`TorchOpt`: An Efficient Library for Differentiable Optimization

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

Abstract

Recent years have witnessed the booming of various differentiable optimization algorithms. These algorithms exhibit different execution patterns, and their execution needs massive computational resources that go beyond a single CPU and GPU. Existing differentiable optimization libraries, however, cannot support efficient algorithm development and multi-CPU/GPU execution, making the development of differentiable optimization algorithms often cumbersome and expensive. This paper introduces `TorchOpt`, a PyTorch-based efficient library for differentiable optimization. `TorchOpt` provides a unified and expressive differentiable optimization programming abstraction. This abstraction allows users to efficiently declare and analyze various differentiable optimization programs with explicit gradients, implicit gradients, and zero-order gradients. `TorchOpt` further provides a high-performance distributed execution runtime. This runtime can fully parallelize computation-intensive differentiation operations (e.g. tensor tree flattening) on CPUs / GPUs and automatically distribute computation to distributed devices. Experimental results show that `TorchOpt` achieves 5.2× training time speedup on an 8-GPU server. `TorchOpt` is available at: `https://github.com/metaopt/torchopt`.

1. Introduction

Recent years have witnessed the booming of differentiable optimization-based algorithms, including MAML [11], OptNet [2], and MGRL [25]. One of the important parts of differentiable optimization is meta-gradient, which is the gradient term of outer-loop variables by differentiating through the inner-loop optimization process. By leveraging meta-gradients, machine learning models can increase the sample efficiency [11] and the final performance [25].

Developing differentiable optimization algorithms poses several challenges. First, developers need to realize different inner-loop optimization and implement algorithms with gradient flows on complex computational graphs. Examples include explicit gradient computation of unrolled optimization [11, 25], implicit gradient for differentiable optimization [1, 2], evolutionary strategies

∗. Equal contribution.
†. Corresponding author.

Table 1: Comparison of `TorchOpt` and other differentiable optimization libraries. Note: ✓̸ indicates that the feature is partially supported.

	Differentiable Optimizer	Implicit Differentiation	Zero-order Gradient	Accelerated Operator	Distributed Training	Debugging Support	Backend
`higher` [13]	✓	✗	✗	✗	✗	✗	PyTorch
`Optax` [4]	✓	✗	✗	✗	✗	✗	JAX
`Torchmeta` [9]	✓	✗	✗	✗	✗	✗	PyTorch
`learn2learn` [3]	✓	✗	✗	✗	✓̸	✗	PyTorch
`JAXopt` [6]	✓	✓	✗	✗	✓	✗	JAX
`HyperTorch` [12]	✓	✓	✗	✗	✗	✗	PyTorch
`Betty` [8]	✓	✓̸	✓	✗	✓̸	✗	PyTorch
`TorchOpt` (ours)	✓	✓	✓	✓	✓	✓	PyTorch

for non-differentiable optimization [10], adjoint methods for differentiable ordinary differentiable equations [7], Gumbel-Softmax to differentiate through discrete distribution [14], and function interpolation for differentiable combinatorial solvers [21], etc. Second, differentiable optimization is computation-intensive. The meta-gradient computation requires heavy Hessian computation [11], high-dimensional linear equations [22], or large task-level batch size [20]. Such computation requirement often goes beyond what a single CPU and GPU can provide.

None of the existing differentiable optimization libraries can provide full support for efficient algorithm development and execution. Most libraries target a limited number of differentiable optimizers [3, 4, 9, 13]. They cannot fully support implicit differentiation [6, 8, 12], zero-order gradient [8], and distributed training [3, 6], as shown in Table 1. As a result, researchers have to implement algorithms in an ad-hoc and application-specific manner, making the development process cumbersome and expensive. Further, critical system optimization techniques (e.g. GPU optimization and distributed execution) are tightly coupled with certain algorithms and they are hard to be enabled for all possible algorithms.

