{c}) \forall k \geq 0$ and $\mathcal{U}(\mathbf{x}^*(\mathbf{c}), \mathbf{c}) = \mathbf{x}^*(\mathbf{c})$. Thus we can iterate just its backward pass (18) repeatedly at $\mathbf{x}^*(\mathbf{c})$, for which the requisite Jacobians Φ and Ψ can be obtained by a just a *single* application of the differentiable update step $\mathcal{U}$. This revised approach leads to the most basic variant of *fixed-point folding*.

The essence of fixed-point folding is to use the computational graph of the update step $\mathcal{U}$ to backpropagate the function $\mathbf{c} \rightarrow \mathbf{x}^*(\mathbf{c})$ by modeling and solving the linear system (DFP), after the optimal solution $\mathbf{x}^*(\mathbf{c})$ is separately furnished by any *blackbox* optimization solver. This is in contrast to unrolled or unfolded optimization, which jointly solves for the optimal solution and its backpropagated gradients by repeated application of $\mathcal{U}$ with automatic differentiation enabled. The separation of the forward and backward pass algorithms, which are typically entangled across iterations of unfolding, is key to enabling several practical advantages as detailed below.

This Section describes a system called *folded optimization*, which encompasses a variety of implementation strategies for fixed-point folding. The paper is also accompanied by an open-source PyTorch library called `fold-opt`, which provides practical implementations of folded optimization variants within a convenient user interface. Its function is to facilitate the conversion of unfolded optimization code into more efficient and reliable JgP-based differentiable optimization. To produce a differentiable mapping $\mathbf{c} \rightarrow \mathbf{x}^*(\mathbf{c})$ in `fold-opt`, two elements are required: a differentiable step $\mathcal{U}$ of an iterative method which solves the problem (1), along with any (blackbox) optimization oracle which provides optimal solutions $\mathbf{x}^*(\mathbf{c})$ given $\mathbf{c}$. Note that both of these elements are always available given any unfolded implementation of (U): the former is equivalent to setting the number of unfolded iterations to one. The backpropagation algorithms employed by `fold-opt` are proposed next.

7.1. Folded Optimization: Algorithms

Given access to an optimization solver $\mathbf{c} \rightarrow \mathbf{x}^*(\mathbf{c})$ and differentiable update step $\mathcal{U}$, the goal is to compute a JgP mapping $\mathbf{g} \rightarrow \mathbf{g}^T \mathbf{J}$ where $\mathbf{g} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^*}$ is the incoming gradient and the matrix $\mathbf{J} = \frac{\partial \mathbf{x}^*(\mathbf{c})}{\partial \mathbf{c}}$ solves the linear system $(\mathbf{I} - \Phi)\mathbf{J} = \Psi$; thus $\mathbf{g}^T \mathbf{J} = \frac{\partial \mathcal{L}}{\partial \mathbf{c}}$. While the Jacobian matrices Φ , Ψ and $\mathbf{J}$ are not known explicitly, the products $\mathbf{g}^T \Phi$ and $\mathbf{g}^T \Psi$ can be computed by backpropagation of any vector $\mathbf{g}$ through the computational graph of $\mathcal{U}(\mathbf{x}^*(\mathbf{c}), (\mathbf{c}))$ backward to $\mathbf{x}^*(\mathbf{c})$ and $\mathbf{c}$, respectively. Thus, the backpropagation algorithms of `fold-opt` are designed to compute the desired mapping

$\mathbf{g} \rightarrow \mathbf{g}^T \mathbf{J}$, by using the available mappings $\mathbf{g} \rightarrow \mathbf{g}^T \Phi$ and $\mathbf{g} \rightarrow \mathbf{g}^T \Psi$, which can be obtained by calling $\mathcal{U}$ only once and saving its computational graph. The library implements three distinct approaches to this end, detailed next.

7.1.1. Linear Fixed-Point Iteration

The first variant of folded optimization mimics unfolding at the fixed point $\mathbf{x}^*$ by solving a linear system for the product $\mathbf{g}^T \mathbf{J}$, using a variation of the LFPI algorithm (LFPI). By construction, it is algorithmically equivalent to the backpropagation of fixed-point unfolding (18).

To see how, write the backpropagation of the loss gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^*}$ through k unfolded steps of (U) at the fixed point $\mathbf{x}^*$ as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^*}^T \left(\frac{\partial \mathbf{x}^k(\mathbf{c})}{\partial \mathbf{c}} \right). \quad (19)$$

We seek to compute the limit $\frac{\partial \mathcal{L}}{\partial \mathbf{c}} = \mathbf{g}^T \mathbf{J}$ where $\mathbf{g} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^*}$, $\mathbf{J} := \lim_{k \rightarrow \infty} \mathbf{J}_k$, and $\mathbf{J}_k = \frac{\partial \mathbf{x}^k(\mathbf{c})}{\partial \mathbf{c}}$. Following the backpropagation rule (18), the expression (19) is equal to

$$\mathbf{g}^T \mathbf{J}_k = \mathbf{g}^T (\Phi \mathbf{J}_{k-1} + \Psi) \quad (20a)$$

$$= \mathbf{g}^T (\Phi^k \Psi + \Phi^{k-1} \Psi + \dots + \Phi \Psi + \Psi) \quad (20b)$$

This expression can be rearranged as

$$\mathbf{g}^T \mathbf{J}_k = \mathbf{v}_k^T \Psi \quad (21)$$

where

$$\mathbf{v}_k^T := (\mathbf{g}^T \Phi^k + \mathbf{g}^T \Phi^{k-1} + \dots + \mathbf{g}^T \Phi + \mathbf{g}^T). \quad (22)$$

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

$$\mathbf{v}_k^T = \mathbf{v}_{k-1}^T \Phi + \mathbf{g}^T \quad (23)$$

which identifies $\mathbf{v} := \lim_{k \rightarrow \infty} \mathbf{v}_k$ as the solution of the linear system

$$\mathbf{v}^T (\mathbf{I} - \Phi) = \mathbf{g}^T \quad (24)$$

under the conditions of Lemma (1), after transposing both sides of (23) and (24).

Once $\mathbf{v}^T$ is calculated by (23), the desired JgP is

$$\mathbf{g}^T \mathbf{J} = \mathbf{v}^T \Psi. \quad (25)$$

Thus the end result $\mathbf{g}^T \mathbf{J}$ is computed by iterating (23) to find $\mathbf{v}$ which solves (24), and then applying (25). The left matrix-vector product with respect to Φ in (23) and Ψ in (25) can be computed by backpropagation through the computational graph of the update function $\mathcal{U}(\mathbf{x}^*(\mathbf{c}), \mathbf{c})$, backward to $\mathbf{x}^*(\mathbf{c})$ and $\mathbf{c}$ respectively. Notice that in contrast to unfolding, this backpropagation method requires to store the computational graph only for a single iteration of the update step, rather than for the entire optimization routine consisting of many iterations.

