Reptile: a Scalable Metalearning Algorithm

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

Abstract

This paper considers metalearning 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 present a remarkably simple metalearning algorithm called Reptile, which learns a parameter initialization that can be fine-tuned quickly on a new task. Reptile works by repeatedly sampling a task, training on it, and moving the initialization towards the trained weights on that task. Unlike MAML, which also learns an initialization, Reptile doesn't require differentiating through the optimization process, making it more suitable for optimization problems where many update steps are required. We show that Reptile performs well on some well-established benchmarks for few-shot classification. We provide some theoretical analysis aimed at understanding why Reptile works.

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. [Sch09; STT12] showed that human subjects can recognize new object categories based on a few example images. Lake et al. [LST15] 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 [WSH+15] 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.

Metalearning 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), metalearning 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 task. From this distribution, we sample a training set and a test set. 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 metalearning 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.

[HYC01] who used LSTMs for next-step prediction and has been followed up by a burst of recent work, for example, Santoro et al. [SBB+16] on few-shot classification, and Duan et al. [DSC+16] 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 [DDS+09]) and fine-tuning on a smaller dataset (such as a dataset of different species of bird [ZDGD14]). 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. [FAL17] 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 [FL17]. 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. (In Section 4, we will discuss the connection between Reptile and first-order MAML, which is an approximation to MAML that ignores the second derivative term [FAL17].)

2 Algorithm

In this paper, we propose a remarkably simple algorithm for metalearning, which we name 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. Let ϕ denote a vector of parameters of a model, and let τ denote a task, and let $\text{SGD}(L, \phi, k)$ denote the function that performs k gradient steps on loss L starting with ϕ and returns the final parameter vector. The Reptile algorithm (at training time) 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 W
  Compute $W = \text{SGD}(L_\tau, \phi, k)$
  Update $\phi \leftarrow \phi + \epsilon(W - \phi)$
end for

In the last step, instead of simply updating ϕ in the direction $W - \phi$, we can treat $(\phi - W)$ as a gradient and plug it into an adaptive algorithm such as Adam [KB15]. (Actually, as we will discuss in Section 4.1, it is most natural to define the Reptile gradient as $(\phi - W)/\alpha$, 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 multiple tasks each iteration.

Algorithm 2 Reptile, batched version

Initialize ϕ
for iteration $= 1, 2, \ldots$ **do**
  Sample tasks $\tau_1, \tau_2, \ldots, \tau_n$
  for $i = 1, 2, \ldots, n$ **do**
    Compute $W_i = \text{SGD}(L_{\tau_i}, \phi, k)$
  end for
  Update $\phi \leftarrow \phi + \epsilon \frac{1}{k} \sum_{i=1}^{n} (W_i - \phi)$
end for

You might be thinking "isn't this the same as training on the expected loss $\mathbb{E}_\tau [L_\tau]$?" and then checking if the date is April 1st. Indeed, if the partial minimization consists of a single gradient step, then this algorithm corresponds to minimizing the expected loss:

$$\mathbb{E}_\tau [\nabla_\phi L_\tau(f_\phi)] = \nabla_\phi \mathbb{E}_\tau [L_\tau(f_\phi)] \tag{1}$$

However, if we perform multiple gradient updates in the partial minimization, then the average update is not equal to the update on the average function. The equality

$$\mathbb{E}_\tau [\text{SGD}(L_\tau, \phi, k)] \stackrel{?}{=} \text{SGD}(\mathbb{E}_\tau [L_\tau], \phi, k) \tag{2}$$

only holds for $k = 1$. For $k > 1$, the expected update (i.e., LHS of Equation (2)) depends on the higher order derivatives of L_τ, and that's why Reptile converges to a solution that's very different from the minimizer of the expected loss $\mathbb{E}_\tau [L_\tau]$.

3 Case Study: One-Dimensional Sine Wave Regression

As a simple case study, let's consider the 1D sine wave regression problem from Finn et al. [FAL17], which is defined as follows.

