TorchOpt: An Efficient Library for Differentiable Optimization

author names withheld

Under Review for OPT 2022

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 bi-level 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 flatten) on CPUs/GPUs and automatically distribute computation to distributed devices. Experimental results show that TorchOpt outperforms state-of-the-art libraries by 7× on an 8-GPU server. TorchOpt will be open source with a permissive Apache-2.0 License.

1. Introduction

Recent years have witnessed the booming of bi-level optimization-based algorithms, including MAML [12], OptNet [3] and MGRL [23]. One of the important parts of bi-level 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 [12] and the final performance [23].

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 [12], implicit gradient for differentiable optimization [2, 3], evolutionary strategies for non-differentiable optimization [11], adjoint methods for differentiable ordinary differentiable equations [8], Gumbel-Softmax to differentiate through discrete distribution [15], and function interpolation for differentiable combinatorial solvers [19], etc. Second, bi-level optimization is computation-intensive. The meta-gradient computation requires heavy Hessian computation [12], high-dimensional linear equations [20], or large task-level batch size [18]. 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 [4, 5, 10, 14]. They cannot fully support implicit differentiation [7, 9, 13], zero-order gradient [9], and distributed training [4, 7], as shown in Table 1. As a result, differentiable optimization 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

© .

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

	Differentiable Optimizer	Implicit Differentiation	Zero-order Gradient	Accelerated Operator	Distributed Training	Debugging Support	Backend
`higher` [14]	✓	✗	✗	✗	✗	✗	PyTorch
`Optax` [5]	✓	✗	✗	✗	✗	✗	JAX
`Torchmeta` [10]	✓	✗	✗	✗	✗	✗	PyTorch
`learn2learn` [4]	✓	✗	✗	✗	✓̸	✗	PyTorch
`JAXopt` [7]	✓	✓	✗	✗	✓	✗	JAX
`HyperTorch` [13]	✓	✓	✗	✗	✗	✗	PyTorch
`Betty` [9]	✓	✓̸	✓	✗	✓̸	✗	PyTorch
`TorchOpt` (ours)	✓	✓	✓	✓	✓	✓	PyTorch

(e.g. GPU optimization and distributed execution) is 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 bi-level optimization. `TorchOpt` provides a general set of Low- / High-level / Functional / Object-Oriented (OO) API to help user flexibly enable differentiable optimization within the computational graphs produced by PyTorch. Specifically, `TorchOpt` supports three differentiation modes for handling bi-level optimization problems: Explicit Gradient for unrolled optimization, Implicit Gradient for differentiable optimization, and Zero-order Gradient estimation for non-smooth/differentiable function.

(2) High-performance and distributed differentiable optimizer implementation. `TorchOpt` realizes a collection of techniques that make differentiable optimizations to fully utilize CPUs and GPUs. These techniques include: (i) CPU/GPU accelerated optimizers (e.g., SGD, RMSProp, Adam) by operator fusion on CPU/GPU. (ii) Parallel OpTree: Optimized PyTree for enabling fast nested structure flattening (Tree Operations) using C++. In addition, `TorchOpt` can automatically identify the inner-loop tasks in differentiable optimizers and dispatch the execution of inner-loop tasks to distributed CPUs and GPUs.

Experimental results show that `TorchOpt` can outperform Higher [14] by 38% faster with a NVIDIA V100 GPU. On a multi-GPU server, `TorchOpt` can reduce the execution time of the MAML [12] algorithm by $5.2\times$ faster by using distributed GPUs.

2. `TorchOpt` Design and Implementation

2.1. `TorchOpt` Design

Figure 1 introduces the key components in `TorchOpt`:

- **Unified and expressive bi-level optimization programming.** The Application Layer aims to offer all the user-oriented high-level and low-level APIs with different levels of abstraction.

- **High-performance and distributed execution runtime.** The Acceleration Layer provides accelerated solutions for application layers, including accelerated operators for both CPU and GPU and distributed training for multi-node multi-GPU settings.

