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]

Appendix

A Code examples

A.1 Code examples for optimality conditions

Our library provides several reusable optimality condition mappings F or fixed points T. We nevertheless demonstrate the ease of writing some of them from scratch.

Proximal gradient fixed point. The proximal gradient fixed point (7) with step size $\eta = 1$ is $T(x, \theta) = \text{prox}_g(x - \nabla_1 f(x, \theta_f), \theta_g)$. It can be implemented as follows.

```
grad = jax.grad(f)

def T(x, theta):
  theta_f, theta_g = theta
  return prox(x - grad(x, theta_f), theta_g)
```

Figure 5: Proximal gradient fixed point $T(x, \theta)$

We recall that when the proximity operator is a projection, we recover the projected gradient fixed point as a special case. Therefore, this fixed point can also be used for constrained optimization. We provide numerous proximal and projection operators in the library.

KKT conditions. As a more advanced example, we now describe how to implement the KKT conditions (6). The stationarity, primal feasibility and complementary slackness conditions read

$$\begin{aligned} \nabla_1 f(z, \theta_f) + [\partial_1 G(z, \theta_G)]^\top \lambda + [\partial_1 H(z, \theta_H)]^\top \nu &= 0 \\ H(z, \theta_H) &= 0 \\ \lambda \circ G(z, \theta_G) &= 0. \end{aligned}$$

Using `jax.vjp` to compute vector-Jacobian products, this can be implemented as

```
grad = jax.grad(f)

def F(x, theta):
  z, nu, lambd = x
  theta_f, theta_H, theta_G = theta

  _, H_vjp = jax.vjp(H, z, theta_H)
  stationarity = (grad(z, theta_f) + H_vjp(nu)[0])

  primal_feasability = H(z, theta_H)

  _, G_vjp = jax.vjp(G, z, theta_G)
  stationarity += G_vjp(lambd)[0]
  comp_slackness = G(z, theta_G) * lambd

  return stationarity, primal_feasability, comp_slackness
```

Figure 6: KKT conditions $F(x, \theta)$

Similar mappings F can be written if the optimization problem contains only equality constraints or only inequality constraints.

Mirror descent fixed point. Letting $\eta = 1$ and denoting $\theta = (\theta_f, \theta_{\text{proj}})$, the fixed point (11) is

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

We can then implement it as follows.

```
grad = jax.grad(f)

def T(x, theta):
  theta_f, theta_proj = params
  x_hat = phi_mapping(x)
  y = x_hat - grad(x, theta_f)
  return bregman_projection(y, theta_proj)
```

Figure 7: Mirror descent fixed point $T(x, \theta)$

Although not considered in this example, the mapping $\nabla\varphi$ could also depend on θ if necessary.

A.2 Code examples for experiments

We now sketch how to implement our experiments using our framework. In the following, jnp is short for jax.numpy. In all experiments, we only show how to compute gradients with the outer objective. We can then use these gradients with gradient-based solvers to solve the outer objective.

Multiclass SVM experiment.

```
X_tr, Y_tr, X_val, Y_val = load_data()

def W(x, theta):  # dual-primal map
  return jnp.dot(X_tr.T, Y_tr - x) / theta

def f(x, theta):  # inner objective
  return 0.5 * theta * jnp.sum(W(x, theta) ** 2)

grad = jax.grad(f)
proj = jax.vmap(projection_simplex)
def T(x, theta):
  return proj(x - grad(x, theta))

@custom_fixed_point(T)
def msvm_dual_solver(theta):
  # [...]
  return x_star  # solution of the dual objective

def outer_loss(lambd):
  theta = jnp.exp(lambd)
  x_star = msvm_dual_solver(theta)  # inner solution
  Y_pred = jnp.dot(W(x_star, theta), X_val)
  return 0.5 * jnp.sum((Y_pred - Y_val) ** 2)

print(jax.grad(outer_loss)(lambd))
```

Figure 8: Code example for the multiclass SVM experiment.

561 **Task-driven dictionary learning experiment.**

```
X_tr, y_tr = load_data()

def f(x, theta):  # dictionary loss
  residual = X_tr - jnp.dot(x, theta)
  return huber_loss(residual)

grad = jax.grad(f)
def T(x, theta):  # proximal gradient fixed point
  return prox_lasso(x - grad(x, theta))

@custom_fixed_point(T)
def sparse_coding(theta):  # inner objective
  # [...]
  return x_star  # lasso solution

def outer_loss(theta, w):  # task-driven loss
  x_star = sparse_coding(theta)  # sparse codes
  y_pred = jnp.dot(x_star, w)
  return logloss(y_tr, y_pred)

print(jax.grad(outer_loss, argnums=(0,1)))
```

Figure 9: Code example for the task-driven dictionary learning experiment.

562 **Dataset distillation experiment.**

```
X_tr, y_tr = load_data()

logloss = jax.vmap(loss.multiclass_logistic_loss)

def f(x, theta, l2reg=1e-3):  # inner objective
  scores = jnp.dot(theta, x)
  distilled_labels = jnp.arange(10)
  penalty = l2reg * jnp.sum(x * x)
  return jnp.mean(logloss(distilled_labels, scores)) + penalty

F = jax.grad(f)

@custom_root(F)
def logreg_solver(theta):
  # [...]
  return x_star

def outer_loss(theta):
  x_star = logreg_solver(theta)  # inner solution
  scores = jnp.dot(X_tr, x_star)
  return jnp.mean(logloss(y_tr, scores))

print(jax.grad(outer_loss)(theta))
```

Figure 10: Code example for the dataset distillation experiment.

