Efficient and Modular Implicit Differentiation

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Automatic differentiation (autodiff) has revolutionized machine learning. It allows expressing complex computations by composing elementary ones in creative ways and removes the burden of computing their derivatives by hand. More recently, differentiation of optimization problem solutions has attracted widespread attention with applications such as optimization as a layer, and in bi-level problems such as hyper-parameter optimization and meta-learning. However, the formulas for these derivatives often involve case-by-case tedious mathematical derivations. In this paper, we propose a unified, efficient and modular approach for implicit differentiation of optimization problems. In our approach, the user defines (in Python in the case of our implementation) a function F capturing the optimality conditions of the problem to be differentiated. Once this is done, we leverage autodiff of F and implicit differentiation to automatically differentiate the optimization problem. Our approach thus combines the benefits of implicit differentiation and autodiff. We show that seemingly simple principles allow to recover many recently proposed implicit differentiation methods and create new ones easily. We demonstrate the ease of formulating and solving bi-level optimization problems using our framework. We also showcase an application to the sensitivity analysis of molecular dynamics.

1 Introduction

Automatic differentiation (autodiff) is now an inherent part of machine learning software. It allows expressing complex computations by composing elementary ones in creative ways and removes the tedious burden of computing their derivatives by hand. The differentiation of optimization problem solutions has found many applications. A classical example is bi-level optimization, which typically involves computing the derivatives of a nested optimization problem in order to solve an outer one. Examples of applications in machine learning include hyper-parameter optimization [19, 63, 57, 30, 10, 11], neural networks [46], and meta-learning [31, 59]. Another line of applications is "optimization as a layer" [42, 6, 51, 27, 36, 7], which usually includes regularization or constraints in the optimization problem in order to impose desirable structure on the layer output. In addition to providing well sought-for interpretability, recent research indicates that such structure could be beneficial in order to improve generalization of neural networks [20].

Since optimization problem solutions typically do not enjoy an explicit formula in terms of their inputs, autodiff cannot be used directly to differentiate these functions. In recent years, two main approaches have been developed to circumvent this problem. The first one consists in unrolling the iterations of an optimization algorithm and to use the final iteration as a proxy for the optimization problem solution [68, 28, 25, 31]. An advantage of this approach is that autodiff through the algorithm iterates can then be used transparently. However, this requires a reimplementation of the algorithm using the autodiff system, and not all algorithms are necessarily autodiff friendly. Moreover, forward-mode autodiff has time complexity that scales linearly with the number of variables and reverse-mode autodiff has memory complexity that scales linearly with the number of algorithm iterations. A second

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

approach is to see optimization problem solutions as implicitly-defined functions of certain optimality conditions. Examples include stationary conditions [9, 46], KKT conditions [19, 35, 6, 53, 52] and the proximal gradient fixed point [51, 10, 11]. An advantage of such implicit differentiation is that a reimplementation is not needed, allowing to build upon state-of-the-art software. However, so far, obtaining the implicit differentiation formulas required a case-by-case tedious mathematical derivation. Recent work [2] attempts to address this issue by adding implicit differentiation on top of cvxpy [26]. This works by reducing all convex optimization problems to a conic program and using conic programming's optimality conditions to derive an implicit differentiation formula. While this approach is very generic, solving a convex optimization problem using a conic programming solver—an ADMM-based splitting conic solver [54] in the case of cvxpy—is rarely the state-of-the-art approach for each particular problem instance.

In this work, we adopt a different strategy, which allows to easily add implicit differentiation on top of existing solvers. In our approach, the user defines (in Python in the case of our implementation) a mapping function F capturing the optimality conditions of the problem solved by the algorithm. Once this is done, we leverage autodiff of F combined with implicit differentiation techniques to automatically differentiate the optimization problem solution. In this way, our approach is very generic and still the efficiency of state-of-the-art solvers. It therefore combines the benefits of implicit differentiation and autodiff. To summarize, we make the following contributions.

- We delineate extremely **general principles** for implicitly differentiating through an optimization problem solution. Our approach can be seen as "hybrid", in the sense that it combines implicit differentiation with autodiff of the optimality conditions.
- We show how to instantiate our framework in order to recover many recently-proposed implicit differentiation schemes, thereby providing a **unifying perspective**. We also obtain new implicit differentiation schemes, such as the one based on the mirror descent fixed point.
- On the theoretical side, we provide new bounds on the **Jacobian error** when the optimization problem is only solved approximately.
- We describe a **JAX implementation** and provide a blueprint for implementing our approach in other frameworks. We will open-source a full-fledged library for implicit differentiation in JAX.
- We implement four **illustrative applications**, demonstrating our framework's ease of use.

In essence, our implementation significantly extends JAX's default autodiff system in context of the numerical optimization domain. From an end user's perspective, autodiff with JAX simply becomes more efficient if they use solvers with implicit differentiation set up by our framework.

Notation. We denote the gradient and Hessian of $f\colon \mathbb{R}^d \to \mathbb{R}$ evaluated at $x \in \mathbb{R}^d$ by $\nabla f(x) \in \mathbb{R}^d$ and $\nabla^2 f(x) \in \mathbb{R}^{d\times d}$. We denote the Jacobian of $F\colon \mathbb{R}^d \to \mathbb{R}^p$ evaluated at $x \in \mathbb{R}^d$ by $\partial F(x) \in \mathbb{R}^{p\times d}$. When f or F have several arguments, we denote the gradient, Hessian and Jacobian in the i^{th} argument by ∇_i, ∇_i^2 and ∂_i, respectively. The standard probability simplex is denoted by $\triangle^d := \{x \in \mathbb{R}^d \colon \|x\|_1 = 1, x \geq 0\}$. For any set $\mathcal{C} \subset \mathbb{R}^d$, we denote by $I_{\mathcal{C}}$ the function $\mathbb{R}^d \to \mathbb{R} \cup \{+\infty\}$ where $I_{\mathcal{C}}(x) = 0$ if $x \in \mathcal{C}$, $I_{\mathcal{C}}(x) = +\infty$ otherwise. For a vector or matrix A, we note $\|A\|$ the Frobenius (or Euclidean) norm, and $\|A\|_{op}$ the operator norm.

2 Proposed framework: combining implicit differentiation and autodiff

2.1 General principles

Overview. Contrary to unrolling of algorithm iterations, implicit differentiation typically involves a manual, sometimes complicated, mathematical derivation. For instance, numerous works [19, 35, 6, 53, 52] use Karush–Kuhn–Tucker (KKT) conditions in order to relate a constrained optimization problem's solution to its inputs, and to manually derive a formula for its derivatives. The derivation and implementation is typically case-by-case.

