Google Research

Efficient and Modular Implicit Differentiation

Mathieu Blondel

Joint work with: Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, J-P. Vert

June 4, 2021

Gradient-based learning

- Gradient-based training algorithms are the workhorse of modern machine learning.
- Deriving gradients by hand is tedious and error prone.
- This becomes quickly infeasible for complex models.
- Changes to the model require rederiving the gradient.
- Deep learning = GPU + data + autodiff
- **This talk: differentiating optimization problem solutions**

Outline

Automatic differentiation

- Evaluates the derivatives of a function at a given point.
- Not the same as numerical differentiation.
- Not the same as symbolic differentiation, which returns a "human-readable" expression.
- In a neural network context, reverse autodiff is often known as backpropagation.

Automatic differentiation

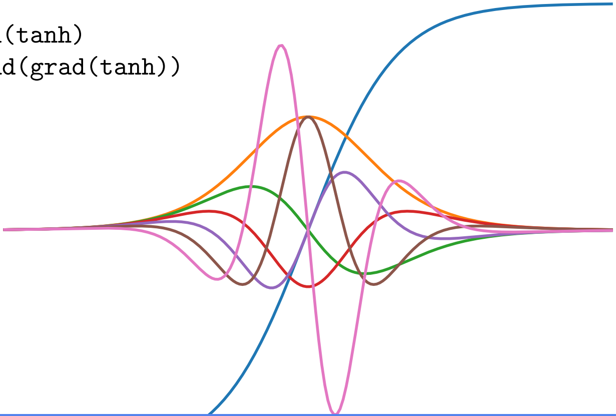
- A program is defined as the composition of primitive operations that we know how to derive.
- The user can focus on the forward computation / model.

```
import jax.numpy as jnp
from jax import grad, jit

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def logprob_fun(params, inputs, targets):
  preds = predict(params, inputs)
  return jnp.sum((preds - targets)**2)

grad_fun = jit(grad(logprob_fun))
```

Automatic differentiation

- Modern frameworks support higher-order derivatives

```
def tanh(x):
  y = jnp.exp(-2.0 * x)
  return (1.0 - y) / (1.0 + y)

fp = grad(tanh)
fpp = grad(grad(tanh))
...
```



Forward-mode vs. Reverse-mode

- Forward-mode
  - Computes Jacobian vector products (JVPs) along the forward pass
  - Each JVP call builds one column of the Jacobian
  - Efficient for tall Jacobians (more outputs than inputs)
  - Need not store intermediate computations
- Reverse-mode
  - Computes vector Jacobian products (VJPs) in reverse order
  - Each VJP call builds one row of the Jacobian
  - Efficient for wide matrices (more inputs than outputs)
  - Needs to store intermediate computations

Key components of an autodiff system

- JVPs and/or VJPs for all primitive operations
- Obtaining the computational graph
  - Ahead of time (from source or using a DSL)
  - Just in time (graph is built while being executed)
- Topological sort
- Forward-mode: forward pass (JVPs)
- Reverse-mode: forward pass + backward pass (VJPs)

Outline

Notation

- Small letters for scalar-valued functions, e.g., f
- The gradient of $f \colon \mathbb{R}^d \to \mathbb{R}$

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_d}(x) \end{bmatrix} \in \mathbb{R}^d$$

- The Hessian of $f \colon \mathbb{R}^d \to \mathbb{R}$ evaluated at $x \in \mathbb{R}^d$

$$\nabla^2 f(x) = \begin{bmatrix} \dfrac{\partial^2 f(x)}{\partial x_1^2} & \cdots & \dfrac{\partial^2 f(x)}{\partial x_1 \, \partial x_d} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f(x)}{\partial x_d \, \partial x_1} & \cdots & \dfrac{\partial^2 f(x)}{\partial x_d^2} \end{bmatrix} \in \mathbb{R}^{d \times d}$$

Notation

- Capital letters for vector-valued functions, e.g., F
- The Jacobian of $F \colon \mathbb{R}^d \to \mathbb{R}^p$ evaluated at $x \in \mathbb{R}^d$

$$\partial F(x) = \begin{bmatrix} \frac{\partial F_1(x)}{\partial x_1} & \cdots & \frac{\partial F_1(x)}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_p(x)}{\partial x_1} & \cdots & \frac{\partial F_p(x)}{\partial x_d} \end{bmatrix} = \begin{bmatrix} \nabla F_1(x)^\top \\ \vdots \\ \nabla F_p(x)^\top \end{bmatrix} \in \mathbb{R}^{p \times d}$$

