`higher`: A Pytorch Meta-Learning Library

Edward Grefenstette*
Facebook AI Research
egrefen@fb.com

Brandon Amos
Facebook AI Research
bda@fb.com

Denis Yarats
NYU & Facebook AI Research
denisyarats@cs.nyu.com

Phu Mon Htut
NYU
pmh330@nyu.edu

Artem Molchanov
University of Southern California
molchano@usc.edu

Franziska Meier
Facebook AI Research
fmeier@fb.com

Douwe Kiela
Facebook AI Research
dkiela@fb.com

Kyunghyun Cho
NYU & Facebook AI Research
kyunghyuncho@fb.com

Soumith Chintala
Facebook AI Research
soumith@fb.com

Abstract

Many (but not all) approaches self-qualifying as "meta-learning" in machine learning fit a common pattern of approximating the solution to a nested optimization problem. In this paper, briefly summarize a formalization of shared pattern, GIMLI. *We discuss the engineering challenges of implementing this class of problems in known differentiable programming libraries, and describe a library addressing these challenges,* `higher`*, which we share with the community to assist and enable future research into these kinds of meta-learning approaches.*

1. Introduction

Although it is by no means a new subfield of machine learning research (see e.g. [23, 4, 15]), there has recently been a surge of interest in meta-learning (e.g. [17, 1, 11]). This is due to the methods meta-learning provides, amongst other things, for producing models that perform well beyond the confines of a single task, outside the constraints of a static dataset, or simply with greater data efficiency or sample complexity. Due to the wealth of options in what could be considered "meta-" to a learning problem, the term itself may have been used with some degree of underspecification. However, it turns out that many meta-learning approaches, in particular in the recent literature, follow the pattern of optimizing the "meta-parameters" of the training process by nesting one or more inner loops in an outer training loop. Such nesting enables training a model for several steps, evaluating it, calculating or approximating the gradients of that evaluation with respect to the meta-parameters, and subsequently updating these meta-parameters.

In this paper, we briefly summarize the **G**eneralized **I**nner Loop **M**eta-**L**earn**i**ng formalism we present in [14] (along with an accompanying analysis of its requirements, and algorithm to implement it). We then describe a lightweight PyTorch library, `higher`, developed based on this analysis and algorithm, that enables the straightforward implementation of any meta-learning approach that fits within the GIMLI framework (or generically, any approach which seeks to backpropagate through an optimizaiton loop) in canonical PyTorch, such that existing codebases require minimal changes, supporting third party module implementations and a variety of optimizers.

2. Generalized Inner Loop Meta-Learning

Whereby "meta-learning" is taken to mean the process of "learning to learn", we can describe it as a nested optimization problem according to which an outer loop optimizes meta-variables controlling the optimization of model parameters within an inner loop. The aim of the outer loop should be to improve the meta-variables such that the inner loop produces models which are more suitable according to some criterion. The general form of such nested optimization problems is described under the GIMLI framework in [14]. Essentially it shows that under well-defined and

*Corresponding author.

provable conditions, an form of backpropagation-through-backpropagation can be specified as a dynamic programming algorithm, and permits the calculation of gradients on "meta-losses" (e.g. validation loss) with regard to arbitrary meta-variables within the training loop, subject to certain conditions being fulfilled. We refer the reader to the details in that paper regarding the theoretical aspects of this claim, in order to focus on solving the engineering challenges of permitting the efficient learning of training loop meta-parameters without needing to write (much) non-canonical code in a popular differentiable programming framework.

2.1. Examples

Many of the papers referenced below contain excellent and thorough reviews of the literature most related to the type of meta-learning they approach. In the interest of brevity, we will not attempt such a review here, but rather focus on giving examples of a few forms of meta-learning that fit the GIMLI framework (and thus are supported by the library presented in Section 3), and briefly explain why.

One popular meta-learning problem is that of learning to optimize hyperparameters through gradient-based methods [4, 17, 16, 12], as an alternative to grid/random search [5] or Bayesian Optimization [18, 21, 6, 24]. Here, select continuous-valued hyperparameters are meta-optimized against a meta-objective, subject to the differentiability of the optimization step, and, where relevant, the loss function. This corresponds GIMLI being run with select optimizer hyperparameters as the meta-variables. To give a simple concrete example, in the approaches of [4] and [17], the only meta-variable is the learning rate α.

A related problem is that of learning the optimizer wholesale as a parametric model [15, 1, 10], typically based on recurrent architectures. Again, here the optimizer's own parameters are the optimizer meta-parameters. As a concrete example, in the work of [1], an RNN with parameters ϕ is meta-learned, and models the updates made to parameters during training. In our formalism, this would correspond to setting as meta-variable the parameters of this update network.

