On First-Order Meta-Learning Algorithms

Alex Nichol and Joshua Achiam and John Schulman
OpenAI
{alex, jachiam, joschu}@openai.com

Abstract

This paper considers meta-learning problems, where there is a distribution of tasks, and we would like to obtain an agent that performs well (i.e., learns quickly) when presented with a previously unseen task sampled from this distribution. We analyze a family of algorithms for learning a parameter initialization that can be fine-tuned quickly on a new task, using only first-order derivatives for the meta-learning updates. This family includes and generalizes first-order MAML, an approximation to MAML obtained by ignoring second-order derivatives. It also includes Reptile, a new algorithm that we introduce here, which works by repeatedly sampling a task, training on it, and moving the initialization towards the trained weights on that task. We expand on the results from Finn et al. showing that first-order meta-learning algorithms perform well on some well-established benchmarks for few-shot classification, and we provide theoretical analysis aimed at understanding why these algorithms work.

1 Introduction

While machine learning systems have surpassed humans at many tasks, they generally need far more data to reach the same level of performance. For example, Schmidt et al. [17, 15] showed that human subjects can recognize new object categories based on a few example images. Lake et al. [12] noted that on the Atari game of Frostbite, human novices were able to make significant progress on the game after 15 minutes, but double-dueling-DQN [19] required more than 1000 times more experience to attain the same score.

It is not completely fair to compare humans to algorithms learning from scratch, since humans enter the task with a large amount of prior knowledge, encoded in their brains and DNA. Rather than learning from scratch, they are fine-tuning and recombining a set of pre-existing skills. The work cited above, by Tenenbaum and collaborators, argues that humans' fast-learning abilities can be explained as Bayesian inference, and that the key to developing algorithms with human-level learning speed is to make our algorithms more Bayesian. However, in practice, it is challenging to develop (from first principles) Bayesian machine learning algorithms that make use of deep neural networks and are computationally feasible.

Meta-learning has emerged recently as an approach for learning from small amounts of data. Rather than trying to emulate Bayesian inference (which may be computationally intractable), meta-learning seeks to directly optimize a fast-learning algorithm, using a dataset of tasks. Specifically, we assume access to a distribution over tasks, where each task is, for example, a classification problem. From this distribution, we sample a training set and a test set of tasks. Our algorithm is fed the training set, and it must produce an agent that has good average performance on the test set. Since each task corresponds to a learning problem, performing well on a task corresponds to learning quickly.

A variety of different approaches to meta-learning have been proposed, each with its own pros and cons. In one approach, the learning algorithm is encoded in the weights of a recurrent network, but gradient descent is not performed at test time. This approach was proposed by Hochreiter et al. [8] who used LSTMs for next-step prediction and has been followed up by a burst of recent work, for example, Santoro et al. [16] on few-shot classification, and Duan et al. [3] for the POMDP setting.

A second approach is to learn the initialization of a network, which is then fine-tuned at test time on the new task. A classic example of this approach is pretraining using a large dataset (such as ImageNet [2]) and fine-tuning on a smaller dataset (such as a dataset of different species of bird [20]). However, this classic pre-training approach has no guarantee of learning an initialization that is good for fine-tuning, and ad-hoc tricks are required for good performance. More recently, Finn et al. [4] proposed an algorithm called MAML, which directly optimizes performance with respect to this initialization—differentiating through the fine-tuning process. In this approach, the learner falls back on a sensible gradient-based learning algorithm even when it receives out-of-sample data, thus allowing it to generalize better than the RNN-based approaches [5]. On the other hand, since MAML needs to differentiate through the optimization process, it's not a good match for problems where we need to perform a large number of gradient steps at test time. The authors also proposed a variant called first-order MAML (FOMAML), which is defined by ignoring the second derivative terms, avoiding this problem but at the expense of losing some gradient information. Surprisingly, though, they found that FOMAML worked nearly as well as MAML on the Mini-ImageNet dataset [18]. (This result was foreshadowed by prior work in meta-learning [1, 13] that ignored second derivatives when differentiating through gradient descent, without ill effect.) In this work, we expand on that insight and explore the potential of meta-learning algorithms based on first-order gradient information, motivated by the potential applicability to problems where it's too cumbersome to apply techniques that rely on higher-order gradients (like full MAML).

We make the following contributions:

- We point out that first-order MAML [4] is simpler to implement than was widely recognized prior to this article.

- We introduce Reptile, an algorithm closely related to FOMAML, which is equally simple to implement. Reptile is so similar to joint training (i.e., training to minimize loss on the expecation over training tasks) that it is especially surprising that it works as a meta-learning algorithm. Unlike FOMAML, Reptile doesn't need a training-test split for each task, which may make it a more natural choice in certain settings. It is also related to the older idea of fast weights / slow weights [7].

- We provide a theoretical analysis that applies to both first-order MAML and Reptile, showing that they both optimize for within-task generalization.

- On the basis of empirical evaluation on the Mini-ImageNet [18] and Omniglot [11] datasets, we provide some insights for best practices in implementation.

2   Meta-Learning an Initialization

We consider the optimization problem of MAML [4]: find an initial set of parameters, ϕ, such that for a randomly sampled task τ with corresponding loss L_τ, the learner will have low loss after k

updates. That is:

$$\underset{\phi}{\text{minimize}}\, \mathbb{E}_\tau \left[L_\tau \left(U_\tau^k(\phi) \right) \right], \tag{1}$$

where U_τ^k is the operator that updates ϕ k times using data sampled from τ. In few-shot learning, U corresponds to performing gradient descent or Adam [10] on batches of data sampled from τ.