Molecular dynamics experiment.

```python
energy_fn = soft_sphere_energy_fun(diameter)
init_fn, apply_fn = jax_md.minimize.fire_descent(
    energy_fun, shift_fun)

x0 = random.uniform(key, (N, 2))
R0 = L * x0  # transform to physical coordinates
R = lax.fori_loop(
    0, num_optimization_steps,
    body_fun=lambda t, state: apply_fn(state, t=t),
    init_val=init_fn(R0)).position
x = R / L

def normalized_forces(x, diameter):
  energy_fn = soft_sphere_energy_fun(diameter)
  normalized_energy_fun = lambda x: energy_fn(L * x)
  return -jax.grad(normalized_energy_fun)(x)

dx = root_jvp(normalized_forces, x, diameter, 1.0,
              solve=linear_solve.solve_bicgstab)
print(dx)
```

Figure 11: Code for the molecular dynamics experiment.

B Jacobian products

Our library provides numerous reusable building blocks. We describe in this section how to compute their Jacobian products. As a general guideline, whenever a projection enjoys a closed form, we leave the Jacobian product to the autodiff system.

B.1 Jacobian products of projections

We describe in this section how to compute the Jacobian products of the projections (in the Euclidean and KL senses) onto various convex sets. When the convex set does not depend on any variable, we simply denote it $\mathcal{C}$ instead of $\mathcal{C}(\theta)$.

Non-negative orthant. When $\mathcal{C}$ is the non-negative orthant, $\mathcal{C} = \mathbb{R}^d_+$, we obtain $\text{proj}_\mathcal{C}(y) = \max(y, 0)$, where the maximum is evaluated element-wise. This is also known as the ReLu function. The projection in the KL sense reduces to the exponential function, $\text{proj}^\varphi_\mathcal{C}(y) = \exp(y)$.

Box constraints. When $\mathcal{C}(\theta)$ is the box constraints $\mathcal{C}(\theta) = [\theta_1, \theta_2]^d$ with $\theta \in \mathbb{R}^2$, we obtain

$$\text{proj}_\mathcal{C}(y, \theta) = \text{clip}(y, \theta_1, \theta_2) := \max(\min(y, \theta_2), \theta_1).$$

This is trivially extended to support different boxes for each coordinate, in which case $\theta \in \mathbb{R}^{d\times 2}$.

Probability simplex. When $\mathcal{C}$ is the standard probability simplex, $\mathcal{C} = \triangle^d$, there is no analytical solution for $\text{proj}_\mathcal{C}(y)$. Nevertheless, the projection can be computed exactly in $O(d)$ expected time or $O(d \log d)$ worst-case time [18, 49, 29, 22]. The Jacobian is given by $\text{diag}(s) - ss^\top/\|s\|_1$, where $s \in \{0, 1\}^d$ is a vector indicating the support of $\text{proj}_\mathcal{C}(y)$ [48]. The projection in the KL sense, on the other hand, enjoys a closed form: it reduces to the usual softmax $\text{proj}^\varphi_\mathcal{C}(y) = \exp(y)/\sum_{j=1}^d \exp(y_j)$.

Box sections. Consider now the Euclidan projection $z^\star(\theta) = \text{proj}_\mathcal{C}(y, \theta)$ onto the set $\mathcal{C}(\theta) = \{z \in \mathbb{R}^d \colon \alpha_i \leq z_i \leq \beta_i, i \in [d]; w^\top z = c\}$, where $\theta = (\alpha, \beta, w, c)$. This projection is a singly-constrained bounded quadratic program. It is easy to check (see, e.g., [52]) that an optimal solution satisfies for all $i \in [d]$

$$z_i^\star(\theta) = [L(x^\star(\theta), \theta)]_i := \text{clip}(w_i x^\star(\theta) + y_i, \alpha_i, \beta_i)$$

where $L\colon \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^d$ is the dual-primal mapping and $x^\star(\theta) \in \mathbb{R}$ is the optimal dual variable of the linear constraint, which should be the root of

$$F(x^\star(\theta), \theta) = L(x^\star(\theta), \theta)^\top w - c.$$

The root can be found, e.g., by bisection. The gradient $\nabla x^\star(\theta)$ is given by $\nabla x^\star(\theta) = B^\top / A$ and the Jacobian $\partial z^\star(\theta)$ is obtained by application of the chain rule on L.

Norm balls. When $\mathcal{C}(\theta) = \{x \in \mathbb{R}^d \colon \|x\| \leq \theta\}$, where $\|\cdot\|$ is a norm and $\theta \in \mathbb{R}_+$, $\text{proj}_\mathcal{C}(y, \theta)$ becomes the projection onto a norm ball. The projection onto the ℓ_1-ball reduces to a projection onto the simplex, see, e.g., [29]. The projections onto the ℓ_2 and ℓ_∞ balls enjoy a closed-form, see, e.g., [55, §6.5]. Since they rely on simple composition of functions, all three projections can therefore be automatically differentiated.

Affine sets. When $\mathcal{C}(\theta) = \{x \in \mathbb{R}^d \colon Ax = b\}$, where $A \in \mathbb{R}^{p \times d}$, $b \in \mathbb{R}^p$ and $\theta = (A, b)$, we get

$$\text{proj}_\mathcal{C}(y, \theta) = y - A^\dagger(Ay - b) = y - A^\top(AA^\top)^{-1}(Ay - b)$$

where $A^\dagger$ is the Moore-Penrose pseudoinverse of A. The second equality holds if $p < d$ and A is full rank. A practical implementation can pre-compute a factorization of the Gram matrix $AA^\top$. Alternatively, we can also use the KKT conditions.

Hyperplanes and half spaces. When $\mathcal{C}(\theta) = \{x \in \mathbb{R}^d \colon a^\top x = b\}$, where $a \in \mathbb{R}^d$ and $b \in \mathbb{R}$ and $\theta = (a, b)$, we get

$$\text{proj}_\mathcal{C}(y, \theta) = y - \frac{a^\top y - b}{\|a\|_2^2} a.$$

When $\mathcal{C}(\theta) = \{x \in \mathbb{R}^d \colon a^\top x \leq b\}$, we simply replace $a^\top y - b$ in the numerator by $\max(a^\top y - b, 0)$.

