Journal of Machine Learning Research 24 (2023) 1-14 Submitted 2/23; Revised 11/23; Published 11/23

TorchOpt: An Efficient Library for Differentiable Optimization

Jie Ren[1,4,*] JIEREN9806@GMAIL.COM
Xidong Feng[2,*] XIDONG.FENG.20@UCL.AC.UK
Bo Liu[5,*] BENJAMINLIU.EECS@GMAIL.COM
Xuehai Pan[3,*] XUEHAIPAN@PKU.EDU.CN
Yao Fu[1] Y.FU@ED.AC.UK
Luo Mai[1,†] LUO.MAI@ED.AC.UK
Yaodong Yang[5,†] YAODONG.YANG@PKU.EDU.CN
[1] *School of Informatics, University of Edinburgh, Edinburgh, United Kingdom*
[2] *Department of Computer Science, University College London, London, United Kingdom*
[3] *School of Computer Science, Peking University, Beijing, China*
[4] *ECRC, KAUST, Thuwal, Saudi Arabia*
[5] *Institute for AI, Peking University, Beijing, China*

Editor: Alexandre Gramfort

Abstract

Differentiable optimization algorithms often involve expensive computations of various meta-gradients. To address this, we design and implement `TorchOpt`, a new PyTorch-based differentiable optimization library. `TorchOpt` provides an expressive and unified programming interface that simplifies the implementation of explicit, implicit, and zero-order gradients. Moreover, `TorchOpt` has a distributed execution runtime capable of parallelizing diverse operations linked to differentiable optimization tasks across CPU and GPU devices. Experimental results demonstrate that `TorchOpt` achieves a 5.2× training time speedup in a cluster. `TorchOpt` is open-sourced at `https://github.com/metaopt/torchopt` and has become a PyTorch Ecosystem project.

Keywords: Differentiable Optimization, Meta Learning, Machine Learning Library

1. Introduction

In recent years, there has been a notable proliferation of differentiable optimization-based algorithms, exemplified by works such as MAML (Finn et al., 2017), OptNet (Amos and Kolter, 2017), and MGRL (Xu et al., 2018). Within the realm of differentiable optimization, a pivotal facet pertains to the concept of meta-gradients. These meta-gradients signify the gradient components associated with outer-loop variables, obtained through the process of differentiating across the inner-loop optimization operations. The utilization of meta-gradients confers advantages to machine learning models, manifesting in heightened sample efficiency (Finn et al., 2017) and amplified final performance outcomes (Xu et al., 2018).

∗. Equal contribution, the order is determined by dice rolling. See Appendix G for more details.
†. Corresponding author.

Table 1: Differentiable optimization libraries. (✓) indicates a partially supported feature.

	Differentiable Optimizer	Implicit Differentiation	Zero-order Gradient	MPMD Training	SPMD Training	Gradient Visualization	Backend
`higher` (Grefenstette et al., 2019)	✓	✗	✗	✗	✗	✗	PyTorch
`Optax` (Babuschkin et al., 2020)	✓	✗	✗	✗	✗	✗	JAX
`Torchmeta` (Deleu et al., 2019)	✓	✗	✗	✗	✗	✗	PyTorch
`learn2learn` (Arnold et al., 2020)	✓	✗	✗	✗	(✓)	✗	PyTorch
`JAXopt` (Blondel et al., 2021)	✓	✓	✓	✗	✓	✗	JAX
`HyperTorch` (Grazzi et al., 2020)	✓	✓	✗	✗	✗	✗	PyTorch
`Betty` (Choe et al., 2022)	✓	(✓)	✓	✗	✓	✗	PyTorch
`TorchOpt`(ours)	✓	✓	✓	✓	✓	✓	PyTorch

Our objective is to develop a library that enables machine learning researchers to efficiently create differentiable optimization algorithms. Through our interactions with these researchers, we have identified several essential library requirements: (i) *Generic bi-level optimization with various meta-gradients*: Researchers require the capability to implement varied inner-loop optimizations within an outer-loop optimization framework. The outer-loop framework needs to compute diverse meta-gradients, including explicit, implicit, and zero-order gradients. (ii) *Generic distributed execution*: Given the significant computational demands of differentiable optimization, distributing computations across nodes (such as CPUs and GPUs) is essential. Depending on algorithm characteristics, distributed differentiable optimization can follow both single-program-multiple-data (SPMD) (e.g., MAML (Finn et al., 2017)) and multiple-program-multi-data (MPMD) (e.g., LOLA (Foerster et al., 2017))[1]. (iii) *Visualizing gradient flow*: The computation of meta-gradients often mandates the incorporation of additional nodes into the gradient flows established by the inner-loop optimization. To ensure accurate computation of meta-gradients, researchers require the ability to visualize and manipulate gradient flows.

Existing differentiable optimization libraries, however, are not fully capable of meeting the aforementioned requirements. We have summarized these libraries in Table 1. The PyTorch libraries, such as `higher` library (Grefenstette et al., 2019) and `learn2learn` (Arnold et al., 2020) solely support explicit differentiation. In contrast, `Torchmeta` (Deleu et al., 2019) offers additional support for implicit differentiation. The `Betty` library supports zero-order gradients and partially covers implicit gradients. In ecosystems beyond PyTorch, JAX-based libraries such as `Optax` (Babuschkin et al., 2020) specializes in explicit differentiation. More comprehensively, `JAXopt` (Blondel et al., 2021) stands out as a state-of-the-art library that extends support to explicit, implicit, and zero-order gradients. However, it only accommodates single-program-multiple-data (SPMD) training for distributed execution, lacking support for more generic multiple-program-multi-data (MPMD). The latter is particularly essential in meta-learning, given its inherently complex and dynamic nature of the training pipeline. Furthermore, `JAXopt` needs users to manually implement the visualization of gradient flow.