To address these issues, this paper introduces `TorchOpt`, a PyTorch library that makes it efficient to develop and execute differentiable optimization algorithms with multiple GPUs. The design and implementation of `TorchOpt` make the following contributions:

(1) Unified and expressive differentiation mode for differentiable optimization. `TorchOpt` provides a general set of low-level / high-level / functional / Object-Oriented (OO) API to help users flexibly enable differentiable optimization within the computational graphs produced by PyTorch. Specifically, `TorchOpt` supports three differentiation modes for handling differentiable optimization problems: (i) Explicit gradient for unrolled optimization, (ii) implicit gradient for differentiable optimization, and (iii) zero-order gradient estimation for non-smooth/differentiable functions.

(2) High-performance and distributed execution runtime. `TorchOpt` aims to enable differentiable optimization algorithms to fully utilize CPUs and GPUs. To achieve this, we design (i) CPU/GPU accelerated optimizers (e.g., SGD, RMSProp, Adam) that realize the fusion of small differentiable operators and a full offloading of these operators to GPUs, (ii) parallel OpTree which can fully parallelize the nested structure flattening (Tree Operations), a key computation-intensive operation in differentiable optimization, on distributed CPUs, and (iii) a distributed auto-grad framework which can automatically identify the inner-loop tasks in differentiable optimizers and dispatch the execution of inner-loop tasks to distributed CPUs and GPUs.

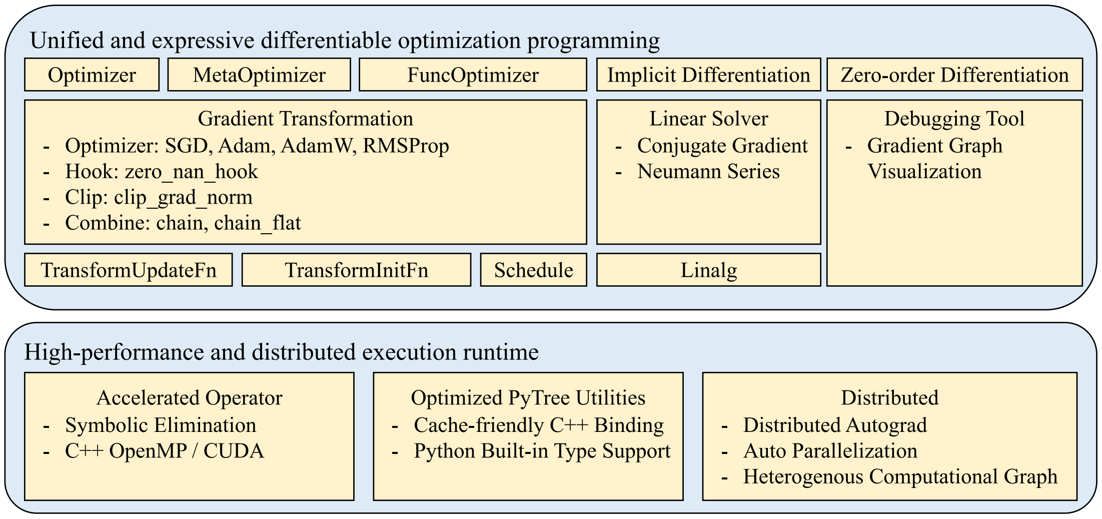

Unified and expressive differentiable optimization programming
Optimizer
MetaOptimizer
FuncOptimizer
Implicit Differentiation
Zero-order Differentiation
Gradient Transformation
- Optimizer: SGD, Adam, AdamW, RMSProp
- Hook: zero_nan_hook
- Clip: clip_grad_norm
- Combine: chain, chain_flat
Linear Solver
- Conjugate Gradient
- Neumann Series
Debugging Tool
- Gradient Graph Visualization
TransformUpdateFn
TransformInitFn
Schedule
Linalg
High-performance and distributed execution runtime
Accelerated Operator
- Symbolic Elimination
- C++ OpenMP / CUDA
Optimized PyTree Utilities
- Cache-friendly C++ Binding
- Python Built-in Type Support
Distributed
- Distributed Autograd
- Auto Parallelization
- Heterogenous Computational Graph

Figure 1: `TorchOpt`'s architecture overview.

