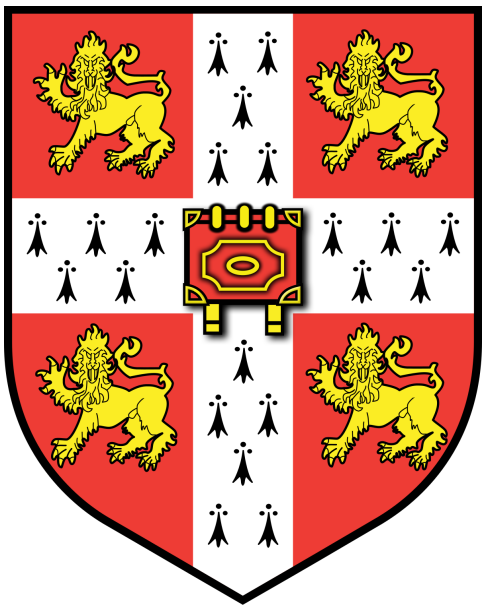


First-Order Approximations for Efficient Meta-Learning: A Review of Reptile

Authors: Adrian Black, Alistair McLeay, Benedict King
Word Count: 4,703
Submitted: March 30, 2022

Contents

1 Introduction

The purpose of this investigation was to explore algorithms for meta-learning, and compare the first-order (and therefore computationally efficient) Reptile algorithm proposed by [8] to its predecessors, Model Agnostic Meta Learning (MAML) and First-Order MAML (FOMAML). We chose the Reptile paper to investigate due to a shared interest in meta-learning. We believe that there is huge value in developing data-efficient machine learning techniques. In our assessment, deeply understanding Reptile and its related approaches was likely to provide a strong foundation to explore this important area further.

In Section 2 we cover the motivations for building meta-learning models and briefly review the related methods that were published before the the Reptile algorithm. This is then followed by discussions of how the Reptile algorithm works and what new contributions it brings to the table.

In Section 4, as a first step in investigating the variations of the Reptile algorithm discussed in [8] we reproduced the sinusoidal function approximation task from the Reptile paper. In this initial replication exercise we applied simple gradient updates for both meta and inner-loop iterations in order to stay consistent with the original paper. However, we suspected that we could improve learning by using more complex adaptive gradient optimisers. Therefore, we extend and discuss sinusoidal regression using Adaptive Moment Estimation (ADAM) [3] and Root Mean Square Propagation (RMSProp) [13].

In Section 5 we extend the 1D sinusoidal regression task to explore how the algorithm handles more complex regression tasks. Specifically, we investigate increasing the meta-batch size from 1 to 2 by adding the additional task of fitting a basic Fourier series. We then investigate scaling the original task up from 1D to 2D and increasing the ratio of the number of points to predict to the number of sampled points by a factor of 200. We utilise both SGD and ADAM optimisers and present a compelling visualisation of Reptile learning as it proceeds through the training process.

Finally, in Section 6 we apply Reptile to few-shot image classification. We reproduce key results from the Reptile paper [8] on both the Omniglot and Mini-ImageNet datasets. We also explore adjusting the inner-loop optimiser and introduce a pretraining protocol to improve the efficiency of the Reptile algorithm when it comes to learning general image features.

2 An Overview of Meta-Learning

Over the past decade, deep neural networks have become increasingly capable at approximating functions via supervised learning, with applications including classification and regression tasks. However, these highly complex networks often contain a huge number of parameters and require large amounts of data and prolonged training. These models also have poor generalisation between tasks, extending the challenges of using them to solve important problems with limited data or limited computational resources.

Meta-learning aims to bridge this gap, and allow neural networks to quickly (i.e. in few training iterations) adapt to contextually similar, but not previously encountered tasks. Moreover, the aim is to achieve this without needing a large amount of data for the new task. In order to attain this, the learning methodology must be adapted to allow the model to encounter and train on a diverse range of problems without overwriting training on previous tasks each time. An early suggestion for how to do this was with Transfer Learning[1] [12].

Transfer learning attempts to improve the generalisation power of a neural network by first bulk

[1] A more thorough description and comparison of meta-learning methods can be found in Appendix A.1.

training on a dataset in one context, and then fine-tuning on a dataset in another (for example bulk training on images of cars and fine-tuning on images of trucks could allow the network to be able to effectively classify both cars and trucks). The mathematical update rules for this can be found in Appendix A.1. The shortcomings of transfer learning are that, firstly, the different tasks have to be very similar to each other to achieve strong performance, and secondly, the fine-tuning process is still data inefficient, and therefore fine-tuning on multiple tasks requires a lot of training and degrades performance on prior tasks. Despite this, the simple ideas captured by transfer learning form the foundations for the more advance meta-learning algorithms.

Meta-learning algorithms can broadly be categorised into 3 classes: model-based, metric-based, and optimisation-based [10]. MAML, FOMAML and Reptile all fall under the class of optimisation based meta-learners, which seek to adjust the optimisation algorithm to pre-initialise network parameters before each inner-loop task in a way that allows the network to train quickly on it. They have the advantage of being independent of the model being used for predictions and are only embedded into the optimisation step, thus giving flexibility to the range of tasks they can be applied to (hence the 'model-agnostic' title in MAML and FOMAML). The base of all three algorithms is MAML, which involves two loops, the meta-loop and the inner-loop. The inner-loop performs rapid-learning, and trains copies of parameters on a batch of different tasks over a small number of iterations using the pre-initialised parameters from the previous meta-loop iteration. The meta-loop then updates the model parameters by stepping in the direction that minimises the gradient of the sum of losses over each task in the batch [7] - details of this are available in Appendix A.1.

Whilst strong at generalisation, a reported issue with MAML was the quadratic cost of calculating the Hessian of the batch loss function with respect to the parameters at every inner-loop and meta-loop iteration [11] (see Appendix A.1 for mathematical justification). For a network with d parameters, n meta-loop iterations, and k rapid-learning/inner-loop iterations, MAML's total complexity is $\mathcal{O}(d^2nk)$. There is however a significant improvement for this in the original code provided by [7]. For a neural network (using backpropagation), the Hessian dot product approximation is used to scale down complexity to $\mathcal{O}(dnk)$, though this does come at the expense of precision of the Hessian's values [1]. First-Order MAML (FOMAML) was also suggested by the original authors as a computationally faster (and not neural network specific) approximation for the gradient update that assumes the learning rate multiplied by the Hessian is minimal compared to the identity matrix (see Appendix A.1). It therefore only computes the gradient operator, again leading to $\mathcal{O}(dnk)$ complexity [11], a significant speed-up for large models.

In certain cases FOMAML's performance was observed to be as good as MAML's [8], which suggests that a first order approximation is a suitable approach to take when computational complexity is an issue. Reptile builds from this hypothesis, and provides a different first-order approximation to achieve similar results to MAML but with lower computational overhead.

3 The Reptile Algorithm

The pseudo-code below in Algorithm 1 outlines the process for Reptile learning, and demonstrates the difference between it and MAML, where there is a need to explicitly compute (or approximate using Hessian dot product approximation) second order derivatives of the loss functions. That said, the benefit of Reptile over FOMAML (which also does not calculate the second derivatives) is that Reptile handles higher order information without explicitly calculating the Hessian. The Reptile gradient can be expanded as the sum of Taylor expansions of each inner-loop gradient about the initial parameters, which includes second-order approximations (see Appendix A.2 for more detail). Therefore, Reptile includes approximate second-order derivative information despite being a first-order algorithm.

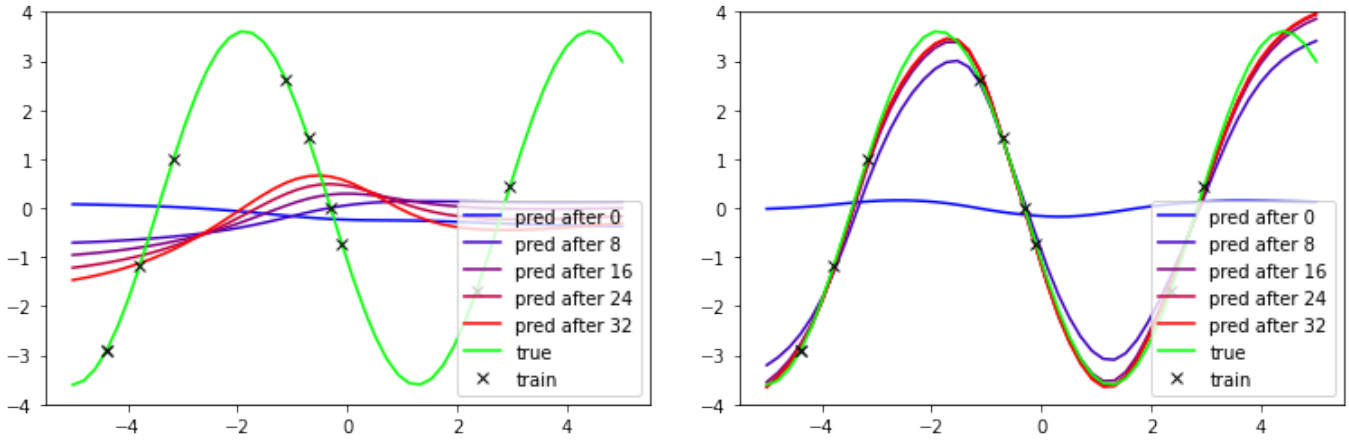
Algorithm 1 The Reptile algorithm (serial version) [8]. n is the meta-learning iteration, and goes up to orders of magnitude 10^4. k is the rapid learning iteration and typically goes up to order of magnitude 10^1.

Initialise parameters $\phi^{(0)}$
for iteration n = 1,2,3,... in meta-loop **do**
  Sample task τ corresponding to task-specific loss $\mathcal{L}_\tau$ from task distribution
  Compute gradient update: $\tilde{\phi} = f(\phi; k, \mathcal{L}_\tau)$ where $f(\cdot; k, \mathcal{L}_\tau)$ is k iterations of an optimiser like SGD
  Update $\phi^{(n)} = \phi^{(n-1)} + \frac{1}{\eta}(\tilde{\phi} - \phi^{(n-1)})$ for η the meta-loop learning rate
end for

4 Sinusoidal Regression

4.1 Replicating Basic Results

We initially replicated the results from Section 4 in the Reptile paper [8] using PyTorch. The results of running the algorithm for 10K meta-iterations, which takes roughly 20 seconds on an Intel i9 CPU, are shown in Figure 1. We see that Reptile is able to find a very accurate (but imperfect) fit after 10K meta-iterations if 32 inner SGD steps are used ($k = 32$ in the algorithm).