In this paper, we present `TorchOpt`, a new PyTorch differentiable optimization library. `TorchOpt` address the above development requirements through two contributions:

(1) Comprehensive differentiation mode. `TorchOpt` furnishes users with a versatile array of APIs, encompassing low-level, high-level, functional, and Object-Oriented (OO) paradigms. This empowers users to seamlessly incorporate differentiable optimization within the computational graphs generated by different PyTorch programs. Notably, `TorchOpt` offers support for three differentiation modes tailored to diverse differentiable

1. We discuss the differences between SPMD and MPMD in Appendix E.1.

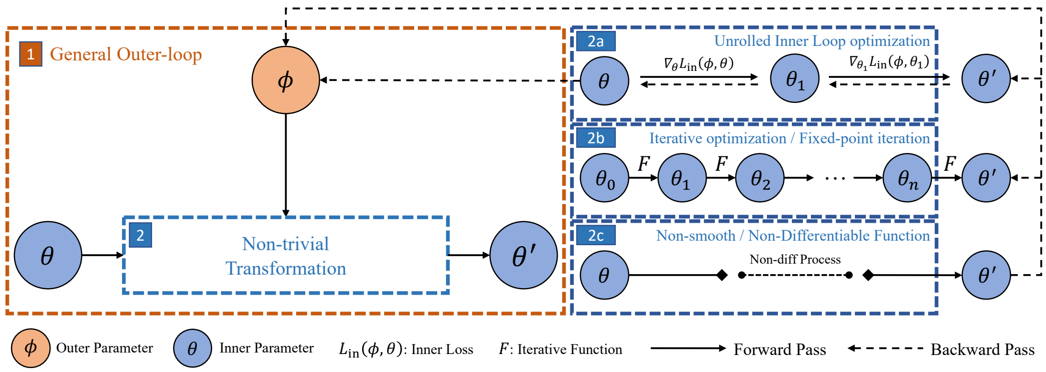


Figure 1: `TorchOpt`'s differentiation modes. A backward pass is denoted by dotted lines.

optimization problems: (i) Explicit gradient for unrolled optimization, (ii) Implicit gradient for differentiable optimization, and (iii) Zero-order gradient estimation for non-smooth or non-differentiable functions.

(2) High-performance distributed execution runtime. `TorchOpt` aims to enable optimal utilization of CPUs and GPUs for differentiable optimization algorithms. To achieve this, we have the following designs: (i) Implementation of CPU/GPU-accelerated optimizers such as SGD, RMSProp, and Adam. These optimizers fuse small differentiable operators and fully offload them to GPUs. (ii) Introduction of fast and efficient PyTree operations, capable of high-throughput flattening of nested structures (Tree Operations) – a crucial computation-intensive task in differentiable optimization. (iii) Establishment of a distributed auto-grad framework that automatically identifies inner-loop tasks within differentiable optimizers. It then efficiently dispatches the execution of these inner-loop tasks to distributed CPUs and GPUs.

2. Comprehensive Differentiation Mode

We describe the differentiable mode of `TorchOpt` in Figure 1 and leave a detailed discussion of TorchOpt's architecture in Appendix A.

Explicit Gradient (EG). Figure 1-2a illustrates the concept behind implementing EG in `TorchOpt`. In this approach, `TorchOpt` treats the gradient step as a differentiable function and facilitates the backpropagation of gradients through the unrolled optimization path. EG suits algorithms where the inner-level solution is obtained through a few gradient steps, as seen in algorithms like MAML (Finn et al., 2017) and MGRL (Xu et al., 2018). Moreover, `TorchOpt` provides users with the flexibility to declare EG within PyTorch programs through both functional and object-oriented APIs. Refer to the code snippet in Appendix C.1 and the EG update scheme in Appendix C for further details.

Implicit Gradient (IG). Figure 1-2b shows the concept behind implementing IG. In this approach, `TorchOpt` treats the inner-loop optimization solution as an implicit function of outer-loop parameters. Hence, it can directly get analytical best-response derivatives by the implicit function theorem (Krantz and Parks, 2002). IG suits algorithms where the inner-level solution is obtained by reaching certain stationary conditions, such as iMAML (Rajeswaran et al., 2019) and DEQ (Bai et al., 2019). `TorchOpt` offers functional and object-oriented API for both conjugate gradient-based (Rajeswaran et al., 2019) and Neumann series-based (Lorraine et al., 2020) method. Refer to the code snippet in Appendix C.2 and the update scheme in C for further details.

Zero-order Differentiation (ZD). As shown in Figure 1-2c, when the inner-loop process is non-differentiable or one wants to eliminate the heavy computation burdens in the previous two modes (brought by Hessian), one can choose ZD. ZD typically gets gradients based on zero-order estimation, such as finite-difference, or Evolutionary Strategy (ES) (Salimans et al., 2017). ESMAML (Song et al., 2019), and NAC (Feng et al., 2021), successfully solve the differentiable optimization problem based on ES. `TorchOpt` also offers functional and OOP API for ES method. Refer to Listing 3 Appendix C.3 for code snippets and Appendix C for illustration.

