Toward Improved Meta-Imitation Learning

Ryan Brand
Department of Computer Science
Columbia University
rmb2208@columbia.edu

Jason Krone
Department of Computer Science
Columbia University
jpk2151@columbia.edu

Ted Moskovitz
Department of Computer Science
Columbia University
thm2118@columbia.edu

Abstract

This paper builds off of the work done in "One-Shot Visual Imitation Learning via Meta-Learning" [1], which demonstrated the success of Model Agnostic Meta-Learning (MAML) [2] for meta-imitation-learning (MIL) of robotic grasping tasks. Our work focuses on answering two questions. First, does the MIL neural network architecture benefit from the addition of an external memory? And second, can we increase the computational efficiency of MIL by using an alternate meta-learning algorithm? To determine if memory is beneficial, we evaluate the performance of various architectures augmented with the memory module proposed in "Learning to Remember Rare Events" [3]. To make MIL less computationally intensive, we adapt and evaluate the performance of a recently proposed, more efficient meta-learning algorithm called Reptile on the simulated pushing task proposed in [1].

1 Introduction

Neural networks have achieved super-human performance on a number of tasks, including many of the Atari 2600 games implemented in The Arcade Learning Environment (ALE) [4]. Yet compared with the human brain, neural networks do not generalize well across tasks and exhibit poor sample complexity. An analysis conducted jointly by researchers at Harvard and MIT found that within ten minutes, the average performance of a human participant on the Atari game Amidar, which he or she had never played, exceeded that of a DDQN reinforcement learning agent trained for 115 hours [5]. This implies that in the context of the game Amidar, human participants were on average 460 times more sample efficient than the reinforcement learning agent. As the authors of this study point out, this is largely because the reinforcement learning agent enters this learning processes without any prior knowledge. The agent must learn to perceive the frames of game play from scratch, while the human subject begins the same task with a working visual system. Moreover, it is likely that the human participant has relevant prior experience playing other types of video games (i.e., performing similar tasks) whereas the reinforcement learning agent has no prior experiences from which to generalize.

The above example illustrates the deficiencies of the standard machine learning paradigm wherein an agent is trained from scratch on a single task. These deficiencies serve as the motivation for meta-learning, a sub-field of machine learning aimed at training models on a number of different tasks such that they can quickly solve new tasks using a small number of examples. This goal is also shared by robotics researchers who want to create general purpose robots that can aid humans in accomplishing everyday tasks. Until recently, relatively little progress had been made toward these

goals. Fortunately, newly developed algorithms have improved meta-learning performance, opening up opportunities for advancement in robotic control and reinforcement learning.

To provide context, in our related works section we introduce the imitation learning problem and define meta-imitation learning with a discussion of [1]. Imitation learning begins with a data-set of demonstrations consisting of input features o and expert actions a. The goal of imitation learning is to learn a policy f that, given the observation o, outputs the action a taken by the expert.

In this paper, we integrate previous techniques for meta-imitation learning and few-shot learning through two separate approaches. First, we leverage a newly released meta-learning algorithm called Reptile [6], applying it for the first time in an imitation learning setting. In the second, we augment a previously developed policy network used for meta-imitation learning [1] with an explicit memory module originally developed for few-shot learning [3]. We provide more detailed descriptions of both the Reptile algorithm as well as the memory module in the following section.

2 Related Work

The current body of literature on meta learning is fairly large. Fortunately, the existing approaches can be grouped based on setting (either supervised or reinforcement learning) and method type (metric based or gradient based). In the supervised setting, meta-learning is often referred to as few-shot classification. Few-shot classification assumes that you are given a labeled dataset $D = \{(x_i, y_i)\}$ consisting of examples x and labels y, and it treats each distinct class y_i as a different task. The goal of few shot classification is to train a model on a set of classes in order to maximize classification accuracy on a held out, disjoint set of test classes. In contrast, in the reinforcement learning setting each unique Markov decision process, which consists of an initial state distribution $\rho_0(o_0)$, transition distribution $P(o_{t+1} \mid o_t, a_t)$, and reward function $R : O \times A \to \mathbb{R}^+$, is treated as a different task, and the objective is to train a policy π on a distribution of training tasks $p_{train}(\mathcal{T})$ such that π maximizes the expected cumulative return $\sum_{\mathcal{T}_i \sim p_{test}(\mathcal{T})} \mu(\pi_\theta, \mathcal{T}_i)$ on a held out distribution of test tasks $p_{test}(\mathcal{T})$, where $\mu(\pi_\theta, \mathcal{T}_i) = \mathbb{E}_{\tau \sim \pi_\theta, \mathcal{T}_i} \left[\sum_{i=0}^{T} r(o_t, a_t) \right]$.