- Jacobian-vector product (JVP) with $u \in \mathbb{R}^d$

$$\partial F(x) u \in \mathbb{R}^p$$

- Vector-Jacobian product (VJP) with $v^\top \in \mathbb{R}^p$

$$v^\top \partial F(x) \in \mathbb{R}^d$$

Argmin differentiation

- Consider the optimization

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

  where $f \colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}$ is twice differentiable

- $x^\star \colon \mathbb{R}^n \to \mathbb{R}^d$ is an **implicit function**
- Extensions: constrained optimization, non-smooth optimization
- How to compute the Jacobian $\partial x^\star(\theta) \in \mathbb{R}^{d \times n}$?
- Autodiff cannot be used as is: $x^\star(\theta)$ has no closed form in general

Argmin differentiation

- **Application 1:** bi-level optimization

$$\underbrace{\operatorname*{argmin}_{\theta \in \mathbb{R}^n} h(\theta) = g(x^\star(\theta))}_{\text{outer problem}} \quad \text{subject to} \quad \underbrace{x^\star(\theta) = \operatorname*{argmin}_{x \in \mathbb{R}^d} f(x, \theta)}_{\text{inner problem}}$$

  Gradient of the outer problem: $\nabla h(\theta) = \partial x^\star(\theta)^\top \nabla g(x^\star(\theta))$

  Useful in hyperparam optimization, meta-learning

- **Application 2:** “optimization as a layer”

$$\dots \to x^\star(\theta) \to \dots$$

  Can impose structure on the output via regularization or constraints

- **Application 3:** sensitivity analysis; $\partial x^\star(\theta)$ may be interesting in its own right (e.g., to answer a scientific question)

Unrolling

- Consider the sequence produced by an iterative algorithm

$$x_0(\theta), x_1(\theta), \ldots, x_K(\theta)$$

  where

$$x_k(\theta) = T(x_{k-1}(\theta), \theta)$$

- If the algorithm is convergent, $\hat{x}(\theta) = x_K(\theta)$ can be used as an approximation of $x^\star(\theta)$
- Idea: use $\partial \hat{x}(\theta)$ as an approximation of $\partial x^\star(\theta)$

Unrolling

- Pros
  - relatively simple (can use autodiff transparently)
  - derivatives $\partial \hat{x}(\theta)$ are consistent with forward pass $\hat{x}(\theta)$
- Cons
  - must reimplement the algorithm from scratch using the autodiff system (cannot reuse state-of-the-art software)
  - not all algorithms are autodiff friendly,
  - complexity scales linearly with n (forward-mode)
  - memory scales linearly with K (reverse-mode), which is especially problematic on GPU
  - the latter can be mitigated by using checkpointing, which trade-offs recomputations for smaller memory requirement

Implicit differentiation

- Use some optimality conditions to mathematically derive an expression of $\partial x^\star(\theta)$
- Examples that have been used in the past:
  - Stationary conditions
  - Karush–Kuhn–Tucker (KKT) conditions
  - Proximal gradient fixed point
- Often involves the resolution of a linear system
- So far, the derivation and implementation were case-by-case and sometimes complicated
- Not flexible: modeling changes require rederiving the expression of $\partial x^\star(\theta)$

cvxpy layers

- cvxpy: an optimization toolbox for easily formulating convex optimization problems
- Reduces all problems to linear conic programing
- cvxpy layers (Agrawal et al 2019): making cvxpy differentiable
- Uses conic programming optimality conditions to derive a formula of the Jacobian
- Pro: very general (supports any convex problem)
- Con: conic solvers are rarely the state-of-the-art for each specific problem instance

Outline

Overview

- Makes it very easy to add implicit differentiation on top of any solver (ability to reuse state-of-the-art implementations)
- The user provides (in Python) a mapping $F\colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$ capturing the optimality conditions solved by the solver
- We combine autodiff of F and implicit differentiation to automatically differentiate $x^\star(\theta)$
- Decouples the implicit differentiation mechanism from the optimality condition speficiation (in previous works, they were intertwined)
- Flexible: no mathematical derivation needed from the user, ability to experiment easily

Example: differentiating ridge regression

```python
X_tr, y_tr = load_data()

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

F = jax.grad(f)  # optimality condition

@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))
```