MAML solves a version of Equation (1) that makes on additional assumption: for a given task τ, the inner-loop optimization uses training samples A, whereas the loss is computed using test samples B. This way, MAML optimizes for generalization, akin to cross-validation. Omitting the superscript k, we notate this as

$$\underset{\phi}{\text{minimize}}\, \mathbb{E}_\tau \left[L_{\tau,B} \left(U_{\tau,A}(\phi) \right) \right], \tag{2}$$

MAML works by optimizing this loss through stochastic gradient descent, i.e., computing

$$g_{\text{MAML}} = \frac{\partial}{\partial \phi} L_{\tau,B}(U_{\tau,A}(\phi)) \tag{3}$$

$$= U'_{\tau,A}(\phi) L'_{\tau,B}(\widetilde{\phi}), \quad \text{where} \quad \widetilde{\phi} = U_{\tau,A}(\phi) \tag{4}$$

In Equation (4), $U'_{\tau,A}(\phi)$ is the Jacobian matrix of the update operation $U_{\tau,A}$. $U_{\tau,A}$ corresponds to adding a sequence of gradient vectors to the initial vector, i.e., $U_{\tau,A}(\phi) = \phi + g_1 + g_2 + \cdots + g_k$. (In Adam, the gradients are also rescaled elementwise, but that does not change the conclusions.) First-order MAML (FOMAML) treats these gradients as constants, thus, it replaces Jacobian $U'_{\tau,A}(\phi)$ by the identity operation. Hence, the gradient used by FOMAML in the outer-loop optimization is $g_{\text{FOMAML}} = L'_{\tau,B}(\widetilde{\phi})$. Therefore, FOMAML can be implemented in a particularly simple way: (1) sample task τ; (2) apply the update operator, yielding $\widetilde{\phi} = U_{\tau,A}(\phi)$; (3) compute the gradient at $\widetilde{\phi}$, $g_{\text{FOMAML}} = L'_{\tau,B}(\widetilde{\phi})$; and finally (4) plug g_{FOMAML} into the outer-loop optimizer.

3 Reptile

In this section, we describe a new first-order gradient-based meta-learning algorithm called Reptile. Like MAML, Reptile learns an initialization for the parameters of a neural network model, such that when we optimize these parameters at test time, learning is fast—i.e., the model generalizes from a small number of examples from the test task. The Reptile algorithm is as follows:

Algorithm 1 Reptile (serial version)

Initialize ϕ, the vector of initial parameters
for iteration $= 1, 2, \ldots$ **do**
    Sample task τ, corresponding to loss L_τ on weight vectors $\widetilde{\phi}$
    Compute $\widetilde{\phi} = U_\tau^k(\phi)$, denoting k steps of SGD or Adam
    Update $\phi \leftarrow \phi + \epsilon(\widetilde{\phi} - \phi)$
end for

In the last step, instead of simply updating ϕ in the direction $\widetilde{\phi} - \phi$, we can treat $(\phi - \widetilde{\phi})$ as a gradient and plug it into an adaptive algorithm such as Adam [10]. (Actually, as we will discuss in Section 5.1, it is most natural to define the Reptile gradient as $(\phi - \widetilde{\phi})/\alpha$, where α is the stepsize

used by the SGD operation.) We can also define a parallel or batch version of the algorithm that evaluates on n tasks each iteration and updates the initialization to

$$\phi \leftarrow \phi + \epsilon \frac{1}{n} \sum_{i=1}^{n} (\widetilde{\phi}_i - \phi) \tag{5}$$

where $\widetilde{\phi}_i = U_{\tau_i}^k(\phi)$; the updated parameters on the ith task.

This algorithm looks remarkably similar to joint training on the expected loss $\mathbb{E}_\tau [L_\tau]$. Indeed, if we define U to be a single step of gradient descent ($k = 1$), then this algorithm corresponds to stochastic gradient descent on the expected loss:

$$g_{\text{Reptile},k=1} = \mathbb{E}_\tau \left[\phi - U_\tau(\phi) \right] / \alpha \tag{6}$$
$$= \mathbb{E}_\tau \left[\nabla_\phi L_\tau(\phi) \right] \tag{7}$$

However, if we perform multiple gradient updates in the partial minimization ($k > 1$), then the expected update $\mathbb{E}_\tau \left[U_\tau^k(\phi) \right]$ does not correspond to taking a gradient step on the expected loss $\mathbb{E}_\tau [L_\tau]$. Instead, the update includes important terms coming from second-and-higher derivatives of L_τ, as we will analyze in Section 5.1. Hence, Reptile converges to a solution that's very different from the minimizer of the expected loss $\mathbb{E}_\tau [L_\tau]$.

Other than the stepsize parameter ϵ and task sampling, the batched version of Reptile is the same as the SimuParallelSGD algorithm [21]. SimuParallelSGD is a method for communication-efficient distributed optimization, where workers perform gradient updates locally and infrequently average their parameters, rather than the standard approach of averaging gradients.

4 Case Study: One-Dimensional Sine Wave Regression

As a simple case study, let's consider the 1D sine wave regression problem, which is slightly modified from Finn et al. [4]. This problem is instructive since by design, joint training can't learn a very useful initialization; however, meta-learning methods can.

- The task $\tau = (a, b)$ is defined by the amplitude a and phase ϕ of a sine wave function $f_\tau(x) = a \sin(x + b)$. The task distribution by sampling $a \sim U([0.1, 5.0])$ and $b \sim U([0, 2\pi])$.

- Sample p points $x_1, x_2, \ldots, x_p \sim U([-5, 5])$

- Learner sees $(x_1, y_1), (x_2, y_2), \ldots, (x_p, y_p)$ and predicts the whole function $f(x)$

- Loss is ℓ_2 error on the whole interval $[-5, 5]$

$$L_\tau(f) = \int_{-5}^{5} \mathrm{d}x \| f(x) - f_\tau(x) \|^2 \tag{8}$$

We calculate this integral using 50 equally-spaced points x.

