GradHpO

Eynullayev Altay, Rubstsov Denis, Karpeev Gleb

February 26, 2026

Abstract

This document presents the architecture and design of `gradhpo`, a JAX-based library for short-horizon gradient-based hyperparameter optimization. The library implements four state-of-the-art algorithms: T1-T2 with DARTS approximation, Greedy gradient-based optimization and Online hyperparameter meta-learning with hypergradient distillation. The design follows a state-based approach with a unified API for all algorithms.

Contents

1 Architecture

1.1 Directory Structure

Path	Description
gradhpo/src	
gradhpo/	Main package directory
core/	Core components
state.py	BilevelState class
base.py	Base optimizer class
types.py	Type definitions
algorithms/	Algorithm implementations
t1_t2.py	T1-T2 with DARTS
greedy.py	Greedy gradient-based
online.py	Online hypergradient distillation
implicit_diff.py	Implicit differentiation
utils/	Utility functions
gradients.py	Gradient utilities
validation.py	Input validation
logging.py	Logging utilities
examples/	Example scripts
tests/	Test suite
docs/	Documentation
setup.py	Package setup script
requirements.txt	Dependencies
README.md	Project readme

Table 1: Project directory structure

2 Core Components

2.1 Type Definitions

2.1.1 Core Types (gradhpo/core/types.py)

```python
from typing import Any, Callable, Dict, NamedTuple, Protocol
import jax.numpy as jnp
from jax import Array

# Type aliases
PyTree = Any  # JAX PyTree (nested structure of arrays)
LossFn = Callable[[PyTree, PyTree, Any], Array]
MetricDict = Dict[str, Union[float, Array]]

class DataBatch(NamedTuple):
    """Structure for a data batch.

    Attributes:
        inputs: Input features [batch_size, ...]
        targets: Target labels [batch_size, ...]
    """
    inputs: Array
    targets: Array

class LossFunctions(NamedTuple):
    """Container for loss functions.

    Attributes:
        train_loss: Training loss function
```

```
        val_loss: Validation loss function
    """
    train_loss: LossFn
    val_loss: LossFn
```

2.2 Bilevel State

2.2.1 `BilevelState` Class (`gradhpo/core/state.py`)

The `BilevelState` class maintains all necessary state information for bilevel optimization.

```
@dataclass
class BilevelState:
    """State container for bilevel optimization process.

    Attributes:
        params: Model parameters (inner level)
        hyperparams: Hyperparameters to optimize (outer level)
        inner_opt_state: State of inner optimizer
        outer_opt_state: State of outer optimizer
        step: Current optimization step
        metadata: Additional information (losses, norms, etc.)
    """
    params: PyTree
    hyperparams: PyTree
    inner_opt_state: OptState
    outer_opt_state: OptState
    step: int
    metadata: Dict[str, Any]
```

Key Methods:

```
@classmethod
def create(cls,
           params: PyTree,
           hyperparams: PyTree,
           inner_opt_state: OptState,
           outer_opt_state: OptState) -> 'BilevelState':
    """Create a new BilevelState with initial values.

    Args:
        params: Initial model parameters
        hyperparams: Initial hyperparameters
        inner_opt_state: Initial inner optimizer state
        outer_opt_state: Initial outer optimizer state

    Returns:
        Initialized state object
    """

def update(self,
           params: Optional[PyTree] = None,
           hyperparams: Optional[PyTree] = None,
           inner_opt_state: Optional[OptState] = None,
           outer_opt_state: Optional[OptState] = None,
           metadata: Optional[Dict[str, Any]] = None) -> 'BilevelState':
    """Create updated state with new values.

    Args:
        params: New parameters (if None, keep current)
        hyperparams: New hyperparameters (if None, keep current)
        inner_opt_state: New inner optimizer state
        outer_opt_state: New outer optimizer state
        metadata: New metadata to merge
```

```
    Returns:
        New state object with updates
    """

def get_metric(self, key: str, default: Any = None) -> Any:
    """Retrieve a metric from metadata.

    Args:
        key: Metric name
        default: Default value if key not found

    Returns:
        Metric value or default
    """
```

