# Betty: An Automatic Differentiation Library for Multilevel Optimization

**Sang Keun Choe**[1], **Willie Neiswanger**[2], **Pengtao Xie**[3*], **Eric Xing**[1*]
Carnegie Mellon University[1], Stanford University[2], University of California San Diego[3]
{sangkeuc, epxing}@cs.cmu.edu, neiswanger@cs.stanford.edu, p1xie@ucsd.edu

## Abstract

Multilevel optimization has been widely adopted as a mathematical foundation for a myriad of machine learning problems, such as hyperparameter optimization, meta-learning, and reinforcement learning, to name a few. Nonetheless, implementing multilevel optimization programs oftentimes requires expertise in both mathematics and programming, stunting research in this field. We take an initial step towards closing this gap by introducing BETTY, a high-level software library for gradient-based multilevel optimization. To this end, we develop an automatic differentiation procedure based on a novel interpretation of multilevel optimization as a dataflow graph. We further abstract the main components of multilevel optimization as Python classes, to enable easy, modular, and maintainable programming. We empirically demonstrate that BETTY can be used as a high-level programming interface for an array of multilevel optimization programs, while also observing up to 11% increase in test accuracy, 14% decrease in GPU memory usage, and 20% decrease in wall time over existing implementations on multiple benchmarks. The code is available at `https://github.com/leopard-ai/betty`.

## 1 Introduction

Multilevel optimization (MLO) addresses nested optimization scenarios, where upper level optimization problems are constrained by lower level optimization problems following an underlying hierarchical dependency. MLO has gained considerable attention as a unified mathematical framework for studying diverse problems including meta-learning [12, 34], hyperparameter optimization [13, 14, 30], neural architecture search [27, 47], and reinforcement learning [23, 26, 35]. While a majority of existing work is built upon bilevel optimization, the simplest case of MLO, there have been recent efforts that go beyond this two-level hierarchy. For example, [33] proposed trilevel optimization that combines hyperparameter optimization with two-level pretraining and finetuning. More generally, conducting joint optimization over machine learning pipelines consisting of multiple models and hyperparameter sets can be approached as deeper instances of MLO [16, 33, 39, 40].

Following its ever-increasing popularity, a multitude of optimization algorithms have been proposed to solve MLO. Among them, *gradient-based (or first-order)* approaches [29, 33, 37] have recently received the limelight from the machine learning community, due to their ability to carry out efficient high-dimensional optimization, under which all of the above listed applications fall. Nevertheless, implementing gradient-based MLO algorithms oftentimes requires both mathematical and programming proficiency [2, 19], which raises a major barrier to MLO research. While such issues could be significantly alleviated through automatic differentiation techniques (*e.g.* backpropagation for neural networks) that abstract away complicated mathematics and implementation details, minimal efforts in this direction have been made to date.

*co-corresponding authors

Preprint. Under review.

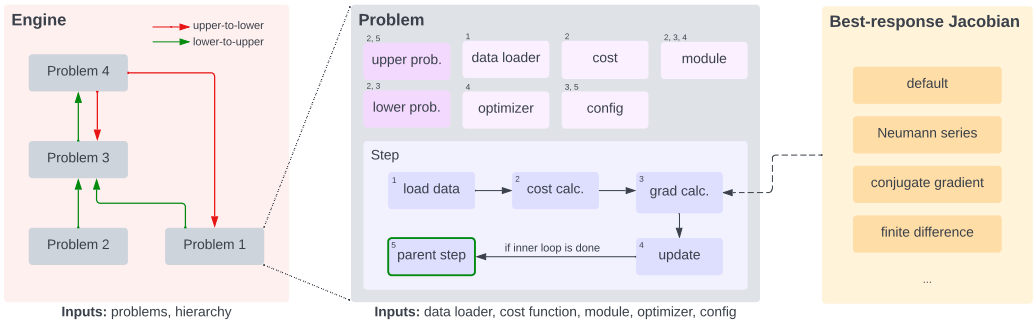


Figure 1: In `Engine` (left), users define their MLO program as a hierarchy/graph of optimization problems. In `Problem` (middle), users define an optimization problem with a data loader, cost function, module, and optimizer, while upper/lower level constraint problems (*i.e.* $\mathcal{U}_k, \mathcal{L}_k$) are injected by `Engine`. The "step" function in `Problem` serves as the base of gradient-based optimization, and we denote which attributes are involved in each part with numeric superscripts. Finally, best-response Jacobian calculation (right) is decoupled from `Problem` & `Engine` to allow a modular ability to choose different methods per users' needs.

In recent years, there has been some work originating in the meta-learning community on developing software libraries that target some aspects of gradient-based MLO [2, 8, 19]. These libraries can be considered *low-level*, as their main goal is to provide basic functionalities (*e.g.* making PyTorch's [32] native optimizers differentiable) to enable gradient calculations in MLO. While they allow some flexibility to build higher-level concepts by combining provided basic functionalities, they lack certain major advantages of low-level libraries—for example better memory efficiency and execution speed—as these aspects are managed by underlying frameworks (*e.g.* PyTorch [32]) that these libraries are built upon. Notably, MLO has several high-level concepts, such as the optimization problems and the hierarchical dependency among them, that are not supported by existing frameworks. Consequently, having to implement such high-level MLO concepts with existing low-level libraries renders resulting code difficult to read, write, debug, and maintain.

In this paper, we attempt to bridge the above-mentioned gap between research and software systems by introducing BETTY, an easy-to-use, modular, and maintainable automatic differentiation library for multilevel optimization. In detail, the main contributions of this paper are as follows:

1. We develop a general automatic differentiation technique for MLO based on the novel interpretation of MLO as a special type of dataflow graph (Section 3.1). In detail, gradient calculation for each optimization problem is carried out by iteratively multiplying best-response Jacobian (defined in Section 2) while reverse-traversing specific paths of this dataflow graph. This reverse-traversing procedure is crucial for efficiency, as it reduces the computational complexity of our automatic differentiation technique from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$. Furthermore, we implement and provide several popular algorithms for best-response Jacobian calculation (*e.g.* implicit differentiation with finite difference [27] or Neumann series [29]) for a user's convenience (Section 3.2).
2. We develop a software library for MLO, BETTY, built upon the aforementioned automatic differentiation technique (Section 4). In particular, BETTY provides two Python classes, `Problem` and `Engine`, with which users can respectively define optimization problems and the hierarchical dependency of a MLO program. Such a design abstracts away the complicated internal mechanism of automatic differentiation behind the API, allowing users without mathematical and programming expertise to easily write MLO code in a modular and maintainable fashion. The architecture of BETTY is shown in Figure 1.
3. We empirically demonstrate that BETTY can be used as a high-level programming interface for a variety of MLO applications by re-implementing several popular MLO benchmarks (Section 5). Interestingly, we observe that trying out different best-response Jacobian algorithms with our modular interface (which only requires changing one line of code) can lead to up to 11% increase in test accuracy, 14% decrease in GPU memory usage, and 20% decrease in wall time on various benchmarks, compared with the original papers' implementations.