First note that the average function is zero everywhere, i.e., $\mathbb{E}_\tau [f_\tau(x)] = 0$, due to the random phase b. Therefore, it is useless to train on the expected loss $\mathbb{E}_\tau [L_\tau]$, as this loss is minimized by the zero function $f(x) = 0$.

On the other hand, MAML and Reptile give us an initialization that outputs approximately $f(x) = 0$ before training on a task τ, but the internal feature representations of the network are such that after training on the sampled datapoints $(x_1, y_1), (x_2, y_2), \ldots, (x_p, y_p)$, it closely approximates

4

the target function f_τ. This learning progress is shown in the figures below. Figure 1 shows that after Reptile training, the network can quickly converge to a sampled sine wave and infer the values away from the sampled points. As points of comparison, we also show the behaviors of MAML and a randomly-initialized network on the same task.

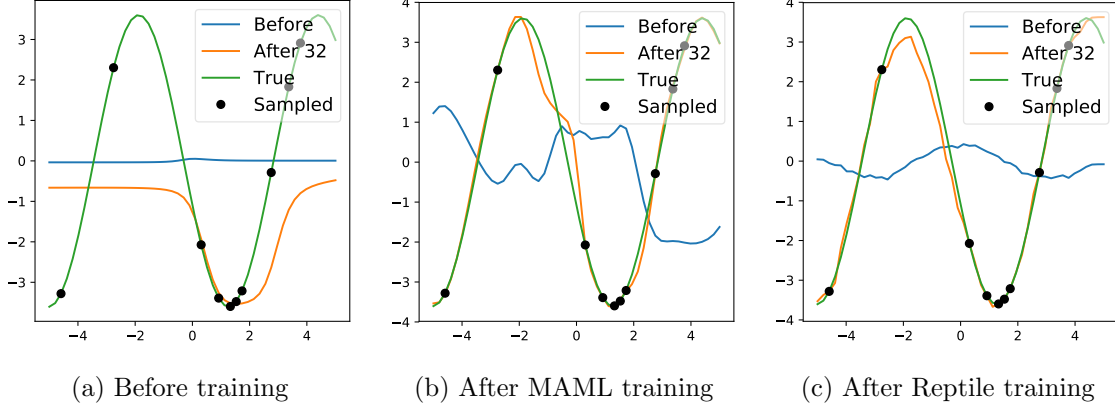(a) Before training   (b) After MAML training   (c) After Reptile training

Figure 1: Demonstration of MAML and Reptile on a toy few-shot regression problem, where we train on 10 sampled points of a sine wave, performing 32 gradient steps on an MLP with layers $1 \to 64 \to 64 \to 1$.

5 Analysis

In this section, we provide two alternative explanations of why Reptile works.

5.1 Leading Order Expansion of the Update

Here, we will use a Taylor series expansion to approximate the update performed by Reptile and MAML. We will show that both algorithms contain the same leading-order terms: the first term minimizes the expected loss (joint training), the second and more interesting term maximizes within-task generalization. Specifically, it maximizes the inner product between the gradients on different minibatches from the same task. If gradients from different batches have positive inner product, then taking a gradient step on one batch improves performance on the other batch.

Unlike in the discussion and analysis of MAML, we won't consider a training set and test set from each task; instead, we'll just assume that each task gives us a sequence of k loss functions $L_1, L_2, \ldots, L_k$; for example, classification loss on different minibatches. We will use the following definitions:

$$g_i = L_i'(\phi_i) \qquad \text{(gradient obtained during SGD)} \tag{9}$$

$$\phi_{i+1} = \phi_i - \alpha g_i \quad \text{(sequence of parameter vectors)} \tag{10}$$

$$\overline{g}_i = L_i'(\phi_1) \qquad \text{(gradient at initial point)} \tag{11}$$

$$\overline{H}_i = L_i''(\phi_1) \qquad \text{(Hessian at initial point)} \tag{12}$$

For each of these definitions, $i \in [1, k]$.

First, let's calculate the SGD gradients to $O(\alpha^2)$ as follows.

$$g_i = L_i'(\phi_i) = L_i'(\phi_1) + L_i''(\phi_1)(\phi_i - \phi_1) + \underbrace{O(\|\phi_i - \phi_1\|^2)}_{=O(\alpha^2)} \quad \text{(Taylor's theorem)} \tag{13}$$

$$= \bar{g}_i + \bar{H}_i(\phi_i - \phi_1) + O(\alpha^2) \quad \text{(using definition of } \bar{g}_i, \bar{H}_i) \tag{14}$$

$$= \bar{g}_i - \alpha \bar{H}_i \sum_{j=1}^{i-1} g_j + O(\alpha^2) \quad \text{(using } \phi_i - \phi_1 = -\alpha \sum_{j=1}^{i-1} g_j) \tag{15}$$

$$= \bar{g}_i - \alpha \bar{H}_i \sum_{j=1}^{i-1} \bar{g}_j + O(\alpha^2) \quad \text{(using } g_j = \bar{g}_j + O(\alpha)) \tag{16}$$

Next, we will approximate the MAML gradient. Define U_i as the operator that updates the parameter vector on minibatch i: $U_i(\phi) = \phi - \alpha L_i'(\phi)$.