Gradient graph visualization. `TorchOpt` provides a visualization tool that draws variable (e.g. network parameters or meta parameters) names on the gradient graph for better analysis. `TorchOpt` fuses the operations within the optimization algorithm (such as Adam) to reduce the complexity and provide a more concise visualization. Refer to the visualization example in Appendix B.

3. High-performance Distributed Execution Runtime

`TorchOpt` offers the following three features to enable efficient differentiable optimization.

High-performance differentiable optimization. We manually write the forward and backward functions, thus achieving a symbolic reduction towards the gradient flow. In addition, we reuse intermediate data during the back-propagation. Our design reduces computation and also benefits numerical stability. We write the accelerated functions in C++ OpenMP and CUDA, bind them by `pybind11` to allow Python can call them, and then we define the forward and backward behavior using `torch.autograd.Function`. Refer to Appendix D for experimental results of CPU/GPU-accelerated optimizers.

High-performance PyTree utilities. The tree operations (e.g., flatten and unflatten) are frequently called by the functional and Just-In-Time (JIT) components in `TorchOpt`. To enable memory-efficient nested structure flattening, we implement a set of high-performance PyTree utilities, named `OpTree`. By optimizing their memory and cache performance (e.g., `absl::InlinedVector`), `TorchOpt` can significantly improve the performance of differentiable optimization at scale. Refer to Appendix F for `OpTree` experimental results.

Distributed differentiable optimization. `TorchOpt` can distribute differentiable optimization to parallel GPUs. Different from MPI-based synchronous training (Mai et al., 2020) and asynchronous model averaging (Koliousis et al., 2019), `TorchOpt` adopts RPC as a flexible and performant communication backend. The distributed GPUs perform parallel differentiable optimization tasks. The GPUs are coordinated by a controller, thus guaranteeing the convergence of the model in various distributed training (including MPMD and SPMD). More details are in Appendix E.

4. Conclusion

This paper introduces TorchOpt, a novel differentiable optimization library for PyTorch. TorchOpt features a comprehensive differentiation mode and a high-performance distributed execution runtime. TorchOpt has been used by numerous researchers on GitHub (Liu et al., 2021), making it a popular library in the PyTorch ecosystem.

Acknowledgments

We would like to thank Vincent Moens, Stefano Woerner, Quentin Bertrand and others for their valuable contributions to TorchOpt. This work was supported by National Key R&D Program of China (2022ZD0114900), Young Elite Scientists Sponsorship Program by CAST (2022QNRC002), and the Edinburgh-Huawei Joint Lab.

References

B. Amos and J. Z. Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, 2017.

S. M. R. Arnold, P. Mahajan, D. Datta, I. Bunner, and K. S. Zarkias. learn2learn: A library for Meta-Learning research. *arXiv preprint arXiv:2008.12284*, 2020.

I. Babuschkin, K. Baumli, A. Bell, S. Bhupatiraju, J. Bruce, P. Buchlovsky, D. Budden, T. Cai, A. Clark, I. Danihelka, C. Fantacci, J. Godwin, C. Jones, R. Hemsley, T. Hennigan, M. Hessel, S. Hou, S. Kapturowski, T. Keck, I. Kemaev, M. King, M. Kunesch, L. Martens, H. Merzic, V. Mikulik, T. Norman, J. Quan, G. Papamakarios, R. Ring, F. Ruiz, A. Sanchez, R. Schneider, E. Sezener, S. Spencer, S. Srinivasan, L. Wang, W. Stokowiec, and F. Viola. The DeepMind JAX Ecosystem, 2020.

S. Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. *Advances in Neural Information Processing Systems*, 32, 2019.

M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert. Efficient and modular implicit differentiation. *arXiv preprint arXiv:2105.15183*, 2021.

S. K. Choe, W. Neiswanger, P. Xie, and E. P. Xing. Betty: An automatic differentiation library for multilevel optimization. *arXiv preprint arXiv:2207.02849*, 2022.

T. Deleu, T. Würfl, M. Samiei, J. P. Cohen, and Y. Bengio. Torchmeta: A Meta-Learning library for PyTorch. *arXiv preprint arXiv:1909.06576*, 2019.

X. Feng, O. Slumbers, Z. Wan, B. Liu, S. McAleer, Y. Wen, J. Wang, and Y. Yang. Neural auto-curricula in two-player zero-sum games. *Advances in Neural Information Processing Systems*, 34:3504–3517, 2021.

C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, 2017.

J. N. Foerster, R. Y. Chen, M. Al-Shedivat, S. Whiteson, P. Abbeel, and I. Mordatch. Learning with opponent-learning awareness. *arXiv preprint arXiv:1709.04326*, 2017.

R. Grazzi, L. Franceschi, M. Pontil, and S. Salzo. On the iteration complexity of hypergradient computation. *Thirty-seventh International Conference on Machine Learning (ICML)*, 2020.

E. Grefenstette, B. Amos, D. Yarats, P. M. Htut, A. Molchanov, F. Meier, D. Kiela, K. Cho, and S. Chintala. Generalized inner loop meta-learning. *arXiv preprint arXiv:1910.01727*, 2019.