## 2 Background: Gradient-based Multilevel Optimization

To introduce MLO, we first define an important concept known as a "constrained problem" [43].

**Definition 1.** *An optimization problem $P$ is said to be **constrained** by $\lambda$ when its cost function $\mathcal{C}$ has $\lambda$ as an argument in addition to the optimization parameter $\theta$ (i.e. $P : \arg\min_\theta \mathcal{C}(\theta, \lambda, \cdots)$).*

Multilevel optimization [31] refers to a field of study that aims to solve a nested set of optimization problems defined on a sequence of so-called *levels*, which satisfy two main criteria: **A1)** upper-level problems are constrained by the *optimal* parameters of lower-level problems while **A2)** lower-level problems are constrained by the *nonoptimal* parameters of upper-level problems. Formally, an $n$-level MLO program can be written as:

$$
\begin{aligned}
P_n: \quad & \theta_n^* = \operatorname*{argmin}_{\theta_n} \mathcal{C}_n(\theta_n, \mathcal{U}_n, \mathcal{L}_n; \mathcal{D}_n) && \triangleright \text{ Level } n \text{ problem} \\
& \ddots \\
P_k: \quad & \text{s.t. } \theta_k^* = \operatorname*{argmin}_{\theta_k} \mathcal{C}_k(\theta_k, \mathcal{U}_k, \mathcal{L}_k; \mathcal{D}_k) && \triangleright \text{ Level } k \in \{2, \ldots, n-1\} \text{ problem} \\
& \qquad \ddots \\
P_1: \quad & \qquad \text{s.t. } \theta_1^* = \operatorname*{argmin}_{\theta_1} \mathcal{C}_1(\theta_1, \mathcal{U}_1, \mathcal{L}_1; \mathcal{D}_k) && \triangleright \text{ Level 1 problem}
\end{aligned}
$$

where, $P_k$ stands for the level $k$ problem, $\theta_k$ / $\theta_k^*$ for corresponding nonoptimal / optimal parameters, and $\mathcal{U}_k$ / $\mathcal{L}_k$ for the sets of constraining parameters from upper / lower level problems. Here, $\mathcal{D}_k$ is the training dataset, and $\mathcal{C}_k$ indicates the cost function. Due to criteria **A1** & **A2**, constraining parameters from upper-level problems should be nonoptimal (*i.e.* $\mathcal{U}_k \subseteq \{\theta_{k+1}, \cdots, \theta_n\}$) while constraining parameters from lower-level problems should be optimal (*i.e.* $\mathcal{L}_k \subseteq \{\theta_1^*, \cdots, \theta_{k-1}^*\}$). Although we denote only one optimization problem per level in the above formulation, each level could in fact have multiple problems. Therefore, we henceforth discard the concept of level, and rather assume that problems $\{P_1, P_2, \cdots, P_n\}$ of a general MLO program are topologically sorted in a "reverse" order (*i.e.* $P_n$ / $P_1$ denote uppermost / lowermost problems).

For example, in hyperparameter optimization formulated as bilevel optimization, hyperparameters and network parameters correspond to upper and lower level parameters ($\theta_2$ and $\theta_1$). Train / validation losses correspond to $\mathcal{C}_1$ / $\mathcal{C}_2$, and validation loss is dependent on optimal network parameters $\theta_1^*$ obtained given $\theta_2$. Thus, constraining sets for each level are $\mathcal{U}_1 = \{\theta_2\}$ and $\mathcal{L}_2 = \{\theta_1^*\}$.

In this paper, we focus in particular on *gradient-based* MLO, rather than zeroth-order methods like Bayesian optimization [7], in order to efficiently scale to high-dimensional problems. Essentially, gradient-based MLO calculates gradients of the cost function $\mathcal{C}_k(\theta_k, \mathcal{U}_k, \mathcal{L}_k)$ with respect to the corresponding parameter $\theta_k$, with which gradient descent is performed to solve for optimal parameters $\theta_k^*$ for every problem $P_k$. Since optimal parameters from lower level problems (*i.e.* $\theta_l^* \in \mathcal{L}_k$) can be functions of $\theta_k$ (criterion **A2**), $\frac{d\mathcal{C}_k}{d\theta_k}$ can be expanded using the chain rule as follows:

$$
\frac{d\mathcal{C}_k}{d\theta_k} = \underbrace{\frac{\partial \mathcal{C}_k}{\partial \theta_k}}_{\text{direct gradient}} + \sum_{\theta_l^* \in \mathcal{L}_k} \underbrace{\frac{d\theta_l^*}{d\theta_k}}_{\text{best-response Jacobian}} \times \underbrace{\frac{\partial \mathcal{C}_k}{\partial \theta_l^*}}_{\text{direct gradient}} \tag{1}
$$

While calculating direct gradients is straightforward with existing automatic differentiation frameworks such as PyTorch [32], a major difficulty in gradient-based MLO lies in best-response Jacobian[2] (*i.e.* $\frac{d\theta_l^*}{d\theta_k}$) calculation, which will be discussed in depth in Section 3. Once gradient calculation is enabled, gradient-based optimization is executed from lower level problems to upper level problems in a topologically reverse order, reflecting the underlying hierarchy.

## 3 Automatic Differentiation for Multilevel Optimization

While Equation (1) serves as a mathematical basis for gradient-based multilevel optimization, how to automatically and efficiently carry out such gradient calculation has not been extensively studied

[2] We abuse the term Jacobian for a total derivative here while it is originally a matrix of partial derivatives

and incorporated into a system. In this section, we discuss the challenges in building an automatic differentiation library for multilevel optimization, and provide solutions to address these challenges.

### 3.1 Dataflow Graph for Multilevel Optimization

One may observe that the best-response Jacobian term in Equation (1) is expressed with a *total derivative* instead of a partial derivative. This is because $\theta_k$ can affect $\theta_l^*$ not only through a direct interaction, but also through multiple indirect interactions via other lower-level optimal parameters. For example, consider the four-problem MLO program illustrated in Figure 2. Here, the parameter of Problem 4 ($\theta_{p_4}$) affects the optimal parameter of Problem 3 ($\theta_{p_3}^*$) in two different ways: 1) $\theta_{p_4} \rightarrow \theta_{p_3}^*$ and 2) $\theta_{p_4} \rightarrow \theta_{p_1}^* \rightarrow \theta_{p_3}^*$. In general, we can expand the best-response Jacobian $\frac{d\theta_l^*}{d\theta_k}$ in Equation (1) by applying the chain rule for all paths from $\theta_k$ to $\theta_l^*$ as

$$\frac{d\mathcal{C}_k}{d\theta_k} = \frac{\partial \mathcal{C}_k}{\partial \theta_k} + \sum_{\theta_l^* \in \mathcal{L}_k} \sum_{q \in \mathcal{Q}_{k,l}} \left( \underbrace{\frac{\partial \theta_{q(1)}^*}{\partial \theta_k}}_{\text{upper-to-lower}} \times \left( \prod_{i=1}^{\text{len}(q)-1} \underbrace{\frac{\partial \theta_{q(i+1)}^*}{\partial \theta_{q(i)}^*}}_{\text{lower-to-upper}} \right) \times \frac{\partial \mathcal{C}_k}{\partial \theta_l^*} \right) \tag{2}$$

where $\mathcal{Q}_{k,l}$ is a set of paths from $\theta_k$ to $\theta_l^*$, and $q(i)$ refers to the index of the $i$-th problem in the path $q$ with the last point being $\theta_l^*$. Replacing a total derivative term in Equation (1) with a product of partial derivative terms using the chain rule allows us to ignore indirect interactions between problems, and only deal with direct interactions.