One remaining detail is to initialize the iterates (23) by choosing $\mathbf{v}_0$. The choice of $\mathbf{v}_0$ does not affect asymptotic convergence of (23). However to make LFPI in `fold-opt` completely equivalent with the backpropagation of fixed-point unfolding, Equation (14) shows that the initial iterate must be chosen as follows:

$$\mathbf{v}_0 = \mathbf{g}^T \Psi. \quad (26)$$

Empirical Illustration. The third row of Figure 3 shows the convergence pattern of backpropagation using *fold-opt* in LFPI mode, on the illustrative example of Section 6. As intended, its error curves are nearly identical to those of the second row, which result from unfolding projected gradient descent at its fixed point. Minute differences between the second and third rows of curves can be attributed to numerical floating-point error.

It is apparent from Figure 3 that due to its equivalence with fixed-point unfolding, backpropagation by `fold-opt` LFPI inherits a dependence on the optimization parameter used to define $\mathcal{U}$, in this case the gradient descent stepsize α . This can be explained by Theorem 1: the stepsize α affects $\mathcal{U}$ and thus Φ , along with its spectral radius. In turn, this determines the asymptotic convergence rate of backpropagation by LFPI.

Figure 4 gives an expanded view of this aspect, showing the relationship between stepsize α , the spectral radius ρ , and error per iteration due to LFPI backpropagation in `fold-opt`. Note that Figure 4 coincides with the third row of Figure 3, for its four values of α . Note the continuous relationship between α and the backward-pass convergence rate, which finds a global maximum within the range of alpha shown. The spectral radius $\rho(\Phi)$ is

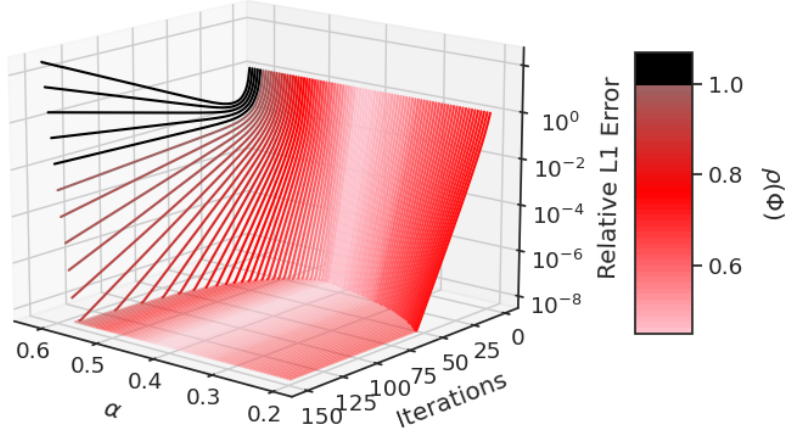


Figure 4: An expanded view of Figure 3’s third row shows backward-pass convergence for fixed-point folding of PGD by GMRes, compared to stepsize α and spectral radius Φ (color scale) on CIFAR100 Multilabel Classification. The main consequences of Theorem 1 are illustrated: convergence rate is maximized when the spectral radius of Φ is minimized, and failure to converge coincides with when the spectral radius exceeds 1.

represented in color coding; as predicted by Theorem 1, it is minimized precisely where the convergence rate is maximized. Further, note that the backward pass fails to converge precisely as $\rho(\Phi)$ exceeds the value 1: that is, the first black curve does not intercept the xy-plane. Taken together, these observations corroborate and provide empirical evidence for the main implications of Theorem 1

7.1.2. Krylov Subspace Methods

The main drawback of backpropagation by LFPI, as described above, is its slow convergence rate. As a generic linear system solver, fixed-point iteration is typically not used in practice due to availability of faster-converging variants including Jacobi, Gauss-Seidel, Successive Over-Relaxation, and Krylov subspace methods [31]. The main advantage of LFPI, which justifies its use in the present context, is its compatibility with solution of the linear system (DFP) by matrix-vector products, since multiplication by Φ and Ψ coincide with backpropagation through $\mathcal{U}$. This allows for the solution of (DFP)

without explicitly computing Φ and Ψ .

Another, more efficient class of linear system solvers which share this characteristic are the Krylov subspace methods [31]. In order to solve a generic linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$, these methods generally act on the basis vectors of a *Krylov subspace* generated by A and a vector r :

$$\mathcal{K}_k(\mathbf{A}; \mathbf{r}) = \text{span}(\{\mathbf{r}, \mathbf{A}\mathbf{r}, \mathbf{A}^2\mathbf{r}, \dots, \mathbf{A}^{k-1}\mathbf{r}\}), \quad (27)$$

but otherwise do not require $\mathbf{A}$ explicitly. Therefore, they can be implemented in the present context to solve (DFP), where only the matrix-vector products with respect to Φ and Ψ are known.

GMRes is the most popular of the general-purpose Krylov subspace methods. It solves at each k^{th} iteration for the minimal-residual vector $\arg\min_x \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2$ which lies within the k^{th} Krylov subspace $\mathcal{K}_k(\mathbf{A}; \mathbf{r})$, where $\mathbf{r} = \mathbf{A}\mathbf{x}_0 - \mathbf{b}$ is the residual of an initial guess. Arnoldi iteration is used to incrementally generate orthonormal bases for the Krylov subspaces, over which the minimal-residual vector can be computed by least-squares via QR decomposition. The decomposition is efficiently updated in order to resolve the least-squares problem at each iteration [33].

The `fold-opt` library implements a variant of GMRes to perform backpropagation as an alternative to LFPI. To see how, note that it was shown above how fixed-point unfolding can be interpreted as solving the equation (24) for $\mathbf{v}$ by LFPI and then applying (25). Here we follow the same pattern, solving (24) instead by GMRes. Since the algorithm is developed for left-sided linear systems, we transpose (24) to make its orientation consistent with the typical GMRes formulation:

$$(\mathbf{I} - \Phi)^T \mathbf{v} = \mathbf{g} \quad (28)$$

where $\mathbf{A} = (\mathbf{I} - \Phi)^T$. Now the Krylov subspace basis vectors can be computed as via multiplication of a vector $\mathbf{r}$ by $(\mathbf{I} - \Phi)^T$ as follows:

$$(\mathbf{I} - \Phi)^T \mathbf{r} = (\mathbf{r} - \mathbf{r}^T \Phi)^T \quad (29)$$

where $\mathbf{r}^T \Phi$ is, once again, computed by backpropagation of $\mathbf{r}$ through $\mathcal{U}$ to $\mathbf{x}^*$. The rest of the method follows a conventional implementation.

The GMRes method typically converges in far fewer iterations than LFPI. This benefit comes at an additional cost of $\mathcal{O}(km)$ flops per each k^{th} iteration, where m is the size of Φ . Additionally, exact convergence of the backward

pass is guaranteed within m iterations [33], at which point the Krylov subspace coincides with $\mathbb{R}^m$. This result is significant, because it highlights the inferior backward-pass convergence properties inherent to unrolling and unfolding optimization, which are limited to the $-\log \rho(\Phi)$ convergence rate of LFPI, by producing equivalent results with faster convergence.