2.3 Base Optimizer

2.3.1 `BilevelOptimizer` Class (`gradhpo/core/base.py`)

Abstract base class for all bilevel optimization algorithms.

```
class BilevelOptimizer(ABC):
    """Abstract base class for bilevel hyperparameter optimizers.

    All algorithms inherit from this class and implement
    the required abstract methods.

    Attributes:
        inner_optimizer: Optax optimizer for parameters
        outer_optimizer: Optax optimizer for hyperparameters
    """

    def __init__(self,
                 inner_optimizer: optax.GradientTransformation,
                 outer_optimizer: optax.GradientTransformation):
        """Initialize bilevel optimizer."""
```

Abstract Methods:

```
@abstractmethod
def init(self,
         params: PyTree,
         hyperparams: PyTree) -> BilevelState:
    """Initialize the bilevel optimization state.

    Args:
        params: Initial model parameters
        hyperparams: Initial hyperparameters

    Returns:
        Initial optimization state
    """

@abstractmethod
def step(self,
         state: BilevelState,
         train_batch: DataBatch,
         val_batch: DataBatch,
         train_loss_fn: LossFn,
         val_loss_fn: LossFn) -> BilevelState:
    """Perform one bilevel optimization step.

    Args:
```

```
        state: Current bilevel state
        train_batch: Training data batch
        val_batch: Validation data batch
        train_loss_fn: Training loss function
        val_loss_fn: Validation loss function

    Returns:
        Updated state
    """

@abstractmethod
def compute_hypergradient(self,
                          state: BilevelState,
                          train_batch: DataBatch,
                          val_batch: DataBatch,
                          train_loss_fn: LossFn,
                          val_loss_fn: LossFn) -> PyTree:
    """Compute hypergradient w.r.t. hyperparameters.

    Args:
        state: Current state
        train_batch: Training batch
        val_batch: Validation batch
        train_loss_fn: Training loss
        val_loss_fn: Validation loss

    Returns:
        Hypergradient with same structure as hyperparams
    """
```

3 Algorithm Implementations

3.1 T1-T2 with DARTS

3.1.1 `T1T2Optimizer` Class (`gradhpo/algorithms/t1_t2.py`)

Implements T1-T2 algorithm using DARTS-style finite difference approximation.

```
class T1T2Optimizer(BilevelOptimizer):
    """T1-T2 algorithm with DARTS approximation.

    Attributes:
        inner_optimizer: Optimizer for model parameters
        outer_optimizer: Optimizer for hyperparameters
        inner_steps: Number of inner loop steps (K)
        epsilon: Step size for finite difference approximation
    """

    def __init__(self,
                 inner_optimizer: optax.GradientTransformation,
                 outer_optimizer: optax.GradientTransformation,
                 inner_steps: int = 1,
                 epsilon: float = 0.01):
        """Initialize T1-T2 optimizer.

        Args:
            inner_optimizer: Optax optimizer for parameters
            outer_optimizer: Optax optimizer for hyperparameters
            inner_steps: Number of inner optimization steps
            epsilon: Finite difference epsilon
        """
```

Key Methods:

```
def init(self,
         params: PyTree,
         hyperparams: PyTree) -> BilevelState:
    """Initialize T1-T2 optimization state."""

@jit
def step(self,
         state: BilevelState,
         train_batch: DataBatch,
         val_batch: DataBatch,
         train_loss_fn: LossFn,
         val_loss_fn: LossFn) -> BilevelState:
    """Perform one T1-T2 optimization step.

    Algorithm:
        1. Update parameters on training loss (inner loop)
        2. Compute hypergradient using DARTS approximation
        3. Update hyperparameters
    """

def compute_hypergradient(self,
                          state: BilevelState,
                          train_batch: DataBatch,
                          val_batch: DataBatch,
                          train_loss_fn: LossFn,
                          val_loss_fn: LossFn) -> PyTree:
    """Compute hypergradient using finite difference.

    Uses DARTS-style approximation from Section 2.3.
    """
```