Experimental results show that `TorchOpt` can reduce PyTorch optimizer forward/backward time, by 5× to 10× on CPU and 5× to 20× on GPU. `TorchOpt` can reduce the training time of the MAML [11] algorithm by 5.2× by distributing MAML computation to 8 GPUs.

2. `TorchOpt` Design and Implementation

2.1. Architecture Overview

Figure 1 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. 2.2. 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. 2.3. 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.

2.2. Programming Abstraction

`TorchOpt` aims to provide (i) high-level APIs that allow users to directly import differentiable optimizers, (ii) low-level APIs that enable automatic differentiation in different applications, and (iii) tools that allow users to analyze gradient flow in gigantic computational graphs.

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.

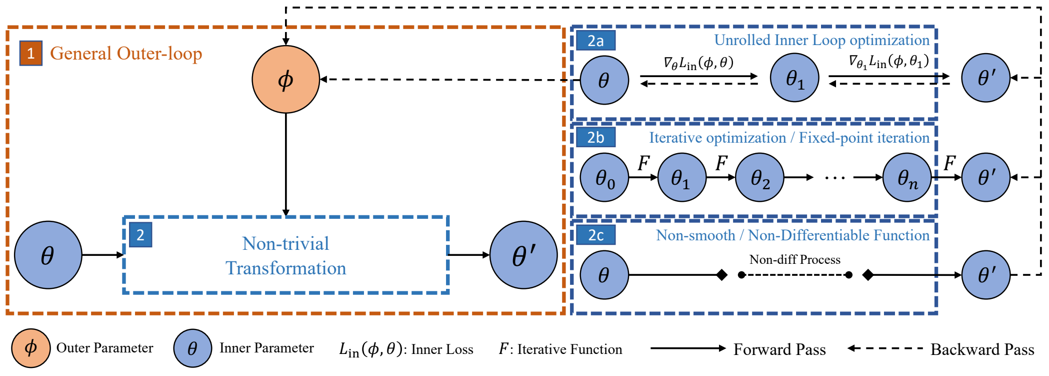

1 General Outer-loop
2 Non-trivial Transformation
2a Unrolled Inner Loop optimization
$\nabla_\theta L_{\text{in}}(\phi,\theta)$
$\nabla_{\theta_1} L_{\text{in}}(\phi,\theta_1)$
2b Iterative optimization / Fixed-point iteration
F
2c Non-smooth / Non-Differentiable Function
Non-diff Process
ϕ Outer Parameter
θ Inner Parameter
$L_{\text{in}}(\phi,\theta)$: Inner Loss
F: Iterative Function
Forward Pass
Backward Pass

Figure 2: `TorchOpt`'s differentiation modes. By formulating the problem as a differentiable problem, `TorchOpt` offers Autograd support for the backward pass (dotted lines).

As shown in Fig. 2, 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.

Explicit Gradient (EG) over unrolled optimization. As shown in Fig. 2-2a, the idea of EG is to treat the gradient step as a differentiable function and try to backpropagate through the unrolled optimization path. This differentiation mode is suitable for algorithms when the inner-level optimization solution is obtained by a few gradient steps, such as MAML [11] and MGRL [25]. `TorchOpt` offers both functional and OOP API. Refer to Listing 1 for the code snippet.

```
# 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.

Implicit Gradient (IG). As shown in Fig. 2-2b, by treating the solution $\boldsymbol{\theta}'$ as an implicit function of $\boldsymbol{\phi}$, the idea of IG is to directly get analytical best-response derivatives $\partial\boldsymbol{\theta}'(\boldsymbol{\phi})/\partial\boldsymbol{\phi}$ by implicit function theorem [16]. This is suitable for algorithms when the inner-level optimal solution is achieved ($\left.\frac{\partial F(\boldsymbol{\theta},\boldsymbol{\phi})}{\partial\boldsymbol{\theta}}\right|_{\boldsymbol{\theta}'} = 0$) or reaches some stationary conditions ($F(\boldsymbol{\theta}',\boldsymbol{\phi}) = 0$), such as iMAML [22] and DEQ [5]. `TorchOpt` offers functional/OOP API for supporting both conjugate gradient-based [22] and Neumann series-based [18] method. Refer to Listing 2 for the code snippet.