Empirical Illustration. The fourth and final row of Figure 3 shows the backward-pass error per iteration due to fixed-point folding with GMRes as described above. As expected, the backward pass converges in far fewer iterations when compared to the other backpropagation rules. Figure 5 shows a more complete view of the relationship between PGD stepsize, spectral radius and convergence pattern. It can be compared directly to Figure 4. Note that backpropagation with GMRes is not subject to the same effect on convergence pattern due to changes in stepsize as in LFPI. This is because convergence of GMRes does not depend on Φ being a contractive mapping with small spectral radius. Note in particular how convergence is reached in few iterations even when $\rho(\Phi) > 1$ (in black). This is a significant improvement over unfolding and fixed-point folding with LFPI, since their backward-pass convergence rate depends on the spectral radius, whose relationship to optimization parameters such α can be difficult to calculate.

7.1.3. Jacobian Extraction

A final alternative method to solve for the backpropagated gradients $\mathbf{g}^T \mathbf{J}$ is to solve the system (24) directly, by first building the Jacobian matrix Φ . This is done by back-propagating the identity matrix through $\mathcal{U}$ backward to $\mathbf{x}^*$. Subsequently, (25) is applied using backpropagation through $\mathcal{U}$ to $\mathbf{c}$. In this approach, any linear equation solver can be applied since the system (24) is known explicitly. The `fold-opt` library allows the user to pass a blackbox linear system solver when this option is chosen.

The downside to this approach is the cost of building the matrix Φ . When $\Phi \in \mathbb{R}^{m \times m}$, this requires backpropagating each of the m columns of the identity matrix in addition to the cost of solving (24). For comparison, backpropagation by GMRes as described in 7.1.2 is guaranteed to reach full convergence within the same number iterations, each requiring one backward pass through $\mathcal{U}$.

7.2. Folded Optimization: Practical Considerations

The Section is concluded with a discussion of some practical aspects when using folded optimization. Here, emphasis is given to the potential pitfalls

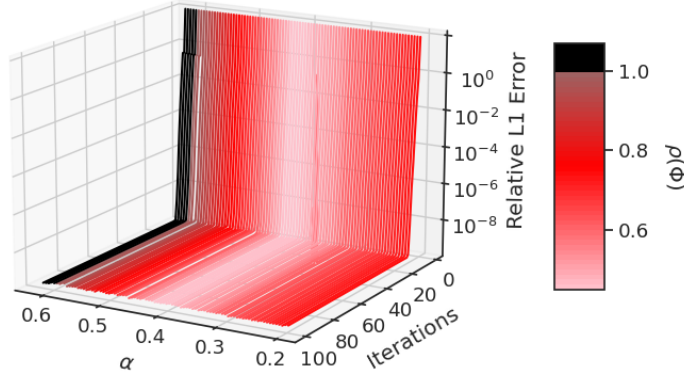


Figure 5: An expanded view of Figure 3’s third row shows backward-pass convergence for fixed-point folding of PGD by GMRes, compared to stepsize α and spectral radius Φ (color scale) on CIFAR100 Multilabel Classification. Because GMRes does not depend on iterating a contractive mapping with low spectral radius, convergence rates are unaffected by the stepsize of PGD used to backpropagate gradients.

of unrolling and unfolding optimization, which are addressed in the folded optimization system.

Blackbox Optimization. One of the primary benefits of folded optimization is the ability to leverage blackbox optimization solvers to compute the forward-pass mapping $\mathbf{c} \rightarrow \mathbf{x}^*(\mathbf{c})$. The ability to accommodate blackbox solvers is an important efficiency advantage that is precluded by unrolled optimization, since it requires the solver to be implemented in an AD environment. Most practical applications of optimization rely on highly optimized software implementations such as Gurobi [34], which can incorporate problem-specific handcrafted heuristics as well as low-level code optimizations to minimize solving time. This is also a major advantage over the existing differentiable optimization library `cvxpy`, which requires converting the problem to a convex cone program before solving it with a specialized operator-splitting method for conic programming [8], rendering it inefficient for many optimization problems.

Parameter Selection. Optimization methods typically require specification of parameters such as gradient stepsizes, which can be chosen as constants

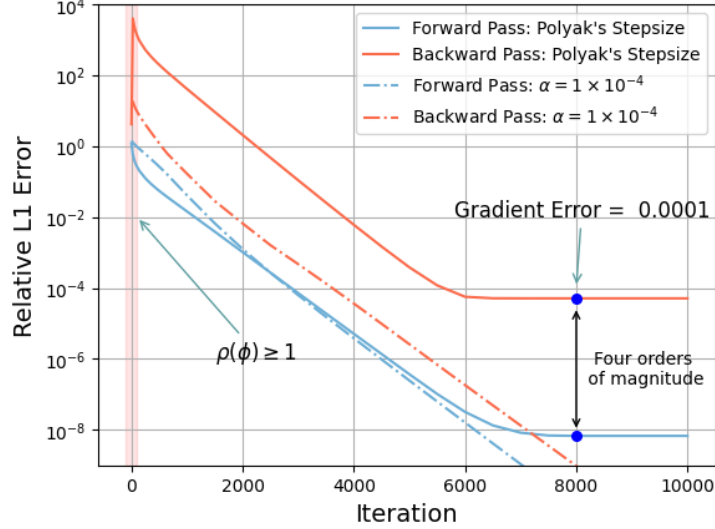


Figure 6: Impact of Polyak’s Stepsize Rule on forward and backward convergence when unfolding PGD on CIFAR100 Multilabel Classification. Convergence in the forward pass is guaranteed, but relies on decaying the stepsize asymptotically to zero, which causes failure to converge in the backward pass.

or adaptively at each iteration. Even when such parameters can be well-chosen for forward-pass convergence, the same values may not perform well for backward-pass convergence in unfolded optimization. This potential hazard of unfolding is illustrated in Figure 6, which again shares the illustrative example of PGD on multiclass selection. Here Polyak’s adaptive stepsize rule is used, which guarantees convergence of PGD [29]. However, since Polyak’s rule decays the stepsize to zero, convergence of the backward pass slows over time, causing it to flatline at four orders of magnitude in error behind the forward pass. The equivalent result due to a constant stepsize (in dotted curves) serves to show how a constant, finite stepsize leads to much more efficient backward-pass convergence. This highlights the importance of separating the forward and backward-pass models in fixed-point folding, so that convergence of both passes can be ensured.

Monitoring Backward Convergence. Error tolerance thresholds are often used to terminate optimization methods when sufficient accuracy is reached. In typical algorithm unrolling, it is not possible to monitor the backward pass for termination by early stopping, since it is fully determined by the forward

pass.