A. Koliousis, P. Watcharapichat, M. Weidlich, L. Mai, P. Costa, and P. R. Pietzuch. Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers. *Proc. VLDB Endow.*, 12(11):1399–1413, 2019. doi: 10.14778/3342263.3342276.

S. G. Krantz and H. R. Parks. *The implicit function theorem: history, theory, and applications.* Springer Science & Business Media, 2002.

B. Liu, X. Feng, J. Ren, L. Mai, R. Zhu, H. Zhang, J. Wang, and Y. Yang. A theoretical understanding of gradient bias in meta-reinforcement learning. *arXiv preprint arXiv:2112.15400*, 2021.

J. Lorraine, P. Vicol, and D. Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *International Conference on Artificial Intelligence and Statistics*, pages 1540–1552. PMLR, 2020.

L. Mai, G. Li, M. Wagenländer, K. Fertakis, A.-O. Brabete, and P. Pietzuch. {KungFu}: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 937–954, 2020.

A. Rajeswaran, C. Finn, S. M. Kakade, and S. Levine. Meta-learning with implicit gradients. *Advances in neural information processing systems*, 32, 2019.

T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

X. Song, W. Gao, Y. Yang, K. Choromanski, A. Pacchiano, and Y. Tang. Es-maml: Simple hessian-free meta learning. *arXiv preprint arXiv:1910.01215*, 2019.

Z. Xu, H. P. van Hasselt, and D. Silver. Meta-gradient reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

S. Zagoruyko. Pytorchviz: A small package to create visualizations of pytorch execution graphs and traces. `https://github.com/szagoruyko/pytorchviz`, 2018.

Appendix A. Architecture Overview

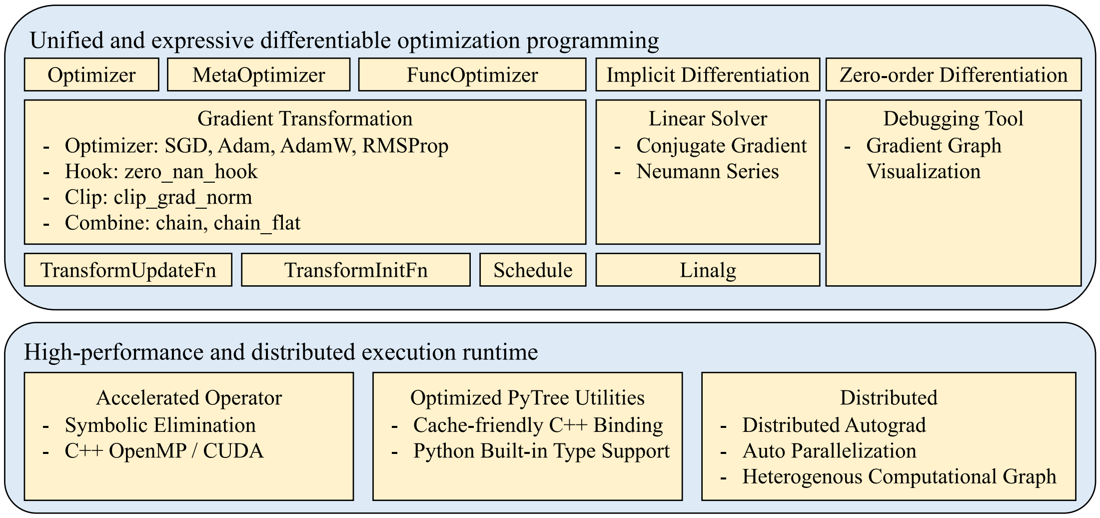


Figure 2: `TorchOpt`'s architecture overview.

Figure 2 gives an overview of the system architecture, `TorchOpt` consists of two different aspects, the unified and expressive differentiable optimization programming lets users easily implement differentiable optimization algorithms, we provide both high-level APIs and low-level APIs for three differentiation modes along with debugging tools, all of which are described in Sec. 2. Then the high-performance and distributed execution runtime contains several accelerated solutions to support fast differentiation with different modes on GPU & CPU and distributed training features for multi-node multi-GPU scenario, which we demonstrate boost performance in Sec. 3. Additionally, we offer `OpTree` to enable fast structure `flatten` and `unflatten`, which is specially designed for our functional programming implementation. We use an optimized structure to avoid memory allocation if the sub-tree is small.

Appendix B. Gradient Graph Visualization

The visualization tool is modified from TorchViz (Zagoruyko, 2018). Fig. 3 shows the visualization example of MAML. We use red squares to represent what each part accomplishes separately. Compared with TorchViz, `TorchOpt` fuses the operations within the Adam together (orange) to reduce the complexity and provides a more straightforward visualization.