```
# 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.

Zero-order Differentiation (ZD). As shown in Fig. 2-2c, 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) [23]. ESMAML [24], and NAC [10], successfully solve the differentiable optimization problem based on ES. Instead of optimizing the objective F, ES optimizes a Gaussion smoothing objective defined as $\tilde{f}_\sigma(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{z}\sim\mathcal{N}(\mathbf{0},\boldsymbol{I}_d)}[f(\boldsymbol{\theta}+\sigma\boldsymbol{z})]$, where σ denotes precision. The gradient of such objective is $\nabla_\theta \tilde{f}_\sigma(\theta) = \frac{1}{\sigma}\mathbb{E}_{\boldsymbol{z}\sim\mathcal{N}(\mathbf{0},\boldsymbol{I}_d)}[f(\boldsymbol{\theta}+\sigma\boldsymbol{z})\boldsymbol{z}]$. `TorchOpt` also offers functional and OOP API for ES method. Refer to Listing 3 for code snippets.

```
# 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
```

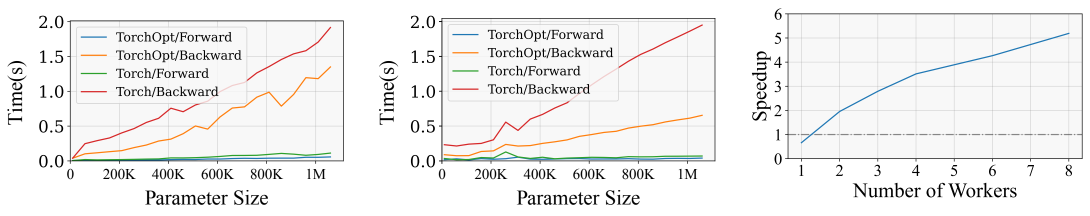
```
# OOP API
class ESModule(torchopt.nn.ZeroOrderGradientModule):
    def sample(self, sample_shape):
        # Customize the noise sampling function in ES
        ...
        return sample_noise

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

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

Gradient graph visualization. Complex gradient graph in meta-learning/differentiable optimization brings a great challenge for managing the gradient graph and debugging the code. `TorchOpt` provides a visualization tool that draws variable (e.g. network parameters or meta parameters) names on the gradient graph for better analysis. The visualization tool is modified from TorchViz [26]. Compared with TorchViz, `TorchOpt` fuses the operations within the optimization algorithm (such as Adam) to reduce the complexity and provide simpler visualization. Refer to the visualization example in Appendix A.

2.3. High-performance and Distributed Runtime