Transportation and Birkhoff polytopes. When $\mathcal{C}(\theta) = \{X \in \mathbb{R}^{p \times d} \colon X\mathbf{1}_d = \theta_1, X^\top \mathbf{1}_p = \theta_2, X \geq 0\}$, the so-called transportation polytope, where $\theta_1 \in \triangle^p$ and $\theta_2 \in \triangle^d$ are marginals, we can compute approximately the projections, both in the Euclidean and KL senses, by switching to the dual or semi-dual [15]. Since both are unconstrained optimization problems, we can compute their Jacobian product by implicit differentiation using the gradient descent fixed point. An advantage of the KL geometry here is that we can use Sinkhorn [24], which is a GPU-friendly algorithm. The Birkhoff polytope, the set of doubly stochastic matrices, is obtained by fixing $\theta_1 = \theta_2 = \mathbf{1}_d / d$.

Order simplex. When $\mathcal{C}(\theta) = \{x \in \mathbb{R}^d \colon \theta_1 \geq x_1 \geq x_2 \geq \cdots \geq x_d \geq \theta_2\}$, a so-called order simplex [37, 14], the projection operations, both in the Euclidean and KL sense, reduce to isotonic optimization [45] and can be solved exactly in $O(d \log d)$ time using the Pool Adjacent Violators algorithm [12]. The Jacobian of the projections and efficient product with it are derived in [27, 16].

Polyhedra. More generally, we can consider polyhedra, i.e., sets of the form $\mathcal{C}(\theta) = \{x \in \mathbb{R}^d \colon Ax = b, Cx \leq d\}$, where $A \in \mathbb{R}^{p \times d}$, $b \in \mathbb{R}^p$, $C \in \mathbb{R}^{m \times d}$, and $d \in \mathbb{R}^m$. There are several ways to differentiate this projection. The first is to use the KKT conditions as detailed in §2.2. A second way is consider the dual of the projection instead, which is the maximization of a quadratic function subject to **non-negative constraints** [55, §6.2]. That is, we can reduce the projection on a polyhedron to a problem of the form (8) with non-negative constraints, which we can in turn implicitly differentiate easily using the projected gradient fixed point, combined with the projection on the non-negative orthant. Finally, we apply the dual-primal mapping , which enjoys a closed form and is therefore amenable to autodiff, to obtain the primal projection.

B.2 Jacobian products of proximity operators

We provide several proximity operators, including for the lasso (soft thresholding), elastic net and group lasso (block soft thresholding). All satisfy closed form expressions and can be differentiated automatically via autodiff. For more advanced proximity operators, which do not enjoy a closed form, recent works have derived their Jacobians. The Jacobians of fused lasso and oscar were derived in [51]. For general total variation, the Jacobians were derived in [65, 21].

C More examples of optimality criteria and fixed points

To demonstrate the generality of our approach, we describe in this section more optimality mapping F or fixed point iteration T.

Newton fixed point. Let x be a root of $G(\cdot, \theta)$, i.e., $G(x, \theta) = 0$. The fixed point iteration of Newton's method for root-finding is

$$T(x, \theta) = x - \eta[\partial_1 G(x, \theta)]^{-1} G(x, \theta).$$

By the chain and product rules, we have

$$\partial_1 T(x, \theta) = I - \eta(...)G(x, \theta) - \eta[\partial_1 G(x, \theta)]^{-1} \partial_1 G(x, \theta) = (1 - \eta)I.$$

Using (3), we get $A = -\partial_1 F(x, \theta) = \eta I$. Similarly,

$$B = \partial_2 T(x, \theta) = \partial_2 F(x, \theta) = -\eta[\partial_1 G(x, \theta)]^{-1} \partial_2 G(x, \theta).$$

Newton's method for optimization is obtained by choosing $G(x, \theta) = \nabla_1 f(x, \theta)$, which gives

$$T(x, \theta) = x - \eta[\nabla_1^2 f(x, \theta)]^{-1} \nabla_1 f(x, \theta). \tag{15}$$

It is easy to check that we recover the same linear system as for the gradient descent fixed point above. A practical implementation can pre-compute an LU decomposition of $\partial_1 G(x, \theta)$, or a Cholesky decomposition if $\partial_1 G(x, \theta)$ is positive semi-definite.

Proximal block coordinate descent fixed point. We now consider the case when $x^\star(\theta)$ is implicitly defined as the solution

$$x^\star(\theta) \coloneqq \operatorname*{argmin}_{x \in \mathbb{R}^d} f(x, \theta) + \sum_{i=1}^m g_i(x_i, \theta),$$

where $g_1, \ldots, g_m$ are possibly non-smooth functions operating on subvectors (blocks) $x_1, \ldots, x_m$ of x. In this case, we can use for $i \in [m]$ the fixed point

$$x_i = [T(x, \theta)]_i = \text{prox}_{\eta_i g_i}(x_i - \eta_i [\nabla_1 f(x, \theta)]_i, \theta), \tag{16}$$

where $\eta_1, \ldots, \eta_m$ are block-wise step sizes. Clearly, when the step sizes are shared, i.e., $\eta_1 = \cdots = \eta_m = \eta$, this fixed point is equivalent to the proximal gradient fixed point (7) with $g(x, \theta) = \sum_{i=1}^n g_i(x_i, \theta)$.