As mentioned above, existing approaches typically fall into two categories: metric based methods and gradient based methods. The metric based methods are for the most part used in the supervised setting for few-shot classification. Generally, metric based approaches work by learning a distance metric for which two examples of the same class will be "close" together and two examples from different classes will be "far apart". To classify a new example x, the distances between x and all other examples seen previously are compared and the class y_{close} belonging to the previously seen example x_{close} which is "closest" to x according to the distance metric is assigned to x. On the other hand, gradient based methods utilize stochastic gradient descent to optimize an objective, which incentivizes rapid adaptation to a previously unseen task. MAML [2] and Reptile [6] both fall into the category of gradient based methods. We refer the reader to the excellent blog post "Learning to Learn" [7] by Chelsea Finn for a more complete survey of the meta-learning literature.

For the remainder of this section we focus on the papers that introduce the methods we make use of in our work. Specifically, these papers are "One-Shot Visual Imitation Learning via Meta-Learning" [1], which introduces MIL, "On First-Order Meta-Learning Algorithms" [6], which introduces Reptile, and "Learning to Remember Rare Events" [3], which introduces the memory module we use in our experiments. The following three sections focus on each of these papers, respectively.

2.1 One-Shot Visual Imitation Learning via Meta-Learning

"One-Shot Visual Imitation Learning via Meta-Learning" by Finn et al. [1] was the first work to apply MAML [2] to imitation learning. This paper focused on teaching robotic agents to leverage information from previous skills to quickly learn new behaviors. Concretely, in their experiments on simulated pushing they teach an agent to imitate an expert demonstration and push an object to a target location. Each demonstration is an expert trajectory $\tau := \{o_1, a_1, ..., o_T, a_T\}$ used to successfully accomplish a task $\mathcal{T}_i$, where o_t is the observation at time-step t and a_t is the action taken by the expert at time-step t.

The main distinction between MIL using MAML and other imitation learning methods is that MAML optimizes for the cumulative performance over a number of tasks after taking a step of stochastic

gradient descent (SGD). This is best understood by inspecting the following MIL training objective:

$$\min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} L_{\mathcal{T}_i}(f_{\theta_i'}) = \sum_{\mathcal{T}_i \sim p(T)} \mathcal{L}_{\mathcal{T}_i}(f_{\theta - \alpha \Delta_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)}), \tag{1}$$

where $\mathcal{L}_{\mathcal{T}_i}$ is defined as

$$\mathcal{L}_{\mathcal{T}_i}(f_\phi) = \sum_{\tau^{(j)} \sim \mathcal{T}_i} \sum_t \left\| f_\phi(o_t^{(j)}) - a_t^{(j)} \right\|_2^2 \tag{2}$$

In the above formula, f is the policy we are training parameterized by a vector ϕ. θ represents the initial parameters of the policy, and $\theta - \alpha \Delta_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ represents the parameters of the policy after taking one step of SGD on the loss function L_{T_i} for task T_i. The loss L_{T_i} is what would be traditionally used to train an imitation learning agent on a single task T_i. In contrast, MIL optimizes the initial parameters of the policy θ over a distribution of tasks $p(T)$ such that the model can be easily fine-tuned on a new task T_i by taking a step $f_{\theta - \alpha \Delta_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)}$ with SGD. Moreover, this framework can incorporate a two-head architecture in which there are two distinct output layers, each with its own parameters. The parameters of the first, known as the "pre-update" head, are not used for the final policy output, while the parameters of the second, called the "post-update" head, are not affected by the demonstration. Both are updated throughout the meta-learning process, essentially allowing a different objective function to be used for the inner loop while maintaining the original objective for the outer loop.

In our Reptile experiments and our external memory experiments, we use the seven layer neural network architecture presented in MIL as the basis for our model. In addition, we evaluate the performance of our models on the simulated pushing task introduced in [2]. This paper also gives the performance of a LSTM baseline that attains a 78.38% success rate on the pushing task, which we benchmark against. It is worth noting that in addition to the pushing task, Finn et al. also evaluated their model on a simulated reaching task and real world placing task. We chose to focus solely on the simulated pushing task because it is the more difficult simulated task, and given our constrained computational resources it was very costly to train on more than one task. We leave the application of our model to real world placing for future work once we have demonstrated a more significant performance gain over the MIL model.