CPU/GPU-accelerated optimizers. We take the optimizer as a whole instead of separating it into several basic operators (e.g., `sqrt` and `div`). Therefore, by manually writing the forward and backward functions, we can perform the symbolic reduction. In addition, we can store some intermediate data that can be reused during the back-propagation. Our design reduces computation and also benefits numerical stability (by explicitly canceling some $0/0$ cases in higher gradient computation). 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



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

Figure 3: 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 multi-GPUs using RPC compared with the sequential implementation.

using `torch.autograd.Function`. The results in Fig. 3(a) and Fig. 3(b) show that our design largely reduces the optimizer forward and backward time. Refer to Appendix B for experimental results comparing `TorchOpt`and Higher [13] on the MAML example.

Memory-efficient and cache-friendly PyTree. 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 the 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 D for `OpTree` experimental results.

Distributed differentiable optimization. `TorchOpt` allows users to reduce training time by using parallel GPUs. Different from existing MPI-based synchronous training [19] and asynchronous model averaging [15] systems, `TorchOpt` adopts RPC as a flexible yet performance communication backend. The distributed GPUs perform differentiable optimization tasks in parallel. These GPUs are coordinated by a chosen GPU device which realizes the synchronous execution of parallel GPUs, thus guaranteeing the convergence of the model in a distributed training setting).

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. Fig. 3(c) shows that `TorchOpt` can achieve linear speed-up with MAML when increasing the number of GPU workers (more details in Appendix C).

3. Conclusion and Future work