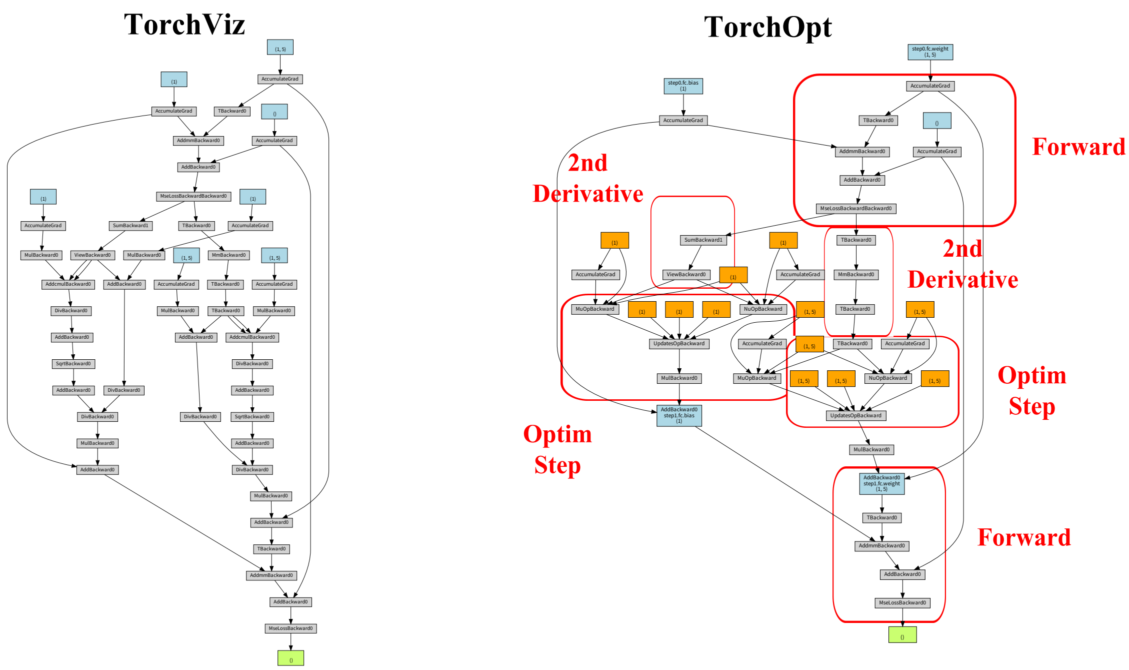


Figure 3: Gradient graph visualization comparison between TorchViz and TorchOpt. Red squares represent what gradient computation each node group accomplishes separately. Compared with TorchViz, `TorchOpt` fuses the operations within the Adam together (orange node) to reduce the complexity and provide a more straightforward visualization.

Appendix C. Differentiable Optimization Updating Scheme

The key challenge of consolidating these high-level and low-level APIs in a single library is that we must have a unified abstraction that allows different differentiable optimization algorithms to be easily declared. To address this, we design a differentiable optimization updating scheme, which can be easily extended to realize various differentiable optimization processes. As shown in Fig. 1, the scheme contains an outer level that has parameters $\boldsymbol{\phi}$ that can be learned end-to-end through the inner level parameters solution $\boldsymbol{\theta}'(\boldsymbol{\phi})$ (treating solution $\boldsymbol{\theta}'$ as a function of $\boldsymbol{\phi}$) by using the best-response derivatives $\partial \boldsymbol{\theta}'(\boldsymbol{\phi})/\partial \boldsymbol{\phi}$. It can be seen that the key component of this algorithm is to calculate the best-response (BR) Jacobian. From the BR-based perspective, `TorchOpt` supports three differentiation modes: explicit gradient over unrolled optimization, implicit differentiation, and zero-order differentiation.

C.1 Explicit Gradient Differentiation

```
# Functional API
opt = torchopt.adam()
# Define meta and inner parameters
meta_params = ...
fmodel, params = make_functional(model)
# Initialize optimizer state
state = opt.init(params)

for iter in range(iter_times):
    loss = inner_loss(fmodel, params, meta_params)
    grads = torch.autograd.grad(loss, params)
    # Apply non-inplace parameter update
    updates, state = opt.update(grads, state, inplace=False)
    params = torchopt.apply_updates(params, updates)

loss = outer_loss(fmodel, params, meta_params)
meta_grads = torch.autograd.grad(loss, meta_params)
```

```
# OOP API
# Define meta and inner parameters
meta_params = ...
model = ...
# Define differentiable optimizer
opt = torchopt.MetaAdam(model)

for iter in range(iter_times):
    # Perform the inner update
    loss = inner_loss(model, meta_params)
    opt.step(loss)

loss = outer_loss(model, meta_params)
loss.backward()
```

Listing 1: `TorchOpt` code snippet for explicit gradient. Left: Similiar to Optax Babuschkin et al. (2020), `TorchOpt` leverages `init`, `update` and `apply_updates` to conduct functional differentiable optimization. Right: OOP API similar with PyTorch `loss.step` API

C.2 Implicit Gradient Differentiation

```
# Functional API for implicit gradient
def stationary(params, meta_params, batch, labels):
    # Stationary condition construction
    ...
    return stationary condition

@torchopt.diff.implicit.custom_root(stationary)
def solve(params, meta_params, batch, labels):
    # Forward optimization process
    ...
    return optimal_params
```

```
# OOP API
class Module(torchopt.nn.ImplicitMetaGradientModule):
    def __init__(self, meta_module, ...):
        ...
    def forward(self, x):
        # Forward process
        ...
    def optimality(self, batch, labels):
        # Stationary condition construction
        ...
    def solve(self, batch, labels):
        # Forward optimization process
        ...
        return self
```

Listing 2: `TorchOpt` code snippet for implicit gradient. Left: Similar to JAXopt Blondel et al. (2021), users need to define the stationary function, and `TorchOpt` provides the decorator to wrap the solve function for enabling implicit gradient computation. Right: The OOP API needs users to implement the `solve` and `optimality` functions. `TorchOpt` will automatically make the `solve` function differentiable.

