Meta-Learning with Adjoint Methods

Shibo Li[1] **Zheng Wang**[1] **Akil Narayan**[2,3] **Robert M. Kirby**[1,2] **Shandian Zhe**[1]
[1]Kahlert School of Computing, University of Utah
[2]Scientific Computing and Imaging (SCI) Institute, University of Utah
[3]Department of Mathematics, University of Utah
{shibo, wzhut, kirby, zhe}@cs.utah.edu, akil@sci.utah.edu

Abstract

Model Agnostic Meta Learning (MAML) is widely used to find a good initialization for a family of tasks. Despite its success, a critical challenge in MAML is to calculate the gradient w.r.t. the initialization of a long training trajectory for the sampled tasks, because the computation graph can rapidly explode and the computational cost is very expensive. To address this problem, we propose Adjoint MAML (A-MAML). We view gradient descent in the inner optimization as the evolution of an Ordinary Differential Equation (ODE). To efficiently compute the gradient of the validation loss w.r.t. the initialization, we use the adjoint method to construct a companion, backward ODE. To obtain the gradient w.r.t. the initialization, we only need to run the standard ODE solver twice — one is forward in time that evolves a long trajectory of gradient flow for the sampled task; the other is backward and solves the adjoint ODE. We need not create or expand any intermediate computational graphs, adopt aggressive approximations, or impose proximal regularizers in the training loss. Our approach is cheap, accurate, and adaptable to different trajectory lengths. We demonstrate the advantage of our approach in both synthetic and real-world meta-learning tasks. The code is available at https://github.com/shib0li/Adjoint-MAML.

1 INTRODUCTION

Meta-learning (Schmidhuber, 1987; Thrun and Pratt, 2012) seeks to develop methods that can quickly adapt a learning model to new tasks or environments, like human learning. A prominent example is the recent model-agnostic meta-learning (MAML) algorithm (Finn et al., 2017), which is successful in learning model initialization for a family of tasks. MAML is a bi-level optimization approach. The inner level starts from the initialization, and optimizes the training loss of the sampled tasks via gradient descent. At the trained model parameters, the outer-level uses back-propagation to calculate the gradient of the validation loss w.r.t the initialization, and optimizes the initialization accordingly.

While successful, a critical challenge of MAML is to back-propagate the gradient from a long training trajectory of the sampled tasks, because the resulting computation graph grows rapidly, can easily explode, and is computationally expensive. To combat these issues, practical usage of MAML performs only one or a few steps of gradient descent in the inner optimization; unfortunately this propagates a trajectory only close to the initialization, and fails to reflect the longer-term learning performance of using that initialization. To bypass this issue, first-order MAML (FOMAML) (Finn et al., 2017) and Reptile (Nichol et al., 2018) employ dropout on the Jacobian to obtain an aggressive approximation. While this is efficient, the approach loses accurate gradient information. The recent iMAML approach (Rajeswaran et al., 2019) uses an implicit method to calculate an accurate gradient w.r.t the initialization. This approach is elegant and successful, but imposes several restrictions. First, an additional regularizer that encourages proximity of the model parameters and the initialization must be added into the training loss. Second, the gradient is accurate only when training reaches the optimum of the regularized loss.

In this paper, we propose A-MAML, an efficient and accurate approach to differentiate long paths of the inner-optimization in meta-learning. See Fig. 1 for an illustration. Our method does not require additional regularizers and can adapt to different trajectory lengths, hence it is well suited to commonly used training strategies, such as early stopping. Specifically, we view the inner optimization (training) as evolving a forward ordinary differential equation (ODE) system, where the states are the model parameters. The standard gradient descent is equivalent to solving this ODE

Proceedings of the 26th International Conference on Artificial Intelligence and Statistics (AISTATS) 2023, Valencia, Spain. PMLR: Volume 206.

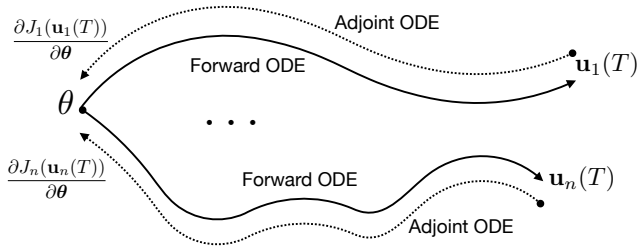


Figure 1: Illustration of A-MAML, where $\boldsymbol{\theta}$ is the initialization, J_n is the validation loss for task n ($n = 1, 2, \ldots$), $\mathbf{u}_n$ are the model parameters for task n, and also the state of the corresponding forward ODE. A-MAML solves the forward ODE to optimize the meta-training loss, and then solves the adjoint ODE backward to obtain the gradient of the meta-validation loss w.r.t $\boldsymbol{\theta}$.

with the forward Euler method. To calculate the gradient of the validation loss w.r.t the model initialization, *i.e.*, the initial state of the ODE, we use the adjoint method to construct a companion ODE. In effect, we only need to run the standard ODE solver twice: First, we solve the forward ODE to evolve a long training trajectory, based on which we compute the initial state of the adjoint ODE. Next, we solve the adjoint ODE backward to obtain the gradient w.r.t the model initialization. To avoid divergence when solving backward, we use high-order solvers in the forward pass and track the states in the trajectory, based on which we use the modified Euler method (second-order) to solve backward. Throughout the procedure, we do not create and grow any intermediate computation graphs, nor do we apply any gradient approximation. The memory cost is linear in the number of model parameters. The accuracy is determined by the numerical precision of the ODE solver, which we can explicitly trade for speed.

For evaluation, we first examined A-MAML in two synthetic benchmark tests, regressing Alpine and Cosine mixture functions. In both task populations, we examined, starting from the given initialization, how the prediction error of the target model varies along with the increase of training epochs. A-MAML leads to much better prediction accuracy and training behaviors compared against MAML, FOMAML, Reptile, and iMAML. Meanwhile, A-MAML dramatically reduces the memory usage and can easily scale to long training trajectories, compared with MAML which utilizes computation graphs. The running time of A-MAML is comparable to FOMAML, Reptile, and iMAML. We then applied A-MAML in three real-world applications of collaborative filtering and two image-classification tasks. In several few-shot learning settings, A-MAML nearly always provides the best initialization, which leads to smaller prediction errors than the competing approaches during the meta-tests. The improvement is often significant.

2 PRELIMINARIES

Suppose we have a family of correlated learning tasks $\mathcal{A}$. The size of $\mathcal{A}$ can be very large or even infinite. For each task, we use the same machine learning model $\mathcal{M}$, which is parameterized by $\mathbf{u} \in \mathbb{R}^d$, *e.g.*, a deep neural network. Our goal is to learn an initialization $\boldsymbol{\theta}$ for $\mathbf{u}$, which can well adapt to all the tasks in $\mathcal{A}$. To this end, we sample N tasks, $\mathcal{S} = \{\mathcal{T}_1, \ldots, \mathcal{T}_N\}$, from a task distribution p on $\mathcal{A}$, and for each $\mathcal{T}_n$, we collect a dataset $\mathcal{D}_n$. We use the N datasets $\widehat{\mathcal{D}} = \{\mathcal{D}_1, \ldots, \mathcal{D}_N\}$ to meta-learn $\boldsymbol{\theta}$. We expect that given any new task $\mathcal{T}^* \in \mathcal{A}$, after initializing $\mathbf{u}$ with $\boldsymbol{\theta}$, the training of $\mathcal{M}$ on $\mathcal{T}^*$ can achieve better performance with the same or fewer training epochs or iterations or examples.