Quadratic programming. We now show how to use the KKT conditions discussed in §2.2 to differentiate quadratic programs, recovering Optnet [6] as a special case. To give some intuition, let us start with a simple equality-constrained quadratic program (QP)

$$\operatorname*{argmin}_{z \in \mathbb{R}^p} f(z, \theta) = \frac{1}{2} z^\top Q z + c^\top z \quad \text{subject to} \quad H(z, \theta) = Ez - d = 0,$$

where $Q \in \mathbb{R}^{p \times p}$, $E \in \mathbb{R}^{q \times p}$, $d \in \mathbb{R}^q$. We gather the differentiable parameters as $\theta = (Q, E, c, d)$. The stationarity and primal feasibility conditions give

$$\begin{aligned} \nabla_1 f(z, \theta) + [\partial_1 H(z, \theta)]^\top \nu = Qz + c + E^\top \nu &= 0 \\ H(z, \theta) = Ez - d &= 0. \end{aligned}$$

In matrix notation, this can be rewritten as

$$\begin{bmatrix} Q & E^\top \\ E & 0 \end{bmatrix} \begin{bmatrix} z \\ \nu \end{bmatrix} = \begin{bmatrix} -c \\ d \end{bmatrix}. \tag{17}$$

We can write the solution of the linear system (17) as the root $x = (z, \nu)$ of a function $F(x, \theta)$. More generally, the QP can also include inequality constraints

$$\operatorname*{argmin}_{z \in \mathbb{R}^p} f(z, \theta) = \frac{1}{2} z^\top Q z + c^\top z \quad \text{subject to} \quad H(z, \theta) = Ez - d = 0, G(z, \theta) = Mz - h \leq 0. \tag{18}$$

where $M \in \mathbb{R}^{r \times p}$ and $h \in \mathbb{R}^r$. We gather the differentiable parameters as $\theta = (Q, E, M, c, d, h)$. The stationarity, primal feasibility and complementary slackness conditions give

$$\begin{aligned} \nabla_1 f(z, \theta) + [\partial_1 H(z, \theta)]^\top \nu + [\partial_1 G(z, \theta)]^\top \lambda = Qz + c + E^\top \nu + M^\top \lambda = 0 \\ H(z, \theta) = Ez - d = 0 \\ \lambda \circ G(z, \theta) = \operatorname{diag}(\lambda)(Mz - h) = 0 \end{aligned}$$

In matrix notation, this can be written as

$$\begin{bmatrix} Q & E^\top & M^\top \\ E & 0 & 0 \\ \operatorname{diag}(\lambda) M & 0 & 0 \end{bmatrix} \begin{bmatrix} z \\ \nu \\ \lambda \end{bmatrix} = \begin{bmatrix} -c \\ d \\ \lambda \circ h \end{bmatrix}$$

While $x = (z, \nu, \lambda)$ is no longer the solution of a linear system, it is the root of a function $F(x, \theta)$ and therefore fits our framework.

Conic programming. We now show that the differentiation of conic linear programs [3, 5], at the heart of differentiating through cvxpy layers [2], easily fits our framework. Consider the problem

$$z^\star(\lambda), s^\star(\lambda) = \operatorname*{argmin}_{z \in \mathbb{R}^p, s \in \mathbb{R}^m} c^\top z \quad \text{subject to} \quad Ez + s = d, s \in \mathcal{K}, \tag{19}$$

where $\lambda = (c, E, d)$, $E \in \mathbb{R}^{m \times p}$, $d \in \mathbb{R}^m$, $c \in \mathbb{R}^p$ and $\mathcal{K} \subseteq \mathbb{R}^m$ is a cone; z and s are the primal and slack variables, respectively. Every convex optimization problem can be reduced to the form (19). Let us form the skew-symmetric matrix

$$\theta(\lambda) = \begin{bmatrix} 0 & E^\top & c \\ -E & 0 & d \\ -c^\top & -d^\top & 0 \end{bmatrix} \in \mathbb{R}^{N \times N},$$

where $N = p + m + 1$. Following [3, 2, 5], we can use the homogeneous self-dual embedding to reduce the process of solving (19) to finding a root of the residual map

$$F(x, \theta) = \theta \Pi x + \Pi^* x = ((\theta - I)\Pi + I)x, \tag{20}$$

where $\Pi = \text{proj}_{\mathbb{R}^p \times \mathcal{K}^* \times \mathbb{R}_+}$ and $\mathcal{K}^* \subseteq \mathbb{R}^m$ is the dual cone. The splitting conic solver [54], which is based on ADMM, outputs a solution $F(x^\star(\theta), \theta) = 0$ which is decomposed as $x^\star(\theta) = (u^\star(\theta), v^\star(\theta), w^\star(\theta))$. We can then recover the optimal solution of (19) using

$$z^\star(\lambda) = u^\star(\theta(\lambda)) \quad \text{and} \quad s^\star(\lambda) = \text{proj}_{\mathcal{K}^*}(v^\star(\theta(\lambda))) - v^\star(\theta(\lambda)).$$

The key oracle whose JVP/VJP we need is therefore Π, which is studied in [4]. The projection onto a few cones is available in our library and can be used to express F.

Frank-Wolfe. We now consider

$$x^\star(\theta) = \operatorname*{argmin}_{x \in \mathcal{C}(\theta) \subset \mathbb{R}^d} f(x, \theta), \tag{21}$$

where $\mathcal{C}(\theta)$ is a convex polytope, i.e., it is the convex hull of vertices $v_1(\theta), \ldots, v_m(\theta)$. The Frank-Wolfe algorithm requires a linear minimization oracle (LMO):

$$s \mapsto \operatorname*{argmin}_{x \in \mathcal{C}(\theta)} \langle s, x \rangle$$

and is a popular algorithm when this LMO is easier to compute than the projection onto $\mathcal{C}(\theta)$. However, since this LMO is piecewise constant, its Jacobian is null almost everywhere. Inspired by SparseMAP [53], which corresponds to the case when f is a quadratic, we rewrite (21) as

$$p^\star(\theta) = \operatorname*{argmin}_{p \in \triangle^m} g(p, \theta) := f(V(\theta)p, \theta),$$

where $V(\theta)$ is a $d \times m$ matrix gathering the vertices $v_1(\theta), \ldots, v_m(\theta)$. We then have $x^\star(\theta) = V(\theta)p^\star(\theta)$. Since we have reduced (21) to minimization over the simplex, we can use the projected gradient fixed point to obtain

$$T(p^\star(\theta), \theta) = \text{proj}_{\triangle^m}(p^\star(\theta) - \nabla_1 g(p^*(\theta), \theta)).$$

We can therefore compute the derivatives of $p^\star(\theta)$ by implicit differentiation and the derivatives of $x^\star(\theta)$ by chain rule. Frank-Wolfe implementations typically maintain the convex weights of the vertices, which we use to get an approximation of $p^\star(\theta)$. Moreover, it is well-known that after t iterations, at most t vertices are visited. We can leverage this sparsity to solve a smaller linear system. Moreover, in practice, we only need to compute VJPs of $x^\star(\theta)$.