(a) Before training.

(b) After 10K meta-iters.

Figure 1: 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 $1 \to 64 \to 64 \to 1$. The amplitude varies from 0 to 5 and the phase varies from 0 to 2π. 10K meta-iterations were used.

4.2 Comparing Optimisers

In this section, we replicate the sinusoid regression results from the original paper [8], as shown above. In these experiments, a 1D sine wave was generated with phase and amplitude that varied every meta-iteration. Thus, each meta-iteration effectively regressed on a unique, randomly generated function. In the inner-loop iteration, we then sample batches of input points from the domain and train k (set to 32 for these experiments) rapid-learning steps on each generated function.

In the original paper, each inner-loop update consisted of k stochastic gradient steps and we replicate this experiment in Figure 2 (left). However, adaptive gradient optimisers like ADAM and RMSProp are broadly effective across deep learning in locating optimal loss function minima [6]. Therefore, we also explored adopting ADAM and RMSProp for the inner-loop optimiser (instead of SGD) as shown in Figures 2 and 3 (middle and right)[2]. In all sub-plots of both figures, the different coloured lines represent a different number of meta-iterations taken prior to fine-tuning/testing[3]. With additional meta-iterations, the model has been trained on more randomly generated unique functions. We thus expect that after many meta-iterations the model will have found a pre-initialisation that allows quick convergence to a low loss on the test sine function.

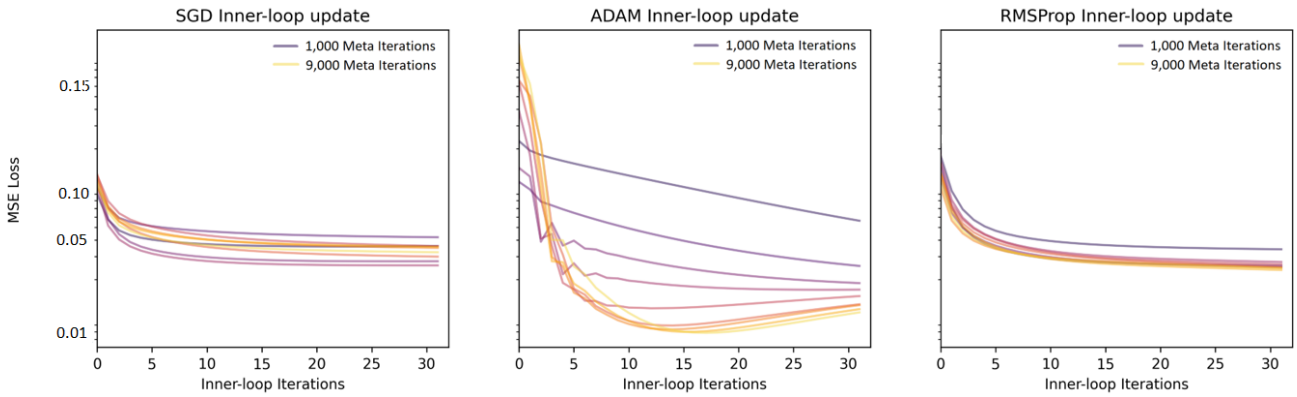


Figure 2: A comparison of three different gradient optimisers on the same training and evaluation tasks. The task was to approximate a sinusoidal function with the different colour lines in each plot representing different numbers of meta-learning iterations to pre-initialise weights before training on the unseen evaluation task (with 32 rapid-learning steps). The darkest purple curve represents 1,000 iterations of meta-learning, and the yellow curve 9,000 iterations.

In Figure 2, as suspected, RMSProp presents an improvement relative to SGD. This is evident by the fewer inner-loop iterations necessary to reach the minimum loss after training for a suitable number of meta-iterations. Moreover, the RMSProp loss curves cluster closer together, suggesting that RMSProp makes Reptile more robust to the pre-initialised weights. Next, note in the leftmost SGD plot that the loss is higher after many meta-iterations. This is likely a consequence of the model overfitting after many meta-iterations, as although functions are randomly generated (by varying the phase and amplitude), they are all simple 1D sine waves.

Perhaps the most interesting thing to observe is the behaviour of ADAM compared to SGD and RMSProp. The difference between these optimisers is principally due to the use of momentum[4], and this provides an intuitive explanation into why the results are different. The update step for a parameter θ with ADAM is as follows [3]:

$$\begin{aligned} \theta_{t+1} &= \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{\nu}_t + \epsilon}} \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ \nu_t &= \beta_2 \nu_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{1}$$

where $\hat{m}_t$ and $\hat{\nu}_t$ are the normalised 1^{st} and 2^{nd} moment parameters $\hat{m}_t = m_t/(1-\beta_1^t)$ and $\hat{\nu}_t = \nu_t/(1-\beta_2^t)$. Therefore, if β_1 is high, there is large information leakage from previous gradient updates

[2]The ρ parameter of RMSProp is set to 0.9, as is often used for RMSProp [6]. The β_1 ADAM parameter is set to 0.1 and the β_2 ADAM parameter is set to 0.99.

[3]The color gradient extends from purple (early in training) to yellow (late in training)

[4]RMSProp actively impedes movement perpendicular to the direction of the minimum to reduce oscillatory behaviour, but does not use momentum to accelerate towards it like ADAM does [6].

to the current update (effectively momentum). This is problematic because Reptile fundamentally depends on sequential steps coming from different mini-batches to appropriately maximise within task generalization [8]. However, momentum allows (undesirably in the case of Reptile) a mini-batch to influence the following steps. Nonetheless, the adverse impact of momentum is softened by setting ADAM's β_1 parameter close to 0[5] and we observe that the ADAM optimiser ultimately locates a lower test MSE loss than the other optimisers.

Although ADAM locates a lower MSE loss than the other optimisers in Figure 2, it is notably slower to converge. An issue we identified is that ADAM's momentum carries gradient information across tasks. Thus, when we sample a new task (a randomly generated function), gradients are still initially moving towards the minimum of the prior task, which introduces instability into the learning process. A possible resolution to this is to reset ADAM's parameters at the start of every meta-iteration and thus ensure moment histories are unique to each randomly generated function. In Figure 3, we present this method on a different set of test functions, and observe that the ADAM optimiser exhibits more stable convergence.

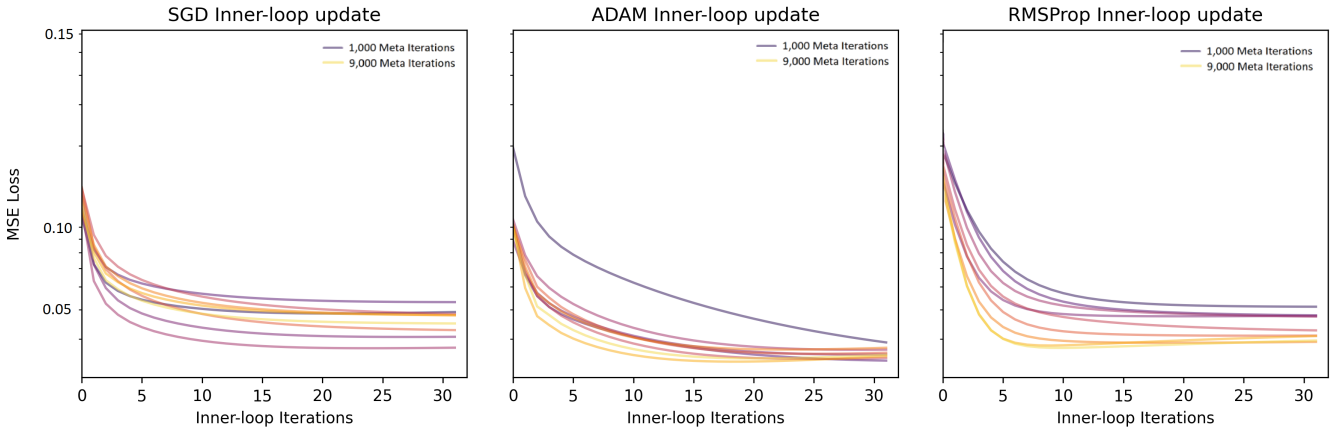


Figure 3: A similar comparison plot to Figure 2 but on a new randomly generated set of test functions and where moment histories are reset every meta-iteration. The Adam optimiser (middle) shows more stable convergence.

Whilst the convergence is more stable under this method (and remains so for higher β_1 values), the downside to resetting ADAM's parameters here is that the minimum MSE reached in Figure 3 is higher than in Figure 2. We hypothesise that this is due to the following: ADAM still has the adverse effect of leaking information across minibatches of a given task, but now has reduced momentum. This means that as k tends to 1, ADAM ceases to contain history information and becomes increasingly similar to SGD. This can be seen from the ADAM loss in Figure 3 (middle) being visually a lot more similar to SGD than Figure 2 (middle). Despite this, for greater than 1,000 meta-iterations a better minimum is still reached by ADAM for $k > 1$ than SGD and RMSProp, and the more stable convergence makes this approach appealing.

[5]The original paper [8] sets $\beta_1 = 0$ to address this. We set our value to $\beta_1 = 0.1$ to better demonstrate the impact that momentum has on training.

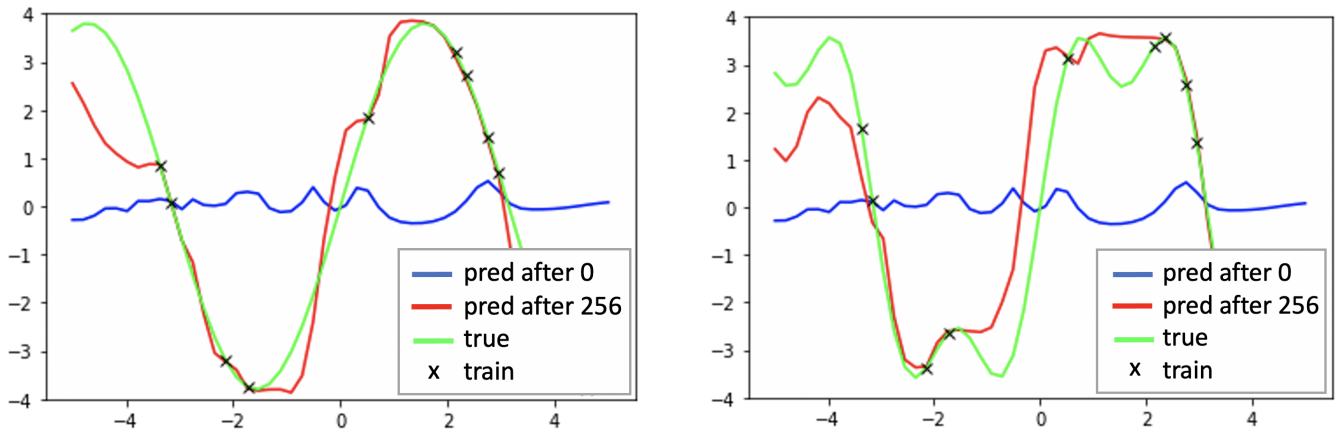
5 Extensions to Function Approximation

After implementing the basic 1D sinusoidal regression results in the Reptile paper (described in Section 4.1), we extended the implementation to two significantly more challenging regression tasks to investigate the limits of Reptile.