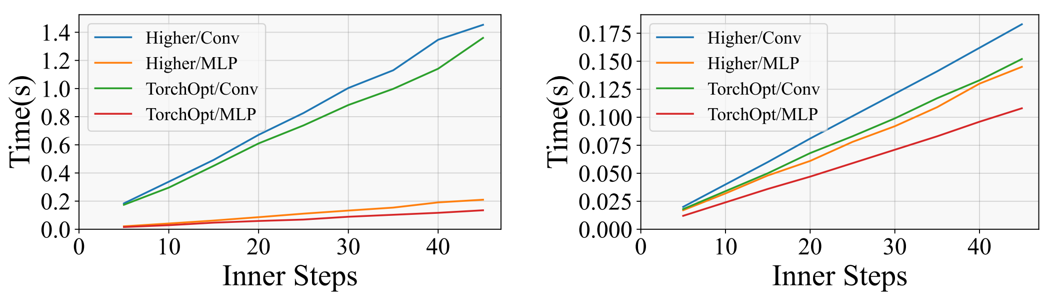
C.3 Zero-order Gradient Differentiation

```python
# Functional API
# Customize the noise sampling function in ES
def sample(sample_shape):
    ...
    return sample_noise

# Specify the method and parameter of ES
@torchopt.diff.zero_order(method, sample)
def forward(params, batch, labels):
    # Forward process
    return output
```

```python
# OOP API
class ESModule(torchopt.nn.ZeroOrderGradientModule):
    def sample(self, sample_shape):
        # Customize the noise sampling function in ES
        ...
        return sample_noise

    def forward(self, batch, labels):
        # Forward process
        ...
        return output
```

Listing 3: `TorchOpt` code snippet for zero-order differentiation.

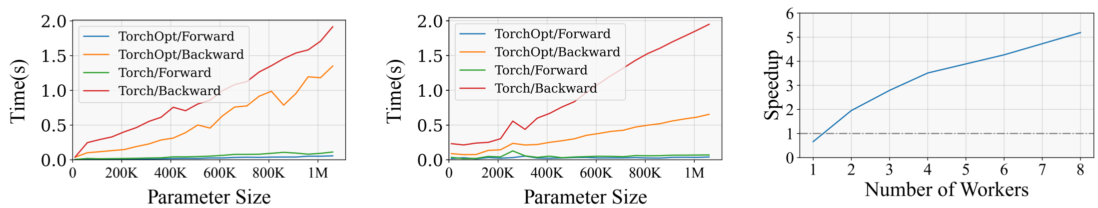
Appendix D. CPU/GPU-Accelerated Optimizers



(a) CPU-accelerated Meta optimization time (b) GPU-accelerated Meta optimization time

Figure 4: Performance of `TorchOpt` compared with Higher using MAML example, (*a*) and (*b*) are the meta-optimization time (Adam optimizer) in different inner steps and model structures.

Fig. 4 shows the meta-optimization time comparison with Higher (Grefenstette et al., 2019) in the CPU and GPU settings. Note that the meta-optimization process consists of extra computation beyond the optimizer, where we do not offer acceleration. However, the acceleration is still significant (around %25) for the MLP model in the CPU setting and both Conv/MLP model in the GPU setting.



(a) CPU-accelerated optimizer (b) GPU-accelerated optimizer (c) Distributed Speedup Ratio

Figure 5: Performance of `TorchOpt`, (*a*) and (*b*) are the forward/backward time (Adam optimizer) in different parameter sizes comparing `TorchOpt` and PyTorch, (*c*) is the speedup ratio on distributed implementation compared with the sequential implementation.

The results in Fig. 5(a) and Fig. 5(b) show that our design largely reduces the optimizer forward and backward time. Fig. 5(c) shows that `TorchOpt` can achieve linear speed-up with MAML when increasing the number of GPU workers.

Appendix E. Distributed Training

E.1 SPMD vs. MPMD in Distributed Optimization

SPMD (Single-Program-Multiple-Data) and MPMD (Multiple-Program-Multi-Data) are parallel processing paradigms pivotal in distributed optimization.
SPMD: Each processor runs an identical program, though on unique data subsets. Such uniformity simplifies task distribution and debugging. All units typically process a shard of the overarching dataset, necessitating synchronization to maintain pace uniformity.
MPMD: Diverse tasks can run different programs on separate processors, each potentially on distinct data subsets. While offering computational flexibility, it demands intricate synchronization, especially if tasks have interdependencies or require data interchange.
In differentiable optimization, the preference between SPMD and MPMD hinges on algorithmic specificity and data nature.

E.2 Distributed Framework

In Fig. 6 we show the overview of our distributed framework. As shown in Fig. 6, `TorchOpt` distributes a differentiable optimization job across multiple GPU workers and executes the workers in parallel. `TorchOpt` users can wrap code in the distributed Autograd module and achieve substantial speedup in training time with only a few changes in existing training scripts.