A particularly successful meta-learning algorithm is model-agnostic meta-learning (MAML) (Finn et al., 2017), which uses a bi-level optimization approach to estimate $\boldsymbol{\theta}$. Specifically, each $\mathcal{D}_n$ is partitioned into a meta-training dataset $\mathcal{D}_n^{\text{tr}}$ and a meta-validation dataset $\mathcal{D}_n^{\text{val}}$. In the inner level, we start with $\boldsymbol{\theta}$ and optimize the training loss $\mathcal{L}(\mathbf{u}, \mathcal{D}_n^{\text{tr}})$ for each task n. Let us denote the trained parameters by $\boldsymbol{\psi}_n(\boldsymbol{\theta})$. In the outer level, we evaluate these trained parameters on the validation loss, and optimize $\boldsymbol{\theta}$ accordingly, *i.e.*, $\boldsymbol{\theta}^* = \min \frac{1}{N}\sum_{j=1}^{N} \mathcal{L}(\boldsymbol{\psi}_n(\boldsymbol{\theta}), \mathcal{D}_n^{\text{val}})$. MAML obtains the gradient w.r.t $\boldsymbol{\theta}$ via automatic differentiation, which essentially computes $\frac{\mathrm{d}\boldsymbol{\psi}_n(\boldsymbol{\theta})}{\mathrm{d}\boldsymbol{\theta}}$ via back-propagation on a computation graph. However, this is very challenging for long training trajectories to obtain $\boldsymbol{\psi}_n(\boldsymbol{\theta})$, since the computation graph can rapidly explode and become very expensive to compute. Therefore, in practice, MAML typically only conducts one or a few gradient descent steps in the inner optimization, *e.g.*, with one step, $\boldsymbol{\psi}_n(\boldsymbol{\theta}) = \boldsymbol{\theta} - \alpha\nabla\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}_n^{\text{tr}})$, where α is the step size. However, with only one step the obtained parameters are frequently too close to the initialization, and inadequately reflect the actual longer-range training performance.

To bypass this issue, First-Order MAML (FOMAML) (Finn et al., 2017) drops out the Jacobian $\frac{\mathrm{d}\boldsymbol{\psi}_n(\boldsymbol{\theta})}{\mathrm{d}\boldsymbol{\theta}}$ and replaces it with the identity matrix $\mathbf{I}$. In so doing, FOMAML can perform many gradient descent steps to obtain $\boldsymbol{\psi}_n$ and update $\boldsymbol{\theta}$ with

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot \frac{1}{N}\sum\nolimits_{n=1}^{N} \frac{\partial \mathcal{L}(\boldsymbol{\psi}_n, \mathcal{D}_n^{\text{val}})}{\partial \boldsymbol{\psi}_n},$$

where η is the learning rate. With the same idea, Reptile (Nichol et al., 2018) instead adjusts the updating direction to $\frac{1}{N}\sum_{j=1}^{N} \frac{\partial \mathcal{L}(\boldsymbol{\psi}_n, \mathcal{D}_n^{\text{val}})}{\partial \boldsymbol{\psi}_n} - \boldsymbol{\theta}$. Despite being efficient, these methods lack accurate gradient information about $\boldsymbol{\theta}$. To overcome this limitation, the recent work, iMAML (Rajeswaran et al., 2019), calculates the accurate gradient via an implicit gradient method. However, it needs to incorporate a proximity regularizer into the training loss to bind $\mathbf{u}$ and

$\boldsymbol{\theta}$ explicitly,

$$\widehat{\mathcal{L}}(\mathbf{u}, \mathcal{D}_n^{\text{tr}}) = \mathcal{L}(\mathbf{u}, \mathcal{D}_n^{\text{tr}}) + \frac{\lambda}{2}\|\mathbf{u} - \boldsymbol{\theta}\|^2.$$

The accurate gradient can be obtained (only) when the training reaches the optimum, *i.e.*, $\boldsymbol{\psi}_n = \text{argmin}_{\mathbf{u}} \widehat{\mathcal{L}}(\mathbf{u}, \mathcal{D}_n^{\text{tr}})$, since we can derive the implicit gradient $\frac{\mathrm{d}\boldsymbol{\psi}_n}{\mathrm{d}\boldsymbol{\theta}}$ from the fact that $\frac{\partial \widehat{\mathcal{L}}}{\partial \boldsymbol{\psi}_n} = \mathbf{0}$.

3 ADJOINT MAML

In this paper, we propose A-MAML, which can accurately and efficiently compute the gradient of the meta loss w.r.t the initialization for long training trajectories, without the need for aggressive approximations or additional regularization, and adapts to different trajectory lengths. Hence, our method can be easily integrated with commonly used training strategies, *e.g.*, early stopping.

3.1 ODE View of Inner Optimization

Specifically, we first view the inner optimization as evolving an ODE system. In more detail, given task n, starting from $\boldsymbol{\theta}$, we run gradient descent for a long time to train the model. The training procedure can be in more general viewed as solving the following ODE,

$$\begin{cases} \mathbf{u}_n(0) & = \boldsymbol{\theta}, \\ \frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}t} & = -\frac{\partial \mathcal{L}(\mathbf{u}_n, \mathcal{D}_n^{\text{tr}})}{\partial \mathbf{u}_n}, \end{cases}$$

where the state $\mathbf{u}_n(t)$ represents the model parameters at time t. Running gradient descent with a step size α essentially solves the ODE with the forward Euler method using temporal step size α, corresponding to the update $\mathbf{u}_n(t+\alpha) \leftarrow \mathbf{u}_n(t) - \alpha \frac{\partial \mathcal{L}(\mathbf{u}_n, \mathcal{D}_n^{\text{tr}})}{\partial \mathbf{u}_n}$. However, the ODE view allows us to apply a variety of more efficient, high-order solvers to fulfill the training, *e.g.*, the Runge-Kutta method (Dormand and Prince, 1980). Suppose we stop at time T, then we evaluate the trained parameters $\mathbf{u}_n(T)$ on the validation dataset via $\mathcal{L}(\mathbf{u}_n(T), \mathcal{D}_n^{\text{val}})$. Therefore, the meta loss is given by

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum\nolimits_{n=1}^{N} \mathcal{L}(\mathbf{u}_n(T), \mathcal{D}_n^{\text{val}}). \tag{1}$$

Note that the stopping time T is not necessarily the same for all the tasks; it can vary for different tasks as determined, say, by an early stopping criterion.

3.2 Efficient Back-Propagation via Solving Adjoint ODEs

To optimize $\boldsymbol{\theta}$ in (1) (in the outer loop), we need to be able to compute the gradient of the validation loss for each task n, *i.e.*, $\frac{\mathrm{d}J_n}{\mathrm{d}\boldsymbol{\theta}}$, where $J_n = \mathcal{L}(\mathbf{u}_n(T), \mathcal{D}_n^{\text{val}})$. We seek to compute this gradient efficiently for large T without creating and growing a computation graph. To this end, we use the adjoint method (Pontryagin, 1987). To simplify the notation, we first define

$$J_n(\mathbf{u}_n(T)) = \mathcal{L}(\mathbf{u}_n(T), \mathbf{D}_n^{\text{val}}),$$
$$f(\mathbf{u}_n, \mathcal{D}_n^{\text{tr}}) = -\left(\frac{\partial \mathcal{L}(\mathbf{u}_n, \mathcal{D}_n^{\text{tr}})}{\partial \mathbf{u}_n}\right)^{\top}. \tag{2}$$

Note that we use the row vector representation of the gradient, *i.e.*, $\frac{\partial \mathcal{L}}{\partial \mathbf{u}_n}$ is a $1 \times d$ vector. This is consistent with the shape of Jacobian matrix, and the chain rule can be expressed as the matrix multiplication from left to right, which is natural and convenient. Accordingly, the ODE for $\mathbf{u}_n(t)$ can be written as

$$\begin{cases} \mathbf{u}_n(0) & = \boldsymbol{\theta}, \\ \frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}t} & = \mathbf{f}(\mathbf{u}_n, \mathbf{D}_n^{\text{tr}}). \end{cases} \tag{3}$$

Next, to construct an adjoint ODE for efficient gradient computation, we augment the validation loss,

$$\widehat{J}_n = J_n\left(\mathbf{u}_n(T)\right) + \int_0^T \boldsymbol{\lambda}(t)^{\top}\left(f(\mathbf{u}_n, \mathbf{D}_n^{\text{tr}}) - \frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}t}\right)\mathrm{d}t, \tag{4}$$

where $\boldsymbol{\lambda}(t)$ is a Lagrange multiplier and a $d \times 1$ vector. According to the ODE constraint (3), the extra integral in (4) is 0 and $\widehat{J}_n = J_n$. Hence, we have

$$\frac{\mathrm{d}J_n}{\mathrm{d}\boldsymbol{\theta}} = \frac{\mathrm{d}\widehat{J}_n}{\mathrm{d}\boldsymbol{\theta}} = \frac{\partial J_n}{\partial \mathbf{u}_n(T)}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}(T) + \int_0^T \boldsymbol{\lambda}^{\top}\left[\frac{\partial \mathbf{f}}{\partial \mathbf{u}_n}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}} - \frac{\mathrm{d}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}t}}{\mathrm{d}\boldsymbol{\theta}}\right]\mathrm{d}t. \tag{5}$$