To formalize the path finding problem, we develop a novel dataflow graph for MLO. Unlike traditional dataflow graphs with no predefined hierarchy among nodes, a dataflow graph for multilevel optimization has two different types of directed edges stemming from criteria **A1** & **A2**: *lower-to-upper* and *upper-to-lower*. Each of these directed edges is respectively depicted with green and red arrows in Figure 2. Essentially, a lower-to-upper edge represents the directed dependency between two optimal parameters (*i.e.* $\theta_i^* \rightarrow \theta_j^*$ with $i < j$), while an upper-to-lower edge represents the directed dependency between nonoptimal and optimal parameters (*i.e.* $\theta_i \rightarrow \theta_j^*$ with $i > j$). Since we need to find paths from the nonoptimal parameter $\theta_k$ to the optimal parameter $\theta_l^*$, the first directed edge must be an upper-to-lower edge (red), which connects $\theta_k$ to some lower-level optimal parameter. Once it reaches the optimal parameter, it can only move through optimal parameters via lower-to-upper edges (green) in the dataflow graph. Therefore, every valid path from $\theta_k$ to $\theta_l^*$ will start with an upper-to-lower edge, and then reach the destination only via lower-to-upper edges. The best-response Jacobian term for each edge in the dataflow graph is also marked with the corresponding color in Equation (2). We implement the above path finding mechanism with a modified depth-first search algorithm in BETTY.

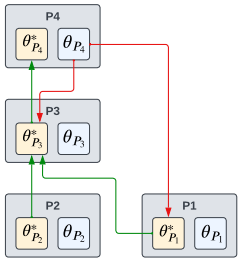

Figure 2: A dataflow graph example of MLO

Furthermore, lower-to-upper directed edges in a dataflow graph can also be used to implement the execution order of an MLO program. For example, let's assume that there is a lower-to-upper edge between problems $P_i$ and $P_j$ (*i.e.* $\theta_i^* \rightarrow \theta_j^*$). When the optimization process of the problem $P_i$ is complete, it can call the problem $P_j$ to start its optimization process through the lower-to-upper edge (see Section 4 for details). The problem $P_j$ waits until all lower level problems in $\mathcal{L}_j$ send their calls, and then starts its optimization process once all the calls from lower levels are received. Hence, to carry out automatic differentiation we only need to call the optimization processes of the lowermost problems in BETTY, as the optimization processes of upper problems will be automatically called from lower problems via lower-to-upper edges in the dataflow graph.

### 3.2 Gradient Calculation with Best-Response Jacobian

Automatic differentiation for MLO can be realized by calculating Equation (2) for each problem $P_k$ ($k = 1, \cdots, n$). However, a naive calculation of Equation (2) could be computationally onerous as it involves multiple matrix multiplications with best-response Jacobians, of which the computational complexity is $\mathcal{O}(n^3)$. To alleviate this issue, we observe that the rightmost term in Equation (2) is a vector, which allows us to reduce the computational complexity of Equation (2) to $\mathcal{O}(n^2)$ by iteratively performing matrix-vector multiplication from right to left (or, equivalently, reverse-traversing a path $q$

in the dataflow graph). As such, matrix-vector multiplication between the best-response Jacobian and a vector serves as a base operation of efficient automatic differentiation for MLO. Mathematically, this problem can be simply written as follows:

$$\text{Calculate} : \quad \frac{\partial w^*(\lambda)}{\partial \lambda} \times v \tag{3}$$

$$\text{Given} : \quad w^*(\lambda) = \underset{w}{\operatorname{argmin}}\, \mathcal{C}(w, \lambda). \tag{4}$$

Two major challenges in the above problems are: 1) approximating the solution of the optimization problem (*i.e.* $w^*(\lambda)$), and 2) differentiating through the (approximated) solution.

In practice, an approximation of $w^*(\lambda)$ is typically achieved by unrolling a small number of gradient steps, which can significantly reduce the computational cost. While we could potentially obtain a better approximation of $w^*(\lambda)$ by running gradient steps until convergence, this procedure alone can take a few days (or even weeks) when the underlying optimization problem is large-scale [9, 10].

Once $w^*(\lambda)$ is approximated, matrix-vector multiplication between the best-response Jacobian $\frac{dw^*(\lambda)}{d\lambda}$ and a vector $v$ is popularly obtained by either iterative differentiation (ITD) or approximate implicit differentiation (AID) [18]. This problem has been extensively studied in bilevel optimization literature [12–14, 27, 29, 30, 34], and we direct interested readers to the original papers, as studying these algorithms is not the focus of this paper.

Rather, we provide several insights about each algorithm here. Roughly speaking, ITD differentiates through the optimization *path* of Equation (4), whereas AID only depends on the (approximated) solution $w^*(\lambda)$. Due to this difference, AID is oftentimes considered to be more memory efficient than ITD. The same observation has also been made based on a theoretical analysis in [24]. Moreover, a dependency to the optimization path requires ITD to track the intermediate states of the parameter during optimization, but existing frameworks like PyTorch override such intermediate states through the use of stateful modules and in-place operations in the optimizer [19]. Hence, ITD requires patching modules and optimizers to support intermediate state tracking as well.

Overall, AID provides two important benefits compared to ITD: it can allow better memory efficiency, and use native modules/optimizers of existing frameworks. Thus, in BETTY we also primarily direct our focus on AID algorithms while also providing an implementation of ITD for completeness. Currently available best-response Jacobian calculation algorithms in BETTY include ITD with reverse-mode automatic differentiation [12], AID with Neumann series [29], AID with conjugate gradient [34], and AID with finite difference [27]. Users can choose whichever algorithm is most-appropriate for each problem in their MLO program, and the chosen algorithm is used to perform the matrix-vector multiplication with best-response Jacobians in Equation (2) for the corresponding problem based on the dataflow graph, accomplishing automatic differentiation for MLO.

