JAXopt

Hardware accelerated (GPU/TPU), batchable and differentiable optimizers in JAX

Mathieu Blondel

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

https://github.com/google/jaxopt

Current state of optimization in SciPy

- **scipy.optimize**: Optimization, root finding, line search algorithms
- Small annoyances (e.g., no support for arbitrary parameter shapes)
- CPU only
  - Implementations are in Python, C / C++, FORTRAN, Cython
- Not built with autodiff in mind (gradients must be provided)
- API incompatible with argmin differentiation

Current state of optimization in JAX

- **Batch optimization**
  - **jax.scipy.optimize**
    - JAX port of a few algorithms from scipy.optimize (BFGS, L-BFGS)
    - Needs to maintain API compatibility with scipy.optimize
    - Not differentiable (neither via unrolling nor via implicit differentiation)

- **Stochastic optimization**
  - **jax.experimental.optimizers, flax.optim, optax**
    - Focus on stochastic optimization
    - Implicit differentiation is not supported

Project vision

- Goal: answer most modern optimization needs of ML and DL users
  - Stochastic optimization of DL models together with Flax or Haiku
  - Constrained and non-smooth optimization
  - Differentiable optimizers / argmin differentiation
    - Bi-level optimization (hyperparameter optimization, meta-learning, robust learning)
    - Optimization layers (structured attention, implicit deep learning, …)

- Leverage JAX's **idiomatic features**
  - Autodiff at the heart of all our design decisions
  - **Hardware acceleration** (pmap, pjit) and **automatic batching** (vmap)

- API designed from the ground up (not necessarily compatible with scipy.optimize)

Basic API

- **User-provided objective function**
  - scalar_value = objective_fun(params, *args, **kwargs)

- **Core methods**
  - **Constructor:** solver = SolverClass(fun=objective_fun, maxiter=1000, …)
  - **Initialization:** params, state = solver.init(init_params, *args, **kwargs)
  - **Performing one iteration:** params, state = solver.update(params, state, *args, **kwargs)

- **Optimization loop methods**
  - **Batch setting:** params, state = solver.run(init_params, *args, **kwargs)
  - **Stochastic setting:** params, state = solver.run_iterator(init_params, iterator, *args, **kwargs)

Batch optimization example

```
def objective_fun(params, l2reg, X, y):
  residuals = jnp.dot(X, params) - y
  return 0.5 * jnp.mean(residuals ** 2) + 0.5 * l2reg * jnp.sum(params ** 2)

solver = GradientDescent(fun=objective_fun, maxiter=100)
init_params = jnp.zeros(X.shape[1])

# loop taken care of by JAXopt
params, state = solver.run(init_params, l2reg, X, y)

# manual loop
params, state = solver.init(init_params)
for _ in range(solver.maxiter):
  params, state = solver.update(params, state, l2reg, X, y)
```

Stochastic optimization example

```
def objective_fun(params, l2reg, data):
  X, y = data
  residuals = jnp.dot(X, params) - y
  return 0.5 * jnp.mean(residuals ** 2) + 0.5 * l2reg * jnp.sum(params ** 2)

solver = OptaxSolver(opt=optax.adam(1e-3), fun=loss_fun, ...)
# solver = PolyakSGD(fun=loss_fun, ...)

# loop taken care of by JAXopt
params, state = solver.run_iterator(init_params, iterator, l2reg=l2reg)

# manual loop
params, state = solver.init(init_params)
for data in iterator:
  params, state = solver.update(params, state, l2reg=l2reg, data=data)
```

JAXopt's current features

- Batch optimization
  - Gradient descent
  - Projected gradient and numerous projection operators
  - Proximal gradient and some proximal operators
  - Block coordinate descent
  - Mirror descent
  - Quadratic programming
  - SciPy wrapper (with pytree and implicit diff support)
- Stochastic optimization
  - Optax wrapper
  - SGD with Polyak adaptive step size
- Root finding
  - Bisection
  - SciPy Wrapper
- Argmin differentiation via unrolling or implicit differentiation

Implicit differentiation out-of-the-box

- Applications
  - Bi-level optimization (hyperparameter optimization, meta-learning, robust learning)
  - Optimization layers (structured attention, implicit deep learning, pathways, expert mixtures)
  - Sensitivity analysis

```
def objective_fun(params, l2regul, X, y):
  residuals = jnp.dot(X, params) - y
  return 0.5 * jnp.mean(residuals ** 2) + 0.5 * l2regul * jnp.sum(params ** 2)

def argmin_solution(l2regul, X, y):
  solver = GradientDescent(fun=objective_fun, maxiter=500, implicit_diff=True)
  init_params = jnp.zeros(X.shape[1])
  return solver.run(init_params, l2regul, X, y).params

# Jacobian w.r.t. l2regul of argmin_solution
print(jax.jacobian(argmin_solution)(l2regul, X, y))
```

