GradHpO

Altay Eynullayev, Rubtsov Denis, Karpeev Gleb

Bayesian Multi-Modeling, Intelligent Systems, MIPT

2026

T1-T2 with DARTS discretization

T_1- training dataset, T_2- validation dataset, $\theta-$ model parameters, $\lambda-$ regularization hyperparameters, $C_1(\theta|\lambda, T_1)-$ training loss of the following form:

$$C_1(\theta|\lambda.T_1) = C(\theta|T_1) + \Omega(\theta, \lambda),$$

$C_2(\theta, T_2)-$ validation loss.
Optimizang parameters:

$$\theta_{t+1} = \theta_t + \eta_1 \nabla_\theta C_1(\theta_t|\lambda_t.T_1),$$

Optimizing hyperparameters:

$$\lambda_{t+1} = \lambda_t + \eta_2 \nabla_\lambda C_2(\theta_{t+1}, T_2) =$$

$$= \lambda_t + \eta_2 \eta_1 \nabla_\theta C_2(\theta_{t+1}, T_2) \nabla_\lambda \nabla_\theta C_1(\theta_t|\lambda_t, T_1).$$

T1-T2 with DARTS discretization

Using DARTS discretization:

$$\nabla_\theta C_2(\theta_{t+1}, T_2)\nabla_\lambda \nabla_\theta C_1(\theta_t|\lambda_t, T_1) \approx$$

$$\approx \frac{\nabla_\lambda C_1(\theta^+|\lambda, T_1) - \nabla_\lambda C_1(\theta^-|\lambda, T_1)}{2\varepsilon},$$

where $\theta^\pm = \theta \pm \varepsilon \nabla_\theta C_2(\theta_{t+1}, T_2)$, we get:

$$\lambda_{t+1} \approx \lambda_t + \eta_2\eta_1 \frac{\nabla_\lambda C_1(\theta^+|\lambda, T_1) - \nabla_\lambda C_1(\theta^-|\lambda, T_1)}{2\varepsilon}.$$

Generalized Greedy Gradient-Based HPO

Algorithm Idea

- **Problem:** Optimize high-dimensional hyperparameters $\lambda \in \mathbb{R}^d$ using validation loss $L_{\text{val}}(\lambda)$.
- **Greedy step:** At each iteration, update only a subset of hyperparameters that yield the largest improvement.
- **Gradient info:** Use hypergradients $\nabla_\lambda L_{\text{val}}$ to rank candidates.
- **Generalization:** Handles constraints, mixed continuous/discrete, and non-smooth objectives.

$$\lambda_{t+1} = \lambda_t - \eta \cdot \mathbf{1}_{\mathcal{S}_t} \odot \nabla_\lambda L_{\text{val}}$$

where $\mathcal{S}_t$ is the selected subset (e.g., top-k by gradient magnitude).

HyperDistill (Lee et al., ICLR 2022)

Setting. Optimize high-dim. hyperparameters $\boldsymbol{\lambda}$ (e.g. per-layer learning rates, warp layers) via $\min_{\boldsymbol{\lambda}} \mathcal{L}_{\mathrm{val}}(\mathbf{w}_T(\boldsymbol{\lambda}), \boldsymbol{\lambda})$.
Hypergradient decomposition:

$$\frac{d\mathcal{L}_{\mathrm{val}}}{d\boldsymbol{\lambda}} = \underbrace{\frac{\partial\mathcal{L}_{\mathrm{val}}}{\partial\boldsymbol{\lambda}}}_{g^{\mathrm{FO}}} + \underbrace{\frac{\partial\mathcal{L}_{\mathrm{val}}}{\partial\mathbf{w}_T}\frac{d\mathbf{w}_T}{d\boldsymbol{\lambda}}}_{g^{\mathrm{SO}}\text{: costly}}$$

Key idea - distill g^{SO} into one JVP

No existing method is simultaneously: high-dim, online, constant memory, horizon > 1 HyperDistill approximates $g_t^{\mathrm{SO}} \approx \pi_t^* \cdot f(\mathbf{w}_t^*, \mathcal{D}_t^*)$ where

- $f(\mathbf{w}, \mathcal{D}) = \sigma\left(\boldsymbol{\alpha}_t \frac{\partial\Phi(\mathbf{w},\boldsymbol{\lambda};\mathcal{D})}{\partial\boldsymbol{\lambda}}\right)$ — single normalized JVP
- $\mathbf{w}_t^* \leftarrow p_t\, \mathbf{w}_{t-1}^* + (1 - p_t)\, \mathbf{w}_{t-1}$ — running avg. of past weights
- $\pi_t^* = c_\gamma(t;\theta)$ — scalar estimated by a cheap linear regressor

Result: one JVP per step; $\gamma \to 0$ recovers 1-step

Architecture & Core Components

Design Philosophy

State-based approach with **unified API** for all algorithms — all optimizers implement `init()`, `step()`, and `compute_hypergradient()` methods

Directory Structure

`gradhpo/core/` — `state.py` (BilevelState), `base.py` (BilevelOptimizer), `types.py`;
`gradhpo/algorithms/` — `t1_t2.py`, `greedy.py`, `online.py`;
`gradhpo/utils/` — gradient utilities, validation, logging;
`gradhpo/examples/` — MNIST, learning rate optimization, data augmentation

Core Classes

BilevelState — encapsulates `params`, `hyperparams`, optimizer states, step counter, and metadata (losses, gradient norms);
BilevelOptimizer — abstract base class with three abstract methods for all algorithms

Dependencies & Integration

Core Dependencies

JAX $\geq$ 0.4.0 — automatic differentiation, JIT compilation, XLA acceleration;
Optax $\geq$ 0.1.7 — gradient transformations and optimization algorithms;
Chex $\geq$ 0.1.8 — testing utilities for JAX (assertions, variants);
NumPy $\geq$ 1.24.0 — array operations and compatibility

Integration Strategy

Optax-compatible — seamless integration with gradient transformations
JAX-first design — pure functions, JIT-compatible, supports `vmap`/`pmap`
Type-annotated — full typing with `PyTree`, `LossFn`, `DataBatch`

Key Features

Unified **BilevelState** for checkpointing; Functional API for all algorithms; Extensible base class for custom methods

Technology Stack

Testing

pytest + **pytest-cov** — unit and integration tests with coverage reports uploaded to **Codecov** via GitHub Actions

Documentation

Sphinx + **sphinx_rtd_theme** — auto-generated API docs deployed to **GitHub Pages** on every push to `master`

Code style

flake8 — PEP 8 compliance checked automatically in CI

Core framework & packaging

JAX — functional transformations, `jit`/`grad`/`vmap`, XLA acceleration; **setuptools** — standard `setup.py` packaging

Motivation: Scale HPO to billion-dim spaces; gradient methods lack unified API.
Goal: Practical framework for short-horizon (T1-T2/DARTS) hypergradient methods.
Outcomes: Prototype on MNIST/CIFAR-10, comparison vs random search, scalability demo.