In general, the proposed automatic differentiation technique has a lot in common with reverse-mode automatic differentiation (*i.e* backpropagation) in neural networks. In particular, both techniques achieve gradient calculation by iteratively multiplying Jacobian matrices while reverse-traversing dataflow graphs. However, the dataflow graph of MLO has two different types of edges, due to its unique constraint criteria, unlike that of neural networks with a single edge type. Furthermore, Jacobian matrices in MLO are generally approximated with ITD or AID while those in neural networks can be analytically calculated.

## 4 Software Design

On top of the automatic differentiation technique developed in Section 3, we take further steps to build an easy-to-use, modular, and maintainable programming interface for multilevel optimization within BETTY. Specifically, we break down MLO into two high-level concepts, namely 1) optimization problems and 2) the hierarchical dependencies among problems, and design abstract Python classes for each of them. From the dataflow graph perspective, each of these concepts respectively corresponds to nodes and edges. The architecture of BETTY is shown in Figure 1.

**Problem** Each optimization problem $P_k$ in MLO is defined by the parameter (or module) $\theta_k$, the sets of the upper and lower constraining problems $\mathcal{U}_k$ & $\mathcal{L}_k$, the dataset $\mathcal{D}_k$, the cost function $\mathcal{C}_k$, the optimizer, and other optimization configurations (*e.g* best-response Jacobian calculation algorithm,

number of unrolling steps). The `Problem` class is an interface where users can provide each of the aforementioned components, except for the constraining problem sets, to define the optimization problem. We intentionally use a design where the constraining problem sets are provided, rather, by the `Engine` class. In doing so, users need to provide the hierarchical problem dependencies only once when they initialize `Engine`, and can avoid the potentially error-prone and cumbersome process of providing constraining problems manually every time they define new problems. As for the remaining components, each one except for the cost function $\mathcal{C}_k$ can be provided through the class constructor, while the cost function can be defined through a *"training_step"* method.

Once all the required components are provided, `Problem` implements a primitive for gradient-based optimization through a *"step"* method. In detail, a *"step"* method abstracts a one-step gradient descent update with four sub-steps: 1) data loading, 2) cost calculation, 3) gradient calculation, and 4) parameter update. `Problem` unrolls a predetermined number of gradient steps to approximate $\theta_k^*$ by running its *"step"* method, and, when the optimization is done, calls the *"step"* methods of upper level problems through lower-to-upper edges as discussed in Section 3.1.

Furthermore, recall the differences between AID and ITD as reviewed in Section 3.2. ITD requires patching modules and optimizers to allow tracking of intermediate parameter states during optimization. To reflect such differences, we build two children classes, `ImplicitProblem` and `IterativeProblem`, by subclassing `Problem`. The example usage of `Problem` is shown in Listing 1.

```
class MyProblem(ImplicitProblem):
    def training_step(self, batch):
        # Users define the cost function here
        return cost_fn(batch, self.module, self.other_probs, ...)

config = Config(type="neumann", neumann_iterations=3, steps=5)
prob = MyProblem(name, config, module, optimizer, data_loader)
```

Listing 1: `Problem` example

Importantly, we provide a modular interface for users to choose different best-response Jacobian algorithms via a one-liner change in `Config`. This allows users without mathematical and programming expertise to easily and flexibly write a MLO code. All in all, our `Problem` and its *"step"* method are similar in concept to PyTorch [32]'s `nn.Module` and its *"forward"* method. Likewise, the parameter in `Problem` before and after the *"step"* method call corresponds to the input and the output of the neural network layer from the dataflow graph perspective.

**Engine** While `Problem` manages each optimization problem, `Engine` handles a dataflow graph based on the user-provided hierarchical problem dependencies. As discussed in Section 3.1, a dataflow graph for MLO has upper-to-lower and lower-to-upper directed edges. We allow users to define two separate graphs, one for each type of edge, using a Python dictionary, in which keys/values respectively represent start/end nodes of the edge. When user-defined dependency graphs are provided, `Engine` compiles them and finds all paths required for automatic differentiation with a modified depth-first search algorithm. Moreover, `Engine` sets constraining problem sets for each problem based on the dependency graphs, as mentioned above. Once all initialization processes are done, users can run a whole MLO program by calling `Engine`'s run method, which repeatedly calls *"step"* methods of lowermost problems. The example usage of `Engine` is provided in Listing 2.

```
prob1 = MyProblem1(...)
prob2 = MyProblem2(...)
depend = {"u2l": {prob1: [prob2]}, "l2u": {prob1: [prob2]}}
engine = Engine(problems=[prob1, prob2], dependencies=depend)
engine.run()
```

Listing 2: `Engine` example

To summarize, BETTY provides a PyTorch-like simplified way of defining multiple optimization problems, which can scale up to large MLO programs with complex dependencies, as well as a modular interface for a variety of best-response Jacobian algorithms, without requiring mathematical and programming proficiency. In its current version, most operations and low-level primitives of BETTY, such as best-response Jacobian algorithms and module/optimizer patching for ITD, are implemented with PyTorch, which provides high efficiency with parallel GPU computing. However, both `Problem` and `Engine` are largely abstract and implemented in a framework-agnostic way. Thus, we may consider supporting other frameworks including Tensorflow [1] or JAX [15] in the future.

# 5 Experiments

To showcase the general applicability of BETTY, we (re-)implement three MLO benchmark tasks: 1) differentiable neural architecture search [27], 2) data reweighting for the class imbalance problem [38], and 3) domain adaptation for a pretraining/finetuning framework [33]. Furthermore, we analyze the effect of different best-response Jacobian algorithms on test accuracy and memory efficiency for the "data reweighting for class imbalance" task. All of our experiments are conducted with one NVIDIA RTX 2080 Ti, and computational costs are reported in terms of GPU memory usage and wall time.

## 5.1 Differentiable Neural Architecture Search

A neural network architecture plays a significant role in deep learning research. However, the search space of neural architectures is so large that manual search is almost impossible. To overcome this issue, DARTS [27] proposes an efficient gradient-based neural architecture search method based on the bilevel optimization formulation:

$$\alpha^* = \underset{\alpha}{\operatorname{argmin}}\ \mathcal{L}_{val}(w^*(\alpha), \alpha) \qquad \triangleright \text{ Architecture Search}$$

$$\text{s.t. } w^*(\alpha) = \underset{w}{\operatorname{argmin}}\ \mathcal{L}_{train}(w; \alpha) \qquad \triangleright \text{ Classification}$$

where $\alpha$ is the architecture weight and $w$ is the network weight.