D Proofs and technical results

Proof of Theorem 1. To simplify notations, we note $A_\star := A(x^\star, \theta)$ and $\hat{A} := A(\hat{x}, \theta)$, and similarly for B and J. We have by definition of the Jacobian estimate function $A_\star J_\star = B_\star$ and $\hat{A}\hat{J} = \hat{B}$. Therefore we have

$$\begin{aligned} J(\hat{x}, \theta) - \partial x^\star(\theta) &= \hat{A}^{-1}\hat{B} - A_\star^{-1} B_\star \\ &= \hat{A}^{-1}\hat{B} - \hat{A}^{-1} B_\star + \hat{A}^{-1} B_\star - A_\star^{-1} B_\star \\ &= \hat{A}^{-1}(\hat{B} - B_\star) + (\hat{A}^{-1} - A_\star^{-1}) B_\star \,. \end{aligned}$$

For any invertible matrices M_1, M_2, it holds that $M_1^{-1} - M_2^{-1} = M_1^{-1}(M_2 - M_1)M_2^{-1}$, so

$$\|M_2^{-1} - M_2^{-1}\|_{op} \leq \|M_1^{-1}\|_{op} \|M_2 - M_1\|_{op} \|M_2^{-1}\|_{op} \,. \tag{22}$$

Therefore,

$$\|\hat{A}^{-1} - A_\star^{-1}\|_{op} \leq \frac{1}{\alpha^2}\|\hat{A} - A_\star\|_{op} \leq \frac{\gamma}{\alpha^2}\|\hat{x} - x^\star(\theta)\| \,. \tag{23}$$

As a consequence, the second term in $J(\hat{x}, \theta) - \partial x^\star(\theta)$ can be upper bounded and we obtain

$$\begin{aligned} \|J(\hat{x}, \theta) - \partial x^\star(\theta)\| &\leq \|\hat{A}^{-1}(\hat{B} - B_\star)\| + \|(\hat{A}^{-1} - A_\star^{-1})B_\star\| \\ &\leq \|\hat{A}^{-1}\|_{\text{op}}\|\hat{B} - B_\star\| + \frac{\gamma}{\alpha^2}\|\hat{x} - x^\star(\theta)\| \, \|B_\star\| \,, \end{aligned}$$

which yields the desired result. □

Corollary 1 (Jacobian precision for gradient descent fixed point)**.** *Let f be such that $f(\cdot, \theta)$ is twice differentiable and α-strongly convex and $\nabla_1^2 f(\cdot, \theta)$ is γ-Lipschitz (in the operator norm) and $\partial_2 \nabla_1 f(x, \theta)$ is β-Lipschitz and bounded in norm by R. The estimated Jacobian evaluated at $\hat{x}$ is then given by*

$$J(\hat{x}, \theta) = -(\nabla_1^2 f(\hat{x}, \theta))^{-1} \partial_2 \nabla_1 f(\hat{x}, \theta) \,.$$

For all $\theta \in \mathbb{R}^n$, and any $\hat{x}$ estimating $x^\star(\theta)$, we have the following bound for the approximation error of the estimated Jacobian

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

Proof of Corollary 1. This follows from Theorem 1, applied to this specific $A(x, \theta)$ and $B(x, \theta)$. □

Corollary 2 (Jacobian precision for proximal gradient descent fixed point)**.** *Let f be such that $f(\cdot, \theta)$ is twice differentiable and α-strongly convex and $\nabla_1^2 f(\cdot, \theta)$ is γ-Lipschitz (in the operator norm) and $\partial_2 \nabla_1 f(x, \theta)$ is β-Lipschitz and bounded in norm by R. Let $g : \mathbb{R}^d \to \mathbb{R}$ be a twice-differentiable μ-strongly convex, λ smooth function (i.e. whose Hessian has a spectrum in $[\mu, \lambda]$, for which $\Gamma(x, \theta) = \nabla^2 g(\text{prox}_{\eta g}(x - \eta \nabla_1 f(x, \theta))$ is κ-Lipschitz in it first argument. The estimated Jacobian evaluated at $\hat{x}$ is then given by*

$$J(\hat{x}, \theta) = -(I_d + \eta\Gamma(x, \theta))(\nabla_1^2 f(\hat{x}, \theta) + \nabla_1^2 g(\hat{x}))^{-1} \partial_2 \nabla_1 f(\hat{x}, \theta)(I_d + \eta\Gamma(x, \theta))^{-1} \,.$$

For all $\theta \in \mathbb{R}^n$, and any $\hat{x}$ estimating $x^\star(\theta)$, we have the following bound for the approximation error of the estimated Jacobian

$$\|J(\hat{x}, \theta) - \partial x^\star(\theta)\| \leq (\tilde{\beta}_\eta / \tilde{\alpha}_\eta + \tilde{R}_\eta \tilde{\rho}_\eta)\|\hat{x} - x^\star(\theta)\| \,,$$

where

$$\tilde{\alpha}_\eta := \frac{\alpha + \mu}{1 + \eta\lambda} \,, \; \tilde{R}_\eta := \frac{R}{1 + \eta\mu} \; \tilde{\rho}_\eta := (1 + \eta\lambda)\frac{\gamma + \kappa}{(\alpha + \mu)^2} + \frac{\eta\kappa}{\alpha + \mu} \; \tilde{\beta}\rho := \frac{\beta}{1 + \eta\mu} + \frac{\eta\kappa R}{(1 + \eta\mu)^2}$$