In this work, we propose a general way to easily add implicit differentiation on top of existing solvers. In our approach, the user defines (in Python in the case of our implementation) a mapping function F capturing the optimality conditions of the problem solved by the algorithm. We provide reusable building blocks to easily express such F. Once this is done, we leverage autodiff of F combined with

```python
X_tr, y_tr = load_data()

def f(x, theta):
  residual = jnp.dot(X_tr, x) - y_tr
  return (jnp.sum(residual ** 2) + theta * jnp.sum(x ** 2)) / 2

F = jax.grad(f)

@custom_root(F)
def ridge_solver(theta):
  XX = jnp.dot(X_tr.T, X_tr)
  Xy = jnp.dot(X_tr.T, y_tr)
  I = jnp.eye(X_tr.shape[0])
  return jnp.linalg.solve(XX + theta * I, Xy)

print(jax.jacobian(ridge_solver)(10.0))
```

Figure 1: Example: adding implicit differentiation on top of a ridge regression solver. The function $f(x, \theta)$ defines the objective function and the mapping F, here simply equation (4), captures the optimality conditions. The decorator `@custom_root` (provided by our library) automatically adds implicit differentiation to the solver for the user. The last line evaluates the Jacobian at $\theta = 10$.

implicit differentiation to automatically differentiate the optimization problem solution. A simple illustrative example is given in Figure 1.

Differentiating a root. Let $F\colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$ be a user-provided mapping, capturing the optimality conditions of a problem. An optimal solution, denoted $x^\star(\theta)$, should be a **root** of F:

$$F(x^\star(\theta), \theta) = 0\,. \tag{1}$$

We can see $x^\star(\theta)$ as an implicitly defined function of $\theta \in \mathbb{R}^n$, i.e., $x^\star\colon \mathbb{R}^n \to \mathbb{R}^d$. Our goal is to differentiate $x^\star(\theta)$ w.r.t. θ. From the **implicit function theorem** [44], if F is continuously differentiable and the Jacobian $\partial_1 F$ evaluated at $x^\star(\theta) \times \theta$ is a square invertible matrix, then $\partial x^\star(\theta)$ exists. Using the chain rule, we know that the Jacobian $\partial x^\star(\theta)$ satisfies

$$\partial_1 F(x^\star(\theta), \theta) \partial x^\star(\theta) + \partial_2 F(x^\star(\theta), \theta) = 0.$$

Computing $\partial x^\star(\theta)$ boils down to the resolution of the linear system of equations

$$\underbrace{-\partial_1 F(x^\star(\theta), \theta)}_{A \in \mathbb{R}^{d \times d}} \underbrace{\partial x^\star(\theta)}_{J \in \mathbb{R}^{d \times n}} = \underbrace{\partial_2 F(x^\star(\theta), \theta)}_{B \in \mathbb{R}^{d \times n}} \tag{2}$$

When (1) is a one-dimensional root finding problem ($d = 1$), (2) becomes particularly simple since $\nabla x^\star(\theta) = B^\top / A$, where A is a scalar value.

We will show that existing and new implicit differentiation methods all reduce to this simple principle. We call our approach hybrid, since it combines implicit differentiation (it involves the resolution of a linear system) with the autodiff of the optimality conditions F. Our approach is **efficient** as it can be added on top of any state-of-the-art solver and **modular** as the optimality conditon specification is decoupled from the implicit differentiation mechanism.

Differentiating a fixed point. We will encounter numerous applications where $x^\star(\theta)$ is implicitly defined through a **fixed point iteration**:

$$x^\star(\theta) = T(x^\star(\theta), \theta)\,,$$

where $T\colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$. This can be seen as a particular case of (1) with

$$F(x^\star(\theta), \theta) = T(x^\star(\theta), \theta) - x^\star(\theta)\,. \tag{3}$$

In this case, using the chain rule, we have

$$A = -\partial_1 F(x^\star(\theta), \theta) = I - \partial_1 T(x^\star(\theta), \theta) \quad \text{and} \quad B = \partial_2 F(x^\star(\theta), \theta) = \partial_2 T(x^\star(\theta), \theta).$$

Computing JVPs and VJPs. In practice, all we need to know from F is how to left-multiply or right-multiply $\partial_1 F$ and $\partial_2 F$ with a vector of appropriate size. These are called vector-Jacobian product (VJP) and Jacobian-vector product (JVP), and are useful for integrating $x^\star(\theta)$ with reverse-mode and forward-mode autodiff, respectively. Often times, F will be explicitly defined. In this case, computing the VJP or JVP can be done via autodiff. Other times, F may itself be implicitly defined, for instance when F involves the solution of a variational problem. In this case, computing the VJP or JVP will itself involve implicit differentiation.

The right-multiplication (JVP) between $J = \partial x^\star(\theta)$ and a vector v, Jv, can be computed efficiently by solving $A(Jv) = Bv$. The left-multiplication (VJP) of $v^\top$ with J, $v^\top J$, can be computed by first solving $A^\top u = v$. Then, we can obtain $v^\top J$ by $v^\top J = u^\top AJ = u^\top B$. Note that when B changes but A and v remain the same, we do not need to solve $A^\top u = v$ once again. This allows to compute the VJP w.r.t. different variables while solving only one linear system.

To solve these linear systems, we can use the conjugate gradient method [39] when A is positive semi-definite and GMRES [61] or BiCGSTAB [66] when A is not. All algorithms are matrix-free, i.e., they only require matrix-vector products (linear maps). Thus, all we need from F is its JVPs or VJPs. An alternative to GMRES/BiCGSTAB is to solve the normal equation $AA^\top u = Av$ using conjugate gradient, which we find faster in some scenarios.

Pre-processing and post-processing mappings. Often times, the goal is not to differentiate θ per se, but the parameters of a function producing θ. One example of such pre-processing is to convert the parameters to be differentiated from one form to another canonical form, such as a quadratic program [6] or a conic program [2]. Another example is when $x^\star(\theta)$ is used as the output of a neural network layer, in which case θ is produced by the previous layer. Likewise, $x^\star(\theta)$ will often not be the final output we want to differentiate. One example of such post-processing is when $x^\star(\theta)$ is the solution of a dual program and we apply the dual-primal mapping to recover the solution of the primal program. Another example is the application of a loss function, in order to reduce $x^\star(\theta)$ to a scalar value. In all these cases, we leave the differentiation of the pre/post-processing mappings to the autodiff system, allowing us to compose functions in complex ways.