We follow the training configurations from the original paper's CIFAR-10 experiment, with a few minor changes. While the original paper performs a finite difference method on the initial network weights, we perform it on the unrolled network weights. This is because we view their best-response Jacobian calculation from the implicit differentiation perspective, where the second-order derivative is calculated based on the unrolled weight. This allows us to unroll the lower-level optimization for more than one step as opposed to strict one-step unrolled gradient descent of the original paper. A similar idea was also proposed in iDARTS [47]. Specifically, we re-implement DARTS with implicit differentiation and finite difference using 1 and 3 unrolling steps. The results are provided in Table 1.

| | Algorithm | Test Acc. | Parameters | Memory | Wall Time |
|---|---|---|---|---|---|
| Random Search [27] | Random | 96.71% | **3.2M** | N/A | N/A |
| DARTS (original) [27] | AID-FD* | 97.24% | 3.3M | 10493MiB | 25.4h |
| DARTS (ours, step=1) | AID-FD | **97.39%** | 3.8M | **10485MiB** | **23.6h** |
| DARTS (ours, step=3) | AID-FD | 97.22% | **3.2M** | **10485MiB** | 28.5h |

Table 1: DARTS re-implementation results. AID-FD refers to implicit differentiation with a finite difference method, and * indicates the difference in the implementation of AID-FD explained above.

Our re-implementation with different unrolling steps achieves a similar performance as the original paper. We also notice that our re-implementation achieves slightly less GPU memory usage and wall time. This is because the original implementation calculates gradients for the architecture weights (upper-level parameters) while running lower-level optimization, while ours only calculates gradients of the parameters for the corresponding optimization stage.

## 5.2 Data Reweighting for Class Imbalance

Many real-world datasets suffer from class imbalance due to underlying long-tailed data distributions. Learning with imbalanced data could have negative societal impacts such as discrimination towards underrepresented groups. Data reweighting has been proposed as one solution to class imbalance problems by assigning higher/lower weights to data from rare/common classes. In particular, Meta-Weight-Net (MWN) [38] proposes to approach data reweighting with bilevel optimization as follows:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\ \mathcal{L}_{val}(w^*(\theta)) \qquad \triangleright \text{ Reweighting}$$

$$\text{s.t. } w^*(\theta) = \underset{w}{\operatorname{argmin}}\ \frac{1}{N}\sum_{i=1}^{n} \mathcal{R}(L^i_{train}(w); \theta) \cdot L^i_{train}(w) \qquad \triangleright \text{ Classification}$$

where $w$ is the network weight, $L^i_{train}$ is the training loss for the $i$-th training sample, and $\theta$ is the MWN $\mathcal{R}$'s weight, which reweights each training sample given its training loss $L^i_{train}$.

We again follow most of the training configurations from the original paper, except for best-response Jacobian algorithms, and minor changes in the learning rate schedule. Specifically, we try out multiple best-response Jacobian algorithms, which only require one-liner changes using BETTY, to study their effect on test accuracy, memory efficiency, and wall time. The experiment results are given in Table 2.

| | Algorithm | IF 200 | IF 100 | IF 50 | Memory | Time |
|---|---|---|---|---|---|---|
| MWN (original) [38] | ITD-RMAD | 68.91 | 75.21 | 80.06 | 2381MiB | 35.8m |
| MWN (ours, step=1) | AID-CG | 66.23±1.88 | 70.88±1.68 | 75.41±0.61 | 2435MiB | 67.4m |
| MWN (ours, step=1) | AID-NMN | 66.45±1.18 | 70.92±1.35 | 75.90 ±1.73 | 2419MiB | 67.1m |
| MWN (ours, step=1) | AID-FD | 75.45±0.63 | 78.11±0.43 | 81.15±0.25 | **2051MiB** | **28.5m** |
| MWN (ours, step=5) | AID-FD | **76.56**±1.19 | **80.45**±0.73 | **83.11**±0.54 | **2051MiB** | 65.5m |

Table 2: MWN experiment results. IF denotes an imbalance factor. AID-CG/NMN/FD respectively stand for implicit differentiation conjugate gradient/Neumann series/finite difference.

We observe that different best-Jacobian algorithms lead to vastly different test accuracy, memory efficiency, and training wall time. Interestingly, we notice that AID-FD with unrolling steps of both 1 and 5 consistently achieve better test accuracy (close to SoTA [41]) and memory efficiency than other methods. This demonstrates that, while BETTY is developed to support large and general MLO programs, it is still useful for simpler bilevel optimization tasks as well.

### 5.3 Domain Adaptation for Pretraining & Finetuning

Pretraining/finetuning paradigms are increasingly adopted with recent advances in self-supervised learning [5, 10, 20]. However, pretraining data are oftentimes from a different distribution than the finetuning data distribution, which could potentially cause negative transfer. Thus, domain adaptation emerges as a natural solution to mitigate negative transfer. As a domain adaptation strategy, [33] proposes to combine data reweighting with a pretraining/finetuning framework to automatically decrease/increase the weight of pretraining samples that cause negative/positive transfer. In contrast with the above two benchmarks, this problem can be formulated as trilevel optimization as follows:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\ \mathcal{L}_{FT}(v^*(w^*(\theta))) \qquad \triangleright \text{ Reweighting}$$

$$\text{s.t. } v^*(w^*(\theta)) = \underset{v}{\operatorname{argmin}}\ \Big(\mathcal{L}_{FT}(v) + \lambda \|v - w^*(\theta)\|_2^2\Big) \qquad \triangleright \text{ Finetuning}$$

$$w^*(\theta) = \underset{w}{\operatorname{argmin}}\ \frac{1}{N}\sum_{i=1}^{n} \mathcal{R}(x_i;\theta)\cdot L_{PT}^i(w) \qquad \triangleright \text{ Pretraining}$$

where $x_i$ / $L_{PT}^i$ stands for the $i$-th pretraining sample/loss, $\mathcal{R}$ for networks that reweight importance for each pretraining sample $x_i$, and $\lambda$ for the proximal regularization parameter. Additionally, $w$, $v$, and $\theta$ are respectively parameters for pretraining, finetuning, and reweighting networks.

We conduct an experiment on the OfficeHome dataset [42] that consists of 15,500 images from 65 classes and 4 domains: Art (Ar), Clipart (Cl), Product (Pr), and Real World (RW). Specifically, we randomly choose 2 domains and use one of them as a pretraining task and the other as a finetuning task. ResNet-18 [21] is used for all pretraining/finetuning/reweighting networks, and Adam [25] with the learning rate of $10^{-4}$ is used as our optimizer. Following [3], the finetuning and the reweighting stages share the same training dataset. For best-response Jacobian calculation, we used AID-FT with the unrolling step of 1 for all stages. We adopted a normal pretraining/finetuning framework without the reweighting stage as our baseline, and the result is presented in Table 3.

| | Algorithm | Cl→Ar | Ar→Pr | Pr→Rw | Rw→Cl | Memory | Time |
|---|---|---|---|---|---|---|---|
| Baseline | N/A | 65.43±0.36 | 87.62±0.33 | 77.43±0.41 | 68.76±0.13 | **3.8GiB** | **290s** |
| Reweight | AID-FD | **67.76**±0.83 | **88.53**±0.42 | **78.58**±0.17 | **69.75**±0.43 | 8.2GiB | 869s |

Table 3: Domain Adaptation for Pretraining & Finetuning results. Reported numbers are classification accuracy on the target domain (right of arrow), after pretraining on the source domain (left of arrow). We note that *Baseline* is a two-layer, and *Baseline + Reweight* a three-layer, MLO program.