5.1 Extending Few-Shot Regression to a mixture of 1D function families

The regression problem outlined in the Reptile paper uses a meta-batch size of one (where the meta-batch consists of the single task of predicting a sine wave using only 10 randomly sampled points). It is interesting to consider the scenario where the number of meta-batches are increased to two by adding an additional function as a new meta-batch (a new task). This provides a more realistic setup for meta-learning applications, where we are trying to learn multiple distinct tasks.

The results of doing this using a basic Fourier series, $y = a\sin(x + p) + 3a\sin((x/3) + p)$, are shown in Figure 4. We see that the model learns an initialisation (in blue) that captures key differences between the two functions after 256 inner steps, despite the size of the network not being increased. However, it is also clear the model is really struggling to model the complexity of the different functions accurately. Using ADAM and a larger network would likely address this.



(a) Sine wave: $y = a\sin(x + p)$.

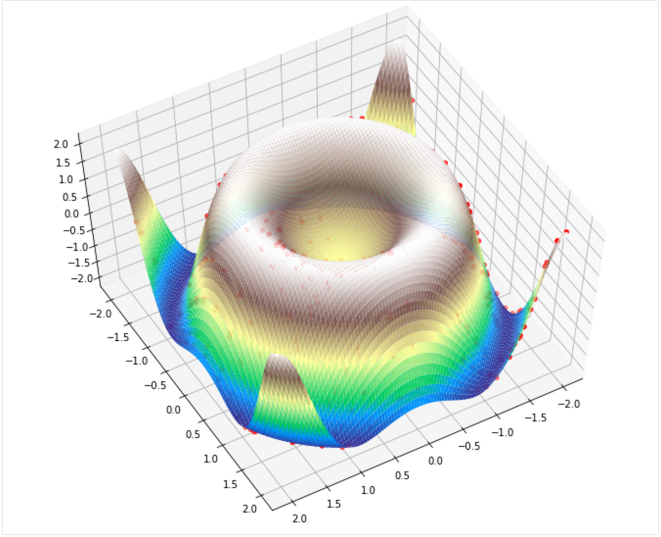
(b) Fourier series: $y = a\sin(x+p)+3a\sin((x/3)+p)$.

Figure 4: Reptile on an extended version of the toy few-shot regression problem, where we train on 10 sampled points of a sine wave, performing 256 gradient steps on an MLP with layers $1 \to 64 \to 64 \to 1$, and where the function is randomly sampled from either a sine wave or a fourier series. For both functions a varies from 0 to 5 and p varies from 0 to 2π. 10K meta-iterations were used.

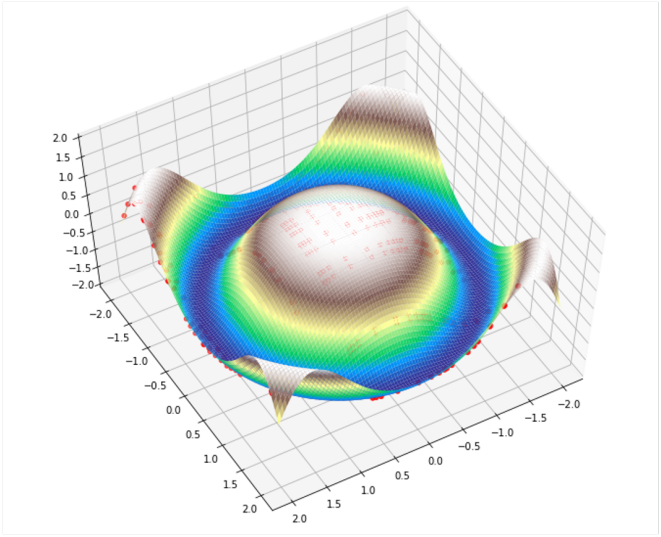
5.2 Extending Few-Shot Regression to Two-Dimensions.

In this experiment we extended the 1D few-shot regression problem into a 2D version using the function $z = a\sin((x^2 + y^2) + p)$. Two sample functions with different phases and amplitudes are shown in Figure 5 to illustrate the new targets of the regression task.

This experiment was designed to test how Reptile handled scaling up the complexity in a different way. Instead of sampling from multiple functions (multiple tasks) it is again using just one task as it does in the 1D problem outlined in the paper. However, unlike the 1D problem the number of points to predict have increased by a factor of 200 (50 to 10K), and the number of sampled points the algorithm has access to has only increased by a factor of 2 (10 to 20). We found that even with a



(a) Sample One.



(b) Sample Two.

Figure 5: Target functions for the 2D version of the 1D few-shot regression sine wave task in the Reptile paper.

significantly larger neural network ($2 \to 128 \to 512 \to 512 \to 128 \to 1$), SGD was unable to achieve a meaningful fit. However, when using ADAM with the same network size, it was able to find a fairly good initialisation in 2900 meta-iterations, as shown in Figure 6.

While a fairly good initialisation is found in Figure 6, we see in Figure 7 that learning is very unstable, even with the momentum set close to 0. This instability is consistent with the findings in Section 3 as the momentum was not being reset after each meta-iteration in this experiment. We expect that the instability can be substantially improved if the momentum is reset after each meta-iteration.

Figure 8 shows the results of this experiment using SGD as the optimiser in a simplified setup, where the phase varies from 0 to 2π and the amplitude is fixed at 2.0. These results are interesting because they provide a clear visualisation of Reptile learning over meta-iterations. After 1000 meta-iterations an initialisation is learned that enables the 'bowl' shape of the function to be modelled. After 2000 epochs the learned initialisation enables the 'tails' of the function to also be modelled.

It would be interesting to replicate these experiments using MAML, FOMAML [7], and other meta-learning approaches to see how they perform compared to Reptile.

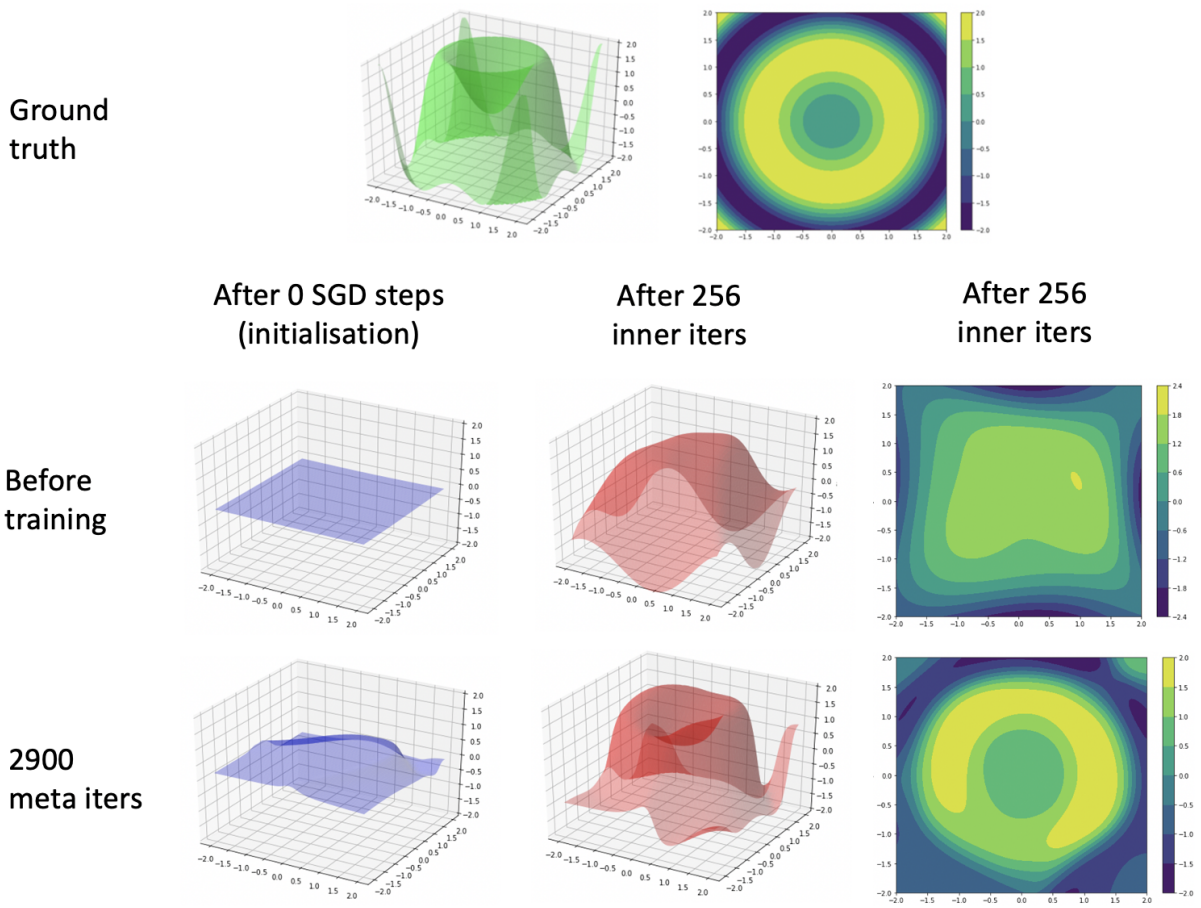


Figure 6: 2D version of the 1D few-shot regression sine wave example in the Reptile paper, using the function $z = a\sin((x^2+y^2)+p)$. The input space is scaled from 50 to 10K points, while the minibatch size is scaled from 10 to 20 (and the number of minibatches is unchanged at 5). A new function is sampled every meta-iteration as it is in the Reptile paper. When sampling the new function, the amplitude is varied from at 0 to 2.0 and the phase is varied from 0 to 2π. An ADAM optimiser is used with $\beta_1 = 0.01$ and $\beta_2 = 0.999$.