Usage and implementation details. Our implementation is based on JAX [17, 34]. JAX's autodiff features enter the picture in at least two ways: (i) we lean heavily on JAX *within* our implementation, and (ii) we integrate the differentiation routines introduced by our framework *into* JAX's existing autodiff system. In doing the latter, we override JAX's default autodiff behavior (e.g. of differentiating transparently through an iterative solver's unrolled iterations).

We delineate here what features are needed from an autodiff system to implement our proposed framework. As mentioned, we only need access to F through the JVP or VJP of $\partial_1 F$ and $\partial_2 F$. Since the definition of F will often include a gradient mapping $\nabla_1 f(x, \theta)$ (see examples in §2.2), second-order derivatives need also be supported. Our library provides two decorators, `custom_root` and `custom_fixed_point`, for adding implicit differentiation on top of a solver, given optimality conditions F or fixed point iteration T. This functionality requires the ability to add custom JVP and/or VJP to a function. All these features are supported by recent autodiff systems, including JAX [17], TensorFlow [1] and PyTorch [56]. Our implementation, made in JAX, also uses JAX-specific features. We make extensive use of automatic batching with `jax.vmap`, JAX's vectorizing map transformation. In order to solve the normal equation $AA^\top u = Av$, we also use JAX's ability to automatically transpose a linear map using `jax.linear_transpose` [33].

2.2 Examples

We now give various examples of mapping F or fixed point iteration T, recovering existing implicit differentiation methods and creating new ones. Each choice of F or T implies different trade-offs in terms of computational **oracles**; see Table 1. Source code examples are given in Appendix A.

Stationary point condition. The simplest example is to differentiate through the implicit function

$$x^\star(\theta) = \operatorname*{argmin}_{x \in \mathbb{R}^d} f(x, \theta),$$

where $f \colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}$ is twice differentiable. In this case, F is simply the gradient mapping

$$F(x, \theta) = \nabla_1 f(x, \theta). \tag{4}$$

Table 1: Summary of optimality condition mappings. Oracles are accessed through their JVP or VJP.

Name	Equation	Solution needed	Oracles needed
Stationary	(4), (5)	Primal	$\nabla_1 f$
KKT	(6)	Primal *and* dual	$\nabla_1 f$, H, G, $\partial_1 H$, $\partial_1 G$
Proximal gradient	(7)	Primal	$\nabla_1 f$, $\text{prox}_{\eta g}$
Projected gradient	(9)	Primal	$\nabla_1 f$, $\text{proj}_{\mathcal{C}}$
Mirror descent	(11)	Primal	$\nabla_1 f$, $\text{proj}_{\mathcal{C}}^{\varphi}$, $\nabla \varphi$
Newton	(15)	Primal	$[\nabla_1^2 f(x,\theta)]^{-1}$, $\nabla_1 f(x,\theta)$
Block proximal gradient	(16)	Primal	$[\nabla_1 f]_j$, $[\text{prox}_{\eta g}]_j$
Conic programming	(20)	Residual map root	$\text{proj}_{\mathbb{R}^p \times \mathcal{K}^* \times \mathbb{R}_+}$

We then have $\partial_1 F(x,\theta) = \nabla_1^2 f(x,\theta)$ and $\partial_2 F(x,\theta) = \partial_2 \nabla_1 f(x,\theta)$, the Hessian of f in its first argument and the Jacobian in the second argument of $\nabla_1 f(x,\theta)$. In practice, we use autodiff to compute Jacobian products automatically. Equivalently, we can use the **gradient descent fixed point**

$$T(x,\theta) = x - \eta \nabla_1 f(x,\theta), \tag{5}$$

which holds for all step sizes $\eta > 0$. Using (3), it is easy to verify that we end up with the same linear system since η cancels out.

KKT conditions. We now show that the KKT conditions, manually differentiated in several works [19, 35, 6, 53, 52], fit our framework. As we will see, the key will be to group the optimal primal and dual variables as our $x^\star(\theta)$. Let us consider the general problem

$$\operatorname*{argmin}_{z \in \mathbb{R}^p} f(z,\theta) \quad \text{subject to} \quad G(z,\theta) \leq 0,\ H(z,\theta) = 0,$$

where $z \in \mathbb{R}^p$ is the primal variable, $f \colon \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}$, $G \colon \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^r$ and $H \colon \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^q$. The stationarity, primal feasibility and complementary slackness conditions give

$$\begin{aligned} \nabla_1 f(z,\theta) + [\partial_1 G(z,\theta)]^\top \lambda + [\partial_1 H(z,\theta)]^\top \nu &= 0 \\ H(z,\theta) &= 0 \\ \lambda \circ G(z,\theta) &= 0, \end{aligned} \tag{6}$$

where $\nu \in \mathbb{R}^q$ and $\lambda \in \mathbb{R}_+^r$ are the dual variables, also known as KKT multipliers. The system of (potentially nonlinear) equations (6) fits our framework, as we can group the primal and dual solutions as $x^\star(\theta) = (z^\star(\theta), \nu^\star(\theta), \lambda^\star(\theta))$ to form the root of a function $F(x^\star(\theta),\theta)$, where $F \colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$ and $d = p + q + r$. The primal and dual solutions can be obtained from a generic solver, such as an interior point method. In practice, the above mapping F will be defined directly in Python (see Figure 6 in Appendix A) and F will be differentiated automatically via autodiff.

Proximal gradient fixed point. Unfortunately, not all algorithms return both primal and dual solutions. Moreover, if the objective contains non-smooth terms, proximal gradient descent may be more efficient. We now discuss its fixed point. Let $x^\star(\theta)$ be implicitly defined as

$$x^\star(\theta) \coloneqq \operatorname*{argmin}_{x \in \mathbb{R}^d} f(x,\theta) + g(x,\theta),$$

where $f \colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}$ is twice-differentiable convex and $g \colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}$ is convex but possibly non-smooth. Let us define the proximity operator associated with g by

$$\text{prox}_g(y,\theta) \coloneqq \operatorname*{argmin}_{x \in \mathbb{R}^d} \frac{1}{2}\|x - y\|_2^2 + g(x,\theta).$$

To implicitly differentiate through $x^\star(\theta)$, we use the fixed point mapping [55, p.150]

$$T(x,\theta) = \text{prox}_{\eta g}(x - \eta \nabla_1 f(x,\theta), \theta), \tag{7}$$

for any step size $\eta > 0$. The proximity operator is 1-Lipschitz continuous [50]. By Rademacher's theorem, it is differentiable almost everywhere. Many proximity operators enjoy a closed form and can easily be differentiated, as discussed in Appendix B.