Our trilevel optimization framework achieves consistent improvements over the baseline for every task combination at the cost of additional memory usage and wall time, which demonstrates the empirical usefulness of multilevel optimization beyond a two-level hierarchy. Finally, we provide an example of (a simplified version of) the code for this experiment in Appendix A to showcase the usability of our library for a general MLO program.

## 6 Related Work

**Bilevel & Multilevel Optimization** There are a myriad of machine learning applications that are built upon bilevel optimization (BLO), the simplest case of multilevel optimization with a two-level hierarchy. For example, neural architecture search [27, 47], hyperparameter optimization [4, 11, 13, 14, 29, 30], reinforcement learning [23, 26, 35], data valuation [36, 38, 44], meta learning [12, 34], and label correction [48] are formulated as BLO with upper level parameters (or meta parameters) being neural architectures, hyperparameters, importance weights of data, etc. In addition to applying BLO to machine learning tasks, a variety of optimization techniques [6, 17, 18, 24, 28, 46] have been developed for solving BLO.

Following the popularity of BLO, MLO with more than a two-level hierarchy has also attracted increasing attention recently [16, 22, 33, 39, 40, 45]. In general, these works construct complex multi-stage ML pipelines, and optimize the pipelines in an end-to-end fashion with MLO. For instance, [16] constructs the pipeline of (data generation)–(architecture search)–(classification) and [22] of (data reweighting)–(finetuning)–(pretraining), all of which are solved with MLO. Furthermore, [37] study gradient-based methods for solving MLO with theoretical guarantees.

**Multilevel Optimization Software** There are several software libraries that are frequently used for implementing a MLO program. Most notably, higher [19] provides two major functionalities of making 1) stateful PyTorch modules stateless and 2) PyTorch optimizers differentiable. In doing so, they allow constructing (higher-order) gradient graphs through optimization paths, allowing gradient calculation for MLO. While higher provides basic ingredients for gradient-based MLO, the resulting implementation may be highly complicated due to its nature as a low-level library. Torchmeta [8] also provides similar functionalities as higher, but it requires users to use its own stateless modules implemented in the library rather than patching general modules as in higher. Thus, it lacks the support for user's custom modules, limiting its applicability. Finally, learn2learn [2] is introduced with a main focus on supporting meta learning. However, since it is specifically developed for bilevel meta-learning problems, extending it beyond two-level hierarchy is not straightforward.

## 7 Conclusion

In this paper, we aim to help establish both mathematical and systems foundations for automatic differentiation in multilevel optimization. To this end, we develop a novel procedure based on an interpretation of MLO as a dataflow graph, and additionally introduce a MLO software library, BETTY, that allows for easy programming of general MLO applications in a modular and maintainable fashion without requiring expert knowledge.

MLO has the power to be a double-edged sword that can have both positive and negative societal impacts. For example, 1) defense/attack in an adversarial game or 2) decreasing/increasing bias in machine learning models can all be formulated as MLO programs, depending on the goal of the uppermost optimization problem, which is defined by users. Thus, research in preventing malicious use cases of MLO is of high importance.

Finally, BETTY has limitations in both machine learning and system aspects. BETTY is currently built upon the assumption that all optimization problems in a MLO program are differentiable with respect to (constraining) parameters. However, this assumption may not hold if non-differentiable processes like sampling are involved. Therefore, machine learning research in enabling best-response Jacobian calculation for non-differentiable optimization problems would broaden the applicability of BETTY. In addition, MLO tends to consume more computation and memory resources compared to traditional single-level optimization, as it involves multiple optimization problems. Thus, incorporating or devising distributed training strategies as well as other memory management techniques (*e.g.* CPU offloading) will be necessary in the future for larger scale MLO programs.

## References


[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: A system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Sébastien MR Arnold, Praateek Mahajan, Debajyoti Datta, Ian Bunner, and Konstantinos Saitas Zarkias. learn2learn: A library for meta-learning research. *arXiv preprint arXiv:2008.12284*, 2020.

[3] Yu Bai, Minshuo Chen, Pan Zhou, Tuo Zhao, Jason Lee, Sham Kakade, Huan Wang, and Caiming Xiong. How important is the train-validation split in meta-learning? In *International Conference on Machine Learning*, pages 543–553. PMLR, 2021.

[4] Atilim Gunes Baydin, Robert Cornish, David Martínez-Rubio, Mark Schmidt, and Frank D. Wood. Online learning rate adaptation with hypergradient descent. *CoRR*, abs/1703.04782, 2017.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] Nicolas Couellan and Wenjuan Wang. On the convergence of stochastic bi-level gradient methods. *Optimization*, 2016.

[7] Hua Cui and Jie Bai. A new hyperparameters optimization method for convolutional neural networks. *Pattern Recognition Letters*, 125:828–834, 2019.

[8] Tristan Deleu, Tobias Würfl, Mandana Samiei, Joseph Paul Cohen, and Yoshua Bengio. Torchmeta: A Meta-Learning library for PyTorch, 2019. Available at: https://github.com/tristandeleu/pytorch-meta.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[11] Matthias Feurer, Jost Springenberg, and Frank Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

[12] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

[13] Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. Forward and reverse gradient-based hyperparameter optimization. In *International Conference on Machine Learning*, pages 1165–1173. PMLR, 2017.

[14] Luca Franceschi, Paolo Frasconi, Saverio Salzo, Riccardo Grazzi, and Massimiliano Pontil. Bilevel programming for hyperparameter optimization and meta-learning. In *International Conference on Machine Learning*, pages 1568–1577. PMLR, 2018.

[15] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, pages 23–24, 2018.

[16] Bhanu Garg, Li Zhang, Pradyumna Sridhara, Ramtin Hosseini, Eric Xing, and Pengtao Xie. Learning from mistakes–a framework for neural architecture search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022.

[17] Saeed Ghadimi and Mengdi Wang. Approximation methods for bilevel programming. *arXiv preprint arXiv:1802.02246*, 2018.

[18] Riccardo Grazzi, Luca Franceschi, Massimiliano Pontil, and Saverio Salzo. On the iteration complexity of hypergradient computation. In *International Conference on Machine Learning*, pages 3748–3758. PMLR, 2020.

[19] Edward Grefenstette, Brandon Amos, Denis Yarats, Phu Mon Htut, Artem Molchanov, Franziska Meier, Douwe Kiela, Kyunghyun Cho, and Soumith Chintala. Generalized inner loop meta-learning. *arXiv preprint arXiv:1910.01727*, 2019.

[20] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[22] Xuehai He, Zhuo Cai, Wenlan Wei, Yichen Zhang, Luntian Mou, Eric Xing, and Pengtao Xie. Towards visual question answering on pathology images. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 708–718, 2021.