2.2 On First-Order Meta-Learning Algorithms

The paper "On First-Order Meta-Learning Algorithms" by Nichol et al. [6] presents the Reptile meta-learning algorithm and conducts a theoretical analysis of other first order meta-learning algorithms. The authors specific focus on "first order" algorithms is at least in part motivated by a desire to address MAML's core weakness (computational inefficiency), which primarily stems from its use of a second order derivative in the update rule. In their experiments, Nichol et al. show that Reptile achieves comparable performance to MAML in the supervised few-shot learning setting on the Omniglot and Mini-Imagenet datasets. The increased computational efficiency and comparable performance of Reptile on few-shot classification are the core reasons why we chose investigate it as an alternate meta-learning algorithm. To the best of our knowledge our work is the first time that Reptile has been applied to imitation learning. The following pseudocode gives a precise description of the Reptile meta-learning algorithm:

Algorithm 1 Reptile

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize network parameters ϕ
2: **for** $i = 1, 2, \ldots$ **do**
3: Sample task τ, corresponding to loss L_τ on weight vectors $\widetilde{\phi}$
4: Compute $\widetilde{\phi} = U_\tau^k(\phi)$, denoting k steps of SGD with Adam optimizer and learning rate α
5: Update $\phi \leftarrow \phi + \beta(\widetilde{\phi} - \phi)$
6: **end for**

2.3 Learning to Remember Rare Events

While augmenting neural networks with explicit memory has become an increasingly well-researched topic in recent years [8] [9], most implementations are highly specialized, not flexible or easy to train enough to integrate into a variety of environments or tasks. Recently, however, progress has been made in improving the portability and trainability of these frameworks. We adapt the memory module proposed in "Learning to Remember Rare Events" (LRRE) by Kaiser et al. [3] for use in our experiments with an external memory. The LRRE memory module $\mathcal{M} = (K, V, A)$ consists of keys K, values V, and ages A where K is $memory - size \times key - size$, V is $memory - size \times value - size$, and A is $memory - size \times 1$. Here $memory - size$ is the number of key-value pairs held in memory, $key - size$ is the dimension of the keys, and $value - size$ is the dimension of the values. In our implementation, the memory module is always connected to the same layer from which it takes its input (that is, it substitutes the output of a layer with a stored value), so $key - size = value - size$, which does not have to be the case. The values in the memory are access via a key-size query q. Given q the memory computes the k-nearest neighbors $(n_1, n_2, n_3, \ldots, n_k)$ of that query using the cosine similarity metric $d_{cosine} = q \cdot K[n_i]$. In our memory we use the representations produced by passing the observations o_t through our convolutional network as keys and the corresponding actions a_t as values.

The memory is trained using a triplet loss chosen to encourage the policy network to maximize the similarity between the query and positive keys (those that map to correct values) and minimize the similarity between the query and negative keys (those that map to incorrect values). Given a query q with value v and nearest neighbor n_1 the loss differs depending if $V[n_1] = v$ or $V[n_1] \neq v$. If $V[n_1] = v$, then the loss

$$\mathcal{L}(\mathcal{M}) = \max\{0, q \cdot k_b - q \cdot k_1 + \alpha\}$$

where b is the smallest index such that $V[n_b] \neq v$ and α is a threshold chosen such that once the similarities between the positive examples and the negative examples are α distance apart, no loss is propagated. When $V[n_1] = v$, the key at location n_1 is updated using the formula $K[n_1] = \frac{q+k_1}{||q+k_1||}$. Alternatively, if $V[n_1] \neq v$, then the loss

$$\mathcal{L}(\mathcal{M}) = \max\{0, q \cdot k_1 - q \cdot k_p + \alpha\}$$

and the key at location n' is set to v where $n' = \arg\max_i(A[i] + r)$, where r is a random integer chosen such that $|r| \ll |\mathcal{M}|$, and p is the smallest index such that $V[n_p] = v$. The overall memory loss is then

$$\mathcal{L}(q, v, \mathcal{M}) = \max\{0, q \cdot k_b - q \cdot k_p + \alpha\} \tag{3}$$

This LRRE memory module has been shown to perform well on the Omniglot few-shot classification task as well as an external module in a LSTM used for machine translation. This previous success on meta-learning in the supervised setting and versatility of the module makes it a good candidate for application to meta imitation learning.