For the second term in the integral, we switch the derivative order and apply integration by parts,

$$\begin{aligned} &\int_0^T \boldsymbol{\lambda}^{\top}\frac{\mathrm{d}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}t}}{\mathrm{d}\boldsymbol{\theta}}\mathrm{d}t = \int_0^T \boldsymbol{\lambda}^{\top}\frac{\mathrm{d}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}}{\mathrm{d}t}\mathrm{d}t \\ &= \boldsymbol{\lambda}^{\top}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}\Big|_0^T - \int_0^T \left(\frac{\mathrm{d}\boldsymbol{\lambda}}{\mathrm{d}t}\right)^{\top}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}\mathrm{d}t \\ &= \boldsymbol{\lambda}(T)^{\top}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}(T) - \boldsymbol{\lambda}(0)^{\top}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}(0) \\ &- \int_0^T \left(\frac{\mathrm{d}\boldsymbol{\lambda}}{\mathrm{d}t}\right)^{\top}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}\mathrm{d}t. \end{aligned}$$

Substituting the above into (5), we obtain

$$\begin{aligned} \frac{\mathrm{d}J_n}{\mathrm{d}\boldsymbol{\theta}} &= \frac{\partial J_n}{\partial \mathbf{u}_n(T)}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}(T) - \boldsymbol{\lambda}(T)^{\top}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}(T) \\ &+ \boldsymbol{\lambda}(0)^{\top}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}(0) \\ &+ \int_0^T \left\{\boldsymbol{\lambda}^{\top}\frac{\partial \mathbf{f}}{\partial \mathbf{u}_n}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}} + \left(\frac{\mathrm{d}\boldsymbol{\lambda}}{\mathrm{d}t}\right)^{\top}\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}\right\}\mathrm{d}t. \end{aligned}$$

The computationally expensive term is the Jacobian $\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}$ (marked as blue), which we efficiently handle by constructing an adjoint ODE for the Lagrange multiplier $\boldsymbol{\lambda}$,

$$\begin{cases} \boldsymbol{\lambda}(T) & = \left(\frac{\partial J_n}{\partial \mathbf{u}_n(T)}\right)^\top, \\ \left(\frac{\mathrm{d}\boldsymbol{\lambda}}{\mathrm{d}t}\right)^\top & = -\boldsymbol{\lambda}(t)^\top \frac{\partial \mathbf{f}}{\partial \mathbf{u}_n}. \end{cases} \tag{6}$$

Note that the ODE (6) runs backward in time starting at the terminal time T. If we can solve (6), the Jacobian terms (blue) will cancel, and the full gradient becomes

$$\frac{\mathrm{d}J_n}{\mathrm{d}\boldsymbol{\theta}} = \boldsymbol{\lambda}(0)^\top \frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}(0) = \boldsymbol{\lambda}(0)^\top, \tag{7}$$

where we have used $\frac{\mathrm{d}\mathbf{u}_n}{\mathrm{d}\boldsymbol{\theta}}(0) = \mathbf{I}$. We see that the gradient is simply the state of $\boldsymbol{\lambda}$ at time 0. To confirm the feasibility of solving (6), we can see from (6) and (2) that $\frac{\partial \mathbf{f}}{\partial \mathbf{u}_n} = \mathbf{H}(\mathbf{u}_n) = -\frac{\partial^2 \mathcal{L}(\mathbf{u}_n, \mathcal{D}_n^{tr})}{\partial \mathbf{u}_n^2}$ is the Hessian matrix of the model parameters. While it seems extremely costly to calculate the Hessian, when we substitute the Hessian into (6) and take the transpose, we find,

$$\begin{cases} \boldsymbol{\lambda}(T) & = \left(\frac{\partial J_n}{\partial \mathbf{u}_n(T)}\right)^\top, \\ \frac{\mathrm{d}\boldsymbol{\lambda}}{\mathrm{d}t} & = -\mathbf{H}(\mathbf{u}_n)\boldsymbol{\lambda}(t). \end{cases} \tag{8}$$

Now it is clear that the dynamics of $\boldsymbol{\lambda}$ is a Hessian-vector product. It is known that we never need to explicitly compute the Hessian matrix. We can first compute the gradient $\mathbf{g} = \frac{\partial \mathcal{L}}{\partial \mathbf{u}_n}$, then the dot product $s = \boldsymbol{\lambda}^\top \mathbf{g}$, and take the gradient of the scalar s again, which gives exactly $\mathbf{H}\boldsymbol{\lambda}$. The complexity is the same as computing the gradient.

Therefore, to calculate $\frac{\mathrm{d}J_n}{\mathrm{d}\boldsymbol{\theta}}$, we only need to run standard ODE solvers twice. First, we run a solver to evolve (3) from time 0 to time T. Note that even a small T can correspond to many gradient descent steps. For example, $T = 10$ corresponds to running 1000 gradient descent steps where the step size is set to 0.01 (a common choice). We can apply high-order methods, like RK45 (Dormand and Prince, 1980) to further improve the speed and accuracy. Next, at the trained parameters $\mathbf{u}(T)$, we jointly solve (8) and (3) backward (note that dynamics of $\boldsymbol{\lambda}$ needs $\mathbf{u}_n$). For solving both ODEs, we never need to create and/or grow new computation graphs. All we need is to compute the dynamics in (3) and (8), and the computational complexity is the same as computing the gradient of the training loss w.r.t the model parameters. The memory cost only involves storage of $\mathbf{u}_n$ and $\boldsymbol{\lambda}$, which is proportional to the number of model parameters. We never need to maintain or calculate any Jacobian matrix. The accuracy is determined by the numerical precision of the ODE solvers, which have been developed for decades, are mature, and can easily effect tradeoffs between precision and speed. Note that our method does not need to add extra regularization into the training loss, although our framework can be easily adjusted to support such regularization.

Empirically, we found that back-solving can be numerically unstable or even diverge when T is relatively large, say, 20 or 100. This is consistent with the observation in the training of neural ODE models (Chen et al., 2018; Gholami et al., 2019; Daulbaev et al., 2020), which also uses the adjoint method. To promote robustness, we track the state $\mathbf{u}_n$ in the training trajectory with a given step size during the forward solve. This can be automatically done via the ODE solver. Then based on the list of states $\{\mathbf{u}_{n,j}\}_j$, we solve the adjoint ODE backward with the modified Euler method (Ascher and Petzold, 1998) whose global accuracy is $\mathcal{O}(h^2)$ where h is the ODE solver step size. Specifically, at each step j, we first calculate an intermediate value $\widetilde{\boldsymbol{\lambda}}_j$ and then the state $\boldsymbol{\lambda}_j$ via,

$$\begin{aligned} \widetilde{\boldsymbol{\lambda}}_j &= \boldsymbol{\lambda}_{j+1} + h\mathbf{H}(\mathbf{u}_{n,j+1})\boldsymbol{\lambda}_{j+1}, \\ \boldsymbol{\lambda}_j &= \boldsymbol{\lambda}_{j+1} + \frac{h}{2}\left[\mathbf{H}(\mathbf{u}_{n,j+1})\boldsymbol{\lambda}_{j+1} + \mathbf{H}(\mathbf{u}_{n,j})\widetilde{\boldsymbol{\lambda}}_j\right]. \end{aligned}$$

While this increases memory requirements, it is still linear with the number of parameters, $\mathcal{O}(\frac{T}{h}d)$, and much cheaper than building a computational graph. While even more memory-efficient approaches are available (Gholami et al., 2019; Daulbaev et al., 2020), our method is simple and convenient to implement. The experiments show that our method has already been able to scale to long training trajectories very economically (see Sec. 5.2). In the Appendix, we examined the trade-off between the number of stored intermediate states and the gradient accuracy (see Sec. A). Our method is summarized in Algorithm 1.