Proof of Corollary 2. First, let us note that $\text{prox}_{\eta g}(y, \theta)$ does not depend on θ, since g itself does not depend on θ, and is therefore equal to classical proximity operator of ηg which, with a slight overload of notations, we denote as $\text{prox}_{\eta g}(y)$ (with a single argument). In other words,

$$\begin{cases} \text{prox}_{\eta g}(y, \theta) &= \text{prox}_{\eta g}(y) \,, \\ \partial_1 \text{prox}_{\eta g}(y, \theta) &= \partial \text{prox}_{\eta g}(y) \,, \\ \partial_2 \text{prox}_{\eta g}(y, \theta) &= 0 \,. \end{cases}$$

Regarding the first claim (expression of the estimated Jacobian evaluated at $\hat{x}$), we first have that $\mathrm{prox}_{\eta g}(y)$ is the solution to $(x' - y) + \eta \nabla g(x') = 0$ in x' - by first-order condition for a smooth convex function. We therefore have that

$$\mathrm{prox}_{\eta g}(y) = (I + \eta \nabla g)^{-1}(y)$$
$$\partial \mathrm{prox}_{\eta g}(y) = (I_d + \eta \nabla^2 g(\mathrm{prox}_{\eta g}(y)))^{-1} ,$$

the first I and inverse being functional identity and inverse, and the second I_d and inverse being in the matrix sense, by inverse rule for Jacobians $\partial h(z) = [\partial h^{-1}(h(z))]^{-1}$ (applied to the prox).

As a consequence, we have, for $\Gamma(x, \theta) = \nabla^2 g(\mathrm{prox}_{\eta g}(x - \eta \nabla_1 f(x, \theta))$ that

$$\begin{aligned} A(x,\theta) &= I_d - (I_d - \eta\nabla_1^2 f(x,\theta))(I_d + \eta\Gamma(x,\theta))^{-1} \\ &= [I_d + \eta\Gamma(x,\theta) - (I_d - \eta\nabla_1^2 f(x,\theta))](I_d + \eta\Gamma(x,\theta))^{-1} \\ &= \eta(\nabla_1^2 f(x,\theta) + \Gamma(x,\theta))(I_d + \eta\Gamma(x,\theta))^{-1} \\ B(x,\theta) &= \eta\partial_2\nabla_1 f(x,\theta)(I_d + \eta\Gamma(x,\theta))^{-1} . \end{aligned}$$

As a consequence, for all $x \in \mathbb{R}^d$, we have that

$$J(x,\theta) = -(I_d + \eta\Gamma(x,\theta))(\nabla_1^2 f(x,\theta) + \Gamma(x,\theta))^{-1}\partial_2\nabla_1 f(x,\theta)(I_d + \eta\Gamma(x,\theta))^{-1} .$$

In the following, we rescale both A and B by a factor η that is cancelled out in the computation of J. Under the same notations for $\hat{A}$, $A_\star$, $\hat{B}$, and $B_\star$, we have that

$$\|\hat{A}^{-1}\|_{\mathrm{op}} \le \frac{1+\eta\lambda}{\alpha+\mu} := \frac{1}{\tilde{\alpha}_\eta} ,\ \|B_\star\| \le \frac{R}{1+\eta\mu} := \tilde{R}_\eta ,$$

and using (22):

$$\|(I_d + \eta\Gamma_\star)^{-1} - (I_d + \eta\hat{\Gamma})^{-1}\|_{\mathrm{op}} \le \frac{\eta\kappa\|\hat{x} - x^\star(\theta)\|}{(1+\eta\mu)^2} .$$

Further, using that for all matrice M_1, M_1', M_2 and M_2', we have

$$\|M_1M_2 - M_1'M_2'\|_{\mathrm{op}} \le \|M_1\|_{\mathrm{op}}\|M_2 - M_2'\|_{\mathrm{op}} + \|M_2'\|_{\mathrm{op}}\|M_1 - M_1'\|_{\mathrm{op}} ,$$

we have

$$\|\hat{A}^{-1} - A_\star^{-1}\|_{\mathrm{op}} \le \left[(1+\eta\lambda)\frac{\gamma+\kappa}{(\alpha+\mu)^2} + \frac{\eta\kappa}{\alpha+\mu}\right]\|\hat{x} - x^\star(\theta)\| := \tilde{\rho}_\eta\|\hat{x} - x^\star(\theta)\|$$
$$\|\hat{B} - B_\star\| \le \left[\frac{\beta}{1+\eta\mu} + \frac{\eta\kappa R}{(1+\eta\mu)^2}\right]\|\hat{x} - x^\star(\theta)\| := \tilde{\beta}_\eta\|\hat{x} - x^\star(\theta)\| .$$

While this does not show that A is Lipschitz, as assumed in Theorem 1, this directly proves that A^{-1} is Lipschitz, which is in fact what we need in the proof of Theorem 1, and that we deduced from the Lipschitzness and well-conditioning of A in (23). Following the proof of Theorem 1 yields as desired

$$\|J(\hat{x},\theta) - \partial x^\star(\theta)\| \le (\tilde{\beta}_\eta/\tilde{\alpha}_\eta + \tilde{R}_\eta\tilde{\rho}_\eta)\|\hat{x} - x^\star(\theta)\| .$$

□

E Experimental setup and additional results

Our experiments use JAX [17], which is Apache2-licensed and scikit-learn [58], which is BSD-licensed.