$$g_{\text{MAML}} = \frac{\partial}{\partial \phi_1} L_k(\phi_k) \tag{17}$$

$$= \frac{\partial}{\partial \phi_1} L_k(U_{k-1}(U_{k-2}(\ldots (U_1(\phi_1))))) \tag{18}$$

$$= U_1'(\phi_1) \cdots U_{k-1}'(\phi_{k-1}) L_k'(\phi_k) \quad \text{(repeatedly applying the chain rule)} \tag{19}$$

$$= \left(I - \alpha L_1''(\phi_1)\right) \cdots \left(I - \alpha L_{k-1}''(\phi_{k-1})\right) L_k'(\phi_k) \quad \text{(using } U_i'(\phi) = I - \alpha L_i''(\phi)) \tag{20}$$

$$= \left(\prod_{j=1}^{k-1} (I - \alpha L_j''(\phi_j))\right) g_k \quad \text{(product notation, definition of } g_k) \tag{21}$$

Next, let's expand to leading order

$$g_{\text{MAML}} = \left(\prod_{j=1}^{k-1} (I - \alpha \bar{H}_j)\right) \left(\bar{g}_k - \alpha \bar{H}_k \sum_{j=1}^{k-1} \bar{g}_j\right) + O(\alpha^2) \tag{22}$$

$$\text{(replacing } L_j''(\phi_j) \text{ with } \bar{H}_j, \text{ and replacing } g_k \text{ using Equation (16))}$$

$$= \left(I - \alpha \sum_{j=1}^{k-1} \bar{H}_j\right) \left(\bar{g}_k - \alpha \bar{H}_k \sum_{j=1}^{k-1} \bar{g}_j\right) + O(\alpha^2) \tag{23}$$

$$= \bar{g}_k - \alpha \sum_{j=1}^{k-1} \bar{H}_j \bar{g}_k - \alpha \bar{H}_k \sum_{j=1}^{k-1} \bar{g}_j + O(\alpha^2) \tag{24}$$

For simplicity of exposition, let's consider the $k = 2$ case, and later we'll provide the general formulas.

$$g_{\text{MAML}} = \bar{g}_2 - \alpha \bar{H}_2 \bar{g}_1 - \alpha \bar{H}_1 \bar{g}_2 + O(\alpha^2) \tag{25}$$

$$g_{\text{FOMAML}} = g_2 = \bar{g}_2 - \alpha \bar{H}_2 \bar{g}_1 + O(\alpha^2) \tag{26}$$

$$g_{\text{Reptile}} = g_1 + g_2 = \bar{g}_1 + \bar{g}_2 - \alpha \bar{H}_2 \bar{g}_1 + O(\alpha^2) \tag{27}$$

As we will show in the next paragraph, the terms like $\bar{H}_2 \bar{g}_1$ serve to maximize the inner products between the gradients computed on different minibatches, while lone gradient terms like $\bar{g}_1$ take us to the minimum of the joint training problem.

When we take the expectation of g_{FOMAML}, g_{Reptile}, and g_{MAML} under minibatch sampling, we are left with only two kinds of terms which we will call AvgGrad and AvgGradInner. In the equations below $\mathbb{E}_{\tau,1,2}[\dots]$ means that we are taking the expectation over the task τ and the two minibatches defining L_1 and L_2, respectively.

- AvgGrad is defined as gradient of expected loss.

$$\text{AvgGrad} = \mathbb{E}_{\tau,1}\left[\bar{g}_1\right] \tag{28}$$

  $(-\text{AvgGrad})$ is the direction that brings ϕ towards the minimum of the "joint training" problem; the expected loss over tasks.

- The more interesting term is AvgGradInner, defined as follows:

$$\text{AvgGradInner} = \mathbb{E}_{\tau,1,2}\left[\overline{H}_2\bar{g}_1\right] \tag{29}$$

$$= \mathbb{E}_{\tau,1,2}\left[\overline{H}_1\bar{g}_2\right] \qquad \text{(interchanging indices } 1, 2) \tag{30}$$

$$= \tfrac{1}{2}\mathbb{E}_{\tau,1,2}\left[\overline{H}_2\bar{g}_1 + \overline{H}_1\bar{g}_2\right] \quad \text{(averaging last two equations)} \tag{31}$$

$$= \tfrac{1}{2}\mathbb{E}_{\tau,1,2}\left[\frac{\partial}{\partial\phi_1}(\bar{g}_1 \cdot \bar{g}_2)\right] \tag{32}$$

  Thus, $(-\text{AvgGradInner})$ is the direction that increases the inner product between gradients of different minibatches for a given task, improving generalization.

Recalling our gradient expressions, we get the following expressions for the meta-gradients, for SGD with $k = 2$:

$$\mathbb{E}\left[g_{\text{MAML}}\right] = (1)\text{AvgGrad} - (2\alpha)\text{AvgGradInner} + O(\alpha^2) \tag{33}$$

$$\mathbb{E}\left[g_{\text{FOMAML}}\right] = (1)\text{AvgGrad} - (\alpha)\text{AvgGradInner} + O(\alpha^2) \tag{34}$$

$$\mathbb{E}\left[g_{\text{Reptile}}\right] = (2)\text{AvgGrad} - (\alpha)\text{AvgGradInner} + O(\alpha^2) \tag{35}$$

In practice, all three gradient expressions first bring us towards the minimum of the expected loss over tasks, then the higher-order AvgGradInner term enables fast learning by maximizing the inner product between gradients within a given task.