Implicit differentiation of custom solvers

- Decorators @custom_root and @custom_fixed_point make it **easy** to add implicit differentiation on top of **existing solvers** (seamless integration with JAX's autodiff)

```
def objective_fun(params, l2reg): # objective function
  residual = jnp.dot(X_tr, params) - y_tr
  return (jnp.sum(residual ** 2) + l2reg * jnp.sum(params ** 2)) / 2

optimality_fun = jax.grad(objective_fun) # optimality condition

@custom_root(optimality_fun)
def ridge_solver(init_params, l2reg):
  del init_params # Initialization not used in this solver
  XX = jnp.dot(X_tr.T, X_tr)
  Xy = jnp.dot(X_tr.T, y_tr)
  I = jnp.eye(X_tr.shape[1])
  return jnp.linalg.solve(XX + l2reg * I, Xy)

print(jax.jacobian(ridge_solver, argnums=1)(None, 10.0))
```

Implicit differentiation in JAXopt: how does it work?

- Let $F: \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$ be a user-provided capturing **optimality conditions**

- Let $x^*(\theta)$ be a **root** of F: $F(x^*(\theta), \theta) = 0$

- From the **implicit function theorem**, the Jacobian $\partial x^*(\theta)$ is given by solving the following linear system of equations:

  $$-\partial_1 F(x^*(\theta), \theta)\, \partial x^*(\theta) = \partial_2 F(x^*(\theta), \theta)$$

- We combine the implicit function theorem with **autodiff** of F

ArXiv preprint: https://arxiv.org/abs/2105.15183

Implicit differentiation: simplest example

- We want to differentiate an **unconstrained optimization** problem solution:

  $x^*(\theta) = \text{argmin}_x f(x, \theta)$

- Optimality condition: $\nabla_1 f(x, \theta) = 0$

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

- $\partial_1 F(x, \theta) = \nabla_1^2 f(x, \theta)$ is the Hessian
- $\partial_2 F(x, \theta) = \partial_2 \nabla_1 f(x, \theta)$ is the cross-derivative

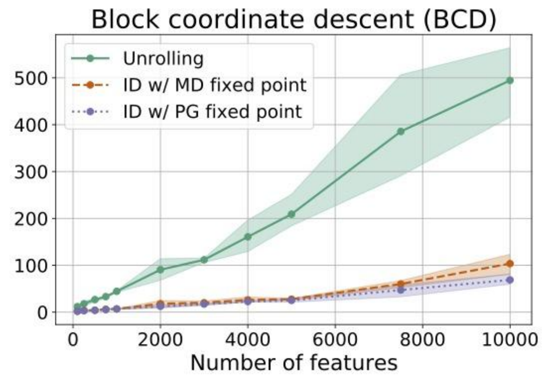
Large catalog of optimality conditions

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	$\text{proj}_{\mathbb{R}^p \times \mathcal{K}^* \times \mathbb{R}_+}$

ArXiv preprint: https://arxiv.org/abs/2105.15183

Google Research

Hyperparameter optimization of multiclass SVMs

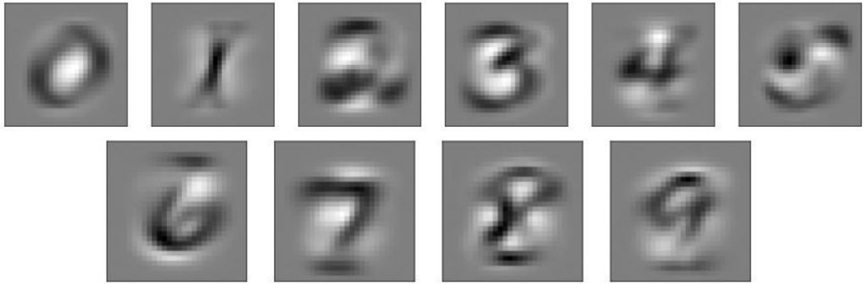


Task-driven dictionary learning

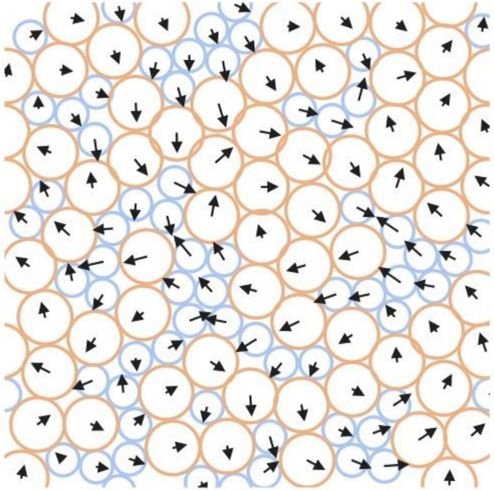
Table 2: Mean AUC (and 95% confidence interval) for the cancer survival prediction problem.

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

Dataset distillation



Sensitivity analysis of molecular dynamics



ArXiv preprint: https://arxiv.org/abs/2105.15183

Google Research

Conclusion

- Hardware accelerated, batchable and differentiable optimizers
- Implicit differentiation out-of-the-box for JAXOpt solvers
- Implicit differentiation for custom solvers thanks to decorators
- We are open-source!
  https://github.com/google/jaxopt
- We're growing fast! Lots of on-going work!