Computational Graph Size. Since folded optimization requires the computational graph of $\mathcal{U}$ for only a single iteration at the fixed point $\mathbf{x}^*$, its time and space efficiency are potentially much higher than that of the typical unrolled optimization, which stores a chain of computational graphs through $\mathcal{U}$ from an arbitrary starting point to the optimal solution at convergence.

Nested Fixed-Point Folding. As noted in Section 5, an optimization method may require in its update step $\mathcal{U}$ the solution of an optimization subproblem; when the subproblem itself requires an iterative method, this leads to nested unrolled loops. It is important to note that as per Definition 2, the innermost optimization loop of a nested unrolling can be considered an unfolding and can be converted to fixed-point folding using the methods of this section. Subsequently, the next outermost loop can now be considered unfolded, and the same process applied until all unrolled loops are replaced solution of their respective analytical models. The process is exemplified by *f-PGDb* (introduced in Section 8), which applies successive fixed-point folding through ADMM for and PGD (described in Section 5) to compose a JgP-based differentiable layer for any optimization problem with a smooth objective function and linear constraints. In particular, quadratic programming by ADMM is used to define $\mathcal{U}$ for a differentiable projection onto linear constraints, resulting in unfolded PGD. Then, fixed-point folding is applied again to replace the unfolded PGD loop. This *f-PGDb* module is used to backpropagate nonconvex quadratic programming, and neither ADMM nor PGD is used to compute the optimal solution in the forward pass. For this, Gurobi solver is used as a black box optimization oracle.

8. Experiments

This Section evaluates folded optimization on five different 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 (see Section 4). 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 blackbox algorithm to produce $\mathbf{x}^*(\mathbf{c})$. In each experiment below, this is used to demonstrate a different advantage.

The experiments test four differentiable optimization modules implemented in `fold-opt`: (1) *f-PGD_a* 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` [6]. (2) *f-PGD_b* is a variation on the former, in which the inner QP step is differentiated by fixed-point folding of the ADMM solver specified in [A.3]. (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 A. Its inner optimization step [O] is a soft thresholding Prox operator, whose simple closed form is differentiated by AD in PyTorch.

Decision-focused Learning Setting. The first three tasks of this Section follow the problem setting known as *Decision-focused Learning*, or *Predict-Then-Optimize*. Here, an optimization problem [I] has unknown coefficients only in its objective function $f(\mathbf{x}, \mathbf{c})$ while the constraints are considered constant. The goal of the supervised learning task is to predict $\hat{\mathbf{c}}$ from feature data such that the resulting $\mathbf{x}^*(\hat{\mathbf{c}})$ optimizes the objective under ground-truth parameters $\bar{\mathbf{c}}$, which is $f(\mathbf{x}^*(\hat{\mathbf{c}}), \bar{\mathbf{c}})$. This is equivalent to minimizing the *regret* loss function:

$$\text{regret}(\hat{\mathbf{c}}, \bar{\mathbf{c}}) = f(\mathbf{x}^*(\hat{\mathbf{c}}), \bar{\mathbf{c}}) - f(\mathbf{x}^*(\bar{\mathbf{c}}), \bar{\mathbf{c}}), \quad (30)$$

which measures the suboptimality, under ground-truth objective data, of decisions $\mathbf{x}^*(\hat{\mathbf{c}})$ resulting from prediction $\hat{\mathbf{c}}$. Since the task amounts to predicting $\hat{\mathbf{c}}$ under ground-truth $\bar{\mathbf{c}}$, a *two-stage* approach is also available which does not require backpropagation through $\mathbf{x}^*$. In the two-stage approach, the loss function $\text{MSE}(\hat{\mathbf{c}}, \bar{\mathbf{c}})$ is used to directly target ground-truth parameters, but the final test criteria is still measured by regret. Since the integrated training minimizes regret directly, it generally outperforms the two-stage.

8.1. 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

$$\mathbf{x}^*(\mathbf{c}, \mathbf{d}) = \underset{\mathbf{0} \leq \mathbf{x}, \mathbf{y} \leq \mathbf{1}}{\operatorname{argmax}} \quad \mathbf{c}^T \mathbf{x} + \mathbf{x}^T \mathbf{Q} \mathbf{y} + \mathbf{d}^T \mathbf{y} \quad (31a)$$

$$\text{s.t.} \quad \sum \mathbf{x} = p, \quad \sum \mathbf{y} = q, \quad (31b)$$

in which two separable linear programs are confounded by a nonconvex quadratic objective term $\mathbf{Q}$. Costs $\mathbf{c}$ and $\mathbf{d}$ are unknown, while p and q are constants. Such programs have numerous industrial applications such as optimal mixing and pooling in gas refining [35]. Here we focus on the difficulty posed by the problem’s form in decision-focused learning, and propose a task in which the unknown parameters $\mathbf{c}$ and $\mathbf{d}$ are correlated with known feature variables and predicted by a 5-layer network. The goal is to predict $\hat{\mathbf{c}}$ and $\hat{\mathbf{d}}$ from features, such that the suboptimality of $\mathbf{x}^*(\hat{\mathbf{c}}, \hat{\mathbf{d}})$ with respect to ground-truth $\mathbf{c}$ and $\mathbf{d}$ is minimized.

It is known that PGD converges to local optima in nonconvex problems [36], therefore the *f-PGDb* module specified above is chosen used to back-propagate the solution of [31] in end-to-end training. Since PGD is not an efficient method for solving the forward-pass mapping (31), the *fold-opt* implementation of this layer uses the Gurobi nonconvex QP solver to find its global optimum. 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. In contrast, the integrated *f-PGDb* layer allows the model to minimize solution regret (i.e., suboptimality) directly as its loss function.

Feature and cost data are generated by the process described in Appendix B. In addition, 15 distinct non-positive semidefinite $\mathbf{Q}$ are randomly generated so that the results of Figure 7(a) are reported on average over all 15 nonconvex decision-focused learning tasks. Notice in Figure 7 how *f-PGDb* achieves much lower regret for each of the 15 nonconvex objectives.

8.2. Cost Prediction for AC-Optimal Power Flow.

The *AC-Optimal Power Flow* (AC-OPF) problem minimizes the cost of generator dispatch that satisfies the power system’s physical and engineering constraints, as shown in Figure 1. In this learning task, the linear and quadratic power generation costs $\mathbf{c}^l$ and $\mathbf{c}^q$ are unknown and must be inferred from known features, such that the overall price of power generation under ground-truth costs is minimized. All other parameters of the optimization

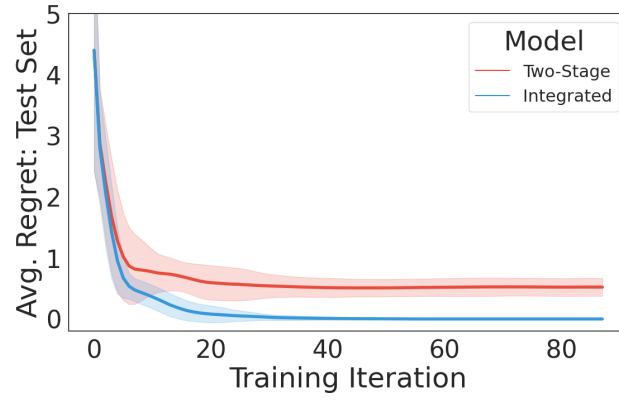


Figure 7: Learning Cost Factors in Bilinear Programming.