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 $x^\star(\theta)$ should be a **root** of F:

$$F(x^\star(\theta), \theta) = 0$$

- Implicit function theorem: $\partial x^\star(\theta)$ exists if $\partial_1 F$ is a square invertible matrix at $(x^\star(\theta), \theta)$
- Using the chain rule, we get

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

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

Differentiating a fixed point

- In many case $x^\star(\theta)$ will be a **fixed point**:

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

  where $T \colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$

- This is of course a special case since we can define

$$F(x^\star(\theta), \theta) = T(x^\star(\theta), \theta) - x^\star(\theta) = 0$$

Gradient descent

- Let $x^\star(\theta)$ be implicitly defined as

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

where $f \colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}$ is twice differentiable

- F is simply the **gradient mapping**

$$F(x, \theta) = \nabla_1 f(x, \theta)$$

- Equivalently, we can use the gradient descent fixed point

$$T(x, \theta) = x - \eta \nabla_1 f(x, \theta)$$

for any $\eta > 0$

KKT conditions

- Consider the problem

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

  where G and H can be vector-valued

- The stationarity, primal feasibility and complementary slackness conditions give

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

  where $\nu \in \mathbb{R}^q$ and $\lambda \in \mathbb{R}^r_+$ are the dual variables

- This can be written as $F(x^\star(\theta), \theta) = 0$ if we denote $x^\star(\theta) = (z^\star(\theta), \nu^\star(\theta), \lambda^\star(\theta))$

KKT conditions

- In code:

```
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
```

Quadratic programming

- Consider the QP

$$\underset{z \in \mathbb{R}^p}{\operatorname{argmin}} f(z, \theta) = \frac{1}{2} z^\top Q z + c^\top z \quad \text{s.t.} \quad \begin{aligned} H(z, \theta) &= Ez - d = 0, \\ G(z, \theta) &= Mz - h \leq 0. \end{aligned}$$

- The KKT conditions for this QP can again be written as $F(x^\star(\theta), \theta) = 0$ if we write

$$x^\star(\theta) = (z^\star(\theta), \nu^\star(\theta), \lambda^\star(\theta))$$
$$\theta = (Q, c, E, d, M, h)$$

- Just need to express f, H and G directly in Python

Proximal gradient fixed point

- Let $x^\star(\theta)$ be implicitly defined as

$$x^\star(\theta) := \underset{x \in \mathbb{R}^d}{\operatorname{argmin}} f(x, \theta) + g(x, \theta)$$

  where $g \colon \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}$ is potentially non-smooth

- We can use the **proximal gradient fixed point**

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

  where we defined the proximity operator

$$\operatorname{prox}_g(y, \theta) := \underset{x \in \mathbb{R}^d}{\operatorname{argmin}} \frac{1}{2}\|x - y\|_2^2 + g(x, \theta)$$

- Proximal operators are Lipschitz continuous and therefore differentiable almost everywhere
- Many enjoy a closed-form (soft thresholding, block soft thresholding, ...)