3.2 Greedy Gradient-Based

3.2.1 `GreedyOptimizer` Class (`gradhpo/algorithms/greedy.py`)

Implements generalized greedy gradient-based hyperparameter optimization.

```
class GreedyOptimizer(BilevelOptimizer):
    """Greedy gradient-based hyperparameter optimization.

    Attributes:
        inner_optimizer: Optimizer for parameters
        outer_optimizer: Optimizer for hyperparameters
        unroll_steps: Number of steps to unroll
    """

    def __init__(self,
                 inner_optimizer: optax.GradientTransformation,
                 outer_optimizer: optax.GradientTransformation,
                 unroll_steps: int = 1):
        """Initialize Greedy optimizer.

        Args:
            inner_optimizer: Optimizer for parameters
            outer_optimizer: Optimizer for hyperparameters
            unroll_steps: Number of inner steps to unroll
        """
```

Key Methods:

```
def init(self,
         params: PyTree,
         hyperparams: PyTree) -> BilevelState:
    """Initialize Greedy optimization state."""
```

```

@jit
def step(self,
         state: BilevelState,
         train_batch: DataBatch,
         val_batch: DataBatch,
         train_loss_fn: LossFn,
         val_loss_fn: LossFn) -> BilevelState:
    """Perform one greedy optimization step."""

def compute_hypergradient(self,
                          state: BilevelState,
                          train_batch: DataBatch,
                          val_batch: DataBatch,
                          train_loss_fn: LossFn,
                          val_loss_fn: LossFn) -> PyTree:
    """Compute hypergradient by unrolling inner optimization."""
```

3.3 Online with Hypergradient Distillation

3.3.1 `OnlineHypergradientOptimizer` Class (`gradhpo/algorithms/online.py`)

Implements online hyperparameter meta-learning with hypergradient distillation.

```
class OnlineHypergradientOptimizer(BilevelOptimizer):
    """Online optimization with hypergradient distillation.

    Attributes:
        inner_optimizer: Optimizer for parameters
        outer_optimizer: Optimizer for hyperparameters
        distillation_weight: Weight for distillation loss
        memory_size: Number of past hypergradients to store
    """

    def __init__(self,
                 inner_optimizer: optax.GradientTransformation,
                 outer_optimizer: optax.GradientTransformation,
                 distillation_weight: float = 0.1,
                 memory_size: int = 10):
        """Initialize Online Hypergradient optimizer.

        Args:
            inner_optimizer: Optimizer for parameters
            outer_optimizer: Optimizer for hyperparameters
            distillation_weight: Weight for distillation term
            memory_size: Size of hypergradient memory buffer
        """
```

Key Methods:

```
def init(self,
         params: PyTree,
         hyperparams: PyTree) -> BilevelState:
    """Initialize state with hypergradient memory."""

@jit
def step(self,
         state: BilevelState,
         train_batch: DataBatch,
         val_batch: DataBatch,
         train_loss_fn: LossFn,
         val_loss_fn: LossFn) -> BilevelState:
    """Perform one online optimization step with distillation."""

```

```
def compute_hypergradient(self,
                          state: BilevelState,
                          train_batch: DataBatch,
                          val_batch: DataBatch,
                          train_loss_fn: LossFn,
                          val_loss_fn: LossFn) -> PyTree:
    """Compute hypergradient using short-horizon approximation."""
```

4 Libraries and Dependencies

Library	Version
JAX	$\geq 0.4.0$
jaxlib	$\geq 0.4.0$
optax	$\geq 0.1.7$
chex	$\geq 0.1.8$
typing-extensions	$\geq 4.5.0$
numpy	$\geq 1.24.0$

Table 2: Core dependencies

4.1 Integration

- **JAX**: Core framework for automatic differentiation and JIT compilation
- **Optax**: Provides gradient transformation and optimization algorithms
- **Chex**: Testing utilities for JAX code (assertions, variants)
- **NumPy**: Array operations and compatibility layer