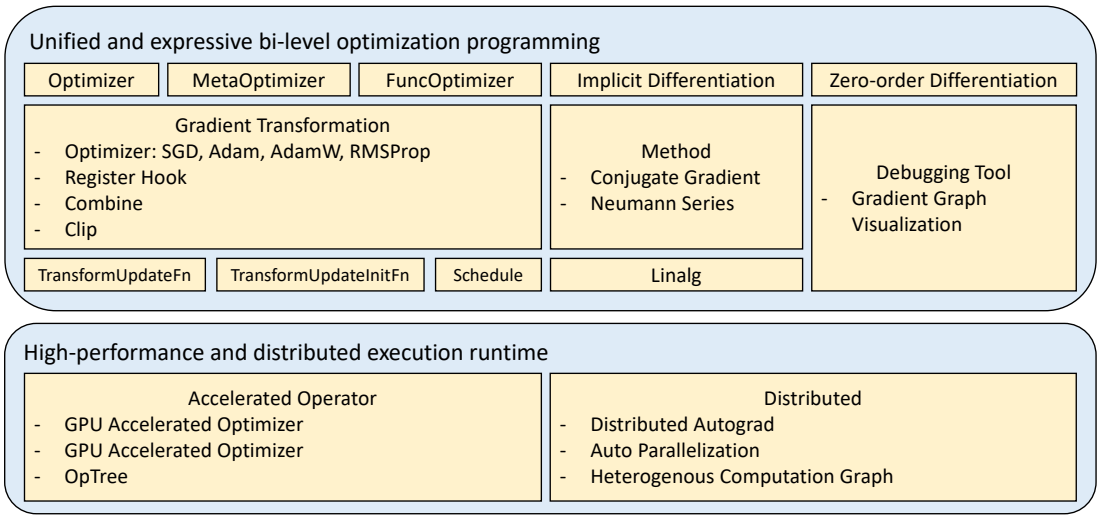


Figure 1: TorchOpt's architecture overview.

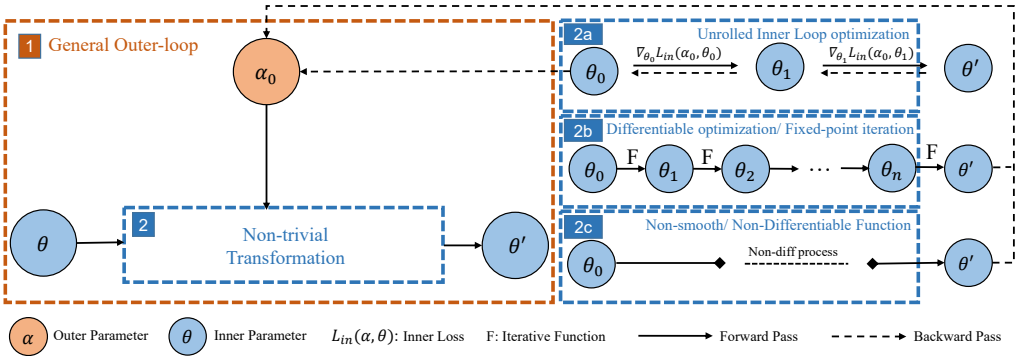


Figure 2: TorchOpt's differentiation modes. By formulating the problem as a Bi-level problem, `TorchOpt` offers the support of automatic backward process (dotted lines).

Together, the Application Layer let 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 detailedly described in Sec. 3.2. Then the Acceleration Layer 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. 3.3. Additionally, in the Backend Layer 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 subtree is small. We build a thread pool to perform tree operations in parallel.

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 bilevel-optimization updating scheme, which can be easily extended to realize various differentiable optimization processes.

As shown in Fig. 2, the scheme contains an outer level which has parameters ϕ that can be learned end-to-end through the inner level parameters solution $\theta^\star(\phi)$ by using the best-response derivatives $\partial\theta^\star(\phi)/\partial\phi$. `TorchOpt` supports three differentiation modes. It can be seen that the key component of this algorithm is to calculate the best-response (BR) Jacobian. From the BR-based perspective, existing gradient methods can be categorized into three groups: explicit gradient over unrolled optimization, implicit differentiation, and zero-order gradient differentiation.

Explicit gradient over unrolled optimization (EG) (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 [12], MGRL [23]. `TorchOpt` offers both functional and OOP API for EG to fit different user applications. Refer to Listing 1 for code snippet.

```python
# Functional API
opt = torchopt.adam()
# Define meta and inner parameters
meta_p = ...
fmodel, p = make_functional(model)
# Initial state
s = opt.init(p)

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

loss = outer_loss(fmodel, p, meta_p)
meta_grads = autograd.grad(loss, meta_p)
```

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

for iter in range(iter_times):
    # perform inner update
    loss = inner_loss(model, meta_p)
    opt.step(loss)

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

Listing 1: `TorchOpt` Code Snippet for Explicit Gradient.

Implicit gradient (IG) (Fig. 2-2b). By treating the solution $\theta^\star$ as an implicit function of ϕ, the idea of IG is to directly get analytical best-response derivatives $\partial\theta^\star(\phi)/\partial\phi$ by implicit function theorem [16]. This is suitable for algorithms when the inner-level optimal solution is achieved ($\frac{\partial F(\theta,\phi)}{\partial\theta}|_{\theta^\star} = 0$) or reaches some stationary conditions ($F(\theta,\phi) = \theta$), such as iMAML [20], DEQ [6]. `TorchOpt` offers functional/OOP API for supporting both conjugate gradient based [20] and neumann series [17] based IG method. Refer to Listing 2 for code snippet.