Proximal gradient fixed point

- In code:

```
grad = jax.grad(f)

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

Projected gradient fixed point

- Let $x^\star(\theta)$ be implicitly defined as

$$x^\star(\theta) = \underset{x \in \mathcal{C}(\theta)}{\text{argmin}}\, f(x, \theta)$$

  where $\mathcal{C}(\theta)$ is a convex set depending on θ

- We can use the **projected gradient fixed point**

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

  where we defined the Euclidean projection operator

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

- Our library provides plenty of reusable projections

Summary of optimality mappings

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

Oracles are accessed through their JVP or VJP.

Computing JVPs and VJPs

- Integrating $x^\star(\theta)$ in forward-mode autodiff requires JVPs

  To obtain the JVP Ju, solve

$$A(Ju) = Bu$$

- Integrating $x^\star(\theta)$ in reverse-mode autodiff requires VJPs

  To obtain the VJP $v^\top J$, solve

$$A^\top u = v$$

  then

$$v^\top J = u^\top AJ = u^\top B$$

Solving the linear systems

- When A is positive semi-definite, we can use conjugate gradient
- When A is indefinite, we can use GMRES or BiCGSTAB
- All algorithms only require access to A or $A^\top$ through matrix-vector products (linear maps)
- Since $A = \partial_1 F$ and $B = \partial_2 F$, we only access to JVPs or VJPs of F
- When A is indefinite, an alternative is the normal equation

$$A^\top A J = A^\top B$$

which can be solved using conjugate gradient

Features needed from an autodiff system

- JVPs and VJPs
- Second derivatives when F includes the gradient mapping $\nabla_1 f(x, \theta)$
- Custom JVPs and VJPs: this is how we are able to create `@custom_root` and `@custom_fixed_point`
- `jax.vmap`: vectorizing map (automatic batching)
- `jax.linear_transpose`: automatic transposition of linear maps

Jacobian bounds

- In practice, we almost never get $x^\star(\theta)$ and thus never solve

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

- Let $J(\hat{x}, \theta)$ be the solution of the linear system at $\hat{x}$ instead of $x^\star(\theta)$
- Under regularity conditions on $\partial_1 F$ and $\partial_2 F$, we can show (Thm 1)

$$\|J(\hat{x}, \theta) - J(x^\star(\theta), \theta)\| = \|J(\hat{x}, \theta) - \partial x^\star(\theta)\| < C\|\hat{x} - x^\star(\theta)\|$$

  i.e., J is Lipschitz

- We then apply this result to the (proximal) gradient descent fixed point under regularity conditions directly on f and prox_g (cf. corollaries 1 and 2)

Outline

Hyperparam optim of multiclass SVMs

- Goal: find hyperparameters that perform well on validation data
- $x^\star(\theta) \in \mathbb{R}^{m \times k}$: optimal dual variables
- $\theta \in \mathbb{R}_+$: regularization parameter
- bi-level optimization problem

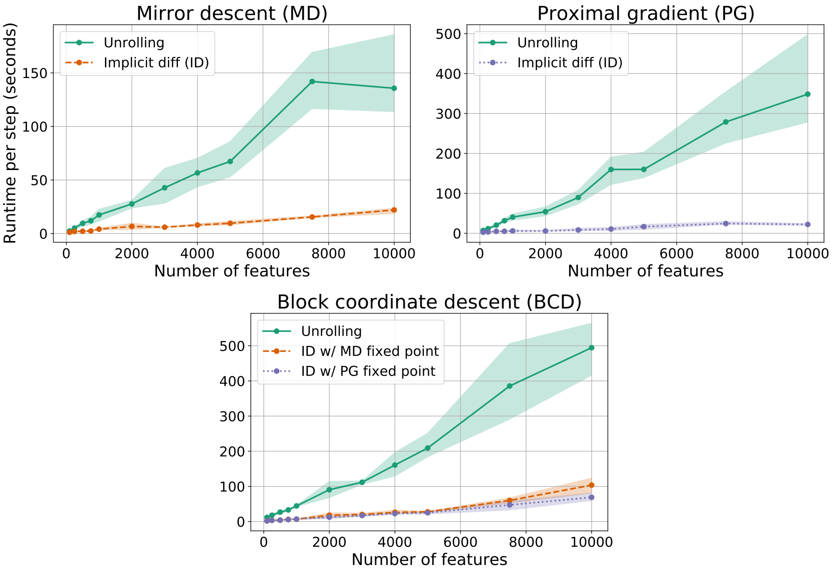
$$\underbrace{\min_{\theta = \exp(\lambda)} \frac{1}{2} \|X_{\texttt{val}} W(x^\star(\theta), \theta) - Y_{\texttt{val}}\|_F^2}_{\text{outer problem}} \text{ s.t. } \underbrace{x^\star(\theta) = \operatorname*{argmin}_{x \in \mathcal{C}} \frac{\theta}{2} \|W(x, \theta)\|_F^2 + \langle x, Y_{\texttt{tr}} \rangle}_{\text{inner problem}}$$

where

$$\mathcal{C} := \triangle^k \times \ldots \triangle^k$$

$$W(x, \theta) := X_{\texttt{tr}}^\top (Y_{\texttt{tr}} - x)/\theta \in \mathbb{R}^{p \times k}$$

Hyperparam optim of multiclass SVMs



Hyperparam optim of multiclass SVMs

```
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))
```

Task-driven dictionary learning

- Goal: breast cancer survival prediction from gene expression data
- $x^\star(\theta) \in \mathbb{R}^{m \times k}$: sparse codes (atom weights for each sample)
- $\theta \in \mathbb{R}^{k \times p}$: dictionary of k atoms
- bi-level optimization problem

$$\underbrace{\min_{\theta \in \mathbb{R}^{k \times p}, w \in \mathbb{R}^k, b \in \mathbb{R}} \sigma(x^\star(\theta) w + b; y_{\mathtt{tr}})}_{\text{outer problem}} \quad \text{s.t.} \quad x^\star(\theta) \in \underbrace{\operatorname*{argmin}_{x \in \mathbb{R}^{m \times k}} f(x, \theta) + g(x)}_{\text{inner problem}}$$

where

$$f(x, \theta) \coloneqq \ell(X_{\mathtt{tr}}, x\theta) : \text{data reconstruction error}$$

$$\sigma : \text{binary logistic loss}$$

Task-driven dictionary learning

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

- 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
- Performs better than using the original features with 100 fewer variables

Task-driven dictionary learning