3 Approach

The focus of our research is two fold. Firstly, we want to make MIL more efficient by exchanging the MAML meta-learning update with the Reptile meta-learning update. Secondly, we aim to determine if the use of the LRRE memory module in the MIL network architecture improves performance. As we have introduced the original formulations of both Reptile and the LRRE memory in our related work section, here we focus on the modifications we made to these algorithms and components respectively.

3.1 Reptile for Meta Imitation Learning

Training

There are two main differences between the version of Reptile that we introduced in the related works section and the variant of reptile that we used for our experiments. Firstly, our modified version samples a batch of tasks at each training iteration and updates the parameters ϕ of the model by the average difference between the pre-update parameters ϕ and the post-update parameters $\widetilde{\phi}_i$ over all

tasks in the batch instead of doing one update per task. We chose to do batched updates because it makes for a fairer comparison with MAML, which also uses batched updates. Secondly, we modified Reptile to use the imitation learning loss function

$$L_{T_i}(f_\phi) = \sum_{\tau^{(j)} \sim T_i} \sum_t \left\| f_\phi(o_t^{(j)}) - a_t^{(j)} \right\|_2^2$$

in place of the traditional supervised loss. Pseudo code for our meta imitation learning variant of Reptile is provided below in algorithm two.

Policy Inference

Now that we have described the training procedure used for Reptile, we move on to discuss the meta test time procedure. To evaluate Reptile's k-way, n-shot performance, where k is the number of examples per task and n is the number of tasks used for the meta update, we first sample a batch of n tasks with k examples each and perform k steps of SGD with Adam [10] on the model parameters ϕ using this batch of data. The purpose of this step is to "teach" the network how to accomplish these previously unseen tasks. Then we roll out the policy π_ϕ that has been trained using reptile on our simulated pushing environment by using it to predict the action a_t to take at each time step given the current observation o_t. At each time-step we check to see if the policy has achieved its goal i.e. succeeded at pushing the given object to the target location. If the agent is able to push the object onto the red target location for 10 time-steps within a 100-timestep episode, we consider the trial a success. This success criterion matches that used in [1].

Algorithm 2 Meta-Imitation Learning with Reptile

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize network parameters ϕ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample demonstration $\tau = \{\mathbf{o}_1, \mathbf{a}_1, \ldots, \mathbf{o}_T, \mathbf{a}_T\}$ from $\mathcal{T}_i$
6: Evaluate $\nabla_\phi \mathcal{L}_{\mathcal{T}_i}(f_\phi)$ using τ and $\mathcal{L}_{\mathcal{T}_i}$ in MIL Equation (2)
7: Compute adapted parameters with SGD for k iterations: $\widetilde{\phi}_i = \phi - \alpha \nabla_\phi \mathcal{L}_{\mathcal{T}_i}(f_\phi)$
8: **end for**
9: Update $\phi \leftarrow \phi - \beta \frac{1}{k} \sum_{i=1}^{n} (\widetilde{\phi}_i - \phi)$
10: **end while**
11: **return** parameters ϕ that can be quickly adapted to new tasks through imitations

3.2 Memory Augmented Meta Imitation Learning

Policy Inference

In our second approach, we integrated the memory module developed by Bengio et al. [3] into the model's policy network (Fig. 1). As in the original paper, we define the memory module $\mathcal{M}$ as a triple, in this case adapting it for multi-dimensional values:

$$\mathcal{M} = (K_{\text{memory-size} \times \text{key-size}}, V_{\text{memory-size} \times \text{value-size}}, A_{\text{memory-size}}).$$

As originally designed, it works by accepting the activations of a selected layer in the network architecture as a query $\mathbf{q}$. The query is then normalized ($||\mathbf{q}|| = 1$) and multiplied by the stored key matrix K, with the nearest neighbor $\text{NN}(\mathbf{q})$ defined as

$$\text{NN}(\mathbf{q}) = \arg\max_i \mathbf{q} \cdot K[i].$$

In addition to the most similar key, we have set the module to compute the k nearest neighbors, calculating the cosine similarities $d_j = \mathbf{q} \cdot K[i_j]$, where i_j is the index of the ith nearest neighbor. Taking a softmax over these values produces a confidence vector $m = \text{softmax}(d_1 t, \ldots, d_k t)$, where t is the inverse of the softmax temperature. The entry $m[i]$ then represents the degree of confidence