Zero-order Differentiation (ZD) (Fig. 2-2c). When the inner-loop process is non-differentiable or one wants to get rid of the heavy computation burdens in the previous two modes (brought by Hessian), one can choose ZD. ZD typically gets gradient based on zero-order estimation, such as finite-difference, or Evolutionary Strategy (ES) [21]. ESMAML [22], and NAC [11] , successfully solve the bi-level optimization problem based on ES. Instead of optimizing the objective F, ES optimize a smoothed objective defined as $\tilde{f}_\sigma(\theta) = \mathbb{E}_{\mathbf{g}\sim\mathcal{N}(0,\mathbb{I}_d)}[f(\theta+\sigma\mathbf{g})]$. The gradient of such smoothed function is $\nabla_\theta\tilde{f}_\sigma(\theta) = \frac{1}{\sigma}\mathbb{E}_{\mathbf{g}\sim\mathcal{N}(0,\mathbf{I}_d)}[f(\theta+\sigma\mathbf{g})\mathbf{g}]$. `TorchOpt` also offers functional and OOP API for the ES method. Refer to Listing 3 for code snippet.

```python
# Functional API for implicit gradient
def stationary(params, meta_params, data):
    # stationary condition construction
    return stationary condition

@torchopt.implicit.custom_root(stationary)
def solve(params, meta_params, data):
    # forward optimization process
    return output
```

```python
# OOP API
@torchopt.implicit.implicit_module()
class Module(nn.Module):
    def forward(self, data):
        ...
    def solve(self, meta_params, data):
        # forward optimization process
        ...
    def stationary(self, meta_params, data):
        # stationary condition construction
        ...
```

Listing 2: `TorchOpt` code snippet for Implicit Gradient.

```python
# Functional API
# Specify method and parameter of ES
@torchopt.zero_order.es_grad(method, params)
def forward(meta_params, data):
    # forward process
    return output
```

```python
# OOP API
@torchopt.zero_order.es_grad_module
class ESModule(nn.Module):
    def __init__(self, module):
        ...

    def forward(self, batch):
        ...
```

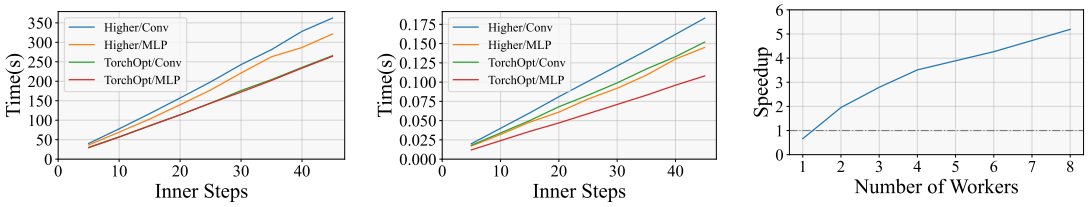
Listing 3: `TorchOpt` code snippet for Zero-order Differentiation.

Besides the differentiation mode, `TorchOpt` also offers **Gradient Graph Visualization** as a debugging tool. Complex gradient flow in meta-learning/bi-level optimization brings a great challenge for managing the gradient flow 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 [1]. Compared with TorchViz, `TorchOpt` fuses the operations within the optimization algorithm (such as Adam) together to reduce the complexity and provide simpler visualization. Because of the space limit, we present a 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 (about 30% of Adam) 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 they can be called by Python, and then we define the forward and backward behavior using `torch.autograd.Function`. The results in Fig. 3(a) and Fig. 3(b) show that our design reduces about 25% of overall time and meta optimization time.

OpTree: Accelerated PyTree. We implement the `PyTree` to enable fast nested structure flatten using C++. The tree operations (e.g., flatten and unflatten) are very important in enabling functional and Just-In-Time (JIT) features of deep learning frameworks. By implementing it in C++, we can use some cache/memory friendly structures (e.g., `boost::container::small_vector`) to improve the performance. We further build a thread pool to perform tree operations in parallel.



(a) Overall Time (b) Meta optimization Time (c) Distributed Speedup

Figure 3: Performance of `TorchOpt`, (a) and (b) are the result of `TorchOpt` on single GPU, (c) is the result on multi GPUs using RPC.