Finally, we can extend these calculations to the general $k \geq 2$ case:

$$g_{\text{MAML}} = \bar{g}_k - \alpha\overline{H}_k\sum_{j=1}^{k-1}\bar{g}_j - \alpha\sum_{j=1}^{k-1}\overline{H}_j\bar{g}_k + O(\alpha^2) \tag{36}$$

$$\mathbb{E}\left[g_{\text{MAML}}\right] = (1)\text{AvgGrad} - (2(k-1)\alpha)\text{AvgGradInner} \tag{37}$$

$$g_{\text{FOMAML}} = g_k = \bar{g}_k - \alpha\overline{H}_k\sum_{j=1}^{k-1}\bar{g}_j + O(\alpha^2) \tag{38}$$

$$\mathbb{E}\left[g_{\text{FOMAML}}\right] = (1)\text{AvgGrad} - ((k-1)\alpha)\text{AvgGradInner} \tag{39}$$

$$g_{\text{Reptile}} = -(\phi_{k+1} - \phi_1)/\alpha = \sum_{i=1}^{k} g_i = \sum_{i=1}^{k}\bar{g}_i - \alpha\sum_{i=1}^{k}\sum_{j=1}^{i-1}\overline{H}_i\bar{g}_j + O(\alpha^2) \tag{40}$$

$$\mathbb{E}\left[g_{\text{Reptile}}\right] = (k)\text{AvgGrad} - \left(\tfrac{1}{2}k(k-1)\alpha\right)\text{AvgGradInner} \tag{41}$$

As in the $k = 2$, the ratio of coefficients of the AvgGradInner term and the AvgGrad term goes MAML > FOMAML > Reptile. However, in all cases, this ratio increases linearly with both the stepsize α and the number of iterations k. Note that the Taylor series approximation only holds for small αk.
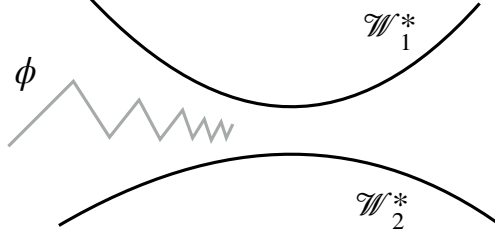
Figure 2: The above illustration shows the sequence of iterates obtained by moving alternately towards two optimal solution manifolds $\mathcal{W}_1$ and $\mathcal{W}_2$ and converging to the point that minimizes the average squared distance. One might object to this picture on the grounds that we converge to the same point regardless of whether we perform one step or multiple steps of gradient descent. That statement is true, however, note that minimizing the expected distance objective $\mathbb{E}_\tau \left[D(\phi, \mathcal{W}_\tau) \right]$ is different than minimizing the expected loss objective $\mathbb{E}_\tau \left[L_\tau(f_\phi) \right]$. In particular, there is a high-dimensional manifold of minimizers of the expected loss L_τ (e.g., in the sine wave case, many neural network parameters give the zero function $f(\phi) = 0$), but the minimizer of the expected distance objective is typically a single point.

5.2 Finding a Point Near All Solution Manifolds

Here, we argue that Reptile converges towards a solution ϕ that is close (in Euclidean distance) to each task τ's manifold of optimal solutions. This is a informal argument and should be taken much less seriously than the preceding Taylor series analysis.

Let ϕ denote the network initialization, and let $\mathcal{W}_\tau$ denote the set of optimal parameters for task τ. We want to find ϕ such that the distance $D(\phi, \mathcal{W}_\tau)$ is small for all tasks.

$$\underset{\phi}{\text{minimize}}\, \mathbb{E}_\tau \left[\tfrac{1}{2} D(\phi, \mathcal{W}_\tau)^2 \right] \tag{42}$$

We will show that Reptile corresponds to performing SGD on that objective.

Given a non-pathological set $S \subset \mathbb{R}^d$, then for almost all points $\phi \in \mathbb{R}^d$ the gradient of the squared distance $D(\phi, S)^2$ is $2(\phi - P_S(\phi))$, where $P_S(\phi)$ is the projection (closest point) of ϕ onto S. Thus,

$$\nabla_\phi \mathbb{E}_\tau \left[\tfrac{1}{2} D(\phi, \mathcal{W}_\tau)^2 \right] = \mathbb{E}_\tau \left[\tfrac{1}{2} \nabla_\phi D(\phi, \mathcal{W}_\tau)^2 \right] \tag{43}$$

$$= \mathbb{E}_\tau \left[\phi - P_{\mathcal{W}_\tau}(\phi) \right], \text{where } P_{\mathcal{W}_\tau}(\phi) = \underset{p \in \mathcal{W}_\tau}{\arg\min}\, D(p, \phi) \tag{44}$$

Each iteration of Reptile corresponds to sampling a task τ and performing a stochastic gradient update

$$\phi \leftarrow \phi - \epsilon \nabla_\phi \tfrac{1}{2} D(\phi, \mathcal{W}_\tau)^2 \tag{45}$$

$$= \phi - \epsilon(\phi - P_{\mathcal{W}_\tau}(\phi)) \tag{46}$$

$$= (1 - \epsilon)\phi + \epsilon P_{\mathcal{W}_\tau}(\phi). \tag{47}$$

In practice, we can't exactly compute $P_{\mathcal{W}_\tau}(\phi)$, which is defined as a minimizer of L_τ. However, we can partially minimize this loss using gradient descent. Hence, in Reptile we replace $W_\tau^*(\phi)$ by the result of running k steps of gradient descent on L_τ starting with initialization ϕ.

6 Experiments

6.1 Few-Shot Classification

We evaluate our method on two popular few-shot classification tasks: Omniglot [11] and Mini-ImageNet [18]. These datasets make it easy to compare our method to other few-shot learning

approaches like MAML.