[23] Mingyi Hong, Hoi-To Wai, Zhaoran Wang, and Zhuoran Yang. A two-timescale framework for bilevel optimization: Complexity analysis and application to actor-critic. *arXiv preprint arXiv:2007.05170*, 2020.

[24] Kaiyi Ji, Junjie Yang, and Yingbin Liang. Bilevel optimization: Convergence analysis and enhanced design. In *International Conference on Machine Learning*, pages 4882–4892. PMLR, 2021.

[25] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[26] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[27] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. In *ICLR*, 2019.

[28] Risheng Liu, Yaohua Liu, Shangzhi Zeng, and Jin Zhang. Towards gradient-based bilevel optimization with non-convex followers and beyond. *Advances in Neural Information Processing Systems*, 34, 2021.

[29] Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics*, pages 1540–1552. PMLR, 2020.

[30] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International conference on machine learning*, pages 2113–2122. PMLR, 2015.

[31] Athanasios Migdalas, Panos M Pardalos, and Peter Värbrand. *Multilevel optimization: algorithms and applications*, volume 20. Springer Science & Business Media, 1998.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[33] Aniruddh Raghu, Jonathan Lorraine, Simon Kornblith, Matthew McDermott, and David K Duvenaud. Meta-learning to improve pre-training. *Advances in Neural Information Processing Systems*, 34, 2021.

[34] Aravind Rajeswaran, Chelsea Finn, Sham M Kakade, and Sergey Levine. Meta-learning with implicit gradients. *Advances in neural information processing systems*, 32, 2019.

[35] Aravind Rajeswaran, Igor Mordatch, and Vikash Kumar. A game theoretic framework for model based reinforcement learning. In *International conference on machine learning*, pages 7953–7963. PMLR, 2020.

[36] Zhongzheng Ren, Raymond Yeh, and Alexander Schwing. Not all unlabeled data are equal: Learning to weight data in semi-supervised learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 21786–21797. Curran Associates, Inc., 2020.

[37] Ryo Sato, Mirai Tanaka, and Akiko Takeda. A gradient method for multilevel optimization. *Advances in Neural Information Processing Systems*, 34, 2021.

[38] Jun Shu, Qi Xie, Lixuan Yi, Qian Zhao, Sanping Zhou, Zongben Xu, and Deyu Meng. Meta-weight-net: Learning an explicit mapping for sample weighting. In *Advances in Neural Information Processing Systems*, pages 1919–1930, 2019.

[39] Sai Ashish Somayajula, Linfeng Song, and Pengtao Xie. A multi-level optimization framework for end-to-end text augmentation. *Transactions of the Association for Computational Linguistics*, 10:343–358, 2022.

[40] Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth Stanley, and Jeffrey Clune. Generative teaching networks: Accelerating neural architecture search by learning to generate synthetic training data. In *International Conference on Machine Learning*, pages 9206–9216. PMLR, 2020.

[41] Kaihua Tang, Jianqiang Huang, and Hanwang Zhang. Long-tailed classification by keeping the good and removing the bad momentum causal effect. *Advances in Neural Information Processing Systems*, 33:1513–1524, 2020.

[42] Hemanth Venkateswara, Jose Eusebio, Shayok Chakraborty, and Sethuraman Panchanathan. Deep hashing network for unsupervised domain adaptation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5018–5027, 2017.

[43] Luis N Vicente and Paul H Calamai. Bilevel and multilevel programming: A bibliography review. *Journal of Global optimization*, 5(3):291–306, 1994.

[44] Yulin Wang, Jiayi Guo, Shiji Song, and Gao Huang. Meta-semi: A meta-learning approach for semi-supervised learning. *CoRR*, abs/2007.02394, 2020.

[45] Pengtao Xie and Xuefeng Du. Performance-aware mutual knowledge distillation for improving neural architecture search. *CVPR*, 2022.

[46] Junjie Yang, Kaiyi Ji, and Yingbin Liang. Provably faster algorithms for bilevel optimization. *Advances in Neural Information Processing Systems*, 34, 2021.

[47] Miao Zhang, Steven W Su, Shirui Pan, Xiaojun Chang, Ehsan M Abbasnejad, and Reza Haffari. idarts: Differentiable architecture search with stochastic implicit gradients. In *International Conference on Machine Learning*, pages 12557–12566. PMLR, 2021.

[48] Guoqing Zheng, Ahmed Hassan Awadallah, and Susan T. Dumais. Meta label correction for learning with weak supervision. *CoRR*, abs/1911.03809, 2019.

# A Code Example

Here, we provide simplified code for our experiments from Section 5. Note that every experiment shares a similar code structure when implemented with BETTY.

## A.1 Differentiable Neural Architecture Search

```
train_loader, valid_loader = setup_dataloader()
arch_module, arch_optimizer = setup_architecture()
cls_module, cls_optimizer, cls_scheduler = setup_classifier()

class Architecture(ImplicitProblem):
    def training_step(self, batch):
        x, target = batch
        alphas = self.module()
        return self.classifier.module.loss(x, alphas, target)

class Classifier(ImplicitProblem):
    def training_step(self, batch):
        x, target = batch
        alphas = self.architecture()
        return self.module.loss(x, alphas, target)

arch_config = Config(type="darts",
                     step=1,
                     retain_graph=True,
                     first_order=True)
cls_config = Config(type="default")

architecture = Architecture(name="architecture",
                            config=arch_config,
                            module=arch_module,
                            optimizer=arch_optimizer,
                            train_data_loader=valid_loader)
classifier = Classifier(name="classifier",
                        config=cls_config,
                        module=cls_module,
                        optimizer=cls_optimizer,
                        scheduler=cls_scheduler,
                        train_data_loader=train_loader)

probs = [architecture, classifier]
u2l = {architecture: [classifier]}
l2u = {classifier: [architecture]}
depends = {"l2u": l2u, "u2l": u2l}

engine = Engine(problems=probs, dependencies=depends)
engine.run()
```

Listing 3: Simplified code of “Differentiable Neural Architecture Search”

### A.2 Data Reweighting for Class Imbalance

```
train_loader, valid_loader = setup_dataloader()
rw_module, rw_optimizer = setup_reweight()
cls_module, cls_optimizer, cls_scheduler = setup_classifier()

class Reweight(ImplicitProblem):
    def training_step(self, batch):
        inputs, labels = batch
        outputs = self.classifier(inputs)
        return F.cross_entropy(outputs, labels)

class Classifier(ImplicitProblem):
    def training_step(self, batch):
        inputs, labels = batch
        outputs = self.module(inputs)
        loss = F.cross_entropy(outputs, labels, reduction="none")
        loss_reshape = torch.reshape(loss, (-1, 1))
        weight = self.reweight(loss_reshape.detach())
        return torch.mean(weight * loss_reshape)

rw_config = Config(type="darts",
                   step=5,
                   retain_graph=True,
                   first_order=True)
cls_config = Config(type="default")

reweight = Reweight(name="reweight",
                    config=rw_config,
                    module=rw_module,
                    optimizer=rw_optimizer,
                    train_data_loader=valid_loader)
classifier = Classifier(name="classifier",
                        config=cls_config,
                        module=cls_module,
                        optimizer=cls_optimizer,
                        scheduler=cls_scheduler,
                        train_data_loader=train_loader)

probs = [reweight, classifier]
u2l = {reweight: [classifier]}
l2u = {classifier: [reweight]}
depends = {"l2u": l2u, "u2l": u2l}

engine = Engine(problems=probs, dependencies=depends)
engine.run()
```