Projected gradient fixed point. As a special case, when $g(x, \theta)$ is the indicator function $I_{\mathcal{C}(\theta)}(x)$, where $\mathcal{C}(\theta)$ is a convex set depending on θ, we obtain

$$x^\star(\theta) = \underset{x \in \mathcal{C}(\theta)}{\operatorname{argmin}} f(x, \theta). \tag{8}$$

The proximity operator prox_g becomes the Euclidean projection onto $\mathcal{C}(\theta)$

$$\text{prox}_g(y, \theta) = \text{proj}_{\mathcal{C}}(y, \theta) \coloneqq \underset{x \in \mathcal{C}(\theta)}{\operatorname{argmin}} \|x - y\|_2^2$$

and (7) becomes the projected gradient fixed point

$$T(x, \theta) = \text{proj}_{\mathcal{C}}(x - \eta \nabla_1 f(x, \theta), \theta). \tag{9}$$

Compared to the KKT conditions, this fixed point is particularly suitable when the projection enjoys a closed form. We discuss how to compute the JVP / VJP for a wealth of convex sets in Appendix B.

Mirror descent fixed point. We again consider the case when $x^\star(\theta)$ is implicitly defined as the solution of (8). We now generalize the projected gradient fixed point beyond Euclidean geometry. Let the Bregman divergence $D_\varphi \colon \operatorname{dom}(\varphi) \times \operatorname{relint}(\operatorname{dom}(\varphi)) \to \mathbb{R}_+$ generated by φ be defined by

$$D_\varphi(x, y) \coloneqq \varphi(x) - \varphi(y) - \langle \nabla \varphi(y), x - y \rangle.$$

We define the Bregman projection of y onto $\mathcal{C}(\theta) \subseteq \operatorname{dom}(\varphi)$ by

$$\text{proj}_{\mathcal{C}}^{\varphi}(y, \theta) \coloneqq \underset{x \in \mathcal{C}(\theta)}{\operatorname{argmin}} D_\varphi(x, \nabla \varphi^*(y)). \tag{10}$$

Definition (10) incudes the mirror map $\nabla \varphi^*(y)$ for convenience. It can be seen as a mapping from $\mathbb{R}^d$ to $\operatorname{dom}(\varphi)$, ensuring that (10) is well-defined. The mirror descent fixed point mapping is then

$$\begin{aligned} \hat{x} &= \nabla \varphi(x) \\ y &= \hat{x} - \eta \nabla_1 f(x, \theta) \\ T(x, \theta) &= \text{proj}_{\mathcal{C}}^{\varphi}(y, \theta). \end{aligned} \tag{11}$$

Because T involves the composition of several functions, manually deriving its JVP/VJP is error prone. This shows that our approach leveraging autodiff allows to handle more advanced fixed point mappings. A common example of φ is $\varphi(x) = \langle x, \log x - \mathbf{1} \rangle$, where $\operatorname{dom}(\varphi) = \mathbb{R}^d_+$. In this case, D_φ is the Kullback-Leibler divergence. An advantage of the Kullback-Leibler projection is that it sometimes easier to compute than the Euclidean projection, as we detail in Appendix B.

Other fixed points. More fixed points are described in Appendix C.

2.3 Jacobian bounds

In practice, either by the limitations of finite precision arithmetic or because we perform a finite number of iterations, we rarely reach the exact solution $x^\star(\theta)$, but instead only reach an approximate solution $\hat{x}$ and apply the implicit differentiation equation (2) at this approximate solution. This motivates the need for approximation guarantees of this approach

Definition 1. *Let $F : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$ be an optimality criterion mapping. Let $A \coloneqq -\partial_1 F$ and $B \coloneqq \partial_2 F$. We define the* ***Jacobian estimate*** *at (x, θ) as the solution to the following linear equation $A(x, \theta) J(x, \theta) = B(x, \theta)$. It is a function $J : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^{d \times n}$.*

It holds by construction that $J(x^\star(\theta), \theta) = \partial x^\star(\theta)$. Computing $J(\hat{x}, \theta)$ for an approximate solution $\hat{x}$ of $x^\star(\theta)$ therefore allows to approximate the true Jacobian $\partial x^\star(\theta)$. In practice, an algorithm used to solve (1) depends on θ. Note however that, what we compute is not the Jacobian of $\hat{x}(\theta)$, unlike works unrolling an algorithm that differentiate through its iterations, but an estimate of $\partial x^\star(\theta)$. We therefore use the notation $\hat{x}$, leaving the dependence on θ implicit.

We develop bounds of the form $\|J(\hat{x}, \theta) - \partial x^\star(\theta)\| < C \|\hat{x} - x^\star(\theta)\|$, hence showing that the error on the estimated Jacobian is at most of the same order as that of $\hat{x}$ as an approximation of $x^\star(\theta)$. These bounds are based on the following main theorem, whose proof is included in Appendix D.

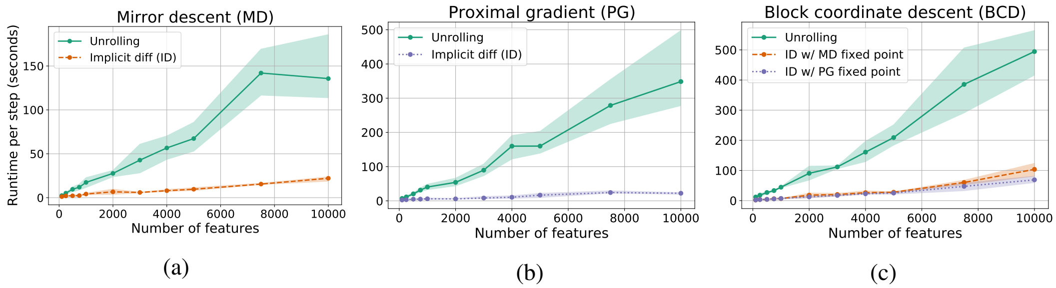

Figure 2: CPU runtime comparison of implicit differentiation and unrolling for hyperparameter optimization of multiclass SVMs for multiple problem sizes. Error bars represent 90% confidence intervals. **(a)** Mirror descent solver, with mirror descent fixed point for implicit differentiation. **(b)** Proximal gradient solver, with proximal gradient fixed point for implicit differentiation. **(c)** Block coordinate descent solver; for implicit differentiation we obtain $x^\star(\theta)$ by BCD but perform differentiation with the mirror descent and proximal gradient fixed points. This showcases that the solver and fixed point can be independently chosen.