In few-shot classification tasks, we have a meta-dataset D containing many classes C, where each class is itself a set of example instances $\{c_1, c_2, ..., c_n\}$. If we are doing K-shot, N-way classification, then we sample tasks by selecting N classes from C and then selecting $K + 1$ examples for each class. We split these examples into a training set and a test set, where the test set contains a single example for each class. The model gets to see the entire training set, and then it must classify a randomly chosen sample from the test set. For example, if you trained a model for 5-shot, 5-way classification, then you would show it 25 examples (5 per class) and ask it to classify a 26th example.

In addition to the above setup, we also experimented with the *transductive* setting, where the model classifies the entire test set at once. In our transductive experiments, information was shared between the test samples via batch normalization [9]. In our non-transductive experiments, batch normalization statistics were computed using all of the training samples and a single test sample. We note that Finn et al. [4] use transduction for evaluating MAML.

For our experiments, we used the same CNN architectures and data preprocessing as Finn et al. [4]. We used the Adam optimizer [10] in the inner loop, and vanilla SGD in the outer loop, throughout our experiments. For Adam we set $\beta_1 = 0$ because we found that momentum reduced performance across the board.[1] During training, we never reset or interpolated Adam's rolling moment data; instead, we let it update automatically at every inner-loop training step. However, we did backup and reset the Adam statistics when evaluating on the test set to avoid information leakage.

The results on Omniglot and Mini-ImageNet are shown in Tables 1 and 2. While MAML, FOMAML, and Reptile have very similar performance on all of these tasks, Reptile does slightly better than the alternatives on Mini-ImageNet and slightly worse on Omniglot. It also seems that transduction gives a performance boost in all cases, suggesting that further research should pay close attention to its use of batch normalization during testing.

Algorithm	1-shot 5-way	5-shot 5-way
MAML + Transduction	$48.70 \pm 1.84\%$	$63.11 \pm 0.92\%$
1st-order MAML + Transduction	$48.07 \pm 1.75\%$	$63.15 \pm 0.91\%$
Reptile	$47.07 \pm 0.26\%$	$62.74 \pm 0.37\%$
Reptile + Transduction	$49.97 \pm 0.32\%$	$65.99 \pm 0.58\%$

Table 1: Results on Mini-ImageNet. Both MAML and 1st-order MAML results are from [4].

Algorithm	1-shot 5-way	5-shot 5-way	1-shot 20-way	5-shot 20-way
MAML + Transduction	$98.7 \pm 0.4\%$	$99.9 \pm 0.1\%$	$95.8 \pm 0.3\%$	$98.9 \pm 0.2\%$
1st-order MAML + Transduction	$98.3 \pm 0.5\%$	$99.2 \pm 0.2\%$	$89.4 \pm 0.5\%$	$97.9 \pm 0.1\%$
Reptile	$95.39 \pm 0.09\%$	$98.90 \pm 0.10\%$	$88.14 \pm 0.15\%$	$96.65 \pm 0.33\%$
Reptile + Transduction	$97.68 \pm 0.04\%$	$99.48 \pm 0.06\%$	$89.43 \pm 0.14\%$	$97.12 \pm 0.32\%$

Table 2: Results on Omniglot. MAML results are from [4]. 1st-order MAML results were generated by the code for [4] with the same hyper-parameters as MAML.

---

[1]This finding also matches our analysis from Section 5.1, which suggests that Reptile works because sequential steps come from different mini-batches. With momentum, a mini-batch has influence over the next few steps, reducing this effect.

6.2 Comparing Different Inner-Loop Gradient Combinations

For this experiment, we used four non-overlapping mini-batches in each inner-loop, yielding gradients g_1, g_2, g_3, and g_4. We then compared learning performance when using different linear combinations of the g_i's for the outer loop update. Note that two-step Reptile corresponds to $g_1 + g_2$, and two-step FOMAML corresponds to g_2.

To make it easier to get an apples-to-apples comparison between different linear combinations, we simplified our experimental setup in several ways. First, we used vanilla SGD in the inner- and outer-loops. Second, we did not use meta-batches. Third, we restricted our experiments to 5-shot, 5-way Omniglot. With these simplifications, we did not have to worry as much about the effects of hyper-parameters or optimizers.

Figure 3 shows the learning curves for various inner-loop gradient combinations. For gradient combinations with more than one term, we ran both a sum and an average of the inner gradients to correct for the effective step size increase.
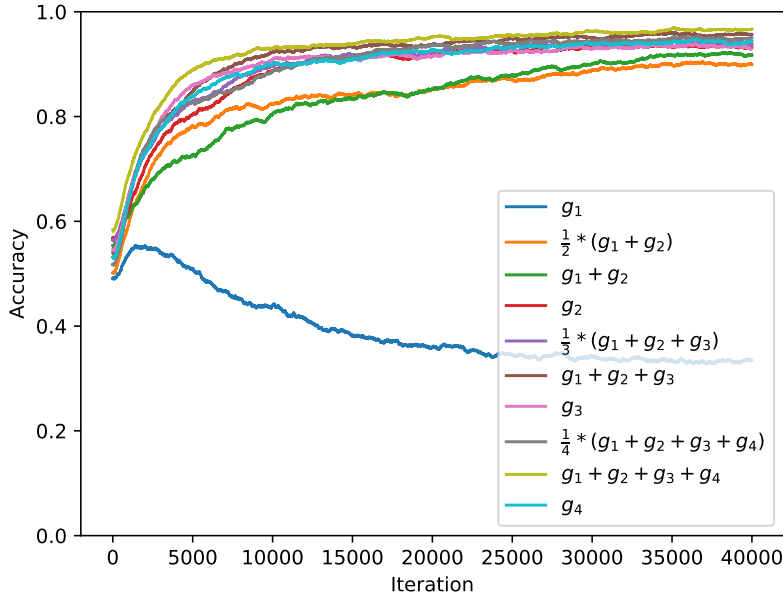

Figure 3: Different inner-loop gradient combinations on 5-shot 5-way Omniglot.