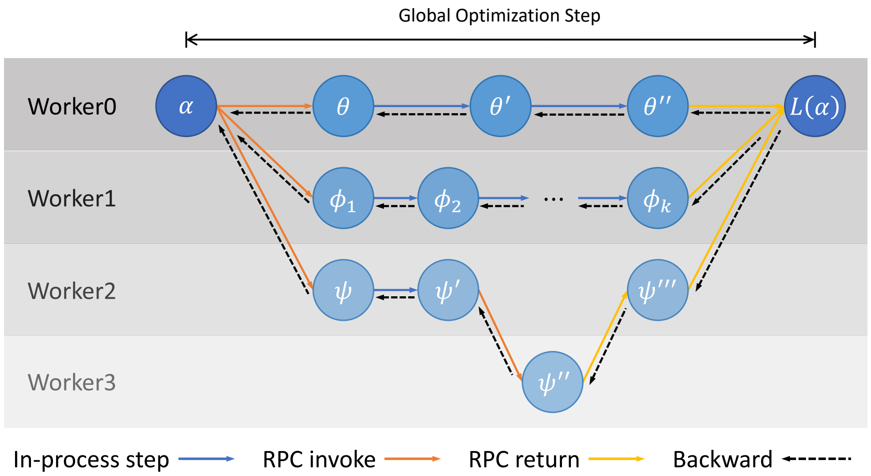


Figure 6: Overview of the Distributed RPC and Autograd framework. The forward and backward pass can be distributed on multiple processes and multiple nodes. The RPC framework supports heterogeneous workloads for different workers.

E.3 Distributed MAML Performance

In Fig. 7, we show the training accuracy and wall time comparison on the MAML Omniglot example. Distributed training achieves better performance and much higher computational efficiency.

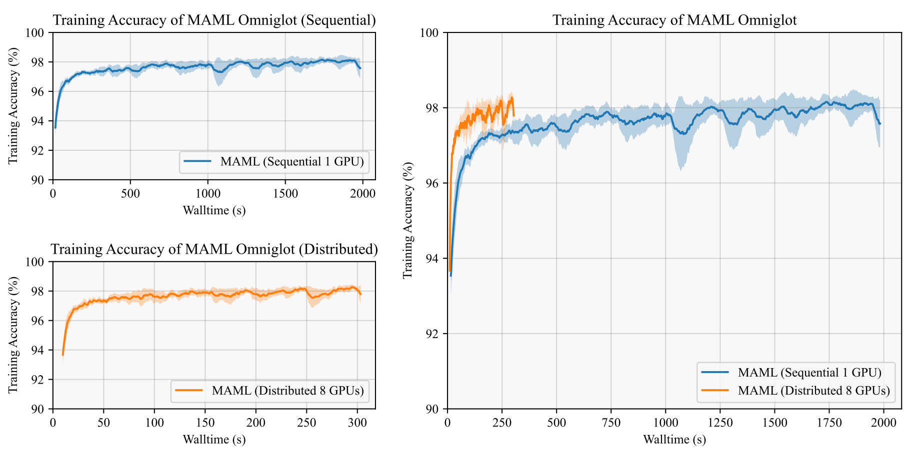


Figure 7: Wall time comparison between sequential training results and distributed training on 8 GPUs for MAML implemented with TorchOpt.

Appendix F. OpTree Performance

In Table. 2 we show the Speedup ratios of tree operations with ResNet models comparing OpTree, JAX XLA, PyTorch, and DM-Tree. In Fig. 8, 9 and 10, we show the time cost of tree-flatten, tree-unflatten, and tree-map trees in a different number of nodes comparing

OpTree, JAX XLA, PyTorch, and DM-Tree. OpTree achieves a large speedup compared with all baselines.

Table 2: Speedup ratios of tree operations with ResNet models. Here, `O`, `J`, `P`, `D` refer to OpTree, JAX XLA, PyTorch, and DM-Tree, respectively.

Module Scale	ResNet18			ResNet50			ResNet101			ResNet152		
Speedup Ratio	J / O	P / O	D / O	J / O	P / O	D / O	J / O	P / O	D / O	J / O	P / O	D / O
Tree Flatten	2.80	27.31	1.49	2.63	26.52	1.40	2.46	25.18	1.38	2.56	23.25	1.28
Tree UnFlatten	2.68	4.47	15.89	2.56	4.16	14.51	2.55	4.32	14.86	2.68	4.51	15.70
Tree Map	2.61	10.17	10.86	2.63	10.18	10.62	2.35	9.26	10.13	2.53	9.69	10.16

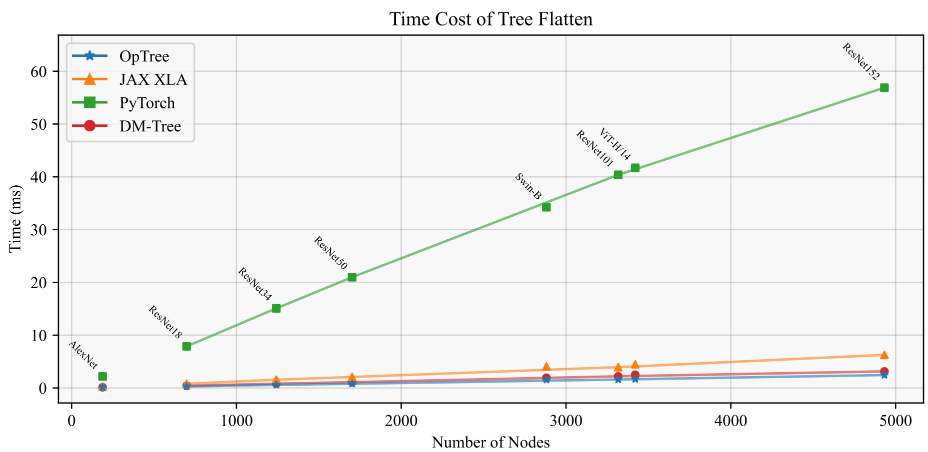


Figure 8: Tree-Flatten time comparison with respect to the tree scale.