Algorithm 1 A-MAML ($p(\mathcal{T})$, T, η, G, ξ)

1: Randomly initialize $\boldsymbol{\theta}$.
2: **repeat**
3: Sample a mini-batch of tasks $\{\mathcal{T}_n\}_{n=1}^B$ from $p(\mathcal{T})$.
4: **for** each task $\mathcal{T}_n$ **do**
5: Calculate $\frac{\partial J_n}{\partial \boldsymbol{\theta}}$ with Algorithm 2.
6: **end for**
7: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot \frac{1}{B}\sum_{n=1}^B \frac{\partial J_n}{\partial \boldsymbol{\theta}}$ (or use ADAM).
8: **until** G iterations are done or the change of $\boldsymbol{\theta}$ is less than ξ
9: Return $\boldsymbol{\theta}$.

Algorithm 2 Adjoint Gradient Computation ($\boldsymbol{\theta}$, J_n, T, h)

1: $\mathbf{u}_n(0) \leftarrow \boldsymbol{\theta}$.
2: Solve forward ODE (3) to time T with RK45, and track the states $\{\mathbf{u}_{n,j}\}_j$ in the trajectory with step size h.
3: $\boldsymbol{\lambda}(T) \leftarrow \frac{\partial J_n}{\partial \mathbf{u}_n(T)}$.
4: Solve the adjoint ODE (8) to time 0 with modified Euler method based on the state list $\{\mathbf{u}_{n,j}\}$.
5: Return $\boldsymbol{\lambda}(0)$.

4 RELATED WORK

Meta-learning (Schmidhuber, 1987; Thrun and Pratt, 2012; Naik and Mammone, 1992) can be (roughly) classified into three categories: (1) metric-learning methods that learn a

metric space (in the outer lever), where the tasks (in the inner level) make predictions by simply matching the training points, *e.g.*, nonparametric nearest neighbors (Koch et al., 2015; Vinyals et al., 2016; Snell et al., 2017; Oreshkin et al., 2018; Allen et al., 2019), (2) black-box methods that train feed-forward or recurrent NNs to take the hyperparameters and task dataset as the input and outright predict the optimal model parameters or parameter updating rules (Hochreiter et al., 2001; Andrychowicz et al., 2016; Li and Malik, 2016; Ravi and Larochelle, 2017; Santoro et al., 2016; Duan et al., 2016; Wang et al., 2016; Munkhdalai and Yu, 2017; Mishra et al., 2017), and (3) optimization-based methods that conduct a bi-level optimization, where the inner level is to estimate the model parameters given the hyperparameters (in each task) and the outer level is to optimize the hyperparameters via a meta-loss (Finn et al., 2017; Finn, 2018; Bertinetto et al., 2018; Lee et al., 2019; Zintgraf et al., 2019; Li et al., 2017; Finn et al., 2018; Zhou et al., 2018; Harrison et al., 2018). There are also hybrid approaches, *e.g.*, (Rusu et al., 2018; Triantafillou et al., 2019).

MAML (Finn et al., 2017) is a popular optimization-based meta-learning method. In addition to FOMAML and Reptile, there are many variants, *e.g.*, (Grant et al., 2018; Finn et al., 2018; Song et al., 2020; Liu et al., 2019). Recently, Denevi et al. (2020); Wang et al. (2020); Denevi et al. (2021) proposed conditional meta learning to leverage the side information (when available) to learn task-specific initializations. The recent work of (Im et al., 2019; Xu et al., 2021) also introduces an ODE view for MAML. However, they use the ODE theory and methods to analyze/improve the outer level optimization, where the inner level still performs one step gradient descent as in standard MAML. They do not consider long training trajectories in the inner level. Im et al. (2019) pointed out the MAML update is a special case of (second-order) Runge-Kutta gradients, and suggested using more refined nodes, weights and even higher-order updates. Xu et al. (2021) showed that if the outer-level optimization of MAML is considered as solving an ODE, it enjoys a linear convergence rate for strongly convex task losses. Based on their analysis, they proposed a bi-phase algorithm to further reduce the cost and improve efficiency. Our work uses the ODE view for inner-level optimization. The adjoint method is a classical and popular framework to estimate the parameters of ODE or dynamic control models (Chen et al., 2018; Eichmeir et al., 2021). If we use Euler method to solve the adjoint ODE, it reduces to the reverse mode differentiation method (Bengio, 2000; Baydin and Pearlmutter, 2014), yet leaving first-order global accuracy ($\mathcal{O}(h)$). Another related work is (Domke, 2012) that provides a general bi-level optimization framework. It can optimize explicit hyper-parameters in the inner-optimization loss, *e.g.*, regularization strength. However, this framework cannot optimize the parameter initialization, since the initialization does not explicitly appear in the loss.

The most recent work (Deleu et al., 2022) (in parallel to ours) also uses an ODE view for the inner-optimization, but it applies the forward sensitivity framework, which constructs an ODE for the state Jacobian, and jointly solves the state and its Jacobian forward. Our work constructs an adjoint ODE for the Lagrange multiplier and solves that ODE backward. In general, the forward sensitivity method is expensive for high-dimensional states, which takes $O(d^2p)$ time and $O(d^2 + dp)$ space complexity to evolve the Jacobian, where d is the state dimension and p is the dimension of the gradient of each state element. Our adjoint method only takes $O(dp)$ complexity, and hence is much more scalable and memory efficient. Deleu et al. (2022) restricted the task to only meta learning the initialization of *linear models*. In other words, the adaptation only applies to the neural network (NN) weights of the last layer. Thereby, it can use the loss structure to greatly simplify the computation and save memory. The method, however, cannot apply to more flexible tasks, *e.g.*, to meta learn the initialization of the weights in the second last layer (or several layers) or the full parameter initialization for any nonlinear model. Our approach is more general in that it does not restrict the model type or the set of parameters. It can meta learn the initialization of all (or any subset of) the parameters in any NN or other linear/nonlinear models. By tracking the intermediate states, our work has also addressed the numerical stability issue when solving the adjoint ODE backward, which is one major concern that motivates (Deleu et al., 2022).

5 EXPERIMENTS

5.1 2D regression

For evaluation, we first examined the proposed approach in two synthetic benchmark tests, namely, meta learning of *CosMixture* and *Alpine* functions (`http://infinity77.net/global_optimization/test_functions.html`), both of which are 2D regression tasks. We considered two families of tasks. In the first family, each task aims to learn a specific *CosMixture* function of the following form,

$$f_1(\mathbf{x}) = -0.1 \sum_{i=1}^{d} A\cos(\omega x_i + \phi) - \sum_{i=1}^{d} x_i^2, \quad (9)$$

where $\mathbf{x} \in [-1, 1]^2$, $d = 2$, $A \in [0.1, 1.0]$, $\omega \in [0.5\pi, 2.0\pi]$, and $\phi \in [3.0, 6.0]$. The second family of tasks learn instances of the *Alpine* function,

$$f_2(\mathbf{x}) = \sum_{i=1}^{d} |x_i \sin(x_i + \phi_i) + 0.1x_i|, \quad (10)$$

where $\mathbf{x} \in [10, 10]^2$, $d = 2$, $\phi_1 \in [-\frac{5}{12}\pi, \frac{5}{12}\pi]$, and $\phi_2 \in [-\frac{5}{12}\pi, \frac{5}{12}\pi]$. An instance of each function is shown in Fig. 2a and 2d. The learning model for both task populations is a neural network with two hidden layers, each consisting of 32 neurons with Tanh activation. To conduct

meta-learning for each task population, we randomly sampled 100 tasks, where for each task, the parameters of the target function, *i.e.*, $\{A, \omega, \phi\}$ in *CosMixture* and $\{\phi_1, \phi_2\}$ in *Alpine*, are uniformly sampled from their ranges. We considered two meta-learning settings: 50shot-50val, where we used 50 examples for meta-training and 50 another examples in meta-validation, and 100shot-100val, where both the meta-training and meta-validation losses employed 100 examples. These examples are non-overlapping and generated by uniformly sampling from the input domain. Given the learned initialization, we tested on 100 new tasks, where the task training data were generated in the same way as in the meta-training and 100 another examples were sampled to evaluate the prediction accuracy.