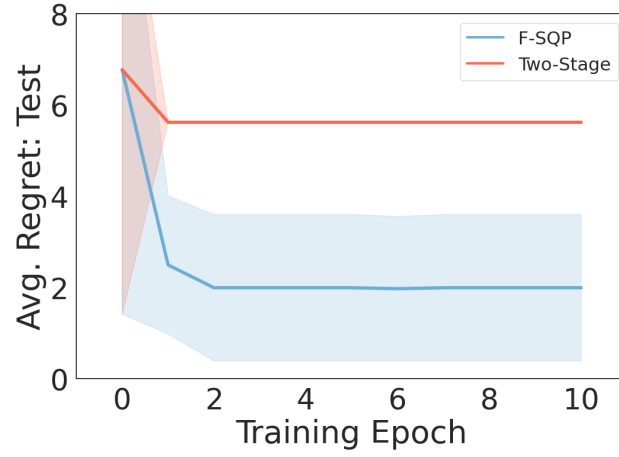


Figure 8: Learning cost coefficients of the AC-OPF problem.

$$\mathcal{O}(\mathbf{c}^l, \mathbf{c}^q) = \operatorname{argmin}_{\mathbf{p}^g, \mathbf{v}} \mathbf{c}^l \cdot \mathbf{p}^g + \mathbf{c}^q \cdot (\mathbf{p}^g)^2 \quad (32)$$

subject to:

$$\dot{v}_i^{\min} \leq v_i \leq \dot{v}_i^{\max} \quad \forall i \in \mathcal{N} \quad (2a)$$

$$-\dot{\theta}_{ij}^\Delta \leq \theta_i - \theta_j \leq \dot{\theta}_{ij}^\Delta \quad \forall (ij) \in \mathcal{E} \quad (2b)$$

$$\dot{p}_i^{g \min} \leq p_i^g \leq \dot{p}_i^{g \max} \quad \forall i \in \mathcal{N} \quad (3a)$$

$$\dot{q}_i^{g \min} \leq q_i^g \leq \dot{q}_i^{g \max} \quad \forall i \in \mathcal{N} \quad (3b)$$

$$(p_{ij}^f)^2 + (q_{ij}^f)^2 \leq \dot{S}_{ij}^{f \max} \quad \forall (ij) \in \mathcal{E} \quad (4)$$

$$p_{ij}^f = \dot{g}_{ij} v_i^2 - v_i v_j (\dot{b}_{ij} \sin(\theta_i - \theta_j) + \dot{g}_{ij} \cos(\theta_i - \theta_j)) \quad \forall (ij) \in \mathcal{E} \quad (5a)$$

$$q_{ij}^f = -\dot{b}_{ij} v_i^2 - v_i v_j (\dot{g}_{ij} \sin(\theta_i - \theta_j) - \dot{b}_{ij} \cos(\theta_i - \theta_j)) \quad \forall (ij) \in \mathcal{E} \quad (5b)$$

$$p_i^g - \dot{p}_i^d = \sum_{(ij) \in \mathcal{E}} p_{ij}^f \quad \forall i \in \mathcal{N} \quad (6a)$$

$$q_i^g - \dot{q}_i^d = \sum_{(ij) \in \mathcal{E}} q_{ij}^f \quad \forall i \in \mathcal{N} \quad (6b)$$

Model 1: AC Optimal Power Flow (AC-OPF)

problem are held constant and obtained from the NESTA energy system test case ACOPF-57 [37].

A five-layer network is used to predict both sets of cost coefficients in Equation 32. Input features are composed of the previous day's temperature, current temperature, and a constant baseline cost vector. Given the predicted cost coefficients, non-convex non-linear solver Interior Point Optimizer, `ipopt`, is used to compute the optimal solution to 32. Given that PGD cannot handle non-linear constraints, the *f-SQP* module is used solve for the backpropagated gradients for training by stochastic gradient descent. Figure 8 shows the relative regret on the test set after each training epoch, compared to a basic two-stage model.

8.3. Portfolio Prediction and Optimization.

A classic problem which combines prediction with optimization is the Markowitz portfolio problem [38]. Here, an investment portfolio must be partitioned to optimize total future return subject to risk constraints, while future asset prices are unknown and must be predicted. This experiment represents a situation in which `cvxpy` makes non negligible errors in the

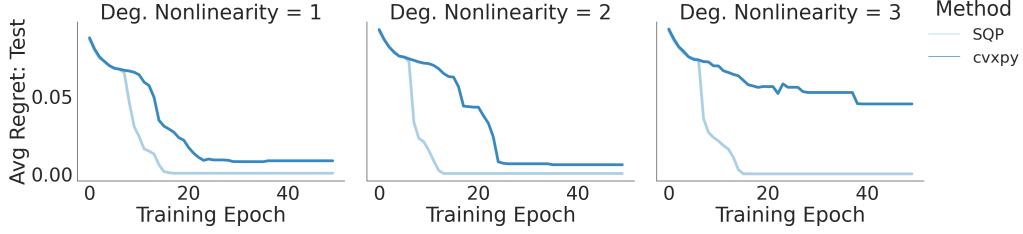


Figure 9: Learning asset prices for portfolio optimization.

forward pass of a problem with nonlinear constraints:

$$\mathbf{x}^*(\mathbf{c}) = \underset{\mathbf{0} \leq \mathbf{x}}{\operatorname{argmax}} \mathbf{c}^T \mathbf{x} \quad \text{s.t.} \quad \mathbf{x}^T \mathbf{V} \mathbf{x} \leq \gamma, \quad \sum \mathbf{x} = 1. \quad (33)$$

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 [28]. 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 (33). 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 9 shows test set regret throughout training, on three synthetically generated datasets of different nonlinearity degrees, following exactly the experimental settings of [28]. 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; this may be due to ill-conditioning of the quadratic constraint in (33). 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.

8.4. Enhanced Total Variation Denoising.

A classic application of proximal optimization models a denoising problem

$$\mathbf{x}^*(\mathbf{d}, \mathbf{D}) = \underset{\mathbf{x}}{\operatorname{argmin}} \quad \frac{1}{2} \|\mathbf{x} - \mathbf{d}\|^2 + \lambda \|\mathbf{D}\mathbf{x}\|_1, \quad (34)$$

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 [29]. Typically, $\mathbf{D}$ is a pairwise differencing matrix so that $\|\mathbf{D}\mathbf{x}\|_1$ represents total variation. The objective function, which balances a combination of distance to the input signal $\mathbf{d}$ with a penalty on variation, aims to find $\mathbf{x}^*$ which removes extraneous noise from $\mathbf{d}$. Here we initialize $\mathbf{D}$ to the classic differencing matrix and *learn* a better operator by treating $\mathbf{D}$ as a learnable parameter.