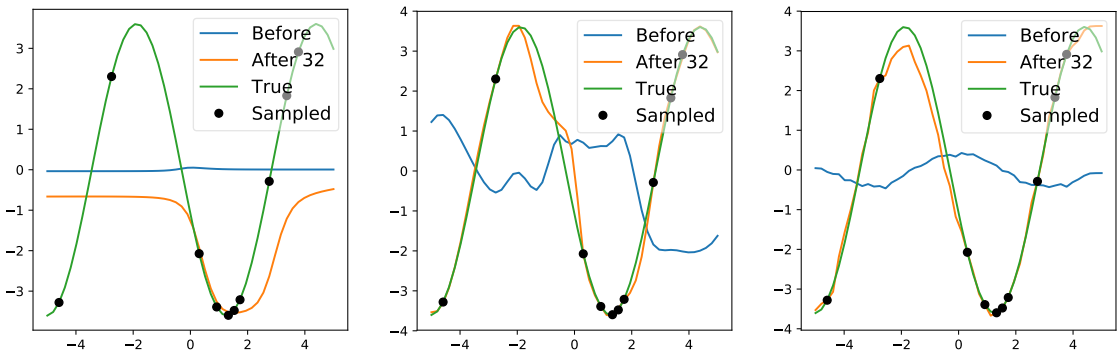
- The task $\tau = (a, b)$ is defined by the amplitude a and phase ϕ of a sine wave function $f_\tau(x) = a \sin(x + b)$. The task distribution by sampling $a \sim U([0.1, 5.0])$ and $b \sim U([0, 2\pi])$.
- Sample p points $x_1, x_2, \ldots, x_p \sim U([-5, 5])$
- Learner sees $(x_1, y_1), (x_2, y_2), \ldots, (x_p, y_p)$ and predicts the whole function $f(x)$
- Loss is ℓ_2 error on the whole interval $[-5, 5]$

$$L_\tau(f) = \int_{-5}^{5} \mathrm{d}x \|f(x) - f_\tau(x)\|^2 \tag{3}$$

  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, Reptile gives 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 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.



(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 $1 \to 64 \to 64 \to 1$.

4 Analysis

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

4.1 Leading Order Expansion of the Update

Here, we'll use a Taylor series to approximate the update performed by Reptile and MAML [FAL17]. For simplicity, we'll consider the case of two steps of SGD for both Reptile and MAML.

Suppose we perform two steps of SGD, first on loss L_0, then on loss L_1. (The subscripts 0 and 1 thus correspond to different minibatches of data for the same task.) Let ϕ denote the initial parameter vector, and let α denote the stepsize. Let the "prime" symbol denote differentiation: $L'(\phi) = \frac{\partial}{\partial \phi} L(\phi), L''(\phi) = \frac{\partial^2}{\partial \phi^2} L(\phi)$, etc.

The updated parameter after two SGD steps is

$$\phi_0 = \phi \tag{4}$$
$$\phi_1 = \phi_0 - \alpha L_0'(\phi_0) \tag{5}$$
$$\phi_2 = \phi_0 - \alpha L_0'(\phi_0) - \alpha L_1'(\phi_1) \tag{6}$$

MAML and Reptile each compute a gradient to update the initial parameter ϕ_0. We will compare the gradients.

First, let's take the Taylor expansion of $L_1'(\phi_1)$, as this term appears in both calculations.

$$L_1'(\phi_1) = L_1'(\phi_0) + L_1''(\phi_0)(\phi_1 - \phi_0) + O(\alpha^2) \tag{7}$$
$$= L_1'(\phi_0) - \alpha L_1''(\phi_0) L_0'(\phi_0) + O(\alpha^2) \tag{8}$$

The Reptile gradient is defined as

$$g_{\text{Reptile}} = (\phi_0 - \phi_2)/\alpha = L_0'(\phi_0) + L_1'(\phi_1) \tag{9}$$
$$= L_0'(\phi_0) + L_1'(\phi_0) - \alpha L_1''(\phi_0) L_0'(\phi_0) + O(\alpha^2) \tag{10}$$

The MAML gradient is as follows

$$g_{\text{MAML}} = \frac{\partial}{\partial \phi_0} L_1(\phi_1) = \frac{\partial \phi_1}{\partial \phi_0} L_1'(\phi_1) = (I - \alpha L_0''(\phi_0)) L_1'(\phi_1) \tag{11}$$

$$= (I - \alpha L_0''(\phi_0))\big(L_1'(\phi_0) - \alpha L_1''(\phi_0) L_0'(\phi_0)\big) + O(\alpha^2) \tag{12}$$

$$= L_1'(\phi_0) - \alpha L_1''(\phi_0) L_0'(\phi_0) - \alpha L_0''(\phi_0) L_1'(\phi_0) + O(\alpha^2) \tag{13}$$