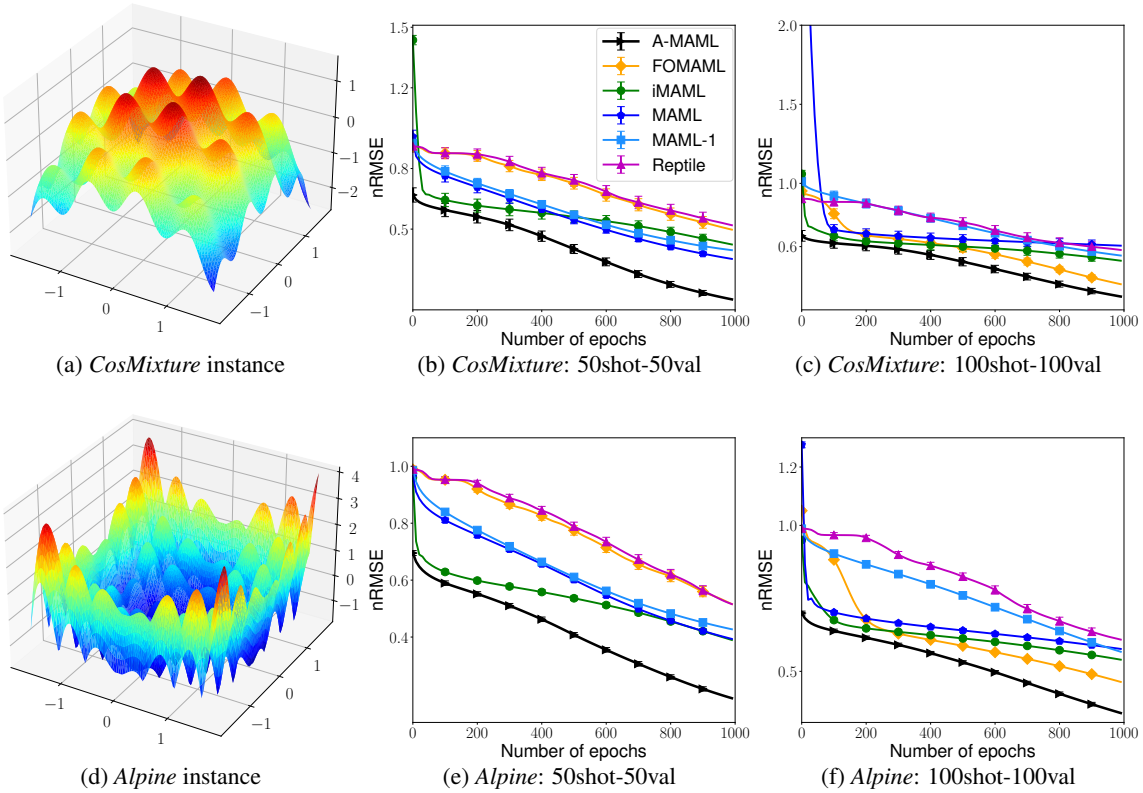
Competing Methods. To examine the effectiveness of our method A-MAML, we tested the following MAML based approaches for an *apples-to-apples* comparison: (1) the original MAML (Finn et al., 2017), (2) First-order MAML (FOMAML) (Finn et al., 2017), which ignores the Jacobian in the gradient computation and uses the gradient w.r.t the trained parameters to update the initialization, (3) Reptile (Nichol et al., 2018), which subtracts the gradient w.r.t the trained parameter by the current initialization as the updating direction, (4) Implicit MAML (iMAML) (Rajeswaran et al., 2019), which introduces a proximal regularizer in the meta-training loss, and uses conjugate gradient to compute the gradient w.r.t the initialization.

All the methods were implemented with PyTorch (Paszke et al., 2019). For MAML, we used a high-quality open source implementation (`https://github.com/dragen1860/MAML-Pytorch`); for iMAML, we used the implementation of the original authors (`https://github.com/aravindr93/imaml_dev`). For our approach A-MAML, we used the Torchdiffeq library (`https://github.com/rtqichen/torchdiffeq`) to accomplish ODE solving with RK45. In the inner optimization, all the competing methods used the standard gradient descent (GD) with step size $\alpha = 0.01$. For iMAML, the strength of the proximal regularizer was chosen as $\lambda = 1$ and 5 CG steps were conducted for Newton-CG optimization. For our method, we used the same step size (*i.e.*, 0.01) to run modified Euler's method (Ascher and Petzold, 1998) for solving the adjoint ODE. In the outer optimization, all the methods used the ADAM algorithm (Kingma and Ba, 2014), and the learning rate was set to 10^{-3}. Each time, a mini-batch of five tasks were sampled to conduct inner-optimization, and then update the initialization in the outer-level. We ran $5,000$ meta-epochs for each method. For the 50shot-50val setting, we ran 200 GD steps for FOMAML, Reptile, iMAML, and for our method A-MAML, set $T = 2$ (that corresponds to 200 GD steps with $\alpha = 0.01$). For the 100shot-100val setting, we ran 500 GD steps for FOMAML, Reptile, iMAML, and set $T = 5$ for A-MAML accordingly. By contrast, MAML ran 20 and 50 GD steps, respectively. Note that MAML cannot run too many GD steps without exhausting computational memory (see Section 5.2). We also evaluated MAML with only one GD step (the most common choice) for both settings; we denote such results by MAML-1. At the adaptation stage (meta-test), we ran the same number of GD steps with the initialization learned by every method: 200 steps for 50shot-50val and 500 steps for 100shot-100val, with the same step size as in the meta training. We executed all the algorithms on a Linux workstation with an NVIDIA GeForce RTX 3090 GPU card that includes 24 GB of G6X memory.

In Fig. 2b,c, e and f, we show that starting with the learned initialization of each method, how the prediction error of the NN model on the test tasks varies along with the increase of training epochs. The prediction error for each task is computed as the normalized root-mean-square error (nRMSE). We averaged the nRMSE over the 100 test tasks and report the standard deviation. As we can see, in all the cases, our approach, A-MAML, always finds the initialization that leads to the best learning progress and performance — the NN models exhibit smaller prediction error throughout the training, as compared with using the initialization from the competing methods. MAML-1 is in general worse than MAML; the discrepancy is particularly evident for learning *Alpine* functions with the 100shot-100val setting (see Fig. 2f). It implies that only performing one-step GD in the inner-optimization might not properly reflect the quality of the initialization in training. Although FOMAML and Reptile can run many GD steps, their performance is often worse than MAML, especially Reptile, which is nearly always inferior to MAML. Such relatively poor performance might be attributed to the use of incorrect gradient information to update the initialization in these approaches. iMAML performed the second best at the beginning, but it was often surpassed by MAML or FOMAML after considerable training epochs. This might be due to (1) the proximity regularizer in the meta-training was not used in the actual training, which introduces some inconsistency, and (2) the inner optimization (though with 200/500 GD steps) has yet to achieve the optimum, and so the obtained gradient w.r.t the initialization is still inaccurate. Note that the nRMSE for 100shot-100val seems a bit higher than 50shot-50val at the early stage, which might because the former involves a double quantity of examples, hence needs more epochs to train better and exhibits slower learning progress. Together these results have demonstrated the advantage of our method in being able to accurately compute the gradient for long inner-optimization trajectories.

5.2 Memory Consumption and Running Time

Next we examined the efficiency of our method in terms of memory usage and computational speed. To this end, we



(a) *CosMixture* instance (b) *CosMixture*: 50shot-50val (c) *CosMixture*: 100shot-100val

(d) *Alpine* instance (e) *Alpine*: 50shot-50val (f) *Alpine*: 100shot-100val

Figure 2: Prediction error of the neural network in learning *CosMixture* and *Alpine* function families, starting from the initialization provided by different meta-learning approaches. (a,d) are the instances of the two types of functions. 50shot-50val means 50 examples were used for meta-training and another 50 examples for meta-validation. 100shot-100val means both the meta-training and meta-validation used 100 examples. The results were averaged over 100 test tasks.

tested the 100shot-100validation setting in the meta learning of *CosMinxture* functions. We varied the number of inner GD steps (with the step size $\alpha = 0.01$) for the competing approaches and the corresponding time ranges $[0, T]$ for ODEs in A-MAML. The average memory usage and running time are reported in Fig. 3 and 4, respectively.