This paper introduces `TorchOpt`, a novel efficient differentiable optimization library for PyTorch. Experimental results show that `TorchOpt` can act as a user-friendly, high-performance, and scalable library when supporting challenging gradient computation with PyTorch. In the future, we aim to support more complex differentiation modes and cover more non-trivial gradient computation problems, such as adjoint methods for the gradient of ODE solutions, RL or Gumbel-Softmax method for differentiating through discrete distribution, and differentiable combinatorial solvers. `TorchOpt` has already been used for meta-gradient research problem [17] and we believe it can be served as an important auto-differentiation tool for more differentiable optimization problems.

Acknowledgments

We thank Vincent Moens for his constructive comments on `TorchOpt`.

References

[1] Akshay Agrawal, Brandon Amos, Shane T. Barratt, Stephen P. Boyd, Steven Diamond, and J. Zico Kolter. Differentiable convex optimization layers. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems*, 2019.

[2] Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, 2017.

[3] Sébastien M R Arnold, Praateek Mahajan, Debajyoti Datta, Ian Bunner, and Konstantinos Saitas Zarkias. learn2learn: A library for Meta-Learning research. *arXiv preprint arXiv:2008.12284*, 2020.

[4] Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, John Quan, George Papamakarios, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Luyu Wang, Wojciech Stokowiec, and Fabio Viola. The DeepMind JAX Ecosystem, 2020. URL http://github.com/deepmind.

[5] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. *Advances in Neural Information Processing Systems*, 32, 2019.

[6] Mathieu Blondel, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, and Jean-Philippe Vert. Efficient and modular implicit differentiation. *arXiv preprint arXiv:2105.15183*, 2021.

[7] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 2018.

[8] Sang Keun Choe, Willie Neiswanger, Pengtao Xie, and Eric P. Xing. Betty: An automatic differentiation library for multilevel optimization. *arXiv preprint arXiv:2207.02849*, 2022.