Training data follows the experimental settings of [6], in which a set of 1D signals is treated as target data and then perturbed by Gaussian noise to generate their corresponding noisy input data $\mathbf{d}$. MSE loss is used to target the true signals while $\mathbf{D}$ is learned. Figure 10(a) shows MSE on the test set throughout training due to *f-FDPG* for various choices of λ . Figure 10(b) shows comparable results from the differentiable QP framework of [6], which converts the problem (34) to an equivalent QP problem:

$$\mathbf{x}^*(\mathbf{D}) = \underset{\mathbf{x}, \mathbf{t}}{\operatorname{argmin}} \quad \frac{1}{2} \|\mathbf{x} - \mathbf{d}\|^2 + \lambda \vec{\mathbf{1}}^T \mathbf{t} \quad (35a)$$

$$s.t. \quad \mathbf{D}\mathbf{x} \leq \mathbf{t} \quad (35b)$$

$$-\mathbf{t} \leq \mathbf{D}\mathbf{x} \quad (35c)$$

in order to differentially solve the denoising problem in `qpth`. Small differences in these results likely stem from solver error tolerance in the forward pass of each method. However, *f-FDPG* computes $\mathbf{x}^*(\mathbf{D})$ up to 40 times faster, by using an optimization method (A.1) which is well-chosen for efficiently solving the denoising problem in its original form (34).

8.5. Multilabel Classification on CIFAR100.

Since the differential fixed-point conditions (DFP) depend on the chosen optimization method (U), we compare the effect of different backpropagation rules in `fold-opt`, based on alternative choices of (U). This experiment compares the backpropagation of both *f-PGD* and *f-SQP* with that of `cvxpy`, since both PGD and SQP methods are suitable for solving the optimization (36). Importantly, each `fold-opt` layer uses the same forward pass, implemented in `cvxpy`. This allows any potential discrepancies in empirical results to be attributed to differences in the backpropagation model.

The experimental task, adapted from [39], learns a smooth top-5 classification model on noisy CIFAR-100. The optimization below maps image

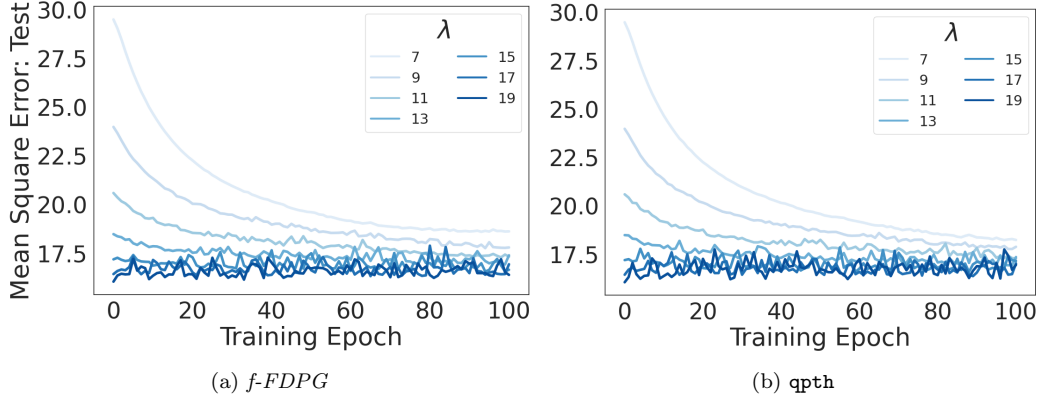


Figure 10: Enhanced Denoising Task: Test Set Loss

feature embeddings $\mathbf{c}$ from DenseNet 40-40 [40], to smoothed top- k binary class indicators (see Appendix B for more details):

$$\mathbf{x}^*(\mathbf{c}) = \underset{\mathbf{0} \leq \mathbf{x} \leq \mathbf{1}}{\operatorname{argmax}} \quad \mathbf{c}^T \mathbf{x} + \sum_i x_i \log x_i \quad s.t. \quad \sum \mathbf{x} = k \quad (36)$$

Figure 11 shows that all three models have indistinguishable classification accuracy throughout training, even after 30 epochs of training on 45k samples. On the other hand, the more sensitive test set shows marginal accuracy divergence between all three methods after a few epochs. This corresponds with a slightly less consistent increase in accuracy throughout training, in which none of the methods holds a clear advantage.

9. Conclusions

This paper introduced folded optimization, a framework for generating efficient and analytically differentiable optimization solvers from unrolled implementations. Theoretically, folded optimization was justified by a novel analysis of unrolled optimization 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. This allowed for the convergence analysis of the backward pass, and evidence that the convergence could be improved by using superior linear system solvers. The paper showed that folded optimization offers substantial advantages over

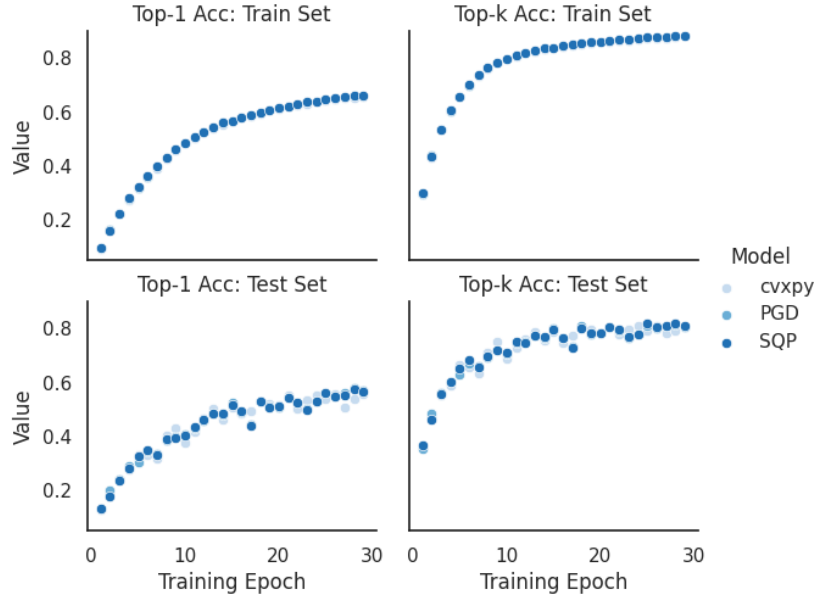


Figure 11: Test and train set accuracy while training multilabel classification on CIFAR-100.

existing both unrolled optimization and 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 grants 2345528, 2334936, 2334448 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