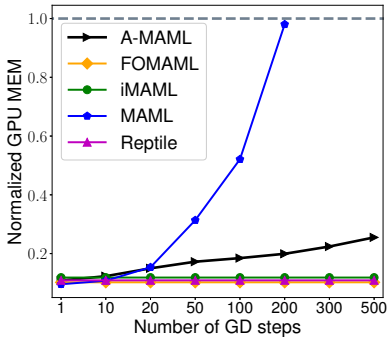


Figure 3: Normalized GPU usage in meta learning of *CosMixutre* with 100shot-100validation. The dashed line indicates the capacity of available GPU memory.

As shown in Fig. 3, MAML always occupies the most memory. With the increase of GD steps, its memory consumption grows exponentially. When MAML runs 200 inner GD steps, the memory is completely exhausted. The result shows the creation and expansion of the computation graphs is very costly. By contrast, A-MAML can accurately compute the gradient in a much more economical way. A-MAML needs to track the states in the training trajectory to robustly solve the adjoint ODE so the memory usage also grows with the number of GD steps, but this growth is much slower (linear) and more affordable than MAML. A-MAML effortlessly supports 500 steps with less than 25% memory usage.

Fig. 4 shows that the running time of A-MAML (per update in the outer-optimization) is comparable to iMAML, FOMAML and Reptile, and much smaller than MAML. This shows that our method is computationally efficient. On the other hand, the running time of MAML indicates that growing the computation graph for more GD steps also incurs a dramatic increase in the computational cost.

	Jester-1		MovieLens100K		MovieLens1M	
	10shot-15val	20shot-30val	10shot-15val	20shot-30val	10shot-15val	20shot-30val
A-MAML	**0.074±0.005**	**0.027±0.002**	**0.053±0.005**	**0.023±0.003**	**0.094±0.008**	**0.035±0.004**
iMAML	0.114±0.007	0.050±0.003	0.082±0.004	0.033±0.002	0.138±0.010	0.052±0.004
MAML	0.120±0.001	0.036±0.000	0.123±0.001	0.050±0.003	0.140±0.002	0.059±0.001
FOMAML	0.292±0.012	0.115±0.004	0.174±0.008	0.068±0.004	0.270±0.011	0.104±0.006
Reptile	0.270±0.012	0.106±0.004	0.166±0.008	0.063±0.003	0.266±0.011	0.101±0.006

Table 1: Meta-test error (nRMSE) with 50 inner GD steps (MAML used 5 GD steps). The results were averaged over 100 tasks.

	Jester-1		MovieLens100K		MovieLens1M	
	10shot-15val	20shot-30val	10shot-15val	20shot-30val	10shot-15val	20shot-30val
A-MAML	**0.069±0.005**	**0.044±0.003**	**0.057±0.006**	**0.021±0.002**	**0.105±0.009**	**0.035±0.004**
iMAML	0.190±0.010	0.103±0.005	0.168±0.007	0.046±0.002	0.130±0.007	0.045±0.004
MAML	0.154±0.001	0.061±0.002	0.123±0.001	0.050±0.002	0.197±0.002	0.083±0.001
FOMAML	0.273±0.012	0.077±0.004	0.191±0.007	0.071±0.004	0.395±0.010	0.119±0.005
Reptile	0.290±0.012	0.100±0.004	0.171±0.008	0.066±0.004	0.408±0.011	0.128±0.006

Table 2: Meta-test error (nRMSE) with 100 inner GD steps (MAML used 10 GD steps). The results were averaged over 100 tasks.

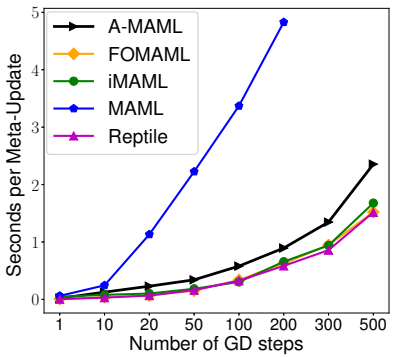


Figure 4: Running time of the inner gradient descent for *CosMixutre*.

5.3 Few-Shot Learning in Collaborative Filtering

Third, we examined our approach in three real-world applications of collaborative filtering. To this end, we used the following datasets. (1) *Jester-1*(`https://goldberg.berkeley.edu/jester-data/`) (Goldberg et al., 2001), which are about joke ratings. There are 100 jokes, rated by $24,983$ users. Each user has rated at least 36 jokes. The ratings are between -10 and 10. (2) *MovieLens-100K* and (3) *MovieLens-1M* (`https://grouplens.org/datasets/movielens/`), movie rating datasets, where the former includes 10K ratings from 1K users on 1.7K movies, and the latter one million movie ratings from 6K users on 4K movies. The ratings are ranged from 0 to 5. Following (Denevi et al., 2020, 2021), we considered predicting the ratings of a given user (on different jokes or movies) as one task.

Different users correspond to different tasks. For each user, we learned a neural network (NN) to predict the rating on a specific joke or movie. The input to the NN is the one-hot encoding of the joke or movie. The NN has two hidden layers, and each layer includes 40 neurons with Tanh activation. We conducted meta learning on each dataset to estimate a good initialization for the corresponding rate prediction model. To prevent scarcity of the task data points, we selected the most frequently rated 100 movies in *MovieLens-100K* and *MovieLens-1M*, and only considered users who had rated at least 20 of them. This gives 489 and 4,985 tasks on *MovieLens-100K* and *MovieLens-1M*, respectively. For *Jester-1*, we used all $24,983$ tasks. For each dataset, we sampled 100 tasks for testing and used the remaining tasks for meta learning. We examined two few-shot settings: 15shot-20val, where 15 examples were used in meta-training and 20 examples in meta-validation, and 20shot-30val where 20 examples were used in meta-training and 30 example in meta-validation. During the meta learning, when the data points of a sampled task are less than the required meta training and validation set size, we re-sample a new task. At the test stage, the training for each task used the same number of examples for few-shot learning and the remaining were used for evaluation.

Method	Ominiglot	Mini-ImageNet
MAML	$95.8 \pm 0.3\%$	$48.70 \pm 1.84\%$
FOMAML	$89.4 \pm 0.5\%$	$48.07 \pm 1.75\%$
Reptile	$89.43 \pm 0.14\%$	$\mathbf{49.97 \pm 0.32\%}$
iMAML-GD	$94.46 \pm 0.42\%$	$48.96 \pm 1.84\%$
iMAML-HF	$96.18 \pm 0.36\%$	$49.30 \pm 1.88\%$
A-MAML($T = 0.5$)	$96.36 \pm 0.39\%$	$49.43 \pm 1.64\%$
A-MAML($T = 1.0$)	$\mathbf{96.79 \pm 0.34\%}$	$49.47 \pm 1.77\%$

Table 3: Meta-test accuracy for 20-way 1-shot on *Omniglot* and 5-way 1-shot on *Mini-ImageNet*.

For all the methods, the step size of the inner training was set to $\alpha = 0.01$, and a mini-batch of 5 tasks were sampled each time to conduct the inner training. We tested two choices of GD steps. First, we performed 50 GD steps for iMAML,

FOMAML and Reptile, and set $T = 0.5$ for A-MAML to solve the forward and adjoint ODEs (corresponding to 50 steps). Second, we performed 100 GD steps for iMAML, FOMAML and Reptile, and accordingly set $T = 1.0$ for A-MAML. In each case, we ran MAML with one tenth of the corresponding steps, *i.e.*, 5 and 10 steps respectively. In the outer-level, all the methods used ADAM optimization with learning rate 10^{-3}. We ran 5000 meta epochs for each method. We computed the average nRMSE and its standard deviation of using the initialization estimated by each method for training and then testing on new tasks.

As shown in Tables 1 and 2, A-MAML achieved the best performance in all the cases — the learned initializations always result in the smallest test error after training ($p < 0.05$), as compared with the competing methods. Consistent with the results in synthetic data (Sec. 5.1), FOMAML and Reptile are still worse than MAML, implying that their updates with inaccurate gradient information do not help improve the performance in these collaborative filtering applications. The results further confirm the advantage of the proposed method A-MAML.