[9] Tristan Deleu, Tobias Würfl, Mandana Samiei, Joseph Paul Cohen, and Yoshua Bengio. Torchmeta: A Meta-Learning library for PyTorch. *arXiv preprint arXiv:1909.06576*, 2019.

[10] Xidong Feng, Oliver Slumbers, Ziyu Wan, Bo Liu, Stephen McAleer, Ying Wen, Jun Wang, and Yaodong Yang. Neural auto-curricula in two-player zero-sum games. *Advances in Neural Information Processing Systems*, 34:3504–3517, 2021.

[11] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, 2017.

[12] Riccardo Grazzi, Luca Franceschi, Massimiliano Pontil, and Saverio Salzo. On the iteration complexity of hypergradient computation. *Thirty-seventh International Conference on Machine Learning (ICML)*, 2020.

[13] 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.

[14] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[15] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter 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. URL `http://www.vldb.org/pvldb/vol12/p1399-koliousis.pdf`.

[16] Steven George Krantz and Harold R Parks. *The implicit function theorem: history, theory, and applications*. Springer Science & Business Media, 2002.

[17] Bo Liu, Xidong Feng, Jie Ren, Luo Mai, Rui Zhu, Haifeng Zhang, Jun Wang, and Yaodong Yang. A theoretical understanding of gradient bias in meta-reinforcement learning. *arXiv preprint arXiv:2112.15400*, 2021.

[18] 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.

[19] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter 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.

[20] Junhyuk Oh, Matteo Hessel, Wojciech M Czarnecki, Zhongwen Xu, Hado P van Hasselt, Satinder Singh, and David Silver. Discovering reinforcement learning algorithms. *Advances in Neural Information Processing Systems*, 33:1060–1070, 2020.

[21] Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*, 2019.

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