E.1 Hyperparameter optimization of multiclass SVMs

Experimental setup. Synthetic datasets were generated using `scikit-learn`'s `sklearn.datasets.make_classification` [58], following a model adapted from [38]. All datasets consist of $m = 700$ training samples belonging to $k = 5$ distinct classes. To simulate problems of different sizes, the number of features is varied as $p \in \{100, 250, 500, 750, 1000, 2000, 3000, 4000, 5000, 7500, 10000\}$, with 10% of features

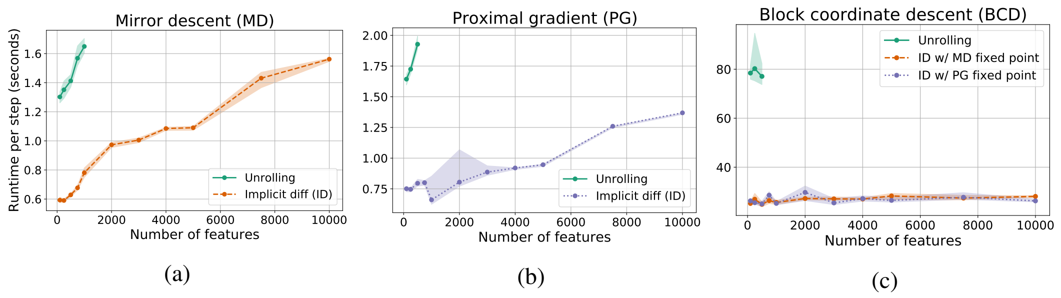

Figure 12: GPU runtime comparison of implicit differentiation and unrolling for hyperparameter optimization of multiclass SVMs for multiple problem sizes (same setting as Figure 2). Error bars represent 90% confidence intervals. Absent data points were due to out-of-memory errors (16 GB maximum).

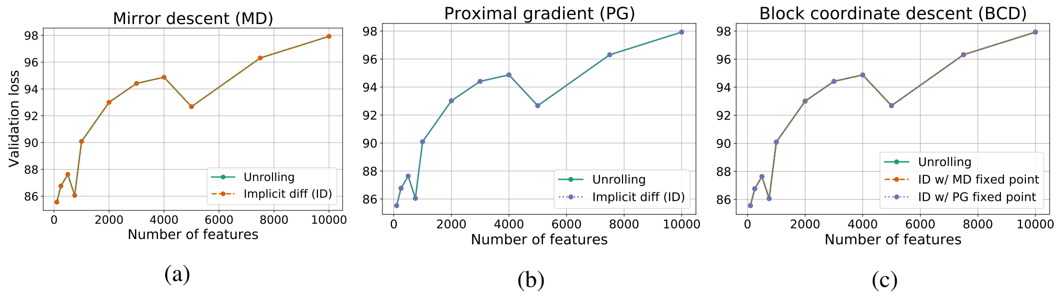

Figure 13: Value of the outer problem objective function (validation loss) for hyperparameter optimization of multiclass SVMs for multiple problem sizes (same setting as Figure 2). As can be seen, all methods performed similarly in terms of validation loss.

being informative and the rest random noise. In all cases, an additional $m_{\text{val}} = 200$ validation samples were generated from the same model to define the outer problem.

For the inner problem, we employed three different solvers: (i) mirror descent, (ii) (accelerated) proximal gradient descent and (iii) block coordinate descent. Hyperparameters for all solvers were individually tuned manually to ensure convergence across the range of problem sizes. For mirror descent, a stepsize of 1.0 was used for the first 100 steps, following a inverse square root decay afterwards up to a total of 2500 steps. For proximal gradient descent, a stepsize of $5 \cdot 10^{-4}$ was used for 2500 steps. The block coordinate descent solver was run for 500 iterations. All solvers used the same initialization, namely, $x_{\text{init}} = \frac{1}{k} 1_{m \times k}$, which satisfies the dual constraints.

For the outer problem, gradient descent was used with a stepsize of $5 \cdot 10^{-3}$ for the first 100 steps, following a inverse square root decay afterwards up to a total of 150 steps.

Conjugate gradient was used to solve the linear systems in implicit differentiation for at most 2500 iterations.

All results reported pertaining CPU runtimes were obtained using an internal compute cluster. GPU results were obtained using a single NVIDIA P100 GPU with 16GB of memory per dataset. For each dataset size, we report the average runtime of an individual iteration in the outer problem, alongside a 90% confidence interval estimated from the corresponding 150 runtime values.

Additional results Figure 12 compares the runtime of implicit differentiation and unrolling on GPU. These results highlight a fundamental limitation of the unrolling approach in memory-limited systems such as accelerators, as the inner solver suffered from out-of-memory errors for most problem sizes ($p \geq 2000$ for mirror descent, $p \geq 750$ for proximal gradient and block coordinate descent). While it might be possible to ameliorate this limitation by reducing the maximum number of iterations in the inner solver, doing so might lead to additional challenges [69] and require careful tuning.

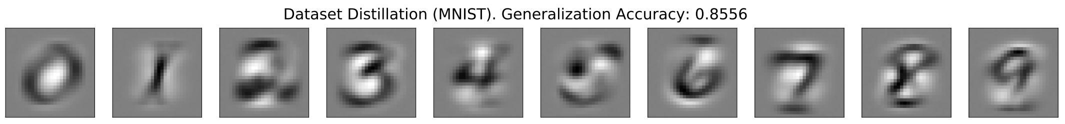
Dataset Distillation (MNIST). Generalization Accuracy: 0.8556

Figure 14: Distilled MNIST dataset $\theta \in \mathbb{R}^{k \times p}$ obtained by solving (14) through unrolled differentiation. Although there is no qualitative difference, the implicit differentiation approach is 4 times faster.

Figure 13 depicts the validation loss (value of the outer problem objective function) at convergence. It shows that all approaches were able to solve the outer problem, with solutions produced by different approaches being qualitatively indistinguishable from each other across the range of problem sizes considered.