Theorem 1 (Jacobian estimate)**.** *Let $F : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$. Assume that there exist $\alpha, \beta, \gamma, \varepsilon, R > 0$ such that $A = -\partial_1 F$ and $B = \partial_2 F$ satisfy, for all $v \in \mathbb{R}^d$, $\theta \in \mathbb{R}^n$ and x such that $\|x - x^\star(\theta)\| \leq \varepsilon$:*

A is well-conditioned, Lipschitz: $\|A(x,\theta)v\| \geq \alpha\|v\|$, $\|A(x,\theta) - A(x^\star(\theta),\theta)\|_{op} \leq \gamma\|x - x^\star(\theta)\|$.

B is bounded and Lipschitz: $\|B(x^\star(\theta),\theta)\| \leq R$, $\|B(x,\theta) - B(x^\star(\theta),\theta)\| \leq \beta\|x - x^\star(\theta)\|$.

Under these conditions, when $\|\hat{x} - x^\star(\theta)\| \leq \varepsilon$, we have

$$\|J(\hat{x},\theta) - \partial x^\star(\theta)\| \leq \left(\beta\alpha^{-1} + \gamma R\alpha^{-2}\right) \|\hat{x} - x^\star(\theta)\| \,.$$

This result is similar to [40, Theorem 7.2], that is concerned with the stability of solutions to inverse problems. Here we consider that $A(\cdot,\theta)$ is uniformly well-conditioned, rather than only at $x^\star(\theta)$. This does not affect the first order in ε of this bound, and makes it valid for all $\hat{x}$. It is also more tailored to applications to equation-specific cases.

Indeed, Theorem 1 can be applied to specific functions F or T for some root and fixed-point equations. In particular, for gradient descent fixed point, where $T(x,\theta) = x - \eta\nabla_1 f(x,\theta)$, this yields

$$A(x,\theta) = \eta\nabla_1^2 f(x,\theta) \text{ and } B(x,\theta) = -\eta\partial_2\nabla_1 f(x,\theta)\,.$$

This guarantees precision on the estimated Jacobian under regularity conditions on f directly; see Corollary 1 in Appendix D.

For proximal gradient descent, where $T(x,\theta) = \text{prox}_{\eta g}(x - \eta\nabla_1 f(x,\theta),\theta)$, this yields

$$\begin{aligned} A(x,\theta) &= I - \partial_1 T(x,\theta) = I - (I - \eta\nabla_1^2 f(x,\theta))\partial_1\text{prox}_{\eta g}(x - \eta\nabla_1 f(x,\theta),\theta) \\ B(x,\theta) &= \partial_2\text{prox}_{\eta g}(x - \eta\nabla_1 f(x,\theta),\theta) - \eta\partial_2\nabla_1 f(x,\theta)\partial_1\text{prox}_{\eta g}(x - \eta\nabla_1 f(x,\theta),\theta)\,. \end{aligned}$$

An important special case is that of a function to minimize in the form $f(x,\theta) + g(x)$, where the prox function g is smooth, and does not depend on θ, as is the case in our experiments in §3.2. For this setting, we derive similar guarantees in Corollary 2 in Appendix D. Recent work also exploits local smoothness of solutions to derive similar bounds [11, Theorem 13].

3 Experiments

To conclude this work, we demonstrate the ease of formulating and solving bi-level optimization problems with our modular framework. We also present an application to the sensitivity analysis of molecular dynamics.

3.1 Hyperparameter optimization of multiclass SVMs

In this example, we consider the hyperparamer optimization of multiclass SVMs [23] trained in the dual. Here, $x^\star(\theta)$ is the optimal dual solution, a matrix of shape $m \times k$, where m is the number of

Table 2: Mean AUC (and 95% confidence interval) for the cancer survival prediction problem.

Method	L_1 logreg	L_2 logreg	DictL + L_2 logreg	Task-driven DictL
AUC (%)	71.6 ± 2.0	72.4 ± 2.8	68.3 ± 2.3	73.2 ± 2.1

training examples and k is the number of classes, and $\theta \in \mathbb{R}_+$ is the regularization parameter. The challenge in differentiating $x^\star(\theta)$ is that each row of $x^\star(\theta)$ is constrained to belong to the probability simplex $\triangle^k$. More formally, let $X_{\texttt{tr}} \in \mathbb{R}^{m \times p}$ be the training feature matrix and $Y_{\texttt{tr}} \in \{0,1\}^{m \times k}$ be the training labels (in row-wise one-hot encoding). Let $W(x, \theta) \coloneqq X_{\texttt{tr}}^\top (Y_{\texttt{tr}} - x)/\theta \in \mathbb{R}^{p \times k}$ be the dual-primal mapping. Then, we consider the following bi-level optimization problem

$$\underbrace{\underset{\theta=\exp(\lambda)}{\operatorname{argmin}} \frac{1}{2}\|X_{\texttt{val}} W(x^\star(\theta), \theta) - Y_{\texttt{val}}\|_F^2}_{\text{outer problem}} \quad \text{subject to} \quad \underbrace{x^\star(\theta) = \underset{x \in \mathcal{C}}{\operatorname{argmin}} f(x, \theta) \coloneqq \frac{\theta}{2}\|W(x, \theta)\|_F^2}_{\text{inner problem}}, \tag{12}$$

where $\mathcal{C} = \triangle^k \times \cdots \times \triangle^k$ is the Cartesian product of m probability simplices. We apply the change of variable $\theta = \exp(\lambda)$ in order to guarantee that the hyper-parameter θ is positive. The matrix $W(x^\star(\theta), \theta) \in \mathbb{R}^{p \times k}$ contains the optimal primal solution, the feature weights for each class. The outer loss is computed against validation data $X_{\texttt{val}}$ and $Y_{\texttt{val}}$.

In order to differentiate $x^\star(\theta)$, several ways are possible using our framework. The first one would be to map (12) to a quadratic program form (18) and use the KKT conditions to form a mapping $F(x, \theta)$. A more direct way is to use proximal gradient fixed point (9). Since $\mathcal{C}$ is a Cartesian product, the projection can be easily computed by row-wise projections on the simplex, which we map over rows (using vectorized operations) via `jax.vmap`. As we explained in §B.1, this projection's Jacobian enjoys a closed form. A third way to differentiate $x^\star(\theta)$ is using the mirror descent fixed point (11). Under the KL geometry, $\text{proj}_{\mathcal{C}}^{\varphi}(y, \theta)$ corresponds to a row-wise softmax. It is therefore easy to compute and differentiate. Figure 2 compares the runtime performance of implicit differentiation vs. unrolling for the latter two fixed points. A code example is included in Figure 8 in the Appendix.