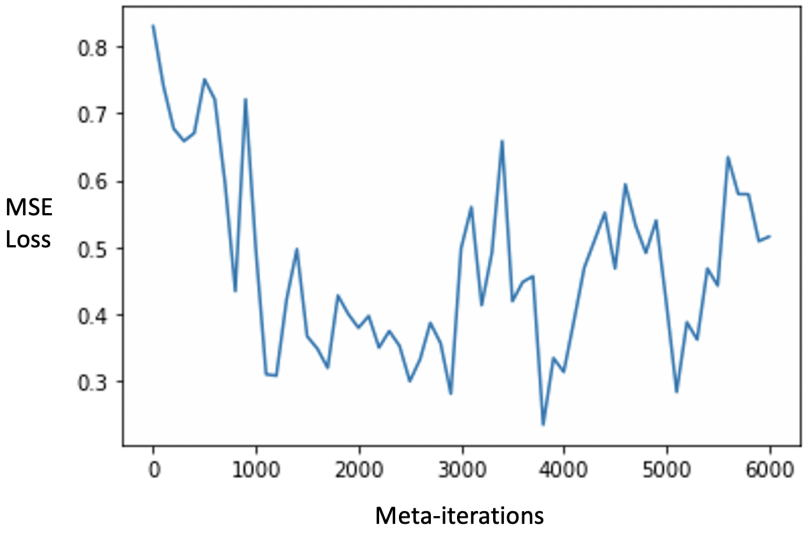


Figure 7: MSE Loss for Reptile when ADAM is applied to the 2D regression task illustrated in Figure 6.

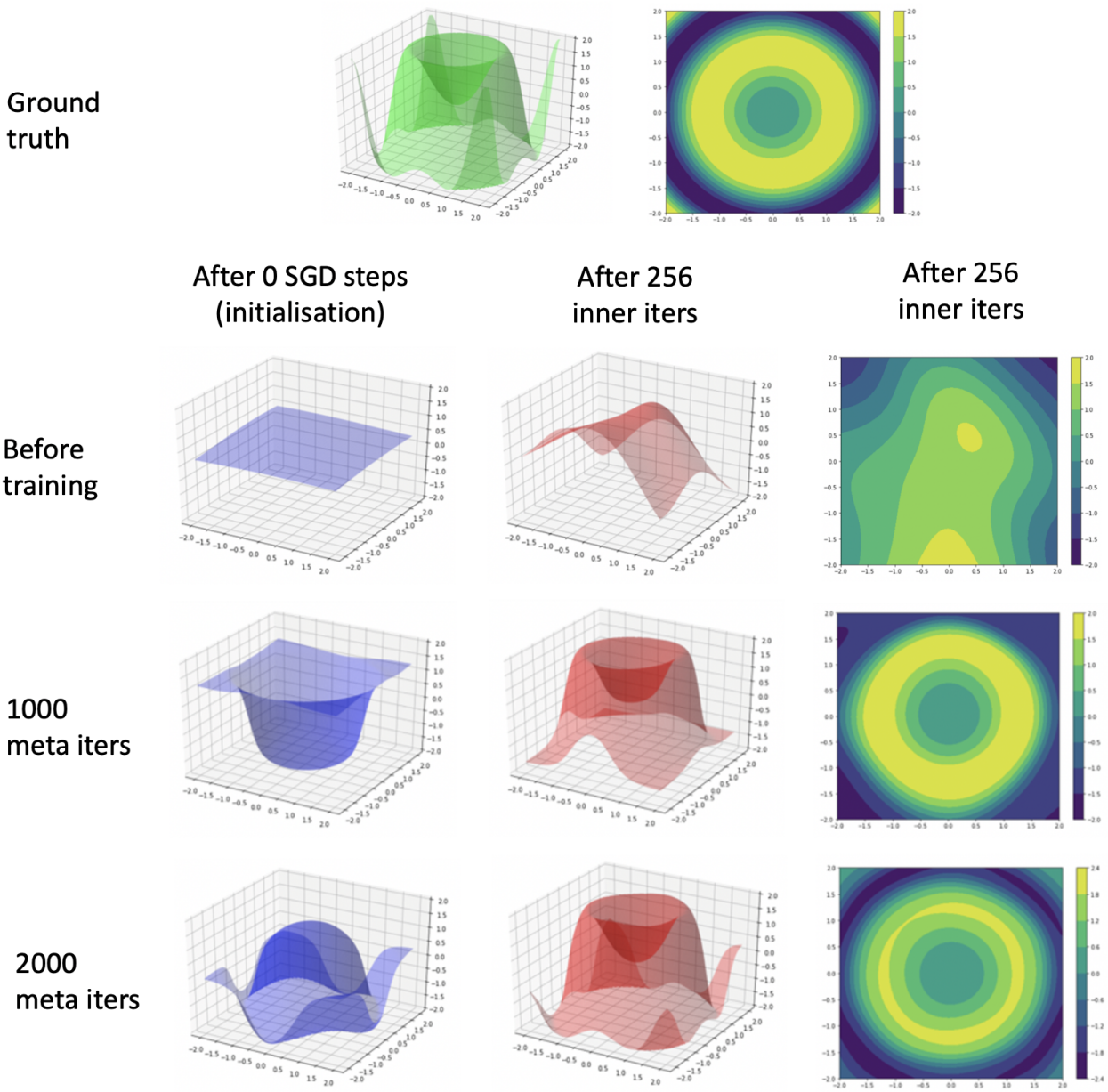


Figure 8: 2D version of the 1D few-shot regression sine wave example in the Reptile paper, using the function $z = \sin(x^2 + y^2)$. The input space is scaled from 50 to 10K points, while the minibatch size is scaled from 10 to 20 (and the number of minibatches is unchanged at 5). A new function is sampled every meta-iteration as it is in the Reptile paper. When sampling the new function, the amplitude is fixed at 2.0 and the phase is varied from 0 to 2π. An SGD optimiser is used.

6 Image Classification

We proceed to apply Reptile to the domain of few-shot image classification. Few shot-image classification involves fast learning of an N-way image classification task. Specifically, we fine-tune a model with K images from each of N unseen classes. We then evaluate the model's performance classifying novel images taken from those N classes. This is framed as N-way, K-shot classification.

The Reptile paper [8], performs a wide variety of experiments on few-shot image classification, exploring many N-way, K-shot tasks. It investigates different inner-loop gradient combinations and the overlap between inner-loop batches in FOMAML. With our limited computational budget we decided to focus on replicating the key Reptile results by running limited N-way, K-shot tasks on two main datasets.

6.1 Datasets

We evaluate Reptile on two datasets commonly benchmarked for meta-learning and few-shot classification: Omniglot and Mini-ImageNet.

The Omniglot dataset consists of black and white hand drawn characters from world alphabets [2]. Specifically, it comprises 20 images of 1623 characters selected from 50 different alphabets. For an illustration of samples see Figure 9.

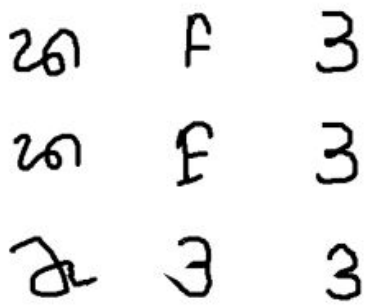


Figure 9: Sample of images from the Omniglot dataset. Three instances of a particular character are shown from the Balinese (left) Armenian (middle) and Cyrillic (right) alphabets.

Mini-ImageNet, the more challenging benchmark, subsets ImageNet to provide a similarly complex modeling challenge without demanding the resources to run the entire Imagenet dataset [5]. Specifically, ImageNet contains 600 colored images from each of 100 classes capturing diverse phenomena like 'nematodes' or 'scoreboards.' Interestingly, there appears to be inconsistency in the literature regarding the precise validation/test class split for Mini-ImageNet. The MAML paper uses a 12 class validation set and 24 class test set, while we follow Reptile's scheme with a 16 class validation set and 20 class test set. For an illustration of samples see Figure 10.

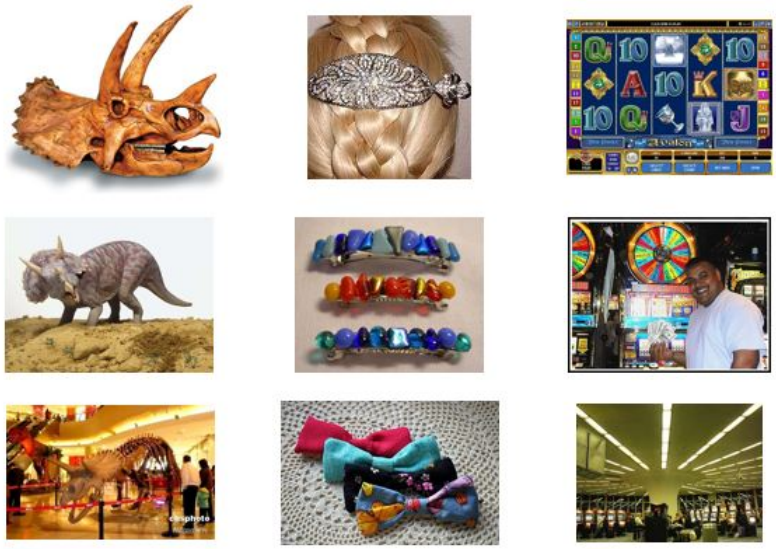


Figure 10: Sample of images from the Mini-ImageNet dataset. Three instances of images are shown for each of three classes: triceratops (left) hair slide (middle) and slot (right). The diversity of images presents a more complex modeling challenge than Omniglot.

6.2 Network Architecture

We employ the same convolutional neural network architecture used by the MAML and Reptile papers [7, 8]. The network comprises 4 blocks each consisting of a 3x3 convolutional layer, batch normalization, and a ReLU activation function. Downscaling is achieved through strided convolutions for Omniglot and a max-pooling layer for Mini-ImageNet. Note additionally that Omniglot uses 64 channel convolutions while Mini-ImageNet uses just 32 channel convolutions as a means to mitigate overfitting. Lastly, the final layer of the final block is fed into a softmax classifier.

6.3 Training Procedure

We closely follow the training protocol of the Reptile paper [8] to ensure a reliable replication. Specifically, we use SGD to update the meta loop and the ADAM [3] optimiser to update the inner loop. Moreover, we effectively drop momentum from ADAM by setting $\beta_1 = 0$. This is both empirically and theoretically motivated. Reptile's authors found momentum consistently reduced performance and also note that Reptile relies on sequential updates from distinct minibatches [8]. Dropping momentum thus helps minimise information leakage between subsequent updates. Finally, we also explore using SGD to update the inner loop though this was not reported in the original paper.