5.4 Few-Shot Learning in Images Classification

Finally, we evaluated A-MAML on popular benchmark datasets in few-shot image classification tasks, *Mini-ImageNet* and *Omniglot*. We followed the standard training and evaluation protocol as in iMAML paper and the prior works (Santoro et al., 2016; Vinyals et al., 2016; Finn et al., 2017), including data splits, NN architecture, *etc*. We tested 5-way 1-shot learning on *Mini-ImageNet* and 20-way 1-shot in *Omniglot*, *because these two settings are more challenging to all the methods.* We ran A-MAML with three settings, $T = 0.5$ and $T = 1.0$. During the adaptation stage, we ran the same number of GD steps with iMAML. The results are reported in Table 3. As we can see, with longer trajectory length, *i.e.*, $T = 1.0$, our method gave the best performance on *Omiglot* and the second best on *Mini-ImageNet*. With shorter length ($T = 0.5$), the performance decreases, but is comparable to or better than the competing methods. This is reasonable and again shows the advantage of being able to carry out longer trajectories during the meta training.

6 CONCLUSION

We have presented A-MAML, a novel meta learning approach of model initializations. We view the inner-optimization as solving a forward ODE, and use the adjoint method to compute the gradient of the meta-loss w.r.t the initialization in an efficient and accurate way. We plan to extend our work to conditional meta learning (Denevi et al., 2021; Wang et al., 2020) so as to further leverage side information to estimate task-specific initializations.

Acknowledgments

This work has been supported by MURI AFOSR grant FA9550-20-1-0358, NSF CAREER Award IIS-2046295 and NSF DMS-1848508.

References

Allen, K., Shelhamer, E., Shin, H., and Tenenbaum, J. (2019). Infinite mixture prototypes for few-shot learning. In International Conference on Machine Learning, pages 232–241. PMLR.

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and De Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. arXiv preprint arXiv:1606.04474.

Ascher, U. M. and Petzold, L. R. (1998). Computer methods for ordinary differential equations and differential-algebraic equations, volume 61. Siam.

Baydin, A. G. and Pearlmutter, B. A. (2014). Automatic differentiation of algorithms for machine learning. arXiv preprint arXiv:1404.7456.

Bengio, Y. (2000). Gradient-based optimization of hyperparameters. Neural computation, 12(8):1889–1900.

Bertinetto, L., Henriques, J. F., Torr, P. H., and Vedaldi, A. (2018). Meta-learning with differentiable closed-form solvers. arXiv preprint arXiv:1805.08136.

Chen, R. T., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. (2018). Neural ordinary differential equations. Advances in neural information processing systems, 31.

Daulbaev, T., Katrutsa, A., Markeeva, L., Gusak, J., Cichocki, A., and Oseledets, I. (2020). Interpolation technique to speed up gradients propagation in neural odes. Advances in Neural Information Processing Systems, 33:16689–16700.

Deleu, T., Kanaa, D., Feng, L., Kerg, G., Bengio, Y., Lajoie, G., and Bacon, P.-L. (2022). Continuous-time meta-learning with forward mode differentiation. In International Conference on Learning Representations.

Denevi, G., Pontil, M., and Ciliberto, C. (2020). The advantage of conditional meta-learning for biased regularization and fine tuning. Advances in Neural Information Processing Systems, 33.

Denevi, G., Pontil, M., and Ciliberto, C. (2021). Conditional meta-learning of linear representations. arXiv preprint arXiv:2103.16277.

Domke, J. (2012). Generic methods for optimization-based modeling. In Artificial Intelligence and Statistics, pages 318–326. PMLR.

Dormand, J. R. and Prince, P. J. (1980). A family of embedded runge-kutta formulae. Journal of computational and applied mathematics, 6(1):19–26.

Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. (2016). Rl2: Fast reinforcement learning via slow reinforcement learning. arXiv preprint arXiv:1611.02779.

Eichmeir, P., Lauß, T., Oberpeilsteiner, S., Nachbagauer, K., and Steiner, W. (2021). The adjoint method for time-optimal control problems. Journal of Computational and Nonlinear Dynamics, 16(2).

Finn, C. (2018). Learning to learn with gradients. PhD thesis, UC Berkeley.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In International Conference on Machine Learning, pages 1126–1135. PMLR.

Finn, C., Xu, K., and Levine, S. (2018). Probabilistic model-agnostic meta-learning. arXiv preprint arXiv:1806.02817.

Gholami, A., Keutzer, K., and Biros, G. (2019). Anode: unconditionally accurate memory-efficient gradients for neural odes. In Proceedings of the 28th International Joint Conference on Artificial Intelligence, pages 730–736.

Goldberg, K., Roeder, T., Gupta, D., and Perkins, C. (2001). Eigentaste: A constant time collaborative filtering algorithm. information retrieval, 4(2):133–151.

Grant, E., Finn, C., Levine, S., Darrell, T., and Griffiths, T. (2018). Recasting gradient-based meta-learning as hierarchical Bayes. In 6th International Conference on Learning Representations, ICLR 2018.

Harrison, J., Sharma, A., and Pavone, M. (2018). Meta-learning priors for efficient online bayesian regression. In International Workshop on the Algorithmic Foundations of Robotics, pages 318–337. Springer.

Hochreiter, S., Younger, A. S., and Conwell, P. R. (2001). Learning to learn using gradient descent. In International Conference on Artificial Neural Networks, pages 87–94. Springer.

Hospedales, T., Antoniou, A., Micaelli, P., and Storkey, A. (2020). Meta-learning in neural networks: A survey. arXiv preprint arXiv:2004.05439.

Im, D. J., Jiang, Y., and Verma, N. (2019). Model-agnostic meta-learning using runge-kutta methods. arXiv preprint arXiv:1910.07368.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Koch, G., Zemel, R., and Salakhutdinov, R. (2015). Siamese neural networks for one-shot image recognition. In ICML deep learning workshop, volume 2. Lille.

Lake, B., Salakhutdinov, R., Gross, J., and Tenenbaum, J. (2011). One shot learning of simple visual concepts. In Proceedings of the annual meeting of the cognitive science society, volume 33.

Lee, K., Maji, S., Ravichandran, A., and Soatto, S. (2019). Meta-learning with differentiable convex optimization. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 10657–10665.

Li, K. and Malik, J. (2016). Learning to optimize. arXiv preprint arXiv:1606.01885.

Li, Z., Zhou, F., Chen, F., and Li, H. (2017). Meta-sgd: Learning to learn quickly for few-shot learning. arXiv preprint arXiv:1707.09835.

Liu, H., Socher, R., and Xiong, C. (2019). Taming maml: Efficient unbiased meta-reinforcement learning. In International Conference on Machine Learning, pages 4061–4071. PMLR.

Mishra, N., Rohaninejad, M., Chen, X., and Abbeel, P. (2017). A simple neural attentive meta-learner. arXiv preprint arXiv:1707.03141.

Munkhdalai, T. and Yu, H. (2017). Meta networks. In International Conference on Machine Learning, pages 2554–2563. PMLR.

Naik, D. K. and Mammone, R. J. (1992). Meta-neural networks that learn by learning. In [Proceedings 1992] IJCNN International Joint Conference on Neural Networks, volume 1, pages 437–442. IEEE.

Nichol, A., Achiam, J., and Schulman, J. (2018). On first-order meta-learning algorithms. arXiv preprint arXiv:1803.02999.

Oreshkin, B. N., Rodriguez, P., and Lacoste, A. (2018). Tadam: Task dependent adaptive metric for improved few-shot learning. arXiv preprint arXiv:1805.10123.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. arXiv preprint arXiv:1912.01703.

Pontryagin, L. S. (1987). Mathematical theory of optimal processes. CRC press.

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

Ravi, S. and Larochelle, H. (2017). Optimization as a model for few-shot learning. In In International Conference on Learning Representations (ICLR).

Rusu, A. A., Rao, D., Sygnowski, J., Vinyals, O., Pascanu, R., Osindero, S., and Hadsell, R. (2018). Meta-learning with latent embedding optimization. arXiv preprint arXiv:1807.05960.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. (2016). Meta-learning with memory-augmented neural networks. In International conference on machine learning, pages 1842–1850. PMLR.

Schmidhuber, J. (1987). Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook. PhD thesis, Technische Universität München.

Snell, J., Swersky, K., and Zemel, R. (2017). Prototypical networks for few-shot learning. Advances in neural information processing systems, 30.

Song, X., Gao, W., Yang, Y., Choromanski, K., Pacchiano, A., and Tang, Y. (2020). Es-maml: Simple hessian-free meta learning. In ICLR.

Sung, F., Yang, Y., Zhang, L., Xiang, T., Torr, P. H., and Hospedales, T. M. (2018). Learning to compare: Relation network for few-shot learning. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1199–1208.