3.2 Task-driven dictionary learning

Task-driven dictionary learning was proposed to learn sparse codes for input data in such a way that the codes solve an outer learning problem [47, 64, 70]. Formally, given a data matrix $X_{\texttt{tr}} \in \mathbb{R}^{m \times p}$ and a dictionary of k atoms $\theta \in \mathbb{R}^{k \times p}$, a sparse code is defined as a matrix $x^\star(\theta) \in \mathbb{R}^{m \times k}$ that minimizes in x a reconstruction loss $f(x, \theta) \coloneqq \ell(X_{\texttt{tr}}, x\theta)$ regularized by a sparsity-inducing penalty $g(x)$. Instead of optimizing the dictionary θ to minimize the reconstruction loss, [47] proposed to optimize an outer problem that depends on the code. For example, given a set of labels $Y_{\texttt{tr}} \in \{0,1\}^m$, we consider a logistic regression problem which results in the bilevel optimization problem:

$$\underbrace{\min_{\theta \in \mathbb{R}^{k \times p}, w \in \mathbb{R}^k, b \in \mathbb{R}} \sigma(x^\star(\theta) w + b; y_{\texttt{tr}})}_{\text{outer problem}} \quad \text{subject to} \quad x^\star(\theta) \in \underbrace{\underset{x \in \mathbb{R}^{m \times k}}{\operatorname{argmin}} f(x, \theta) + g(x)}_{\text{inner problem}}. \tag{13}$$

When ℓ is the squared Frobenius distance between matrices, and g the elastic net penalty, [47, Eq. 21] derive manually, using optimality conditions (notably the support of the codes selected at the optimum), an explicit re-parameterization of $x^\star(\theta)$ as a linear system involving θ. This closed-form allows for a *direct* computation of the Jacobian of $x^\star$ w.r.t. θ. Similarly, [64] derive first order conditions in the case where ℓ is a β-divergence, while [70] propose to use unrolling of ISTA iterations. Our approach bypasses all of these computations, giving the user more leisure to focus directly on modeling (loss, regularizer) aspects (see code snippet in Figure 9 in Appendix).

We illustrate this on a problem of breast cancer survival prediction from gene expression data, framed as a binary classification problem to discriminate patients who survive longer than 5 years ($m_1 = 200$) vs patients who die within 5 years of diagnosis ($m_0 = 99$), from $p = 1,000$ gene expression values. As shown in Table 2, solving (13) (Task-driven DictL) reaches a classification performance competitive with state-of-the-art L_1 or L_2 regularized logistic regression with 100 times fewer variables. See Appendix E.2 for more details.

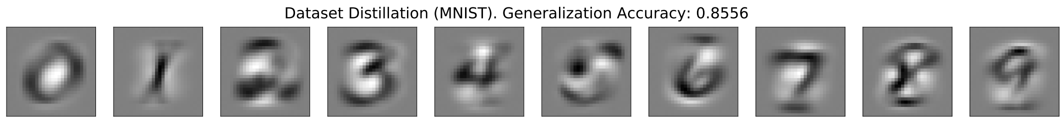

Figure 3: Distilled MNIST dataset $\theta \in \mathbb{R}^{k \times p}$ obtained by solving (14). We learn one image per class such that a logistic regression model trained on θ achieves the lowest logistic loss on the MNIST training set. Implicit differentiation was 4 times faster than unrolling.

3.3 Dataset distillation

Dataset distillation [67, 46] aims to learn a small synthetic training dataset such that a model trained on this learned data set achieves small loss on the original training set. Formally, let $X_{\mathtt{tr}} \in \mathbb{R}^{m \times p}$ and $y_{\mathtt{tr}} \in [k]^m$ denote the original training set. The distilled dataset will contain one prototype example for each class and therefore $\theta \in \mathbb{R}^{k \times p}$. The dataset distillation problem can then naturally be cast as a bi-level problem, where in the inner problem we estimate a logistic regression model $x^\star(\theta) \in \mathbb{R}^p$ trained on the distilled images $\theta \in \mathbb{R}^{k \times p}$, while in the outer problem we want to minimize the loss achieved by $x^\star(\theta)$ over the training set:

$$\underbrace{\operatorname*{argmin}_{\theta \in \mathbb{R}^{k \times p}} f(x^\star(\theta), X_{\mathtt{tr}}; y_{\mathtt{tr}})}_{\text{outer problem}} \quad \text{subject to} \quad x^\star(\theta) \in \underbrace{\operatorname*{argmin}_{x \in \mathbb{R}^p} f(x, \theta; [k]) + \varepsilon \|x\|^2}_{\text{inner problem}}, \tag{14}$$

where $f(x, X; y) := \ell(Xx, y)$, ℓ denotes the multiclass logistic regression loss, and $\varepsilon = 10^{-3}$ is a regularization parameter that we found had a very positive effect on convergence.

In this problem, and unlike in the general hyperparameter optimization setup, *both* the inner and outer problems are high-dimensional, making it an ideal test-bed for gradient-based bi-level optimization methods. For this experiment, we use the MNIST dataset. The number of parameters in the inner problem is $p = 28^2 = 784$. while the number of parameters of the outer loss is $k \times p = 7840$. We solve this problem using gradient descent on both the inner and outer problem, with the gradient of the outer loss computed using implicit differentiation, as described in §2. This is fundamentally different from the approach used in the original paper, where they used differentiation of the unrolled iterates instead. For the same solver, we found that the implicit differentiation approach was 4 times faster than the original one. The obtained distilled images θ are visualized in Figure 3 and a code example is given in Figure 10 in the Appendix.