The code used to carry out the image classification was provided by the authors of the Reptile paper [8][6]. A few adaptations were necessary as the code was provided as-is and had not been updated since 2017. Principally, the code was written in the now discontinued TensorFlow V1 library and had to be made compatible with TensorFlow V2. Additionally, the data collection script was broken and needed to be reconfigured with a functional Mini-ImageNet repository[7]. Note all such revisions altered implementation details, not the underlying algorithm. However, in Section 6.5 we make more extensive modifications to the code to support a pre-training protocol.

[6]https://github.com/openai/supervised-reptile
[7]https://www.kaggle.com/c/minimagenet/data

In Tables 1 and 2 we specify the hyperparemeters used during training. They are essentially identical to those presented in the Reptile paper [8]. Note we only explore SGD updates for the Omniglot dataset and use a higher learning rate than with the ADAM optimiser. Lastly, we investigate 1-shot, 5-shot, and 10-shot classification for Mini-ImageNet. In line with the Reptile paper [8] we use a larger evaluation inner-batch size for 5-shot classification than 1-shot classification. As the Reptile paper does not report results for 10-shot classification, we choose to repeat the evaluation inner-batch size from 5-shot classification.

Note that all experiments were carried out in the transductive setting. That is, the model classifies the entire test-set batch at once and thus test examples share information due to batch normalization. The Reptile authors note that transduction empirically boosts performance and with our limited computational budget we focused on replicating the best results.

Lastly, we conducted all training runs using cloud-based GPUs provided through Microsoft Azure. Specifically, we used a machine with a single Nvidida Tesla K80 GPU.

Table 1: Reptile Hyperparameters for Omniglot

Parameter	Value
ADAM (inner) learning rate	.001
SGD (inner) learning rate	.003
Inner batch size	10
Inner iterations	5
Training shots	10
Meta step size	1
Meta iterations	100k
Meta-batch size	5
Eval. inner iterations	50
Eval. inner batch	5

Table 2: Reptile Hyperparameters for Mini-ImageNet

Parameter	1-shot	5-shot	10-shot
ADAM (inner) learning rate	.001	.001	.001
Inner batch size	10	10	10
Inner iterations	8	8	8
Training shots	15	15	15
Meta step size	1	1	1
Meta iterations	100k	100k	100k
Meta-batch size	5	5	5
Eval. inner iterations	50	50	50
Eval. inner batch	5	15	15

6.4 Results and Discussion

6.4.1 Omniglot

We first detail results for the simpler Omniglot dataset. In Table 3 we observe that on 1-shot, 5-way Omniglot classification our Reptile experiment achieves the same 97.7% accuracy as in the original paper [8]. We also note that substituting SGD for the inner-loop optimiser (instead of ADAM) reaches nearly the same performance after 100k iterations of training. However, by evaluating test

accuracy over the course of training (Figure 11), we see that SGD appears to undergo momentarily sub-optimal exploration around 25,000 iterations, although eventually it escapes this local minimum. Lastly, note that Reptile still slightly underperforms both MAML and FOMAML [7] on Omniglot. However, we will see the reverse is true with Mini-ImageNet.

Table 3: Results on Omniglot for 1-shot, 5-way classification.

Algorithm	Test Accuracy
Reptile	97.7%
Reptile [8]	97.7%
Reptile (SGD)	97.4%
MAML [7]	98.3%
FOMAML [7]	98.7%

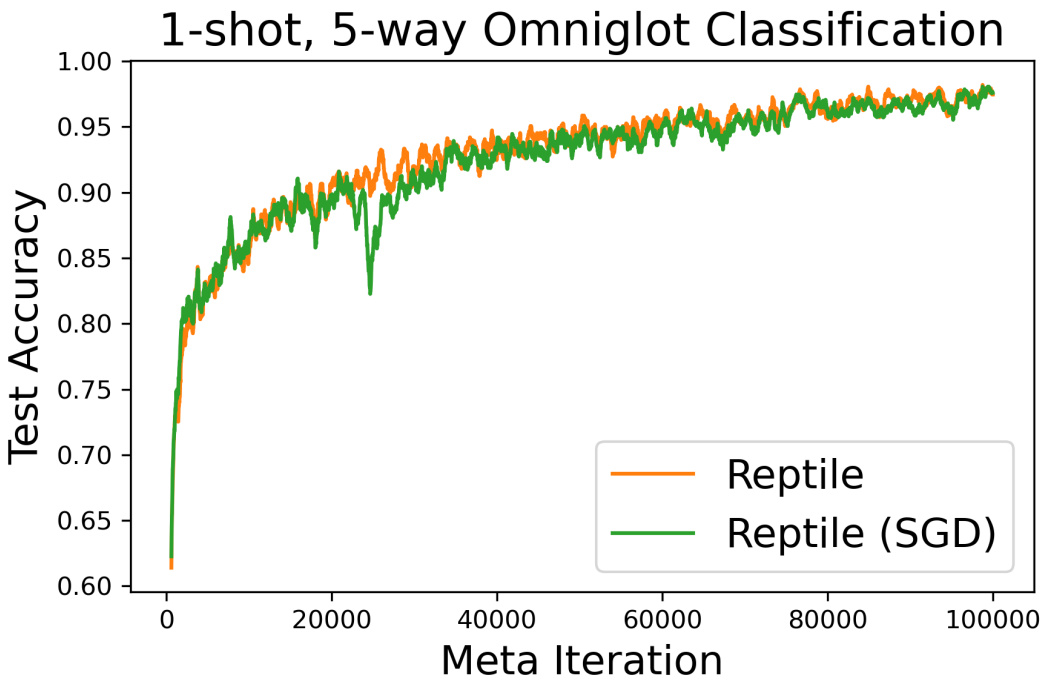


Figure 11: Test accuracy evaluated over the course of Reptile training (every 20 meta iterations). Adopting SGD (instead of ADAM) for the inner-loop leads to momentarily sub-optimal exploration but ultimately comparable performance

6.4.2 Mini-ImageNet

On Mini-ImageNet, we observe that our Reptile experiment achieves slightly superior performance to those reported in the Reptile paper [8] (Table 4). This is a bit surprising and we suspect it may stem from the confusion in the literature, alluded to earlier, over the exact test/validation class splits for Mini-ImageNet. That said, we were careful to use the exact Mini-ImageNet split provided in the code associated with the Reptile paper. Nevertheless, both our Reptile experiment and the original result slightly outperform both MAML and FOMAML on 1-shot, 5-way and 5-shot, 5-way image classification. Finally, we also extend our results to 5-way, 10-shot classification and demonstrate that providing additional fine-tuning data can offer a meaningful further performance boost.

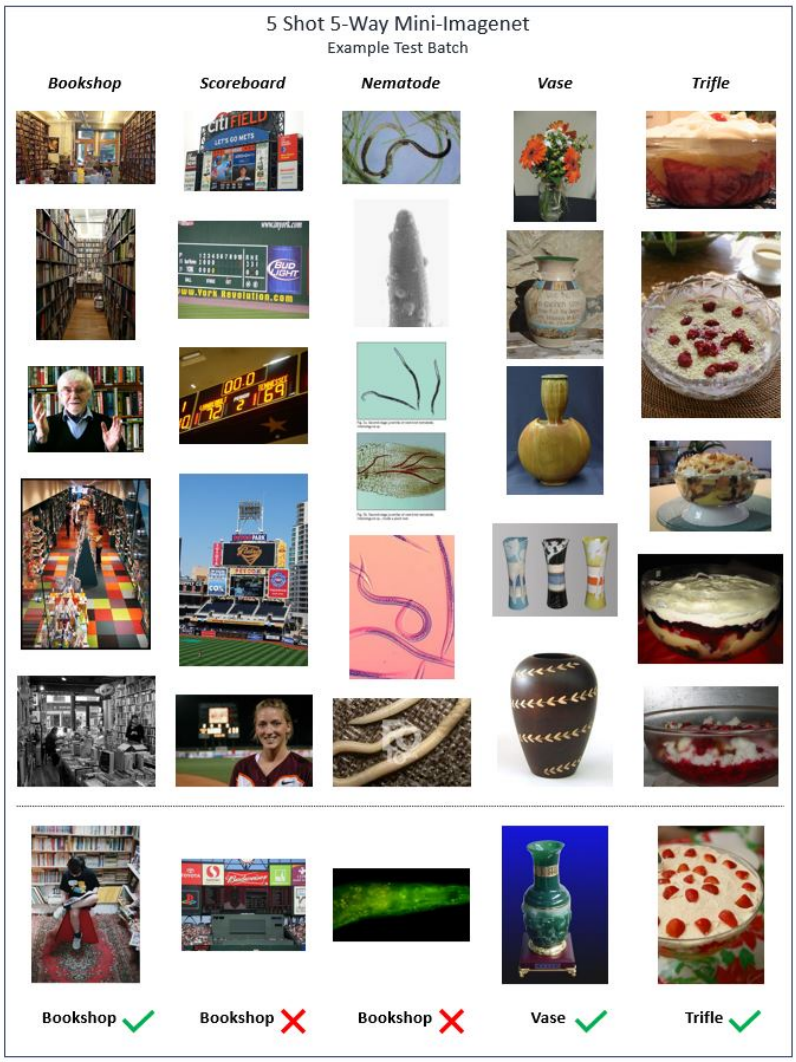


Figure 12: Example evaluation on Mini-ImageNet for 5-shot, 5-way classification task. The model is finetuned on the top 25 images and then evaluated on an image from each class in the bottom row. Predicted labels are shown below each test image.

Table 4: Results on Mini-ImageNet for few-shot classification.

Algorithm	1-shot	5-shot	10-shot
Reptile	51.6%	67.4%	72.6%
Reptile [8]	50.0%	66.0%	-
MAML [7]	48.7%	63.1%	-
FOMAML [7]	48.07%	63.2%	-

In Figure 12 we provide an example test batch evaluation for 5-shot, 5-way classification. The model is fine-tuned with 25 images (5 images each from 5 classes) and then evaluated on a test-batch of images evaluated all at once in the transductive setting.

6.5 Extension: Pre-training

We now motivate, detail, and evaluate an extension to improve the efficiency of the Reptile algorithm.