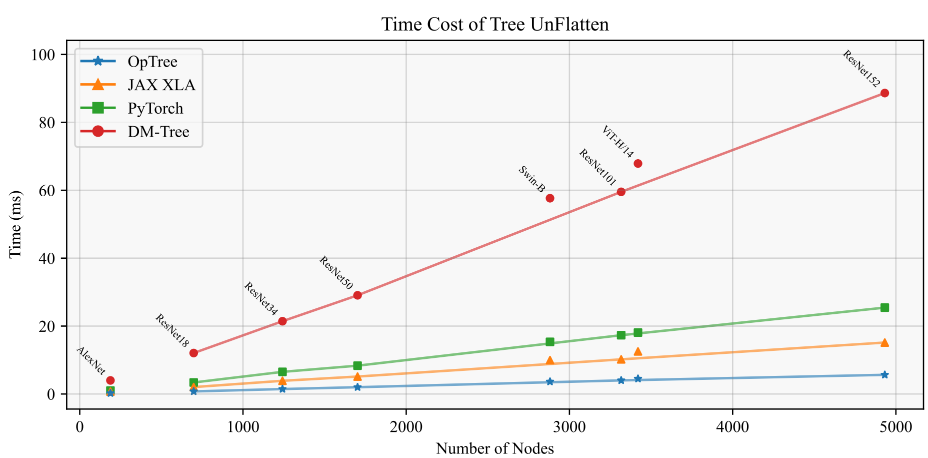


Figure 9: Tree-UnFlatten time comparison with respect to the tree scale.

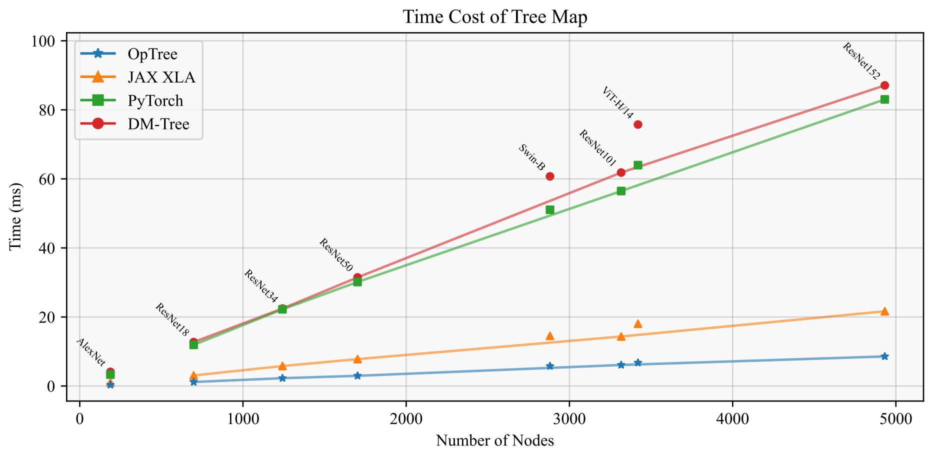


Figure 10: Tree-Map time comparison with respect to the tree scale.

Appendix G. Author Contributions

We summarise the main contributions from each of the authors as follows:
Bo Liu, Xidong Feng and Jie Ren created development roadmap for TorchOpt.
Jie Ren and Xuehai Pan implemented CPU/GPU-accelerated Adam operator.
Jie Ren and Bo Liu implemented differentiable optimizers.
Xuehai Pan and Jie Ren implemented optimized PyTree utilities.
Jie Ren, Xidong Feng and Bo Liu implemented explicit gradient differentiation functional API.
Xidong Feng, Jie Ren and Xuehai Pan, Bo Liu implemented explicit gradient differentiation OOP API.
Jie Ren, Xidong Feng and Bo Liu implemented implicit gradient differentiation functional API.
Xuehai Pan designed and implemented implicit gradient differentiation OOP API.
Xidong Feng and Jie Ren and Xuehai Pan implemented zero-order gradient differentiation functional and OOP API.
Xuehai Pan implemented distributed framework for differentiable optimization.
Xidong Feng, Bo Liu, Xuehai Pan and Jie Ren implemented tutorials for this project.
Bo Liu, Xidong Feng, Xuehai Pan and Jie Ren implemented examples for this project.
Xuehai Pan, Bo Liu and Jie Ren designed continuous integration and continuous delivery pipeline for this project.
Bo Liu, Xuehai Pan, Xidong Feng, Jie Ren implemented documentation for this project.
Xuehai Pan wrote the packaging tool for release distribution.
Xidong Feng wrote the README for this project.
Yao Fu designed and implemented differentiable RMSProp optimizer.
Luo Mai and Yaodong Yang led the project from its inception.
Xidong Feng, Bo Liu, Xuehai Pan, Jie Ren, Yao Fu, Luo Mai and Yaodong Yang wrote the paper.