[23] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[24] Xingyou Song, Wenbo Gao, Yuxiang Yang, Krzysztof Choromanski, Aldo Pacchiano, and Yunhao Tang. Es-maml: Simple hessian-free meta learning. *arXiv preprint arXiv:1910.01215*, 2019.

[25] Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-gradient reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

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

Appendix A. Gradient Graph Visualization

Fig. 4 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.

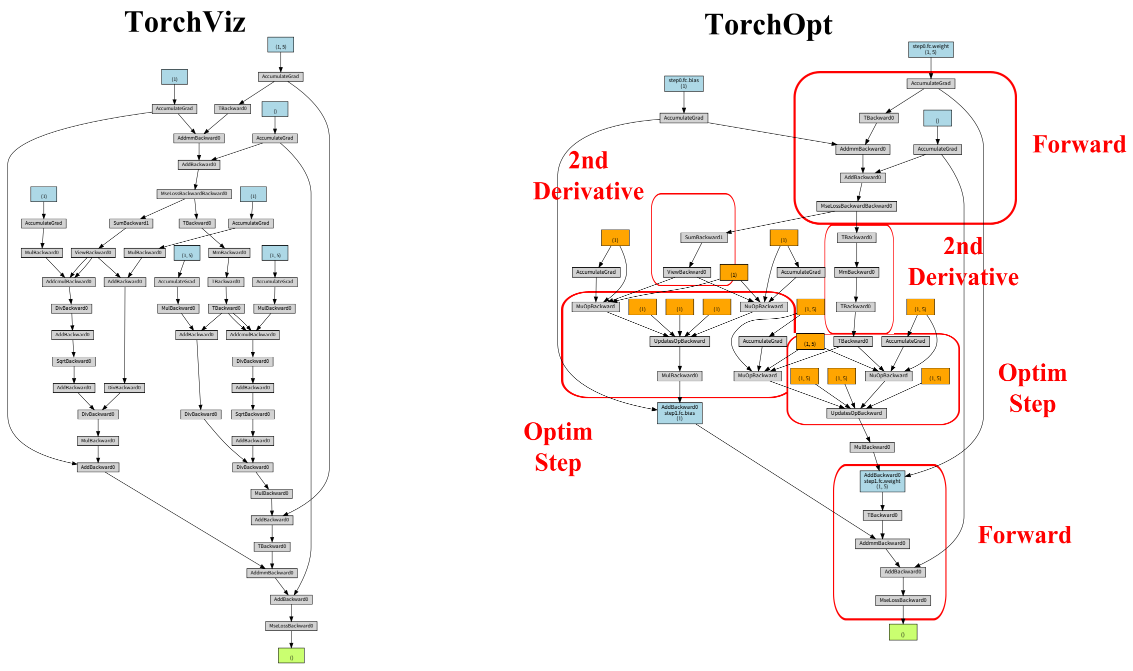
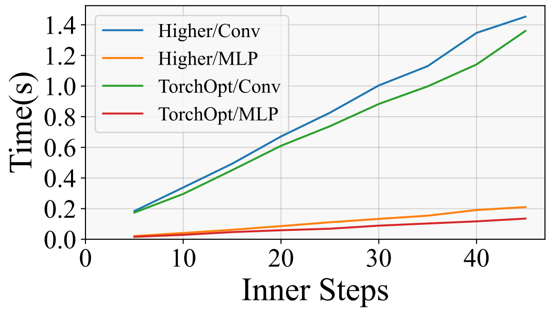


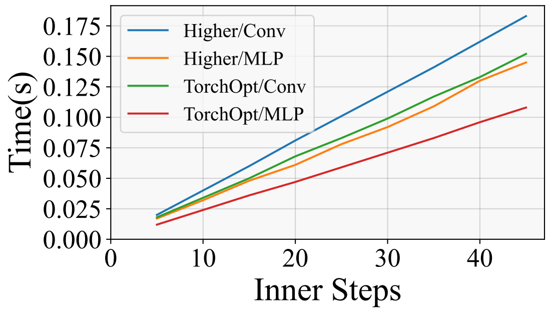
Figure 4: Gradient graph visualization comparison between TorchViz and TorchOpt.

Appendix B. CPU/GPU-Accelerated Optimizers

Fig. 5 shows the meta-optimization time comparison with Higher [13] 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 Meta optimization time



(*b*) GPU-accelerated Meta optimization time

Figure 5: 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.

Appendix C. Distributed Training

C.1. Distributed Framework

In Fig. 6 we show the overview of our distributed framework.

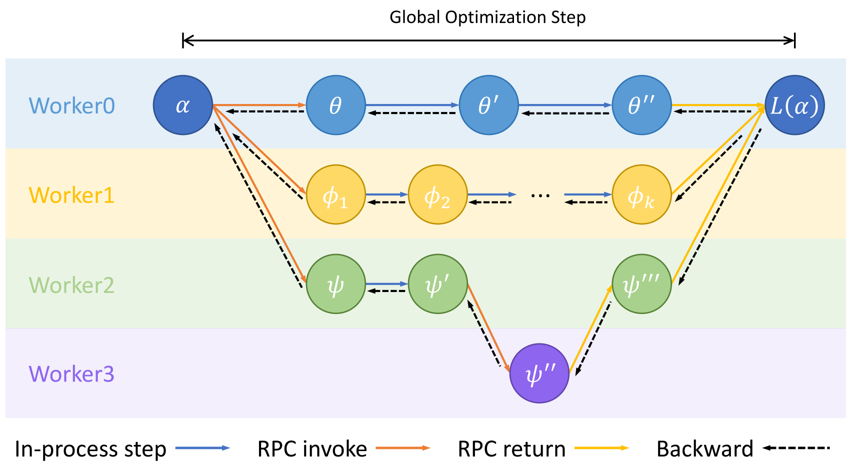


Figure 6: Overview of the Distributed RPC and Autograd framework. The forward and backward pass can be distributed on multiple processes and multiple nodes. The RPC framework supports heterogeneous workloads for different workers.

C.2. 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.

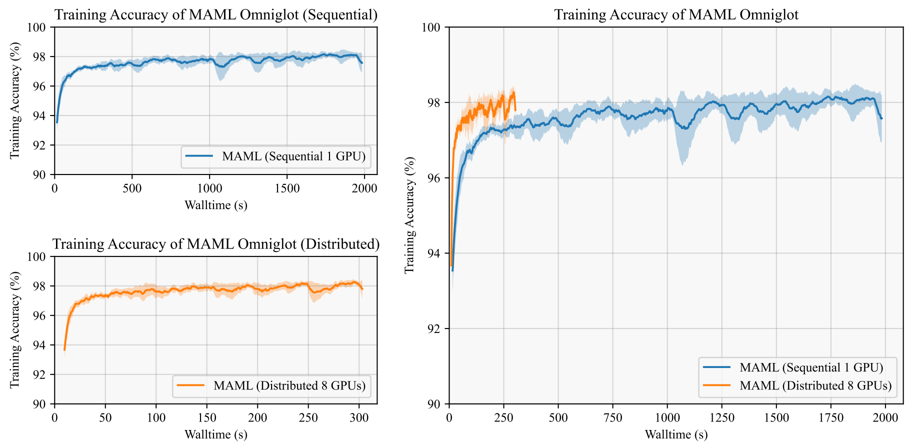


Figure 7: Wall time comparison between sequential training results and distributed training on 8 GPUs for MAML implemented with TorchOpt.

Appendix D. 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

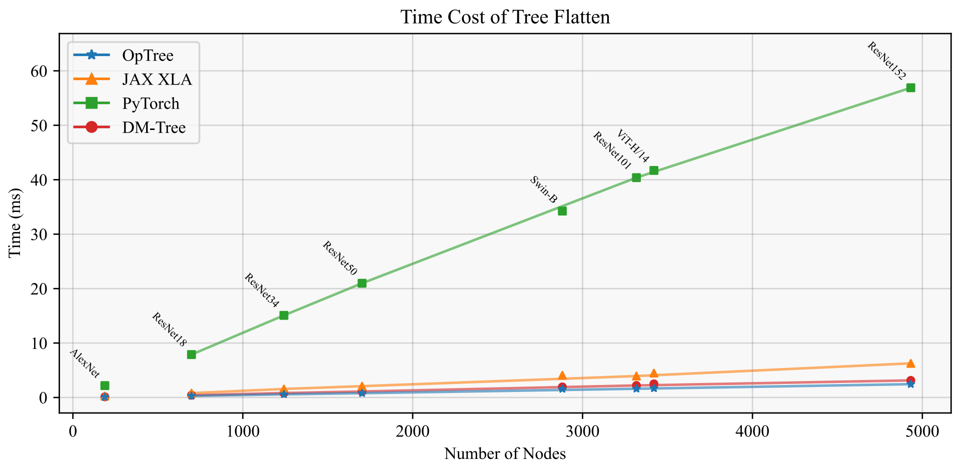


Figure 8: Tree-Flatten time comparison with respect to the tree scale.

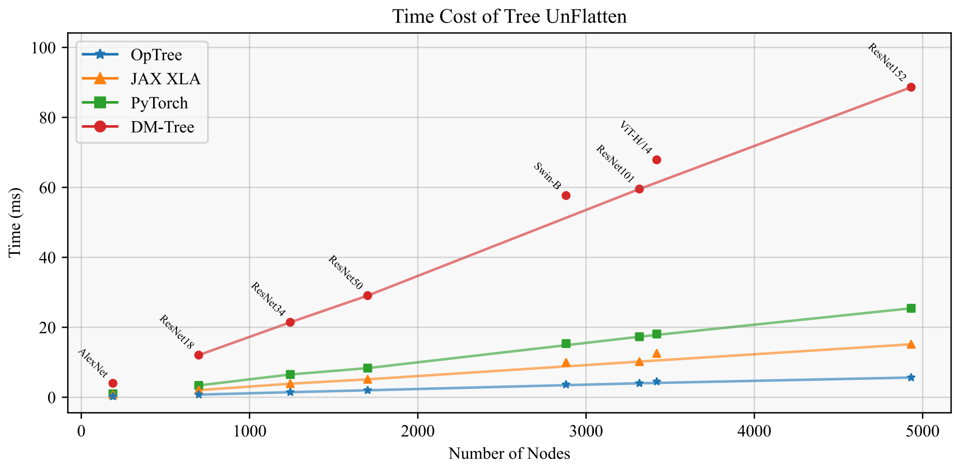


Figure 9: Tree-UnFlatten time comparison with respect to the tree scale.

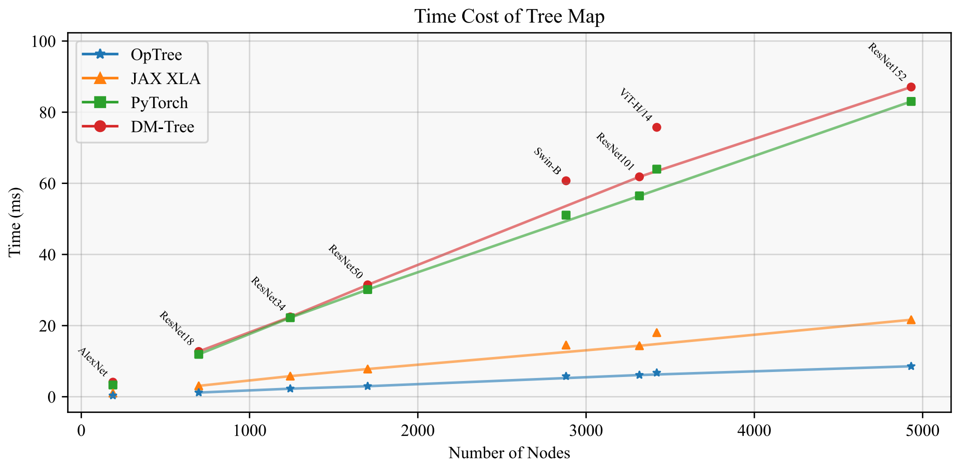


Figure 10: Tree-Map time comparison with respect to the tree scale.