Listing 4: Simplified code of “Data Reweighting for Class Imbalance”

### A.3 Domain Adaptation for Pretraining & Finetuning

```
# Get module, optimizer, lr_scheduler, data loader for each problem
pt_module, pt_optimizer, pt_scheduler, pt_loader = setup_pretrain()
ft_module, ft_optimizer, ft_scheduler, ft_loader = setup_finetune()
rw_module, rw_optimizer, rw_scheduler, rw_loader = setup_reweight()

# Define each optimization problem
class Pretrain(ImplicitProblem):
    def training_step(self, batch):
        inputs, targets = batch
        outs = self.module(inputs)
        loss_raw = F.cross_entropy(outs, targets, reduction="none")

        logit = self.reweight(inputs)
        weight = torch.sigmoid(logit)
        return torch.mean(loss_raw * weight)

class Finetune(ImplicitProblem):
    def training_step(self, batch):
        inputs, targets = batch
        outs = self.module(inputs)
        loss = F.cross_entropy(outs, targets, reduction="none")
        loss = torch.mean(ce_loss)
        for (n1, p1), p2 in zip(self.module.named_parameters(), self.
    pretrain.module.parameters()):
            lam = 0 if "fc" in n1 else args.lam
            loss += lam * (p1 - p2).pow(2).sum()
        return loss

class Reweight(ImplicitProblem):
    def training_step(self, batch):
        inputs, targets = batch
        outs = self.finetune(inputs)
        return F.cross_entropy(outs, targets)

# Define optimization configurations
reweight_config = Config(type="darts", step=1, retain_graph=True)
finetune_config = Config(type="default", step=1)
pretrain_config = Config(type="default", step=1)

pretrain = Pretrain("pretrain", pt_config, pt_module, pt_optimizer
                    pt_scheduler, pt_loader)
finetune = Finetune("finetune", ft_config, ft_module, ft_optimizer
                    ft_scheduler, ft_loader)
reweight = Reweight("reweight", rw_config, rw_module, rw_optimizer
                    rw_scheduler, rw_loader)

probs = [reweight, finetune, pretrain]
u2l = {reweight: [pretrain]}
l2u = {pretrain: [finetune], finetune: [reweight]}
depends = {"u2l": u2l, "l2u": l2u}
engine = Engine(problems=probs, dependencies=depends)
engine.run()
```

Listing 5: Simplified code of “Domain Adaptation for Pretraining & Finetuning”

## B Experiment Details

While we mostly follow the training configurations of the original papers that we reproduce, we provide further training details of each experiment for the completeness of the paper.

### B.1 Differentiable Neural Architecture Search

**Dataset** Follwing the original paper [27], we use the first half of the CIFAR-10 training dataset as our inner-level training dataset (*i.e.* classification network) and the other half as the outer-level training dataset (*i.e.* architecture network). Training accuracy reported in the main text is measured on the CIFAR-10 validation dataset.

**Architecture Network** We adopt the same architecture search space as in the original paper [27] with 8 operations, and 7 nodes per convolutional cell. The architecture parameters are initialized to zero to ensure equal softmax values, and trained with the Adam optimizer [25] whose learning rate is fixed to 0.0003, momentum values to (0.5, 0.999), and weight decay value to 0.001 throughout training. Training is performed for 50 epochs.

**Classification Network** Given the above architecture parameters, we set our classification network to have 8 cells and the initial number of channels to be 16. The network is trained with the SGD optimizer whose initial learning rate is set to 0.025, momentum to 0.9, and weight decay value to 0.0003. Training is performed for 50 epochs, and the learning rate is decayed following the cosine annealing schedule without restart to the minimum learning rate of 0.001 by the end of training.

### B.2 Data Reweighting for Class Imbalance

**Dataset** We reuse the long-tailed CIFAR-10 dataset from the original paper [38] as our inner-level training dataset. More specifically, the imbalance factor is defined as the ratio between the number of training samples from the most common class and the most rare class. The number of training samples of other classes are defined by geometrically interpolating the number of training samples from the most common class and the most rare class. We randomly select 10 samples from the validation set to construct the upper-level (or meta) training dataset, and use the rest of it as the validation dataset, on which classification accuracy is reported in the main text.

**Meta-Weight-Network** We adopt a MLP with one hidden layer of 100 neurons (*i.e.* 1-100-1) as our Meta-Weight-Network (MWN). It is trained with the Adam optimizer [25] whose learning rate is set to 0.00001 throughout the whole training procedure, momentum values to (0.9, 0.999), and weight decay value to 0. MWN is trained for 10,000 iterations and learning rate is fixed throughout training.

**Classification Network** Following the original MWN work [38], we use ResNet32 [21] as our classification network. It is trained with the SGD optimizer whose initial learning rate is set to 0.1, momentum value to 0.9, and weight decay value to 0.0005. Training is performed for 10,000 iterations, and we decay the learning rate by a factor of 10 on the iterations of 5,000 and 7,500.

### B.3 Domain Adaptation for Pretraining & Finetuning

**Dataset** We split each domain of the OfficeHome dataset [42] into training/validation/test datasets with a ratio of 5:3:2. The pretraining network is trained on the training set of the source domain. Finetuning and reweighting networks are both trained on the training set of the target domain following the strategy proposed in [3]. The final performance is measured by the classification accuracy of the finetuning network on the test dataset of the target domain.

**Pretraining Network** We use ResNet18 [21] pretrained on the ImageNet dataset [9] for our pretraining network. Following the popular transfer learning strategy, we split the network into two parts, namely the feature (or convolutional layer) part and the classifier (or fully-connected layer) part, and each part is trained with different learning rates. Specifically, learning rates for the feature and the classifier parts are respectively set to 0.001 and 0.0001 with the Adam optimizer. They share the same weight decay value of 0.0005 and momentum values of (0.9, 0.999). Furthermore, we encourage

the network weight to stay close to the pretrained weight by introducing the additional proximal regularization with the regularization value of 0.001. Training is performed for 1,000 iterations, and the learning rate is decayed by a factor of 10 on the iterations of 400 and 800.

**Finetuning Network** The same architecture and optimization configurations as the pretraining network are used for the finetuning network. The proximal regularization parameter, which encourages the finetuning network parameter to stay close to the pretraining network parameter, is set to 0.007.

**Reweighting Network** The same architecture and optimization configurations as the pretraining network are used for the reweighting network, except that no proximal regularization is applied to the reweighting network.