6.5.1 Motivation

Training a single Reptile run (i.e. 100k meta iterations on the Omniglot or Mini-ImageNet datasets) took on the order of days even with a powerful Nvidia Tesla K80 GPU. A major contributor to Reptile's inefficiency is that the algorithm must execute many inner-loop update steps before a single meta-loop parameter update. In particular, Reptile takes 25 meta-steps per meta-step using the Omniglot hyperparameters (5 inner-iterations X 5 tasks in meta-batch) and 40 inner-steps per meta-step using the Mini-ImageNet hyperparamters (8 inner-iterations X 5 tasks in meta-batch). This is not particularly efficient, especially if we consider what our model is really learning.

We suppose that much of what our convolutional neural network learns is pertinent to general image recognition. For instance, we'd expect the first layers of our convolutional neural network to focus on recognizing low level image features (corners, blobs) rather than specializing towards the meta-learning objective. Thus, it may be advantageous to apply transfer learning by, for example, taking a model like ResNet-50 [4] that has been pretrained on a general image recognition task (like ImageNet classification) and then fine-tuning the model to the meta-learning objective with Reptile. In doing so, we do not incur the high computational cost of Reptile to learn general image structure. However, we take a slightly different approach as we desire comparable results and thus do not want to modify the network architecture outlined in Section 6.2. We emphasise that this decision was made for experimental consistency and fine-tuning a more powerful architecture like ResNet-50 is still a worthwhile avenue for future exploration.

6.5.2 Pre-training Protocol

We outline a simple pre-training protocol as follows: We first adapt the network architecture in Section 6.2 such that the final softmax layer now has an output unit for every class in the training dataset (e.g. 64 for Mini-ImageNet). We then train a standard classifier on the training set until validation loss plateaus. Finally, we use the weights learned by this model (excluding the final softmax layer) as initial weights for the Reptile model and run Reptile as before.

Although we've added a pre-training procedure that incurs non-trivial time and compute, it runs much faster than Reptile training as we update parameters with every network forward-pass. To run the necessary pre-training update iterations took less than an hour and this seems a reasonable cost to speedup the days-long Reptile training procedure.

We used weight decay as a means to regularise pre-training (with the weight decay coefficient tuned to 0.999 for omniglot and 0.99 for MiniImageNet). Learning curves for pre-training, with both train and test set accuracies[8], are presented in Figures 13 and 14. Note we still observe some overfitting, particularly on Mini-ImageNet.

6.5.3 Results and Discussion

In Figure 15 we observe that pre-training for 1-shot, 5-way Omniglot provides Reptile with an early, but ultimately unsustained, training advantage. At 100k iterations, both Reptile and Reptile + pre-training reach an accuracy of 97.7%. On MiniImagenet, we also observe comparable results for Reptile and Reptile + pre-training (all results are outlined in Table 5).

[8]The pre-training classifier seeks to correctly classify images as belonging to any class in the training set and thus accuracies tend to be lower than the 5-way classification task used in Reptile. Moreover, training and test set classes are identical in pre-training (they are the training classes used by Reptile). The train/test distinction here is that test images are not seen during pre-training.

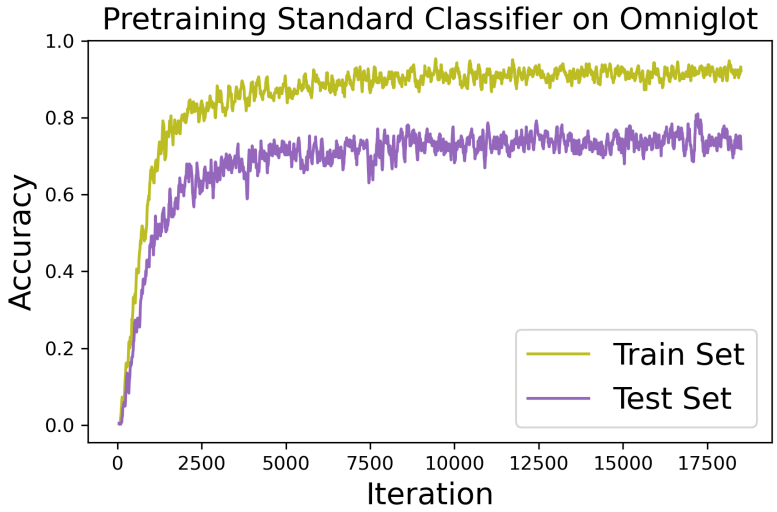


Figure 13: Accuracy evaluated over the course of pre-training a standard classifier for Omniglot.

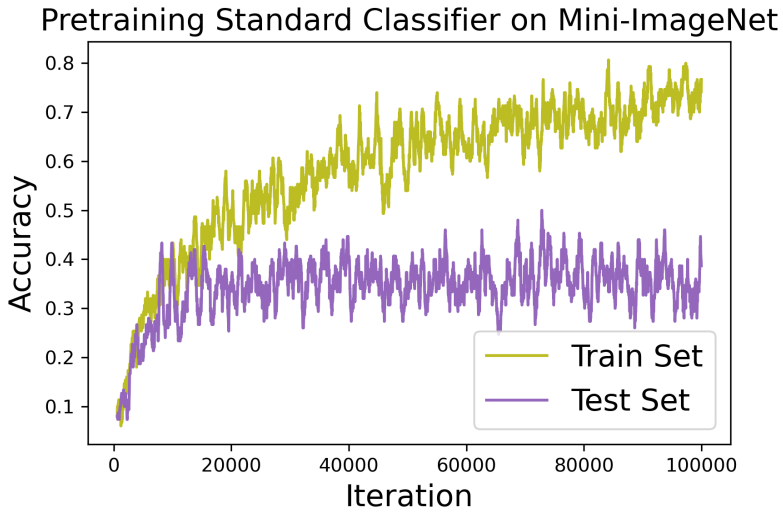


Figure 14: Accuracy evaluated over the course of pre-training a standard classifier for Mini-Imagenet.

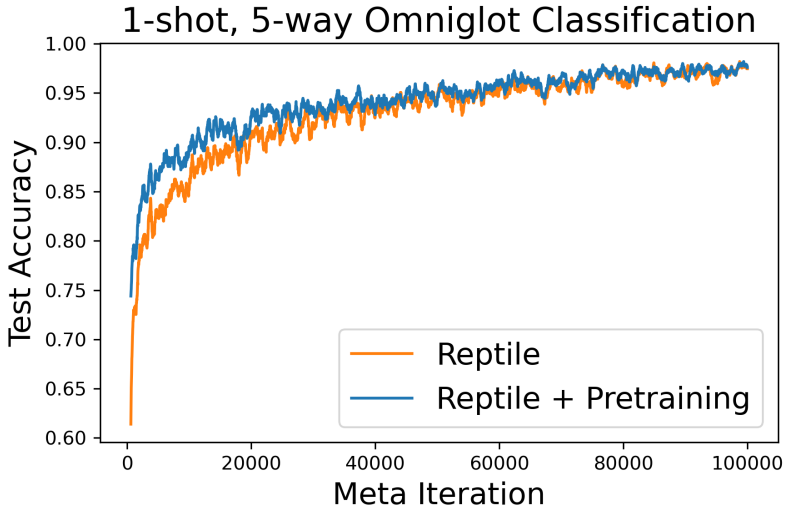


Figure 15: Test accuracy evaluated over the course of Omniglot training for both Reptile and Reptile + Pre-training. Although intialising weights from a pretrained classifier provides Reptile with an early training advantage, it reaches optimal performance no earlier than standard initialised Reptile.

Table 5: Results comparing Reptile to Reptile + pre-training on 1-shot, 5-way classification for both the Omniglot and Mini-ImageNet datasets

Algorithm	1-shot, 5-way Omniglot	1-shot,5-way Mini-ImageNet
Reptile	97.7%	51.6%
Reptile + pre-training	97.7%	49.4%

This is a bit disappointing. We may expect that provided enough time and compute that Reptile will reach similar performance to Reptile + pre-training. However, we would hope that the pre-training protocols allows Reptile + pre-training to achieve optimal performance earlier than Reptile, but this does not quite appear to be the case.

Nonetheless, we were unable to extensively explore and refine the pre-training protocol due to computational budget. For example, we implemented a function to freeze earlier network layers during Reptile training but were unable to evaluate its effectiveness. Likewise, it could be worthwhile to lower the Reptile learning rate if we are initialising from pretrained weights. Moreover, as noted earlier, the pretrained classifiers are not perfectly regularised and it may be beneficial to refine their regularisation so we do not initialise Reptile from overfitted weights. Finally, it would be intriguing to compare Reptile performance on a pretrained model like ResNet-50 versus the same uninitialised model.

All together, although this first attempt at pre-training Reptile produced underwhelming results, we believe that there are still some interesting refinements to the pre-training protocol to explore.

7 Conclusion

In this report we replicate fundamental results from the research paper on Reptile meta-learning [8] and investigate several modifications and extensions. In particular, we motivate and compare different inner-loop optimisers, extend the Reptile algorithm to a more complex set of function approximation tasks, and explore a pre-training protocol to increase efficiency on few-shot image classification.

Meta-learning algorithms seek to generalize across a distribution of tasks by learning to (quickly) learn a previously unseen task from that distribution. Reptile [8] and MAML [7] belong to a family of meta-learning algorithms that seek to obtain an initialisation of a neural network that can be quickly fine-tuned to new tasks at test time. In particular, Reptile provides a fast approximation to the MAML algorithm. Moreover, unlike the first-order MAML (FOMAML) approximation, Reptile retains information from higher-order derivatives. In a sentence, Reptile runs by repeatedly sampling different tasks, fine-tuning the network on the sampled tasks, and finally adjusting the network towards the fine-tuned weights.

Firstly, we replicated the results of the 1D sinusoid regression experiment from the original paper [8] and extended them to investigate inner-loop adaptive gradient optimisers. We found that the ADAM optimiser can locate more optimal local minima. However, we must be careful to reduce the leakage of sequential within-task minibatch gradients (via ADAM's momentum) for Reptile to work optimally. Moreover, by resetting Adam moment histories between tasks, we can attain more stable convergence.

Next, we extended the 1D sinusoidal regression task in the Reptile paper to investigate how the algorithm handles more complex regression tasks. We investigated increasing the meta-batch size from 1 to 2 by adding an additional task of fitting a basic Fourier series. We found Reptile was able to find an initialisation that captures some of the key features of each function, but it was unable to

provide an accurate fit (likely due to the neural network being too small to capture the additional complexity). We then investigated scaling the original task up from 1D to 2D and increasing the ratio of the number of points to predict to the number of sampled points by a factor of 200. We increased the size of the neural network to capture this increased complexity, and utilised both SGD and ADAM optimisers. We found promising results using ADAM but again did find limitations in the approach. These limitations were at least partially due to the momentum term not being reset after each meta-iteration, and it would be valuable to address this in the future. Finally, we presented a compelling visualisation of Reptile learning as it proceeds through the training process.