In the paper introducing MAML, Finn et al. described an algorithm they called first-order MAML, which we will abbreviate as FOMAML. In FOMAML, the gradient of the first update, $L_0'(\phi_0)$, is treated as a constant. Thus, in Equation (11), $\frac{\partial \phi_1}{\partial \phi_0}$ is replaced by $\frac{\partial}{\partial \phi_0}(\phi_0 - \text{SG}(L_0'(\phi_0))) = I$, where SG denotes "stop gradient", and therefore $g_{\text{FOMAML}} = L_1'(\phi_1)$. Hence, FOMAML corresponds to performing two steps of SGD, and taking the second step to be the gradient of the metalearning objective. Recalling Equation (8), we get

$$g_{\text{FOMAML}} = L_1'(\phi_1) = L_1'(\phi_0) - \alpha L_1''(\phi_0) L_0'(\phi_0) + O(\alpha^2) \tag{14}$$

We are interested in the Reptile, MAML, and FOMAML gradients when we take the expectation over task and minibatch sampling. In the equations below $\mathbb{E}_{\tau,0,1}[\dots]$ means that we are taking the expectation over the task τ and the two minibatches defining L_0 and L_1, respectively. We can see that there are only two kinds of terms in the Reptile and MAML gradient expressions:

1. $\text{AvgGrad} = \mathbb{E}_{\tau,0}[L_0'(\phi)]$, the gradient of expected loss. $(-\text{AvgGrad})$ is the direction that brings ϕ towards the minimum of the "joint training" problem; the expected loss over tasks.
2. $\text{AvgGradInner} = \mathbb{E}_{\tau,0,1}[L_0''(\phi) L_1'(\phi)] = \frac{1}{2}\mathbb{E}_{\tau,0,1}[L_0''(\phi) L_1'(\phi) + L_1''(\phi) L_0'(\phi)]$
   $= \frac{1}{2}\mathbb{E}_{\tau,0,1}\left[\frac{\partial}{\partial \phi}(L_0'(\phi) \cdot L_1'(\phi))\right]$. 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.

$$\mathbb{E}[g_{\text{Reptile}}] = 2\text{AvgGrad} - \alpha \cdot (\text{AvgGradInner}) + O(\alpha^2) \tag{15}$$

$$\mathbb{E}[g_{\text{MAML}}] = \text{AvgGrad} - 2\alpha \cdot (\text{AvgGradInner}) + O(\alpha^2) \tag{16}$$

$$\mathbb{E}[g_{\text{FOMAML}}] = \text{AvgGrad} - \alpha \cdot (\text{AvgGradInner}) + O(\alpha^2) \tag{17}$$

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.

4.2 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.

Let ϕ denote the network initialization, and $W = \phi + \Delta\phi$ denote the network weights after performing some sort of update. Let $\mathcal{W}_\tau^*$ denote the set of optimal network weights for task τ. We want to find ϕ such that the distance $D(\phi, \mathcal{W}_\tau^*)$ is small for all tasks.

$$\underset{\phi}{\text{minimize}}\, \mathbb{E}_\tau\left[\tfrac{1}{2} D(\phi, \mathcal{W}_\tau^*)^2\right] \tag{18}$$

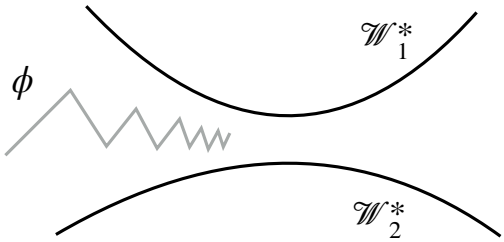


Figure 2: The above illustration shows the sequence of iterates obtained by moving alternately towards two optimal solution manifolds $\mathcal{W}_1^*$ and $\mathcal{W}_2^*$ and converging to the point that minimizes the average squared distance. One might object to this picture on the grounds that we 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.

Reptile can be seen as SGD on that objective. The gradient of the squared Euclidean distance between a point ϕ and a set S is the vector $2(\phi - p)$, where p is the closest point in S to ϕ. Thus,

$$\nabla_\phi \mathbb{E}_\tau \left[\tfrac{1}{2} D(\phi, \mathcal{W}_\tau^*)^2\right] = \mathbb{E}_\tau \left[\tfrac{1}{2} \nabla_\phi D(\phi, \mathcal{W}_\tau^*)^2\right] \tag{19}$$

$$= \mathbb{E}_\tau \left[\phi - W_\tau^*(\phi)\right], \text{where } W_\tau^*(\phi) = \operatorname*{arg\,min}_{W \in \mathcal{W}_\tau^*} D(W, \phi) \tag{20}$$

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

$$\phi \leftarrow \phi - \epsilon \nabla_\phi \tfrac{1}{2} D(\phi, \mathcal{W}^*)^2 \tag{21}$$

$$= \phi - \epsilon(W_\tau^*(\phi) - \phi) \tag{22}$$

$$= (1 - \epsilon)\phi + \epsilon W_\tau^*(\phi). \tag{23}$$

In practice, we can't exactly compute $W_\tau^*(\phi)$, 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 $\text{SGD}(L_\tau, \phi, k)$, i.e., k steps of gradient descent starting with ϕ.