Thrun, S. and Pratt, L. (2012). Learning to learn. Springer Science & Business Media.

Triantafillou, E., Zhu, T., Dumoulin, V., Lamblin, P., Evci, U., Xu, K., Goroshin, R., Gelada, C., Swersky, K., Manzagol, P.-A., et al. (2019). Meta-dataset: A dataset of datasets for learning to learn from few examples. arXiv preprint arXiv:1903.03096.

Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., and Wierstra, D. (2016). Matching networks for one shot learning. arXiv preprint arXiv:1606.04080.

Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. (2016). Learning to reinforcement learn. arXiv preprint arXiv:1611.05763.

Wang, R., Demiris, Y., and Ciliberto, C. (2020). Structured prediction for conditional meta-learning. Advances in Neural Information Processing Systems, 33.

Xu, R., Chen, L., and Karbasi, A. (2021). Meta learning in the continuous time limit. In International Conference on Artificial Intelligence and Statistics, pages 3052–3060. PMLR.

Yoon, J., Kim, T., Dia, O., Kim, S., Bengio, Y., and Ahn, S. (2018). Bayesian model-agnostic meta-learning. In Proceedings of the 32nd International Conference on Neural Information Processing Systems, pages 7343–7353.

Zhou, F., Wu, B., and Li, Z. (2018). Deep meta-learning: Learning to learn in the concept space. arXiv preprint arXiv:1802.03596.

Zintgraf, L., Shiarli, K., Kurin, V., Hofmann, K., and Whiteson, S. (2019). Fast context adaptation via meta-learning. In International Conference on Machine Learning, pages 7693–7702. PMLR.

A Trade-off Analysis of Backward Solving

In this section, we examined the trade-off between the number of stored intermediate states and the accuracy of meta-gradient computation. To this end, we first considered a nonlinear ODE system, for which the gradient w.r.t the initial state has a closed form:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -2x^3 - 2ty. \tag{11}$$

The solution of the ODE is

$$y(t) = 1 - t^2 + ce^{-t^2} \tag{12}$$

where c is an arbitrary number and determined by the initial state, $c = y(0) - 1$. We then define a synthetic objective function,

$$\mathcal{L}(y(t)) = (y(t) - 3)^6. \tag{13}$$

Via the chain rule, we can obtain the gradient of the objective w.r.t to the initial state $y(0)$, *i.e.*, the meta gradient,

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y(0)} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y(t)} \cdot \frac{\mathrm{d}y(t)}{\mathrm{d}y(0)} = 6(y(t) - 3)^5 e^{-t^2}. \tag{14}$$

We then examined the relative L_2 error of the meta gradient calculation by our method, with different T's and numbers of intermediate states tracked in the back-solving process. For comparison, we tested the automatic differentiation (Autodiff) method based on computational graphs. To be fair, we used the same number of states to set the step size in the forward solving with the modified Euler method, and then applied Autodiff to compute the meta gradient. We repeated the experiment for 20 times, and each time we used a random initial state. The results are reported in Table 4. Note that our method uses DPORI5 (Runge-Kutta of order 5 of Dormand-Prince-Shampine) for the forward solving and the modified Euler for the backward solving. We can see that the accuracy of our method is better than or comparable to Autodiff in all the cases. Tracking more intermediate states consistently improves the accuracy, yet bringing more memory consumption and computational cost. Hence, it enables us to select the cost and accuracy trade-off.

Number of Intermediate States	$T = 0.1$		$T = 0.5$	
	Autodiff	Adjoint	Autodiff	Adjoint
10	6.24e-7 ± 8.87e-8	1.02e-6 ± 9.21e-7	3.34e-4 ± 1.06e-7	2.67e-4 ± 1.79e-4
20	2.01e-7 ± 1.08e-7	6.34e-7 ± 7.28e-7	7.34e-5 ± 1.61e-7	6.17e-5 ± 4.07e-5
50	1.48e-7 ± 1.20e-7	6.88e-7 ± 7.49e-7	1.08e-5 ± 2.49e-7	9.35e-6 ± 5.94e-6
100	6.75e-7 ± 2.00e-7	5.87e-7 ± 5.31e-7	2.64e-6 ± 2.59e-7	2.17e-6 ± 1.33e-6
200	7.32e-7 ± 7.61e-7	4.49e-7 ± 3.62e-7	9.18e-7 ± 4.13e-7	6.73e-7 ± 4.93e-7
500	7.69e-7 ± 1.13e-6	3.01e-7 ± 1.99e-7	1.12e-6 ± 6.59e-7	7.36e-7 ± 7.44e-7
1000	7.09e-7 ± 7.82e-7	1.45e-7 ± 8.58e-8	4.55e-6 ± 1.94e-6	6.94e-7 ± 4.73e-7

(a)

Number of Intermediate States	$T = 1.0$		$T = 2.0$	
	Autodiff	Adjoint	Autodiff	Adjoint
10	2.51e-3 ± 9.23e-4	2.48e-3 ± 7.28e-8	4.74e-1 ± 4.13e-3	2.92e-1 ± 1.41e-7
20	5.82e-4 ± 1.89e-4	5.04e-4 ± 1.64e-7	7.33e-2 ± 4.91e-4	4.77e-2 ± 2.00e-7
50	8.90e-5 ± 2.71e-5	7.17e-5 ± 2.14e-7	9.28e-3 ± 5.93e-5	6.10e-3 ± 2.51e-7
100	2.19e-5 ± 6.69e-6	1.73e-5 ± 3.40e-7	2.16e-3 ± 1.38e-5	1.42e-3 ± 2.88e-7
200	5.54e-6 ± 1.92e-6	4.34e-6 ± 4.82e-7	5.23e-4 ± 3.76e-6	3.44e-4 ± 5.31e-7
500	1.16e-6 ± 1.02e-6	9.91e-7 ± 5.42e-7	8.14e-5 ± 1.27e-6	5.39e-5 ± 6.10e-7
1000	1.51e-6 ± 1.51e-6	9.96e-7 ± 7.64e-7	2.01e-5 ± 1.34e-6	1.32e-5 ± 1.05e-6

(b)

Table 4: The relative L_2 error of meta-gradient computation. The results were averaged over 20 random initializations.

Next, we examined the trade-off in the 2D regression problem, *CosMixture* (see Sec. 5.1). In this problem, we do not have the ground-truth of the meta gradient. To evaluate the trade-off, we used the meta-gradient computed with $1,000$ intermediate states by our method as a reference. We then examined how the computed gradients using different numbers of states are close

to the reference. We varied T from $\{0.1, 0.3, 0.5\}$, and the number of intermediate steps from $\{10, 20, 50, 100, 200, 500\}$. We computed the relative L_2 error w.r.t the reference gradient. We tested on 20 random initializations. The results are reported in Table 5. As we can see, when only using 100 or 200 states, the computed meta gradient has already been very close to the one computed with 1,000 states. It implies that the gain of the accuracy is minor after a certain number of intermediate states. Hence, it is unnecessary to use too many states, and we can use much fewer to improve both the memory and computation efficiency.

Number of Intermediate States	$T = 0.1$	$T = 0.3$	$T = 0.5$
10	7.62e-4 $\pm$ 2.04e-4	4.82e-3 $\pm$ 9.07e-4	1.17e-2 $\pm$ 2.17e-3
20	3.55e-4 $\pm$ 9.49e-5	2.26e-3 $\pm$ 4.21e-4	5.46e-3 $\pm$ 1.01e-3
50	1.33e-4 $\pm$ 3.55e-5	8.63e-4 $\pm$ 1.61e-4	2.09e-3 $\pm$ 3.85e-4
100	6.24e-5 $\pm$ 1.67e-5	4.19e-4 $\pm$ 7.79e-5	1.03e-3 $\pm$ 1.88e-4
200	2.76e-5 $\pm$ 7.34e-6	2.01e-4 $\pm$ 3.74e-5	4.99e-4 $\pm$ 9.16e-5
500	6.87e-6 $\pm$ 1.83e-6	7.16e-5 $\pm$ 1.33e-5	1.87e-4 $\pm$ 3.42e-5

Table 5: The relative L_2 error w.r.t the gradient computed with 1K intermediate states on the *CosMixture* problem. The results were averaged from 20 random initializations.