We then reproduce several important results in the domain of few-shot image classification. We show that Reptile achieves a 98.3% accuracy on 1-shot, 5-way classification for the Omniglot dataset. This is in-line with the original paper [8] but slightly trails both MAML and FOMAML [7] accuracies. Nonetheless, we proceed to show that Reptile achieves superior performance to MAML/FOMAML on the more complex Mini-ImageNet dataset (for both 1-shot, 5-way and 5-shot, 5-way classification).

Finally, we investigate a pre-training protocol motivated by Reptile's relatively inefficient update dynamics. Specifically, we surmised a neural net could more rapidly learn general image features via a standard classifier than the Reptile procedure. Although our warm-starting procedure outperforms Reptile early in training, it does not maintain a definitive advantage. Nonetheless, we were unable to extensively refine or tune the pre-training protocol and future work could explore, for example, freezing layers or lowering the learning rate. It may also be worthwhile to investigate warm-starting with other pretrained image networks (e.g., ResNet-50).

References

[1] Justin Domke. *Hessian-vector products*. Feb. 2009. URL: https://justindomke.wordpress.com/2009/01/17/hessian-vector-products/.

[2] Brenden Lake et al. "One shot learning of simple visual concepts". In: *Proceedings of the annual meeting of the cognitive science society*. Vol. 33. 33. 2011.

[3] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[4] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[5] Sachin Ravi and Hugo Larochelle. "Optimization as a model for few-shot learning". In: (2016).

[6] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[7] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-agnostic meta-learning for fast adaptation of deep networks". In: *International conference on machine learning*. PMLR. 2017, pp. 1126–1135.

[8] Alex Nichol, Joshua Achiam, and John Schulman. "On first-order meta-learning algorithms". In: *arXiv preprint arXiv:1803.02999* (2018).

[9] Alex Nichol, Joshua Achiam, and John Schulman. *Reptile: a Scalable Metalearning Algorithm*. 2018.

[10] Lilian Weng. *Meta-Learning: Learning to Learn Fast*. Nov. 2018. URL: https://lilianweng.github.io/lil-log/2018/11/30/meta-learning.html.

[11] Alireza Fallah, Aryan Mokhtari, and Asuman E. Ozdaglar. "On the Convergence Theory of Gradient-Based Model-Agnostic Meta-Learning Algorithms". In: *ArXiv* abs/1908.10400 (2020).

[12] Boyang Zhao. *Basics of few-shot learning with optimization-based meta-learning*. Aug. 2021. URL: https://boyangzhao.github.io/posts/few_shot_learning.

[13] Geoffrey Hinton. *Neural Networks for Machine Learnin Lecture 6a Overview of mini-batch gradient descent*. Accessed: 2022-03-30. URL: %5Curl%7Bhttps://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf%7D.

A Appendix

A.1 A Comparison of Meta-Learning Techniques

To understand the motivation behind Reptile and inspiration for its mathematical formulation, it helps to briefly cover the progression of the meta-learning models that preceded it.

Vanilla Transfer Learning

Transfer learning in its simplest form involves bulk training on one training data-set, and fine-tuning on another [12]. In this way it exposes the model to data of different contexts and improves generalisation somewhat over single-training-set training, but is still restricted by poor knowledge transfer to novel tasks in the test-set that have not been seen before (sometimes referred to as 'long-tail' tasks). Mathematically, the procedure of training can be summarised as (assuming one update iteration for simplicity):

1. Bulk training on the first training set:
$$\theta_{pre} = \theta_0 - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}^{pre,tr}) \tag{2}$$
2. Fine-tuning on the new training set:
$$\theta = \theta_{pre} - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}^{tr}) \tag{3}$$

Model-Agnostic Meta-Learning (MAML)

Whilst an improvement to generalisation ability, there is still no framework in transfer learning as written above that can allow for good model performance on completely novel data. As an approach to this, meta-learning was suggested, which takes a novel approach of *learning to learn* [7].

The difference now is that MAML seeks to find the optimal parameters during fine tuning that generalise to multiple other test sets over a *distribution* of tasks, τ. Therefore, for a set of tasks $\tau = \{\tau_1, ..., \tau_n\} \sim \mathcal{P}(\tau)$:

$$\begin{aligned} \phi_i &= \theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}_i^{tr}) \\ \theta &= \theta - \beta \nabla_\theta \sum_{i=1}^{n} \mathcal{L}(\phi_i, \mathcal{D}_i^{ts}) \end{aligned} \tag{4}$$

(Note that in the top line, θ are the initial model parameters, and ϕ_i the task-specific trained ones.) We can also re-express Equation (4) by expanding the grad operator via the chain rule:

$$\begin{aligned} \nabla_\theta \mathcal{L}(\phi_i, \mathcal{D}_i^{ts}) &= \nabla_\phi \mathcal{L}(\phi_i, \mathcal{D}_i^{ts}) \nabla_\theta \phi_i \\ &= \nabla_\phi \mathcal{L}(\phi_i, \mathcal{D}_i^{ts}) \nabla_\theta \left[\theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}_i^{tr})\right] \\ &= \nabla_\phi \mathcal{L}(\phi_i, \mathcal{D}_i^{ts}) \left[\hat{I} - \alpha \nabla_\theta^2 \mathcal{L}(\theta, \mathcal{D}_i^{tr})\right] \end{aligned} \tag{5}$$

where $\nabla_\theta^2 \mathcal{L}(\theta, \mathcal{D}_i^{tr})$ is the Hessian matrix of $\mathcal{L}(\theta, \mathcal{D}_i^{tr})$.

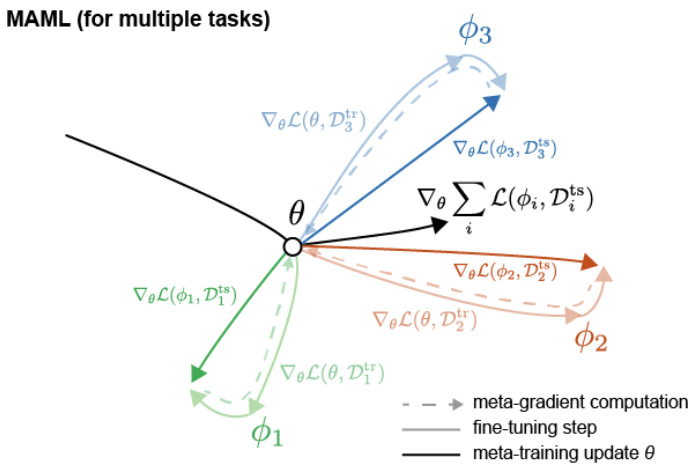


Figure 16: A visualisation of a single update step of the MAML algorithm. The net update step is the sum of the steps for all of the individual tas ks in the batch in parameter space. Figure taken from [12].

First-Order Approximation of MAML (FOMAML)

The method above therefore needs to calculate the Hessian for every parameter update, which scales with $\mathcal{O}(d^2)$ when there are d parameters in the model (although as mentioned in Section 1, the original code provided computational speed-up to this Hessian calculation in the specific case of neural networks by approximating the Hessian dot product using the finite difference approximation [1]). FOMAML therefore provides a faster approximation for MAML in general use by expanding Equations (5) and (4) to the first-order derivative:

$$\begin{aligned} \nabla_\theta \mathcal{L}(\phi_i, \mathcal{D}_i^{ts}) &\approx \nabla_\phi \mathcal{L}(\phi_i, \mathcal{D}_i^{ts}) \\ \theta &\approx \theta - \beta \sum_{i=0}^{n} \nabla_\phi \mathcal{L}(\phi_i, \mathcal{D}_i^{ts}) \end{aligned} \tag{6}$$

Reptile

It has been demonstrated [7, 8] that FOMAML can give approximations good enough that the accuracy is similar to MAML, suggesting first-order approximations are a strong approach. This is the motivation behind the Reptile algorithm which is discussed extensively throughout the report.

Reptile works by expressing the first updated parameters as an update operator function acting on the initial parameters θ: $\phi_l := U_{\tau_l}^k(\theta, \mathcal{D}_l^{tr})$, where k is the number of iterations ran on the meta-batch of sampled tasks (treated as a single task in all of the cases above for simplicity). There is a meta loop, which on each iteration samples tasks from a distribution as before (each task denoted $\tau_l \in \tau \sim \mathcal{P}(\tau)$), and then iteratively trains the parameters on each of these tasks, each over an inner loop of k iterations ($i = 1, ..., k$). In this way there is rapid learning of specific tasks in the inner loop, and gradual learning of generalisation to many tasks via the meta loop.

Therefore, the update process can be summarised as follows (where θ are the initial parameters):

1. For each iteration of the meta loop (gradual-learning), we first sample L tasks from $P(\tau)$: $\tau_l \in \tau \sim P(\tau)$ for $l = 1, ..., L$.

2. For each of the tasks l in the batch, the initial parameters are passed into the inner-loop

(rapid-learning), which iterates k times:

$$\phi_l^{(i+1)} = \phi_l^{(i)} + \beta(\phi_l^{(i)} - \theta) \text{ for } i = 1, ..., k \tag{7}$$

3. After k iterations of gradient update inner-loop for each task, the initial parameters are updated via:

$$\theta = \theta + \frac{\beta}{k} \sum_{\tau_l \in \tau} \left[\phi_l^{(k)} - \theta \right] \tag{8}$$

4. This process is then repeated for the next iteration of the meta loop, where a new set of tasks is sampled.

5. The effect of this meta-learning is that, on evaluation on a new unseen task, the model is able to rapidly generalise to the new task with less data, demonstrating few-shot learning.

In the above framework, it seems like this is equivalent to joint learning, where the model is trained on the expected loss over a sample of tasks at each iteration:

$$\mathbb{E}_{\tau_l \in \tau} \left[\nabla_\theta \mathcal{L}_{\tau_l}(\theta) \right] = \nabla_\theta \mathbb{E}_{\tau_l \in \tau} \left[\mathcal{L}_{\tau_l}(\theta) \right] \tag{9}$$

However, this only holds true if $k = 1$, otherwise, Reptile deviates from joint learning and the equality breaks, as the Reptile update becomes dependent on higher order derivatives as demonstrated below in Appendix A.2.