More recently, meta-learning approaches such as MAML [11, 2] and its variants/extensions have sought to use higher-order gradients to meta-learn model/policy initializations in few-shot learning settings. In GIMLI, this corresponds to setting the initial model state at the beginning of task-specific loops as the meta-variable.

Finally, recent work by [7] has introduced the ML^3 framework for learning unconstrained loss functions as parametric models, through exploiting second-order gradients of a meta-loss with regard to the parameters of the inner loss. This corresponds, in GIMLI, to learning a parametric model of the loss parameterized by by the meta-variable(s).

3. The `higher` library

In this section, we provide a high-level description of the design and capabilities of `higher`,[1] a PyTorch [20] library aimed at enabling implementations of GIMLI with as little reliance on non-vanilla PyTorch as possible. In this section, we first discuss the obstacles that would prevent us from implementing this in popular deep learning frameworks, how we overcame these in PyTorch to implement GIMLI. Additional features, helper functions, and other considerations when using/extending the library are provided in its documentation.

3.1. Obstacles

Many deep learning frameworks offer the technical functionality required to implement GIMLI, namely the ability to take gradients of gradients. However, there are two aspects of how we implement and train parametric models in such frameworks which inhibit our ability to flexibly implement the GIMLI Algorithm of [14].

The first obstacle is that models are typically implemented statefully (e.g. `torch.nn` in PyTorch, `keras.layers` in Keras [8], etc.), meaning that the model's parameters are encapsulated in the model implementation, and are implicitly relied upon during the forward pass. Therefore while such models can be considered as functions theoretically, they are not pure functions practically, as their output is not uniquely determined by their explicit input, and equivalently the parameters for a particular forward pass typically cannot be trivially overridden or supplied at call time. This prevents us from tracking and backpropagating over the successive values of the model parameters θ within the inner loop, through an implicit or explicit graph.

The second issue is that even though the operations used within popular optimizers are mathematically differentiable functions of the parameters, gradients, and select hyperparameters, these operations are not tracked in various framework's implicit or explicit graph/gradient tape implementations when an optimization step is run. This is with good reason: updating model parameters in-place is memory efficient, as typically there is no need to keep references to the previous version of parameters. Ignoring the gradient dependency formed by allowing backpropagation through an optimization step essentially makes it safe to release memory allocated to historical parameters and intermediate model states once an update has been completed.

3.2. Making stateful modules stateless

As we wish to track and backpropagate through intermediate states of parameters during the inner loop, we keep a

[1] Available at `https://github.com/facebookresearch/higher`.

record of such states which can be referenced during the backward pass stage of the outer loop in the GIMLI Algorithm of [14]. The typical way this is done in implementations of meta-learning algorithms such as MAML is to rewrite a "stateless" version of the inner loop's model, permitting the use, in each invocation of the model's forward pass, of weights which are otherwise tracked on the gradient graph/tape. While this addresses the issue, it is an onerous and limiting solution, as exploring new models within such algorithms invariably requires their reimplementation in a stateless style. This typically prevents the researcher from experimenting with third-party codebases, complicated models, or those which requiring loading pretrained weights, without addressing a significant and unwelcome engineering challenge.

A more generic solution, permitting the use of existing stateful modules (including with pre-loaded activations), agnostic to the complexity or origin of the code which defines them, is to modify the run-time instance of the model's parent class to render them effectively function, a technique often referred to as "monkey-patching". The high-level function `higher.monkeypatch()` does this by taking as argument a `torch.nn.Module` instance and the structure of its nested sub-modules. As it traverses this structure, it clones the parent classes of submodule instances, leaving their functionality intact save for that of the `forward` method which implements the forward pass. Here, it replaces the call to the forward method with one which first replaces the stateful parameters of the submodule with ones provided as additional arguments to the patched `forward`, before calling the original class's bound `forward` method, which will now used the parameters provided at call time. This method is generic and derived from first-principles analysis of the `torch.nn.Module` implementation, ensuring that any first or third-party implementation of parametric models which are subclasses of `torch.nn.Module` and do not abuse the parent class at runtime will be supported by this recursive patching process.

3.3. Making optimizers differentiable

Again, as part of our need to make the optimization process differentiable, the typical solution is to write a version of SGD which does not modify parameters in-place, but treated as a differentiable function of the input parameters and hyperparameters akin to any other module in the training process. While this is often considered a satisfactory solution in the meta-learning literature due to its simplicity, it too is limiting. Not only does the inability to experiment with other inner loop optimizers prevent research into the applicability of meta-learning algorithms to other optimization processes, the restriction to SGD also means that existing state-of-the-art methods used in practical domains cannot be extended using meta-learning methods such as those described in Section 2.1, lest they perform competitively when trained with SGD.