As expected, using only the first gradient g_1 is quite ineffective, since it amounts to optimizing the expected loss over all tasks. Surprisingly, two-step Reptile is noticeably worse than two-step FOMAML, which might be explained by the fact that two-step Reptile puts less weight on AvgGradInner relative to AvgGrad (Equations (34) and (35)). Most importantly, though, all the methods improve as the number of mini-batches increases. This improvement is more significant when using a sum of all gradients (Reptile) rather than using just the final gradient (FOMAML). This also suggests that Reptile can benefit from taking many inner loop steps, which is consistent with the optimal hyper-parameters found for Section 6.1.
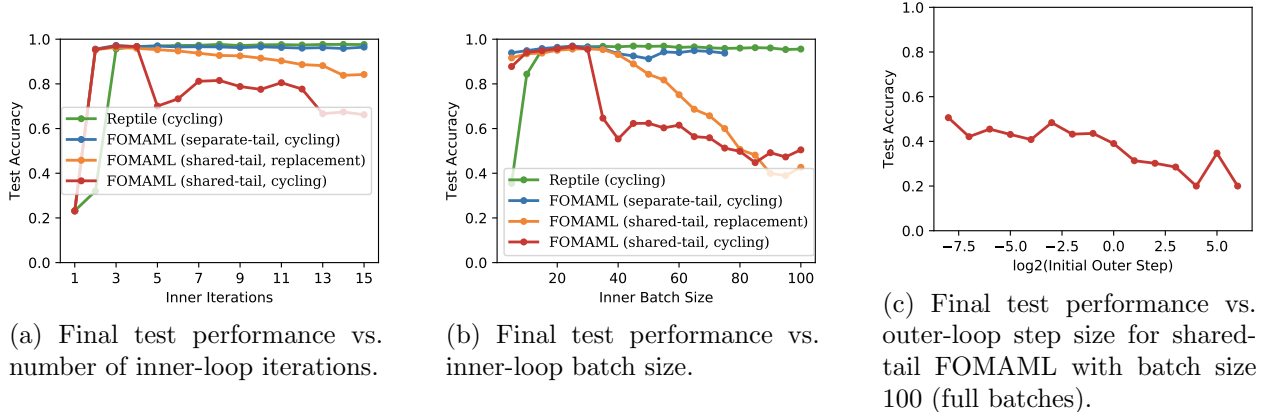
(a) Final test performance vs. number of inner-loop iterations.

(b) Final test performance vs. inner-loop batch size.

(c) Final test performance vs. outer-loop step size for shared-tail FOMAML with batch size 100 (full batches).

Figure 4: The results of hyper-parameter sweeps on 5-shot 5-way Omniglot.

6.3 Overlap Between Inner-Loop Mini-Batches

Both Reptile and FOMAML use stochastic optimization in their inner-loops. Small changes to this optimization procedure can lead to large changes in final performance. This section explores the sensitivity of Reptile and FOMAML to the inner loop hyperparameters, and also shows that FOMAML's performance significantly drops if mini-batches are selected the wrong way.

The experiments in this section look at the difference between *shared-tail FOMAML*, where the final inner-loop mini-batch comes from the same set of data as the earlier inner-loop batches, to *separate-tail FOMAML*, where the final mini-batch comes from a disjoint set of data. Viewing FOMAML as an approximation to MAML, *separate-tail FOMAML* can be seen as the more correct approach (and was used by Finn et al. [4]), since the training-time optimization resembles the test-time optimization (where the test set doesn't overlap with the training set). Indeed, we find that separate-tail FOMAML is significantly better than shared-tail FOMAML. As we will show, shared-tail FOMAML degrades in performance when the data used to compute the meta-gradient ($g_{\mathrm{FOMAML}} = g_k$) overlaps significantly with the earlier batches; however, Reptile and separate-tail MAML maintain performance and are not very sensitive to the inner-loop hyperparameters.

Figure 4a shows that when minibatches are selected by cycling through the training data (shared-tail, cycle), shared-tail FOMAML performs well up to four inner-loop iterations, but drops in performance starting at five iterations, where the final minibatch (used to compute $g_{\mathrm{FOMAML}} = g_k$) overlaps with the earlier ones. When we use random sampling instead (shared-tail, replacement), shared-tail FOMAML degrades more gradually. We hypothesize that this is because some samples still appear in the final batch that were not in the previous batches. The effect is stochastic, so it makes sense that the curve is smoother.

Figure 4b shows a similar phenomenon, but here we fixed the inner-loop to four iterations and instead varied the batch size. For batch sizes greater than 25, the final inner-loop batch for shared-tail FOMAML necessarily contains samples from the previous batches. Similar to Figure 4a, here we observe that shared-tail FOMAML with random sampling degrades more gradually than shared-tail FOMAML with cycling.

In both of these parameter sweeps, separate-tail FOMAML and Reptile do not degrade in performance as the number of inner-loop iterations or batch size changes.

There are several possible explanations for above findings. For example, one might hypothesize that shared-tail FOMAML is only worse in these experiments because its effective step size is much lower than that of separate-tail FOMAML. However, Figure 4c suggests that this is not the

11

case: performance was equally poor for every choice of step size in a thorough sweep. A different hypothesis is that shared-tail FOMAML performs poorly because, after a few inner-loop steps on a sample, the gradient of the loss for that sample does not contain very much useful information about the sample. In other words, the first few SGD steps might bring the model close to a local optimum, and then further SGD steps might simply bounce around this local optimum.