It is also worth noting that, unlike (FO)MAML, Reptile does not require splitting into support and query sets, which is advantageous when data availability is low. Also, in the original paper [8], they only sample one task per iteration i (i.e. the sum over l vanishes in Equation (8)).

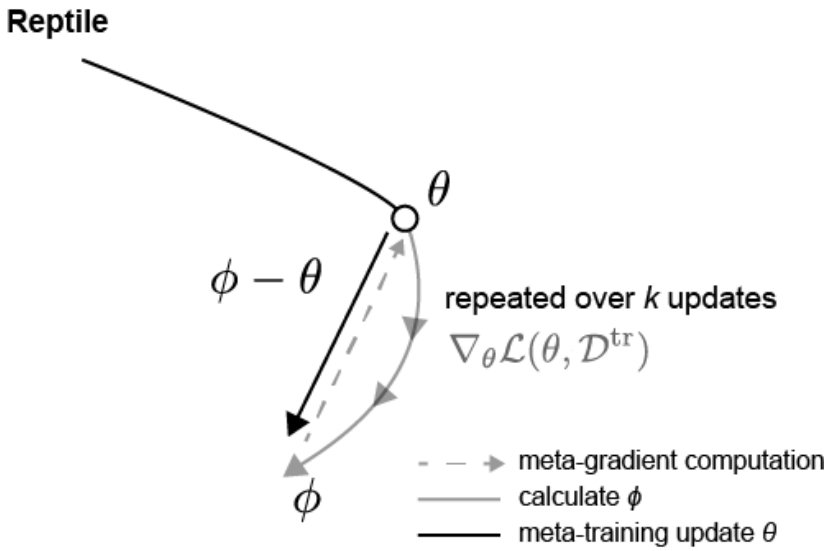


Figure 17: A similar plot to Figure 16, but demonstrating the concept behind Reptile, for one sampled task in a meta loop iteration and three inner-loop iterations. In the batch-sample variation, the trajectory would again be piece-wise (due to inner-loop iterations), but with a sum of trajectories over different tasks in the batch, similar to Figure 16. Figure taken from [12].

NB: we used θ and ϕ to denote initial and updated parameters respectively in this Reptile discussion in order for the notation to be consistent with the other models. However, in the next section, and in the rest of the report, Reptile uses $\bar{\phi}$ to mean the

initial parameters, and $\phi_l^{(i)}$ to mean the i^{th} iteration updated parameters for task τ_l in sampled batch τ. Below, we also use the short-hand notation that ϕ_i represents the parameters trained on an entire batch of tasks for iteration i of the inner loop, where the batches are different.

A.2 Justification for Using Reptile

This section aims to provide an idea of why the Reptile algorithm works, and how it can be shown to perform in a similar way to (FO)MAML. To do this, we first introduce the new parameters similar to those adopted in the main body of the updated notation report[9] [9]:

$$\begin{aligned} g_i &:= \mathcal{L}_i'(\phi_i) \\ H_i &:= \mathcal{L}_i''(\phi_i) \\ \phi_{i+1} &:= \phi_i - \alpha g_i \\ \bar{g}_i &:= \mathcal{L}_i'(\bar{\phi}) \\ \bar{H}_i &:= \mathcal{L}_i''(\bar{\phi}) \end{aligned} \tag{10}$$

where the subscript on the loss corresponds to different mini-batches of data for the same task (i.e. different iterations of the inner-loop where different mini-batches are sampled), where we do not necessarily have to update parameters (hence the i subscript on the loss does not always have to line-up with the i subscript on the ϕ). This is a slight deviation from the algorithm, where the parameters are automatically updated with each iteration of the inner-loop.

A Gradient-Based Formalism

With this notation, we can express approximations for the Reptile gradient via a Taylor expansion about $\phi_i = \bar{\phi}$ for each $\mathcal{L}_i'(\phi_i)$ [8]:

$$\begin{aligned} g^{\text{Reptile}} = \sum_{i=1}^{k} \mathcal{L}_i'(\phi_i) &\approx \sum_{i=1}^{k} \left\{ \mathcal{L}_i'(\bar{\phi}) + (\phi_i - \bar{\phi}) \mathcal{L}_i''(\bar{\phi}) \right\} \\ &= \sum_{i=1}^{k} \left\{ \bar{g}_i + (\phi_i - \bar{\phi}) \bar{H}_i \right\} \\ &= \sum_{i=1}^{k} \left\{ \bar{g}_i + \bar{H}_i \left[\phi_{i-1} - \beta g_{i-1} - \bar{\phi} \right] \right\} \\ &= \sum_{i=1}^{k} \left\{ \bar{g}_i - \beta \bar{H}_i \sum_{j=1}^{i-1} g_j \right\} \approx \sum_{i=1}^{k} \left\{ \bar{g}_i - \beta \bar{H}_i \sum_{j=1}^{i-1} \bar{g}_j \right\} \end{aligned} \tag{11}$$

Therefore, to a first-order expansion, we can write the Reptile gradient as:

$$\boxed{g^{\text{Reptile}} \approx \sum_{i=1}^{k} \left\{ \bar{g}_i - \beta \bar{H}_i \sum_{j=1}^{i-1} \bar{g}_j \right\}} \tag{12}$$

[9]The updated version contains the same experiments and algorithms to the original, but with improved notation to clarify ambiguities in the derivations.

Similarly, we can expand the MAML gradient to the first order in learning rate (via the chain rule) [7]:

$$\begin{aligned} g^{\text{MAML}} &= \frac{\partial}{\partial \bar{\phi}} \mathcal{L}_k(\phi_k) = \prod_{i=1}^{k-1} \nabla_{\bar{\phi}} \phi_i \cdot \mathcal{L}'_k(\phi_k) \\ &= \prod_{i=1}^{k-1} \left[\hat{I} - \beta \nabla^2_{\phi} \mathcal{L}_i(\phi_i) \right] \mathcal{L}'_k(\phi_k) = \prod_{i=1}^{k-1} \left[\hat{I} - \beta H_i \right] g_k \end{aligned} \tag{13}$$

which on expansion approximates to:

$$\begin{aligned} g^{\text{MAML}} &= \prod_{i=1}^{k-1} \left[\hat{I} - \beta H_i \right] g_k \approx \prod_{i=1}^{k-1} \left[\hat{I} - \beta H_i \right] \left\{ \bar{g}_k - \beta \bar{H}_k \sum_{j=1}^{k-1} \bar{g}_j \right\} \\ &\approx \bar{g}_k - \beta \bar{g}_k \sum_{j=1}^{k-1} \bar{H}_j - \beta \bar{H}_k \sum_{j=1}^{k-1} \bar{g}_j \end{aligned} \tag{14}$$

So we get the MAML gradient expansion:

$$\boxed{g^{\text{MAML}} \approx \bar{g}_k - \beta \bar{g}_k \sum_{j=1}^{k-1} \bar{H}_j - \beta \bar{H}_k \sum_{j=1}^{k-1} \bar{g}_j} \tag{15}$$

The final gradient expansion is then for FOMAML, which treats the Hessian as being equal to 0, and thus approximates the MAML gradient as [9]:

$$\boxed{g^{\text{FOMAML}} = \bar{g}_k - \beta \bar{H}_k \sum_{j=1}^{k=-1} \bar{g}_j \approx \mathcal{L}'_k(\phi_k)} \tag{16}$$

Therefore, if we arbitrarily take $k = 2$ (as it makes the explanation in the next section clearer, but this holds for general k, see [8]), we get the following relationship:

$$\begin{aligned} g^{\text{FOMAML}} &\approx \bar{g}_2 - \beta \bar{H}_2 \bar{g}_1 \\ g^{\text{MAML}} &\approx \bar{g}_2 - \beta \bar{g}_2 \bar{H}_1 - \beta \bar{H}_2 \bar{g}_1 \\ g^{\text{Reptile}} &\approx \bar{g}_1 + \bar{g}_2 - \beta \bar{H}_2 \bar{g}_1 \end{aligned} \tag{17}$$

Generalising through Inner Products

As the final stage of this somewhat lengthy justification, we define two new properties: the expectation over tasks of the loss gradient on the first minibatch with respect to the inital parameters (prior to any inner-loop steps), denoted AvgGrad; and the AvgGradInner, which is the expectation over tasks of the inner product between gradients on the first and second minibatches with respect to the initial parameters (note that we have chosen $k = 2$ still for simplicity):

$$\begin{aligned} \text{AvgGrad} &= \mathbb{E}_\tau \left[\bar{g}_1\right] \\ \text{AvgGradInner} &= \mathbb{E}_\tau \left[\bar{H}_2 \bar{g}_1\right] = \mathbb{E}_\tau \left[\bar{H}_1 \bar{g}_2\right] \end{aligned} \tag{18}$$

The equality on the last line gives rise to the identity:

$$\text{AvgGradInner} = \frac{1}{2}\mathbb{E}_\tau \left[\bar{H}_2 \bar{g}_1 + \bar{H}_1 \bar{g}_2\right] = \frac{1}{2}\mathbb{E}_\tau \left[\frac{\partial}{\partial \bar{\phi}}(\bar{g}_1 \cdot \bar{g}_2)\right] \tag{19}$$

Rewriting Equation (20) in terms of these new definitions for all 3 algorithms now gives:

$$\begin{aligned} \mathbb{E}_\tau[g^{\text{FOMAML}}] &\approx \text{AvgGrad} - \beta\text{AvgGradInner} \\ \mathbb{E}_\tau[g^{\text{MAML}}] &\approx \text{AvgGrad} - 2\beta\text{AvgGradInner} \\ \mathbb{E}_\tau[g^{\text{Reptile}}] &\approx 2\text{AvgGrad} - \beta\text{AvgGradInner} \end{aligned} \tag{20}$$

Stepping in the direction of negative gradient therefore steps in a way to minimise AvgGrad and maximise AvgGradInner.

Maximising the AvgGradInner is equivalent to maximising the dot product between gradients on two minibatches of a given task and hence improves within-task generalisation of the model. The minimisation of the AvgGrad term is equivalent to minimising the gradient on each single minibatch, similar to standard deep learning.

Equation (20) therefore indicates that MAML is more concerned with maximising minibatch generalisation than both FOMAML and Reptile, suggesting that MAML would be more accurate over the same number of minibatches. This was however not found to be consistently the case in the Omniglot and Mini-ImageNet experiments (Table 3 and Table 4), inviting the question of whether the approximations in this mathematical reasoning are insufficient, or if instead this was a consequence of a lack of extensive hyper parameter tuning affecting the results.