```python
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)))
```

Dataset distillation

- Goal: learn a small "distilled" datataset such that a model trained on this data performs well on the original data
- $x^\star(\theta) \in \mathbb{R}^{p \times k}$: logistic regression weights
- $\theta \in \mathbb{R}^{k \times p}$: distilled images ("class prototypes")
- bi-level optimization problem

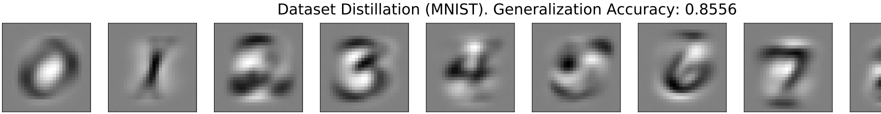
$$\underbrace{\min_{\theta \in \mathbb{R}^{k \times p}} f(x^\star(\theta), X_{\mathtt{tr}}; y_{\mathtt{tr}})}_{\text{outer problem}} \quad \text{s.t.} \quad x^\star(\theta) \in \underbrace{\operatorname*{argmin}_{x \in \mathbb{R}^{p \times k}} f(x, \theta; [k]) + \varepsilon \|x\|^2}_{\text{inner problem}}$$

where

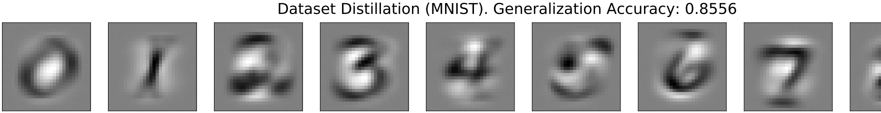
$$f(W, X; y) := \ell(y, XW)$$
$$\ell: \text{ multiclass logistic loss}$$

Dataset distillation (MNIST)

- Via implicit diff



- Via unrolling (4x slower)



Dataset distillation

```
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))
```

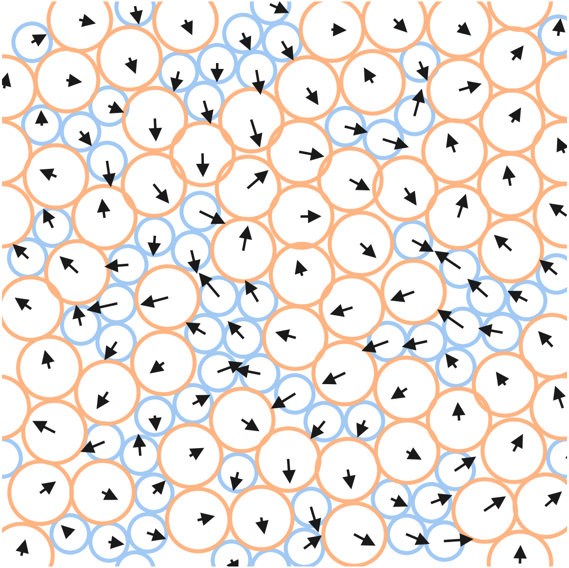
Molecular dynamics

- Goal: sensitivity analysis of molecular dynamics
- $x^\star(\theta) \in \mathbb{R}^{k \times 2}$: coordinates of k particles
- $\theta \in \mathbb{R}_+$: diameter of small particles
- optimization problem

$$x^\star(\theta) = \underset{x \in \mathbb{R}^{k \times m}}{\operatorname{argmin}} f(x, \theta) := \sum_{i,j} U(x_{i,j}, \theta)$$

where $U(x_{i,j}, \theta)$ is the pairwise potential energy function

Molecular dynamics: $\partial x^\star(\theta) \in \mathbb{R}^{k\times 2}$



Conclusion

- A general framework combining implicit differentiation with autodiff of optimality conditions
- Flexibility to try out ideas easily
- Ability to add implicit differentiation on top of existing solvers
- Arxiv preprint: https://arxiv.org/abs/2105.15183
- Open-source release: coming soon!
- Thank you for your attention!