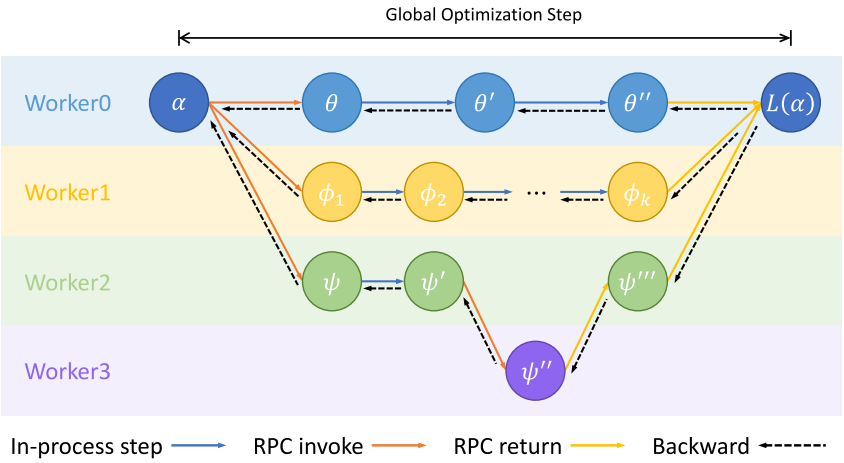


Figure 4: Overview of the Distributed RPC and Autograd framework.

Distributed Training. `TorchOpt` provides distributed training features based on the PyTorch RPC module for better training speed and multi-node multi-GPU support. Different from the MPI-like parallelization paradigm, which uses multiple homogenous workers and requires carefully designed communication hooks, the RPC APIs allow users to build their optimization pipeline more flexibly. Illustrated in Fig. 4, the RPC framework distributes workloads across multiple workers and executes in parallel. Users can wrap their code with the distributed autograd and benefit from the instant speedup with minimal code changes. Fig. 3(c) presents the experimental results of MAML, which illustrates the linear relationship between the speed-up ratio and the number of workers.

3. Conclusion and Future work

This paper presents `TorchOpt` a high-performance advanced optimization library for PyTorch. `TorchOpt` provides a user-friendly and efficient framework to fill the vacancy of complex gradient computation problems in PyTorch. In the future, `TorchOpt` aims at supporting more complex differentiation modes and covering more non-trivial gradient computation problems, such as the adjoint method for the gradient of ODE solution, RL or Gumbel-Softmax method for differentiating through discrete distribution, and differentiable combinatorial solver. We believe `TorchOpt` can be served as an important auto-differentiation tool for related machine learning research and applications.

References

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

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] Riccardo Grazzi, Luca Franceschi, Massimiliano Pontil, and Saverio Salzo. On the iteration complexity of hypergradient computation. 2020.

[14] 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.

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

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

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

Appendix A. Visualization

Fig. 5 show 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 simpler visualization.

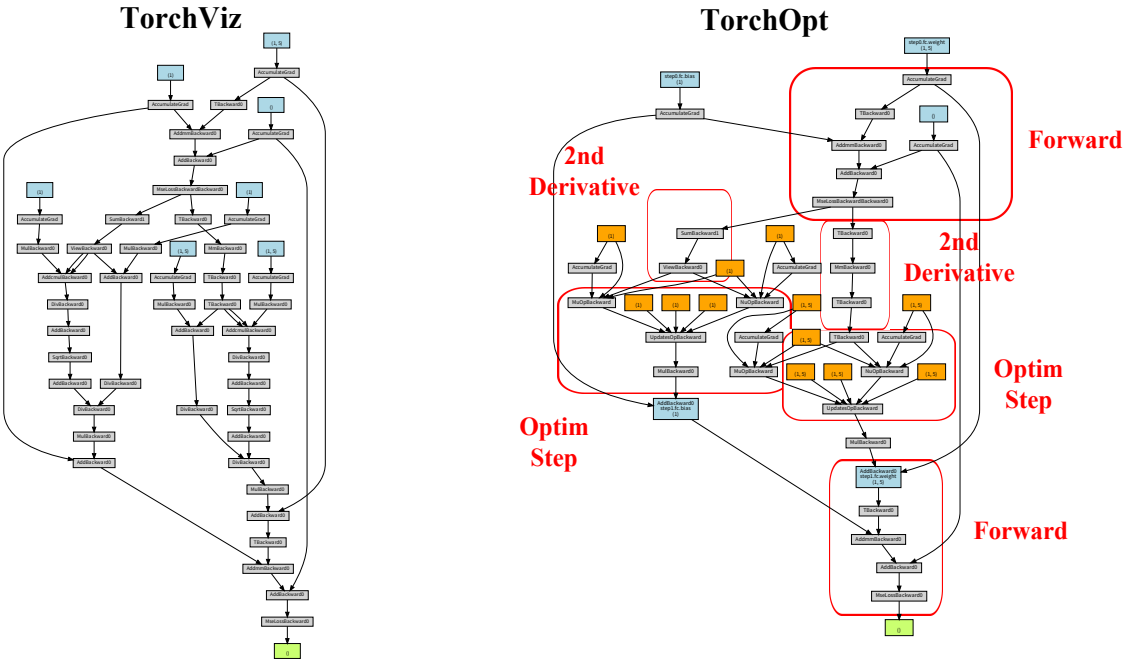


Figure 5: Visualization comparison between TorchViz and TorchOpt