- [1] J. Kotary, F. Fioretto, P. Van Hentenryck, B. Wilder, [End-to-end constrained optimization learning: A survey](#), in: Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21, 2021, pp. 4475–4482. [doi:10.24963/ijcai.2021/610](#). URL <https://doi.org/10.24963/ijcai.2021/610>

- [2] A. Martins, R. Astudillo, From softmax to sparsemax: A sparse model of attention and multi-label classification, in: International conference on machine learning, PMLR, 2016, pp. 1614–1623.
- [3] R. P. Adams, R. S. Zemel, Ranking via sinkhorn propagation, arXiv preprint arXiv:1106.1925 (2011).
- [4] J. Kotary, F. Fioretto, P. Van Hentenryck, Z. Zhu, End-to-end learning for fair ranking systems, in: Proceedings of the ACM Web Conference 2022, 2022, pp. 3520–3530.
- [5] B. Wilder, B. Dilkina, M. Tambe, Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization, in: AAAI, Vol. 33, 2019, pp. 1658–1665.
- [6] B. Amos, J. Z. Kolter, Optnet: Differentiable optimization as a layer in neural networks, in: International Conference on Machine Learning, PMLR, 2017, pp. 136–145.
- [7] V. Monga, Y. Li, Y. C. Eldar, Algorithm unrolling: Interpretable, efficient deep learning for signal and image processing, IEEE Signal Processing Magazine 38 (2) (2021) 18–44.
- [8] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, J. Z. Kolter, Differentiable convex optimization layers, Advances in neural information processing systems 32 (2019).
- [9] J. Kotary, M. H. Dinh, F. Fioretto, Backpropagation of unrolled solvers with folded optimization, arXiv preprint arXiv:2301.12047 (2023).
- [10] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in pytorch, in: NIPS-W, 2017, p. 1.
- [11] J. Domke, Generic methods for optimization-based modeling, in: Artificial Intelligence and Statistics, PMLR, 2012, pp. 318–326.
- [12] N. Shlezinger, Y. C. Eldar, S. P. Boyd, Model-based deep learning: On the intersection of deep learning and optimization, arXiv preprint arXiv:2205.02640 (2022).

- [13] S. Gould, B. Fernando, A. Cherian, P. Anderson, R. S. Cruz, E. Guo, On differentiating parameterized argmin and argmax problems with application to bi-level optimization, arXiv preprint arXiv:1607.05447 (2016).
- [14] B. Amos, V. Koltun, J. Z. Kolter, The limited multi-label projection layer, arXiv preprint arXiv:1906.08707 (2019).
- [15] T. Konishi, T. Fukunaga, End-to-end learning for prediction and optimization with gradient boosting, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2021, pp. 191–207.
- [16] P. Donti, B. Amos, J. Z. Kolter, Task-based end-to-end model learning in stochastic optimization, in: NIPS, 2017, pp. 5484–5494.
- [17] M. Blondel, O. Teboul, Q. Berthet, J. Djolonga, Fast differentiable sorting and ranking, in: International Conference on Machine Learning, PMLR, 2020, pp. 950–959.
- [18] J. Mandi, T. Guns, Interior point solving for lp-based prediction+optimisation, in: Advances in Neural Information Processing Systems (NeurIPS), 2020, p. 1.
- [19] A. Agrawal, S. Barratt, S. Boyd, E. Busseti, W. M. Moursi, Differentiating through a cone program, arXiv preprint arXiv:1904.09043 (2019).
- [20] A. Nemirovski, Advances in convex optimization: conic programming, in: International Congress of Mathematicians, Vol. 1, 2007, pp. 413–444.
- [21] E. Busseti, W. M. Moursi, S. Boyd, Solution refinement at regular points of conic problems, Computational Optimization and Applications 74 (3) (2019) 627–643.
- [22] M. C. Grant, S. P. Boyd, Graph implementations for nonsmooth convex programs, in: Recent advances in learning and control, Springer, 2008, pp. 95–110.
- [23] A. Ferber, B. Wilder, B. Dilkina, M. Tambe, Mipaal: Mixed integer program as a layer, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 34, 2020, pp. 1504–1511.

- [24] Q. Berthet, M. Blondel, O. Teboul, M. Cuturi, J.-P. Vert, F. Bach, Learning with differentiable perturbed optimizers, *Advances in neural information processing systems* 33 (2020) 9508–9519.
- [25] M. Paulus, D. Choi, D. Tarlow, A. Krause, C. J. Maddison, Gradient estimation with stochastic softmax tricks, *Advances in Neural Information Processing Systems* 33 (2020) 5691–5704.
- [26] M. V. Pogančić, A. Paulus, V. Musil, G. Martius, M. Rolinek, Differentiation of blackbox combinatorial solvers, in: *International Conference on Learning Representations*, 2019, p. 1.
- [27] S. Sekhar Sahoo, M. Vlastelica, A. Paulus, V. Musil, V. Kuleshov, G. Martius, Gradient backpropagation through combinatorial algorithms: Identity with projection works, *arXiv e-prints* (2022) arXiv–2205.
- [28] A. N. Elmachtoub, P. Grigas, Smart “predict, then optimize”, *Management Science* (2021).
- [29] A. Beck, *First-order methods in optimization*, SIAM, 2017.
- [30] 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) (2011) 1–122.
- [31] A. Quarteroni, R. Sacco, F. Saleri, *Numerical mathematics*, Vol. 37, Springer Science & Business Media, 2010.
- [32] J. R. Munkres, *Analysis on manifolds*, CRC Press, 2018.
- [33] T. Sauer, *Numerical analysis*, Addison-Wesley Publishing Company, 2011.
- [34] Gurobi Optimization, LLC, [Gurobi Optimizer Reference Manual](https://www.gurobi.com) (2023). URL <https://www.gurobi.com>
- [35] C. Audet, J. Brimberg, P. Hansen, S. L. Digabel, N. Mladenović, Pooling problem: Alternate formulations and solution methods, *Management science* 50 (6) (2004) 761–776.