5 Experiments

5.1 Few-Shot Classification

We evaluate our method on two popular few-shot classification tasks: Omniglot [LSGT11] and Mini-ImageNet [RL17]. 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 26^{th} 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. 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. [FAL17] use transduction for evaluating MAML.

For our experiments, we used the same CNN architectures and data preprocessing as Finn et al. [FAL17]. We used the Adam optimizer [KB15] with $\beta_1 = 0$ throughout our experiments. We chose $\beta_1 = 0$ because we found that momentum reduced performance across the board (matching the intuition that momentum is harmful when the optimization objective is re-sampled every few iterations). 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.

We found that we could improve performance by using meta-batches (Algorithm 2), similar to Finn et al. [FAL17]. With meta-batches, we trained separately on several different tasks and averaged together the resulting update directions. In these experiments, we ran each task in the meta-batch sequentially. (The order matters because of Adam, which we did not reset after every task in the meta-batch).

The resulting performance on Omniglot and Mini-Imagenet are shown in Tables 1 and 2 below.

Algorithm	1-shot 5-way	5-shot 5-way
MAML + Transduction	$48.70 \pm 1.84\%$	$63.11 \pm 0.92\%$
1^{st}-order MAML + Transduction	$48.07 \pm 1.75\%$	$63.15 \pm 0.91\%$
Reptile	$45.79 \pm 0.44\%$	$61.98 \pm 0.69\%$
Reptile + Transduction	$48.21 \pm 0.69\%$	$66.00 \pm 0.62\%$

Table 1: Results on Mini-ImageNet

Algorithm	1-shot 5-way	5-shot 5-way	1-shot 20-way	5-shot 20-way
MAML + Transduction	$98.7 \pm 0.4\%$	$99.9 \pm 0.1\%$	$95.8 \pm 0.3\%$	$98.9 \pm 0.2\%$
1^{st}-order MAML + Transduction	$98.3 \pm 0.5\%$	$99.2 \pm 0.2\%$	$89.4 \pm 0.5\%$	$97.9 \pm 0.1\%$
Reptile	$95.32 \pm 0.05\%$	$98.87 \pm 0.02\%$	$88.27 \pm 0.30\%$	$97.07 \pm 0.12\%$
Reptile + Transduction	$97.97 \pm 0.08\%$	$99.47 \pm 0.04\%$	$89.36 \pm 0.20\%$	$97.47 \pm 0.10\%$

Table 2: Results on Omniglot

We optimized most of our hyper-parameters using CMA-ES [Han06]. 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.

We found that, most of the time, the hyper-parameters found for non-transductive Reptile were also good for transductive Reptile. However, for 1-shot 5-way classification, we were able to achieve better performance using hyperparameters tuned for transduction.

Table 3: Hyper-parameters for Omniglot.

Parameter	1-shot 5-way	1-shot 20-way	5-shot 5-way	5-shot 20-way
Adam learning rate	4.4×10^{-4}	4.4×10^{-4}	1.5×10^{-3}	4.6×10^{-4}
Inner batch size	5	15	10	20
Inner iterations	12	12	5	12
Training shots	12	9	10	12
Outer step size	1.0	1.0	0.7	1.0
Outer iterations	200K	200K	100K	200K
Meta-batch size	3	3	5	3
Eval. inner batch size	5	10	6	10
Eval. inner iterations	86	97	100	97

Table 4: Hyper-parameters for Mini-ImageNet.

Parameter	1-shot 5-way	5-shot 5-way
Adam learning rate	1.2×10^{-3}	2.2×10^{-4}
Inner batch size	3	10
Inner iterations	19	8
Training shots	12	15
Outer step size	0.235	1.0
Outer iterations	200K	200K
Meta-batch size	2	5
Eval. inner batch size	3	15
Eval. inner iterations	55	88

Table 5: Hyper-parameters for transductive 1-shot 5-way.

Parameter	Omniglot	Mini-ImageNet
Adam learning rate	1×10^{-3}	1×10^{-3}
Inner batch size	25	5
Inner iterations	3	15
Training shots	15	15
Outer step size	1	1
Outer iterations	100K	100K
Meta-batch size	10	10
Eval. inner batch size	5	5
Eval. inner iterations	5	10

5.2 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.