Here, while less generic, the solution provided by the high-level function `higher.get_diff_optim()` is to render a PyTorch optimizer instance differentiable by mapping its parent class to a differentiable reimplementation of the instance's parent class. The reimplementation is typically a copy of the optimizer's step logic, with in-place operations being replaced with gradient-tracking ones (a process which is syntactically simple to execute in PyTorch). To this, we add wrapper code which copies the optimizer's state, and allows safe branching off of it, to permit "unrolling" of the optimization process within an inner loop without modifying the initial state of the optimizer (e.g. to permit several such unrolls, or to preserve state if inner loop optimizer is used elsewhere in the outer loop). Most of the optimizers in `torch.optim` are covered by this method. Here too, a runtime modification of the parent optimizer class could possibly be employed as was done for `torch.nn.Modules`, but this would involve modifying Python objects at a far finer level of granularity. We find that supporting a wide and varied class of optimizers is a sufficient compromise to enable further research.[2]

4. Related Work

The first is work by [13] which describes how several meta-learning and hyperparameter optimization approaches can be cast as a bi-level optimization process, akin to our own formalization in [14]. This fascinating and relevant work is highly complementary to the formalization and discussion presented in our paper. Whereas we focus on the requirements according to which gradient-based solutions to approaches based on nested optimization problems can be found in order to drive the development of a library which permits such approaches to be easily and scalably implemented, their work focuses on analysis of the conditions under which exact gradient-based solutions to bi-level optimization processes can be approximated, and what convergence guarantees exist for such guarantees. In this sense, this is more relevant to those who wish to analyze and extend alternatives to first-order approximations of algorithms such as MAML, e.g. see work of [19] or [22].

On the software front, the library `learn2learn` [3] addresses similar problems to that which we will present in Section 3. This library focuses primarily on providing implementations of existing meta-learning algorithms and their training loops that can be extended with new models. In contrast, the library we present in Section 3 is "closer

[2]We also provide documentation as to how to write differentiable third party optimizers and supply helper functions such as `higher.register_optim()` to register them for use with the library at runtime.

to the metal", aiming to support the development of new meta-learning algorithms fitting the GIMLI definitions with as little resort to non-canonical PyTorch as possible. A recent parallel effort, Torchmeta [9] also provides a library aiming to assist the implementation of meta-learning algorithms, supplying useful data-loaders for meta-training. However, unlike our approach described in 3, it requires reimplementation of models using their functional/stateless building blocks, and for users to reimplement the optimizers in a differentiable manner.

5. Conclusion

To summarize, we have briefly summarized GIMLI, a general formulation of a wide class of existing and potential meta-learning approaches, and described a lightweight library, `higher`, which extends PyTorch to enable the easy and natural implementation of such meta-learning approaches at scale.

References

[1] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems*, pages 3981–3989, 2016.

[2] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your maml. *arXiv preprint arXiv:1810.09502*, 2018.

[3] Séb Arnold, Praateek Mahajan, Debajyoti Datta, and Ian Bunner. learn2learn. `https://github.com/learnables/learn2learn`, 2019.

[4] Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900, 2000.

[5] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[6] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[7] Yevgen Chebotar, Artem Molchanov, Sarah Bechtle, Ludovic Righetti, Franziska Meier, and Gaurav Sukhatme. Meta-learning via learned loss. *arXiv preprint arXiv:1906.05374*, 2019.

[8] François Chollet et al. Keras, 2015.

[9] 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.

[10] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[11] 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.

[12] Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. Forward and reverse gradient-based hyperparameter optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1165–1173. JMLR. org, 2017.

[13] Luca Franceschi, Paolo Frasconi, Saverio Salzo, Riccardo Grazzi, and Massimilano Pontil. Bilevel programming for hyperparameter optimization and meta-learning. *arXiv preprint arXiv:1806.04910*, 2018.

[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] Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

[16] Jelena Luketina, Mathias Berglund, Klaus Greff, and Tapani Raiko. Scalable gradient-based tuning of continuous regularization hyperparameters. In *International conference on machine learning*, pages 2952–2960, 2016.

[17] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[18] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.

[19] Alex Nichol and John Schulman. Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999*, 2, 2018.

[20] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[21] Martin Pelikan, David E Goldberg, and Erick Cantú-Paz. Boa: The bayesian optimization algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*, pages 525–532. Morgan Kaufmann Publishers Inc., 1999.

[22] Aravind Rajeswaran, Chelsea Finn, Sham Kakade, and Sergey Levine. Meta-learning with implicit gradients. *arXiv preprint arXiv:1909.04630*, 2019.

[23] Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

[24] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.