- [36] H. Attouch, J. Bolte, 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) (2013) 91–129.
- [37] C. Coffrin, D. Gordon, P. Scott, Nesta, the nicta energy system test case archive, *arXiv preprint arXiv:1411.0359* (2014).
- [38] Y. Zhang, X. Li, S. Guo, Portfolio selection problems with markowitz’s mean–variance framework: a review of literature, *Fuzzy Optimization and Decision Making* 17 (2018) 125–158.
- [39] L. Berrada, A. Zisserman, M. P. Kumar, Smooth loss functions for deep top-k classification, *ArXiv abs/1802.07595* (2018).
- [40] G. Huang, Z. Liu, L. Van Der Maaten, K. Q. Weinberger, Densely connected convolutional networks, in: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

Appendix A. Optimization Models

Soft Thresholding Operator. The soft thresholding operator defined below arises in the solution of denoising problems proximal gradient descent variants as the proximal operator to the $\|\cdot\|_1$ norm:

$$\mathcal{T}_\lambda(\mathbf{x}) = [|\mathbf{x}| - \lambda \mathbf{e}]_+ \cdot \text{sgn}(\mathbf{x})$$

Fast Dual Proximal Gradient Descent. The following is an FDPG implementation from [29], specialized to solve the denoising problem

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

of Section 8. Letting $\mathbf{u}_k$ be the primal solution iterates, with $t_0 = 1$ and arbitrary $\mathbf{w}_0 = \mathbf{y}_0$:

$$\mathbf{u}_k = \mathbf{D}^T \mathbf{w}_k + \mathbf{d} \tag{A.1a}$$

$$\mathbf{y}_{k+1} = \mathbf{w}_k - \frac{1}{4} \mathbf{D} \mathbf{u}_k + \frac{1}{4} \mathcal{T}_{4\lambda}(\mathbf{D} \mathbf{u}_k - 4 \mathbf{w}_k) \tag{A.1b}$$

$$t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2} \tag{A.1c}$$

$$\mathbf{w}_{k+1} = \mathbf{y}_{k+1} + \left(\frac{t_k - 1}{t_{k+1}} \right) (\mathbf{y}_{k+1} - \mathbf{y}_k) \tag{A.1d}$$

Quadratic Programming by ADMM. A Quadratic Program is an optimization problem with convex quadratic objective and linear constraints. The following ADMM scheme of [30] solves any quadratic programming problem of the standard form:

$$\underset{\mathbf{x}}{\operatorname{argmax}} \quad \frac{1}{2} \mathbf{x}^T \mathbf{Q} \mathbf{x} + \mathbf{p}^T \mathbf{x} \tag{A.2a}$$

$$s.t. \quad \mathbf{A} \mathbf{x} = \mathbf{b} \tag{A.2b}$$

$$\mathbf{x} \geq \mathbf{0} \tag{A.2c}$$

by declaring the operator splitting

$$\underset{\mathbf{x}}{\operatorname{argmax}} \quad f(\mathbf{x}) + g(\mathbf{z}) \tag{A.3a}$$

$$s.t. \quad \mathbf{x} = \mathbf{z} \tag{A.3b}$$

with $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{p}^T\mathbf{x}$, $\text{dom}(f) = \{\mathbf{x} : \mathbf{A}\mathbf{x} = \mathbf{b}\}$, $g(\mathbf{x}) = \delta(\mathbf{x} \geq 0)$ and where δ is the indicator function.

This results in the following ADMM iterates:

1. Solve
$$\begin{bmatrix} \mathbf{P} + \rho\mathbf{I} & \mathbf{A}^T \\ \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{k+1} \\ \boldsymbol{\nu} \end{bmatrix} = \begin{bmatrix} -\mathbf{q} + \rho(\mathbf{z}_k - \mathbf{u}_k) \\ \mathbf{b} \end{bmatrix}$$
2. $\mathbf{z}_{k+1} = (\mathbf{x}_{k+1} + \mathbf{u}_k)_+$
3. $\mathbf{u}_{k+1} = \mathbf{u}_k + \mathbf{x}_{k+1} - \mathbf{z}_{k+1}$

Where (1) represents the KKT conditions for equality-constrained minimization of f , (2) is projection onto the positive orthant, and (3) is the dual variable update.

Sequential Quadratic Programming. For an optimization mapping defined by Problem [\(1\)](#) where f , g and h are continuously differentiable, define the operator $\mathcal{T}$ as:

$$\mathcal{T}(\mathbf{x}, \boldsymbol{\lambda}) = \underset{\mathbf{d}}{\text{argmin}} \quad \nabla f(\mathbf{x})^T \mathbf{d} + \mathbf{d}^T \nabla^2 \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) \mathbf{d} \quad (\text{A.4a})$$

$$\text{s.t.} \quad h(\mathbf{x}) + \nabla h(\mathbf{x})^T \mathbf{d} = 0 \quad (\text{A.4b})$$

$$g(\mathbf{x}) + \nabla g(\mathbf{x})^T \mathbf{d} \leq 0 \quad (\text{A.4c})$$

where dependence of each function on parameters $\mathbf{c}$ is hidden. The function $\mathcal{L}$ is a Lagrangian function of Problem [\(1\)](#). Then given initial estimates of the primal and dual solution (x_0, λ_0) , sequential quadratic programming is defined by

$$(\mathbf{d}, \boldsymbol{\mu}) = \mathcal{T}(\mathbf{x}_k, \boldsymbol{\lambda}_k) \quad (\text{A.5a})$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d} \quad (\text{A.5b})$$

$$\boldsymbol{\lambda}_{k+1} = \alpha_k (\boldsymbol{\mu} - \boldsymbol{\lambda}_k) \quad (\text{A.5c})$$

Here, the inner optimization $\mathcal{O} = \mathcal{T}$ as in Section [5](#).

Appendix B. Experimental Details

Additional details for each experiment of Section [8](#) are described in their respective subsections below. Note that in all cases, the machine learning models compared in Section [8](#) use identical settings within each study, with the exception of the optimization components being compared.

Appendix B.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 $\mathbf{c}$ and $\mathbf{d}$. We train model with Adam optimizer on learning rate of 10^{-2} and batch size 32 for 5 epochs. Nonconvex objective coefficients $\mathbf{Q}$ 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 $\mathbf{c}$ and $\mathbf{d}$. 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.

Appendix B.2. AC-Optimal Power Flow

Data Generation. A 57-node power system is used to generate our dataset. Specifications of generators, demand loads, and buses are adapted directly from the NESTA energy system test case [37]. Cost coefficients are randomly perturbed from the original generator costs and altered by a non-linear function of the temperature variations $x \times (1 + \frac{|t_{previous} - t_{current}|}{100})$. Temperature variations are represented by the previous day temperature $t_{previous}$, sampled uniformly in the interval $[20, 110]$, and the current day temperature $t_{current}$ which is computed by sampling the change in temperature normally with a mean of 0 and variation of 20. The demand loads are also modified by a non-linear function of the temperature variations $x + |\frac{t_{current} - 65}{45}|$. To train the model, 1000 points were sampled to create the dataset with a 90/10 train/test split.

Settings. A five-layer ReLU network with hidden layer size 64 is trained to predict generator costs $\mathbf{c} \in \mathbb{R}^{7 \times 3}$ using SGD optimizer with learning rate 10^{-2} and batch size 32.

Appendix B.3. Portfolio Optimization

Data Generation. The data generation follows exactly the prescription of Appendix D in [28]. 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 non-linearity is introduced by exponentiation of its elements to a chosen degree. The studies in Section 8 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.

Appendix B.4. 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 $\mathbf{D} \in \mathbb{R}^{99 \times 100}$ to the differencing operator, for which $D_{i,i} = 1$ and $D_{i,i+1} = -1 \ \forall i$ with all other values 0.

Appendix B.5. Multilabel Classification

Dataset. We follow the experimental settings and implementation provided by [39]. 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.