E.2 Task-driven dictionary learning

We downloaded from `http://acgt.cs.tau.ac.il/multi_omic_benchmark/download.html` a set of breast cancer gene expression data together with survival information generated by the TCGA Research Network (`https://www.cancer.gov/tcga`) and processed as explained by [60]. The gene expression matrix contains the expression value for p=20,531 genes in m=1,212 samples, from which we keep only the primary tumors (m=1,093). From the survival information, we select the patients who survived at least five years after diagnosis ($m_1 = 200$), and the patients who died before five years ($m_0 = 99$), resulting in a cohort of $m = 299$ patients with gene expression and binary label. Note that non-selected patients are those who are marked as alive but were not followed for 5 years.

To evaluate different binary classification methods on this cohort, we repeated 10 times a random split of the full cohort into a training (60%), validation (20%) and test (20%) sets. For each split and each method, 1) the method is trained with different parameters on the training set, 2) the parameter that maximizes the classification AUC on the validation set is selected, 3) the method is then re-trained on the union of the training and validation sets with the selected parameter, and 4) we measure the AUC of that model on the test set. We then report, for each method, the mean test AUC over the 10 repeats, together with a 95% confidence interval defined a mean $\pm$ 1.96 $\times$ standard error of the mean.

We used Scikit Learn's implementation of logistic regression regularized by ℓ_1 (lasso) and ℓ_2 (ridge) penalty from `sklearn.linear_model.LogisticRegression`, and varied the `C` regularization parameter over a grid of 10 values: $\{10^{-5}, 10^{-3}, \ldots, 10^{4}\}$. For the unsupervised dictionary learning experiment method, we estimated a dictionary from the gene expression data in the training and validation sets, using `sklearn.decomposition.DictionaryLearning(n_components=10, alpha=2.0)`, which produces sparse codes in $k = 10$ dimensions with roughly 50% nonzero coefficients by minimizing the squared Frobenius reconstruction distance with lasso regularization on the code. We then use `sklearn.linear_model.LogisticRegression` to train a logistic regression on the codes, varying the ridge regularization parameter `C` over a grid of 10 values $\{10^{-1}, 10^{0}, \ldots, 10^{8}\}$.

Finally, we implemented the task-driven dictionary learning model (13) with our toolbox, following the pseudo-code in Figure 9. Like for the unsupervised dictionary learning experiment, we set the dimension of the codes to $k = 10$, and a fixed elastic net regularization on the inner optimization problem to ensure that the codes have roughly 50% sparsity. For the outer optimization problem, we solve an ℓ_2 regularized ridge regression problem, varying again the ridge regularization parameter `C` over a grid of 10 values $\{10^{-1}, 10^{0}, \ldots, 10^{8}\}$. Because the outer problem is non-convex, we minimize it using the Adam optimizer [43] with default parameters.

E.3 Dataset Distillation

Experimental setup. For the inner problem, we used gradient descent with backtracking line-search, while for the outer problem we used gradient descent with momentum and a fixed step-size. The momentum parameter was set to 0.9 while the step-size was set to 1.

Fig. 3 was produced after 4000 iterations of the outer loop on CPU (Intel(R) Xeon(R) Platinum P-8136 CPU @ 2.00GHz), which took 1h55. Unrolled differentiation took instead 8h:05 (4 times

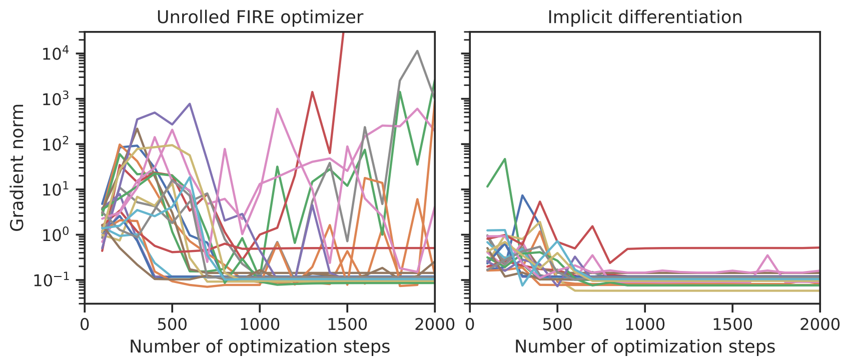

Figure 15: L1 norm of position sensitivities in the molecular dynamics simulations, for 40 different random initial conditions (different colored lines). Gradients through the unrolled FIRE optimizer [13] for many initial conditions do not converge, in contrast to implicit differentiation.

more) to run the same number of iterations. As can be seen in Fig. 14, the output is the same in both approaches.

E.4 Molecular dynamics

Our experimental setup is adapted from the JAX-MD example notebook available at `https://github.com/google/jax-md/blob/master/notebooks/meta_optimization.ipynb`.

We emphasize that calculating the gradient of the total energy objective, $E(x, \theta) = \sum_{ij} U_{ij}(x_{ij}, \theta)$, with respect to the diameter θ of the smaller particles, $\partial E / \partial \theta$, does not require implicit differentiation or unrolling. This is because $\nabla_1 E(x, \theta) = 0$ at $x = x^\star(\theta)$:

$$\nabla_\theta E(x^\star(\theta), \theta) = \partial x^\star(\theta)^\top \nabla_1 E(x^\star(\theta), \theta) + \nabla_2 E(x^\star(\theta), \theta) = \nabla_2 E(x^\star(\theta), \theta).$$

This is known as Danskin's theorem or envelope theorem. Thus instead, we consider sensitivities of position $\partial x^\star(\theta)$ directly, which does require implicit differentiation or unrolling.

Our results comparing implicit and unrolled differentiation for calculating the sensitivty of position are shown in Fig. 15. We use BiCGSTAB [66] to perform the tangent linear solve. Like in the original JAX-MD experiment, we use $k = 128$ particles in two dimensions.