7   Discussion

Meta-learning algorithms that perform gradient descent at test time are appealing because of their simplicity and generalization properties [5]. The effectiveness of fine-tuning (e.g. from models trained on ImageNet [2]) gives us additional faith in these approaches. This paper proposed a new algorithm called Reptile, whose training process is only subtlely different from joint training and only uses first-order gradient information (like first-order MAML).

We gave two theoretical explanations for why Reptile works. First, by approximating the update with a Taylor series, we showed that SGD *automatically* gives us the same kind of second-order term that MAML computes. This term adjusts the initial weights to maximize the dot product between the gradients of different minibatches on the same task—i.e., it encourages the gradients to generalize between minibatches of the same task. We also provided a second informal argument, which is that Reptile finds a point that is close (in Euclidean distance) to all of the optimal solution manifolds of the training tasks.

While this paper studies the meta-learning setting, the Taylor series analysis in Section 5.1 may have some bearing on stochastic gradient descent in general. It suggests that when doing stochastic gradient descent, we are automatically performing a MAML-like update that maximizes the generalization between different minibatches. This observation partly explains why fine tuning (e.g., from ImageNet to a smaller dataset [20]) works well. This hypothesis would suggest that *joint training plus fine tuning* will continue to be a strong baseline for meta-learning in various machine learning problems.

8   Future Work

We see several promising directions for future work:

- Understanding to what extent SGD automatically optimizes for generalization, and whether this effect can be amplified in the non-meta-learning setting.

- Applying Reptile in the reinforcement learning setting. So far, we have obtained negative results, since joint training is a strong baseline, so some modifications to Reptile might be necessary.

- Exploring whether Reptile's few-shot learning performance can be improved by deeper architectures for the classifier.

- Exploring whether regularization can improve few-shot learning performance, as currently there is a large gap between training and testing error.

- Evaluating Reptile on the task of few-shot density modeling [14].

References

[1] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, 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] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[3] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. RL^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[4] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

[5] Chelsea Finn and Sergey Levine. Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. *arXiv preprint arXiv:1710.11622*, 2017.

[6] Nikolaus Hansen. The CMA evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer, 2006.

[7] Geoffrey E Hinton and David C Plaut. Using fast weights to deblur old memories. In *Proceedings of the ninth annual conference of the Cognitive Science Society*, pages 177–186, 1987.

[8] 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.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[11] Brenden M. Lake, Ruslan Salakhutdinov, Jason Gross, and Joshua B. Tenenbaum. One shot learning of simple visual concepts. In *Conference of the Cognitive Science Society (CogSci)*, 2011.

[12] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[13] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. In *International Conference on Learning Representations (ICLR)*, 2017.

[14] Scott Reed, Yutian Chen, Thomas Paine, Aäron van den Oord, SM Eslami, Danilo Rezende, Oriol Vinyals, and Nando de Freitas. Few-shot autoregressive density estimation: Towards learning to learn distributions. *arXiv preprint arXiv:1710.10304*, 2017.

[15] Ruslan Salakhutdinov, Joshua Tenenbaum, and Antonio Torralba. One-shot learning with a hierarchical nonparametric bayesian model. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, pages 195–206, 2012.

[16] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pages 1842–1850, 2016.

[17] Lauren A Schmidt. *Meaning and compositionality as statistical induction of categories and constraints*. PhD thesis, Massachusetts Institute of Technology, 2009.

[18] Oriol Vinyals, Charles Blundell, Tim Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *Advances in Neural Information Processing Systems*, pages 3630–3638, 2016.

[19] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[20] Ning Zhang, Jeff Donahue, Ross Girshick, and Trevor Darrell. Part-based R-CNNs for fine-grained category detection. In *European conference on computer vision*, pages 834–849. Springer, 2014.

[21] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.

A Hyper-parameters

For all experiments, we linearly annealed the outer step size to 0. We ran each experiment with three different random seeds, and computed the confidence intervals using the standard deviation across the runs.

Initially, we tried optimizing the Reptile hyper-parameters using CMA-ES [6]. However, we found that most hyper-parameters had little effect on the resulting performance. After seeing this result, we simplified all of the hyper-parameters and shared hyper-parameters between experiments when it made sense.

Table 3: Reptile hyper-parameters for the Omniglot comparison between all algorithms.

Parameter	5-way	20-way
Adam learning rate	0.001	0.0005
Inner batch size	10	20
Inner iterations	5	10
Training shots	10	10
Outer step size	1.0	1.0
Outer iterations	100K	200K
Meta-batch size	5	5
Eval. inner iterations	50	50
Eval. inner batch	5	10

Table 4: Reptile hyper-parameters for the Mini-ImageNet comparison between all algorithms.

Parameter	1-shot	5-shot
Adam learning rate	0.001	0.001
Inner batch size	10	10
Inner iterations	8	8
Training shots	15	15
Outer step size	1.0	1.0
Outer iterations	100K	100K
Meta-batch size	5	5
Eval. inner batch size	5	15
Eval. inner iterations	50	50

Table 5: Hyper-parameters for Section 6.2. All outer step sizes were linearly annealed to zero during training.

Parameter	Value
Inner learning rate	3×10^{-3}
Inner batch size	25
Outer step size	0.25
Outer iterations	40K
Eval. inner batch size	25
Eval. inner iterations	5

Table 6: Hyper-parameters Section 6.3. All outer step sizes were linearly annealed to zero during training.

Parameter	Figure 4b	Figure 4a	Figure 4c
Inner learning rate	3×10^{-3}	3×10^{-3}	3×10^{-3}
Inner batch size	-	25	100
Inner iterations	4	-	4
Outer step size	1.0	1.0	-
Outer iterations	40K	40K	40K
Eval. inner batch size	25	25	25
Eval. inner iterations	5	5	5