3.4 Sensitivity analysis of molecular dynamics

Many applications of physical simulations require solving optimization problems, such as energy minimization in molecular [62] and continuum [8] mechanics, structural optimization [41] and data assimilation [32]. However, even fully differentiable simulators may not have efficient or accurate derivatives. We revisit an example from JAX-MD [62], the problem of finding energy minimizing configurations to a system of packed particles in an m-dimensional box of width ℓ,

$$x^\star(\theta) = \operatorname*{argmin}_{x \in \mathbb{R}^{k \times m}} \sum_{ij} U(x_{ij} \bmod \ell, \theta),$$

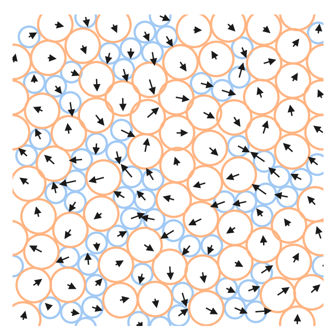

Figure 4: Particle positions and position sensitivity vectors, with respect to increasing the diameter of the blue particles.

where $U(x_{ij}, \theta)$ is the pairwise potential energy function, with half the particles at diameter 1 and half at diameter $\theta = 0.6$, which we optimize with a domain-specific optimizer [13]. Here we consider sensitivity of particle position with respect to diameter $\partial x^\star(\theta)$, rather than sensitivity of the total energy from the original experiment. Figure 4 shows results calculated via forward-mode implicit differentiation (JVP). Whereas differentiating the unrolled optimizer happens to work for total energy, here it typically does not even converge (see Appendix Fig. 15), due the discontinuous optimization method.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter. Differentiable convex optimization layers. *arXiv preprint arXiv:1910.12430*, 2019.

[3] A. Agrawal, S. Barratt, S. Boyd, E. Busseti, and W. M. Moursi. Differentiating through a cone program. *arXiv preprint arXiv:1904.09043*, 2019.

[4] A. Ali, E. Wong, and J. Z. Kolter. A semismooth newton method for fast, generic convex programming. In *International Conference on Machine Learning*, pages 70–79. PMLR, 2017.

[5] B. Amos. *Differentiable optimization-based modeling for machine learning*. PhD thesis, PhD thesis. Carnegie Mellon University, 2019.

[6] B. Amos and J. Z. Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *Proc. of ICML*, pages 136–145, 2017.

[7] S. Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. *arXiv preprint arXiv:1909.01377*, 2019.

[8] A. Beatson, J. Ash, G. Roeder, T. Xue, and R. P. Adams. Learning composable energy surrogates for pde order reduction. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 338–348. Curran Associates, Inc., 2020.

[9] Y. Bengio. Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900, 2000.

[10] Q. Bertrand, Q. Klopfenstein, M. Blondel, S. Vaiter, A. Gramfort, and J. Salmon. Implicit differentiation of lasso-type models for hyperparameter optimization. In *Proc. of ICML*, pages 810–821, 2020.

[11] Q. Bertrand, Q. Klopfenstein, M. Massias, M. Blondel, S. Vaiter, A. Gramfort, and J. Salmon. Implicit differentiation for fast hyperparameter selection in non-smooth convex learning. *arXiv preprint arXiv:2105.01637*, 2021.

[12] M. J. Best, N. Chakravarti, and V. A. Ubhaya. Minimizing separable convex functions subject to simple chain constraints. *SIAM Journal on Optimization*, 10(3):658–672, 2000.

[13] E. Bitzek, P. Koskinen, F. Gähler, M. Moseler, and P. Gumbsch. Structural relaxation made simple. *Phys. Rev. Lett.*, 97:170201, Oct 2006.

[14] M. Blondel. Structured prediction with projection oracles. In *Proc. of NeurIPS*, 2019.

[15] M. Blondel, V. Seguy, and A. Rolet. Smooth and sparse optimal transport. In *Proc. of AISTATS*, pages 880–889. PMLR, 2018.

[16] M. Blondel, O. Teboul, Q. Berthet, and J. Djolonga. Fast differentiable sorting and ranking. In *Proc. of ICML*, pages 950–959, 2020.

[17] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. Jax: composable transformations of python+ numpy programs, 2018. *URL http://github.com/google/jax*, 4:16, 2020.

[18] P. Brucker. An $O(n)$ algorithm for quadratic knapsack problems. *Operations Research Letters*, 3(3):163–166, 1984.

[19] O. Chapelle, V. Vapnik, O. Bousquet, and S. Mukherjee. Choosing multiple parameters for support vector machines. *Machine learning*, 46(1):131–159, 2002.

[20] X. Chen, Y. Zhang, C. Reisinger, and L. Song. Understanding deep architecture with reasoning layer. *Advances in Neural Information Processing Systems*, 33, 2020.

[21] H. Cherkaoui, J. Sulam, and T. Moreau. Learning to solve tv regularised problems with unrolled algorithms. *Advances in Neural Information Processing Systems*, 33, 2020.

[22] L. Condat. Fast projection onto the simplex and the ℓ_1 ball. *Mathematical Programming*, 158(1-2):575–585, 2016.

[23] K. Crammer and Y. Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of machine learning research*, 2(Dec):265–292, 2001.

[24] M. Cuturi. Sinkhorn distances: lightspeed computation of optimal transport. In *Advances in Neural Information Processing Systems*, volume 2, 2013.

[25] C.-A. Deledalle, S. Vaiter, J. Fadili, and G. Peyré. Stein unbiased gradient estimator of the risk (sugar) for multiple parameter selection. *SIAM Journal on Imaging Sciences*, 7(4):2448–2487, 2014.

[26] S. Diamond and S. Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[27] J. Djolonga and A. Krause. Differentiable learning of submodular models. *Proc. of NeurIPS*, 30:1013–1023, 2017.

[28] J. Domke. Generic methods for optimization-based modeling. In *Artificial Intelligence and Statistics*, pages 318–326. PMLR, 2012.

[29] J. C. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra. Efficient projections onto the ℓ_1-ball for learning in high dimensions. In *Proc. of ICML*, 2008.

[30] L. Franceschi, M. Donini, P. Frasconi, and M. Pontil. Forward and reverse gradient-based hyperparameter optimization. In *International Conference on Machine Learning*, pages 1165–1173. PMLR, 2017.

[31] L. Franceschi, P. Frasconi, S. Salzo, R. Grazzi, and M. Pontil. Bilevel programming for hyperparameter optimization and meta-learning. In *International Conference on Machine Learning*, pages 1568–1577. PMLR, 2018.

[32] T. Frerix, D. Kochkov, J. A. Smith, D. Cremers, M. P. Brenner, and S. Hoyer. Variational data assimilation with a learned inverse observation operator. 2021.

[33] R. Frostig, M. Johnson, D. Maclaurin, A. Paszke, and A. Radul. Decomposing reverse-mode automatic differentiation. In *LAFI 2021 workshop at POPL*, 2021.