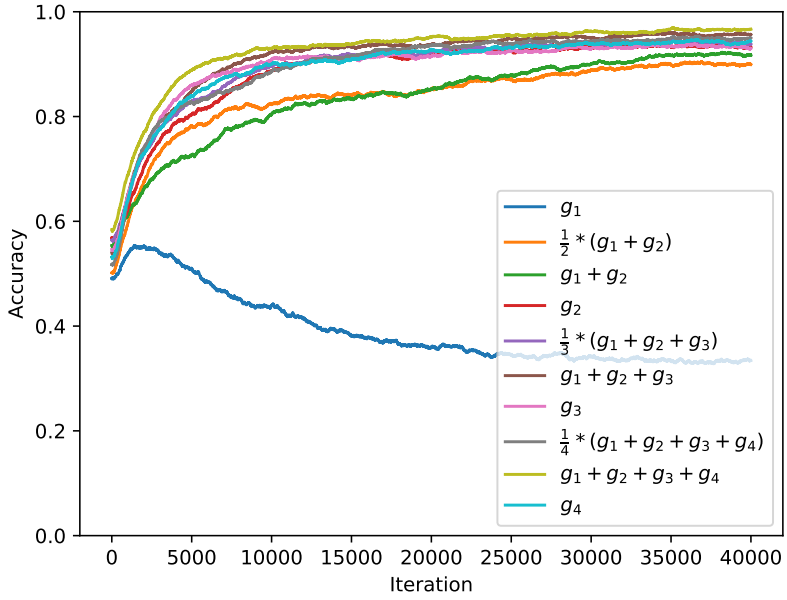


Figure 3: Different inner-loop gradient combinations on 5-shot 5-way Omniglot.

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. 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 5.1.

Table 6: Hyper-parameters for comparison between different inner-loop gradient combinations. 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

6 Discussion

In metalearning problems, we assume access to a training set of tasks, which we use to train a fast learner. We described a surprisingly simple approach for metalearning, which works by repeatedly optimizing on a single task, and moving the parameter vector towards the parameters learned on that task. This algorithm performs similarly to MAML [FAL17], while being significantly simpler to implement.

We gave two theoretical explanations for why Reptile works. First, by approximating the update with a Taylor series, we showed that the key leading-order term matches the gradient from MAML [FAL17]. 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 metalearning setting, the Taylor series analysis in Section 4.1 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 [ZDGD14]) works well—SGD automatically gives us an initialization that generalizes well to similar tasks. This hypothesis would suggest that *joint training plus fine tuning* will continue to be a strong baseline for metalearning in various machine learning problems.

References

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, IEEE, 2009, pp. 248–255.

[DSC+16] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, "RL^2: Fast reinforcement learning via slow reinforcement learning," *ArXiv preprint arXiv:1611.02779*, 2016.

[FAL17] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," *ArXiv preprint arXiv:1703.03400*, 2017.

[FL17] C. Finn and S. Levine, "Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm," *ArXiv preprint arXiv:1710.11622*, 2017.

[Han06] N. Hansen, "The cma evolution strategy: A comparing review," in *Towards a new evolutionary computation*, Springer, 2006, pp. 75–102.

[HYC01] S. Hochreiter, A. S. Younger, and P. R. Conwell, "Learning to learn using gradient descent," in *International Conference on Artificial Neural Networks*, Springer, 2001, pp. 87–94.

[KB15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015.

[LSGT11] B. M. Lake, R. Salakhutdinov, J. Gross, and J. B. Tenenbaum, "One shot learning of simple visual concepts," in *Conference of the Cognitive Science Society (CogSci)*, 2011.

[LST15] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015.

[RL17] S. Ravi and H. Larochelle, "Optimization as a model for few-shot learning," in *International Conference on Learning Representations (ICLR)*, 2017.

[STT12] R. Salakhutdinov, J. Tenenbaum, and A. Torralba, "One-shot learning with a hierarchical nonparametric bayesian model," in *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, 2012, pp. 195–206.

[SBB+16] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap, "Meta-learning with memory-augmented neural networks," in *International conference on machine learning*, 2016, pp. 1842–1850.

[Sch09] L. A. Schmidt, "Meaning and compositionality as statistical induction of categories and constraints," PhD thesis, Massachusetts Institute of Technology, 2009.

[WSH+15] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *ArXiv preprint arXiv:1511.06581*, 2015.

[ZDGD14] N. Zhang, J. Donahue, R. Girshick, and T. Darrell, "Part-based r-cnns for fine-grained category detection," in *European conference on computer vision*, Springer, 2014, pp. 834–849.