[34] R. Frostig, M. J. Johnson, and C. Leary. Compiling machine learning programs via high-level tracing. *Machine Learning and Systems (MLSys)*, 2018.

[35] S. Gould, B. Fernando, A. Cherian, P. Anderson, R. S. Cruz, and E. Guo. On differentiating parameterized argmin and argmax problems with application to bi-level optimization. *arXiv preprint arXiv:1607.05447*, 2016.

[36] S. Gould, R. Hartley, and D. Campbell. Deep declarative networks: A new hope. *arXiv preprint arXiv:1909.04866*, 2019.

[37] S. Grotzinger and C. Witzgall. Projections onto order simplexes. *Applied mathematics and Optimization*, 12(1):247–270, 1984.

[38] I. Guyon. Design of experiments of the nips 2003 variable selection benchmark. In *NIPS 2003 workshop on feature extraction and feature selection*, volume 253, 2003.

[39] M. R. Hestenes, E. Stiefel, et al. *Methods of conjugate gradients for solving linear systems*, volume 49. NBS Washington, DC, 1952.

[40] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, second edition, 2002.

[41] S. Hoyer, J. Sohl-Dickstein, and S. Greydanus. Neural reparameterization improves structural optimization. 2019.

[42] Y. Kim, C. Denton, L. Hoang, and A. M. Rush. Structured attention networks. *arXiv preprint arXiv:1702.00887*, 2017.

[43] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[44] S. G. Krantz and H. R. Parks. *The implicit function theorem: history, theory, and applications*. Springer Science & Business Media, 2012.

[45] C. H. Lim and S. J. Wright. Efficient bregman projections onto the permutahedron and related polytopes. In *Proc. of AISTATS*, pages 1205–1213. PMLR, 2016.

[46] J. Lorraine, P. Vicol, and D. Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics*, pages 1540–1552. PMLR, 2020.

[47] J. Mairal, F. Bach, and J. Ponce. Task-driven dictionary learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(4):791–804, 2012.

[48] A. F. Martins and R. F. Astudillo. From softmax to sparsemax: A sparse model of attention and multi-label classification. In *Proc. of ICML*, 2016.

[49] C. Michelot. A finite algorithm for finding the projection of a point onto the canonical simplex of $\mathbb{R}^n$. *Journal of Optimization Theory and Applications*, 50(1):195–200, 1986.

[50] J.-J. Moreau. Proximité et dualité dans un espace hilbertien. *Bulletin de la S.M.F.*, 93:273–299, 1965.

[51] V. Niculae and M. Blondel. A regularized framework for sparse and structured neural attention. In *Proc. of NeurIPS*, 2017.

[52] V. Niculae and A. Martins. Lp-sparsemap: Differentiable relaxed optimization for sparse structured prediction. In *International Conference on Machine Learning*, pages 7348–7359, 2020.

[53] V. Niculae, A. Martins, M. Blondel, and C. Cardie. Sparsemap: Differentiable sparse structured inference. In *International Conference on Machine Learning*, pages 3799–3808. PMLR, 2018.

[54] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications*, 169(3):1042–1068, 2016.

[55] N. Parikh and S. Boyd. Proximal algorithms. *Foundations and Trends in optimization*, 1(3):127–239, 2014.

[56] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[57] F. Pedregosa. Hyperparameter optimization with approximate gradient. In *International conference on machine learning*. PMLR, 2016.

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

[59] A. Rajeswaran, C. Finn, S. Kakade, and S. Levine. Meta-learning with implicit gradients. *arXiv preprint arXiv:1909.04630*, 2019.

[60] N. Rappoport and R. Shamir. Multi-omic and multi-view clustering algorithms: review and cancer benchmark. *Nucleic Acids Res.*, 46:10546–10562, 2018.

[61] Y. Saad and M. H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[62] S. Schoenholz and E. D. Cubuk. Jax md: A framework for differentiable physics. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 11428–11441. Curran Associates, Inc., 2020.

[63] M. W. Seeger. Cross-validation optimization for large scale structured classification kernel methods. *Journal of Machine Learning Research*, 9(6), 2008.

[64] P. Sprechmann, A. M. Bronstein, and G. Sapiro. Supervised non-euclidean sparse nmf via bilevel optimization with applications to speech enhancement. In *2014 4th Joint Workshop on Hands-free Speech Communication and Microphone Arrays (HSCMA)*, pages 11–15. IEEE, 2014.

[65] S. Vaiter, C.-A. Deledalle, G. Peyré, C. Dossal, and J. Fadili. Local behavior of sparse analysis regularization: Applications to risk estimation. *Applied and Computational Harmonic Analysis*, 35(3):433–451, 2013.

[66] H. A. v. d. Vorst and H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[67] T. Wang, J.-Y. Zhu, A. Torralba, and A. A. Efros. Dataset distillation. *arXiv preprint arXiv:1811.10959*, 2018.

[68] R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

[69] Y. Wu, M. Ren, R. Liao, and R. B. Grosse. Understanding short-horizon bias in stochastic meta-optimization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[70] J. Zarka, L. Thiry, T. Angles, and S. Mallat. Deep network classification by scattering and homotopy dictionary learning. *arXiv preprint arXiv:1910.03561*, 2019.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] Our paper builds on the premise that reinstantiating the implicit function theorem for every optimization problem a user may encounter is cumbersome. We make the case that a modular approach is needed to bypass that issue. This approach raises several challenges, notably in the way these implicit solvers can be automatically instantiated, consistently, across the large corpus of optimization approaches favored by users.
   (b) Did you describe the limitations of your work? [Yes] , We discuss several limitations in our work. For instance, implicit differentiation requires $\hat{x}$ to be sufficiently close to $x^\star$ to be meaningful. This is the main topic of §2.3
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] As a purely methodological paper, we do not foresee negative societal impacts of our work.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] We confirm our paper conforms to those guidelines.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] , The paper contains one theoretical section, §2.3.
   (b) Did you include complete proofs of all theoretical results? [Yes] , All proofs are included in the Appendix D
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] At the time of submission, we are in the course of an approval process for open-source release required by our organization. We believe that the library itself comprises a contribution, and will have it available in open source by the time of this paper's publication (at the latest).
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] , Experiments were mostly run with minimal parameter tuning to reflect the simplicity of the approach we advocate. This is reflected in §3 and Appendix E
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] , see Figure 2 and std for the dictionary learning task.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] , see Appendix E
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] , see §3 and Appendix E
   (b) Did you mention the license of the assets? [Yes] , see Appendix E
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]