the model has that the stored value for the ith nearest neighbor is a "good" output. The definition of a "good" output varies based on the type of task and where the memory module output is fed back into the policy network. In a classification task with the memory module attached to the output layer, a "good" output would be regarded as the correct label. In the control-based task covered here, a "good output" for a memory module linked to the output layer can be loosely viewed as an action - in this case, a torque applied to the robot limb - that will move it closer to its goal, i.e. push the target object closer to the desired spot. Theoretically, the memory module could be connected to a hidden layer, such as the location where the convolutional network embedding is concatenated with the current robot configuration (see Fig. 1). In this case, the query to the memory module would represent the network's understanding of its current environment, with a "good" output being considered an embedding that is easily interpreted by the downstream fully-connected layers and assists in selecting the correct action. However, training such a module would be challenging, as its current formulation requires supervised examples of intended module outputs, which would require the identification and collection of "good" embeddings to be used as part of a labeled dataset. We leave this to future work (see Section 5).

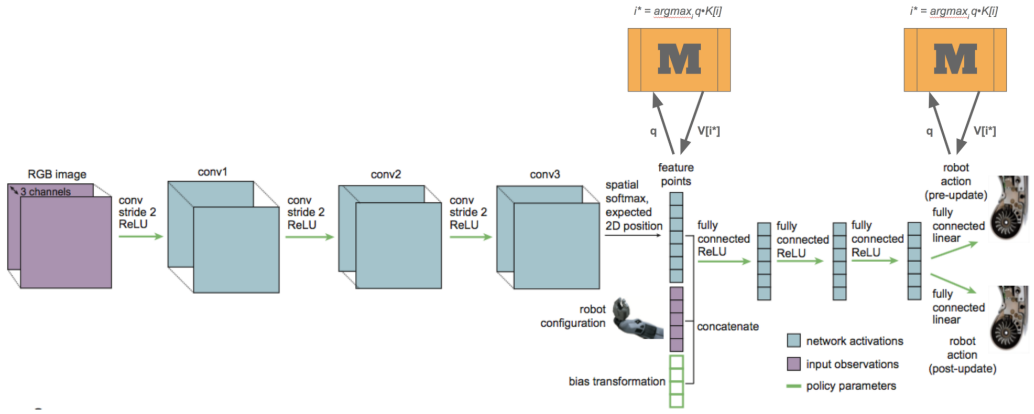

Figure 1: Two-headed policy architecture showing two possible locations for memory augmentation. We used only one memory module in our implementation, achieving the best performance by attaching it to the output layer. In theory, however, multiple memory modules could be attached.

As part of our implementation, with the memory module linked to the policy output layer, the confidence vector is then returned by the memory module and the output action $\hat{\mathbf{a}}_t$ is determined as follows:

$$\hat{\mathbf{a}}_t = \begin{cases} \mathbf{V}[\mathbf{i}^*] & \text{if } m[i^*] > \delta \\ \mathbf{q} & \text{otherwise} \end{cases}, \tag{4}$$

where δ is a constant threshold. Thus, the network selects the returned action from the memory module only if it has sufficient confidence that the remembered torque represents a beneficial action. Otherwise, it defaults to the original action (the query $\mathbf{q}$). We use $\delta = 0.5$ in all experiments.

In the two-headed architecture, the memory module is shared between both heads. This has interesting implications, as since the weights are independent between the pre-update and the post-update heads, the shared memory module allows them to benefit from equal access to previously beneficial actions, while retaining the flexibility afforded by creating separate heads in the first place. As demonstrated in Section 4.2, this produces a marked boost in performance in comparison to attaching the memory module to one head only.

Training

The training algorithm is the same as used by Finn et al. [1]. That is, it uses MAML to perform meta-imitation learning (see RELATED WORK). The memory module is also trained in the manner as introduced by Bengio et al. [3]. Because the outputs are real-valued torques, the selected action from the memory value array $\mathbf{V}[\mathbf{i}^*]_t$ will never be exactly equal to the labeled demonstration action $\mathbf{a}_t$. We therefore experimented with two measures of proximity, the L_1 norm and cosine similarity, to

weight the nearest neighbors determined by $\mathbf{q}_t K$. More specifically, for the L_1 norm, we compute the quality measure ρ_t as

$$\rho_t = 1 - \min\{1, |\mathbf{a}_t - \mathbf{V}[\mathbf{i}]_t|\}, \tag{5}$$

for $i = 1$ to k (where k is the number of nearest neighbors). When using cosine similarity, $|\mathbf{a}_t - \mathbf{V}[\mathbf{i}]_t|$ is replaced by $\frac{\mathbf{a}_t \cdot \mathbf{V}[\mathbf{i}]_t}{||\mathbf{a}_t|| ||\mathbf{V}[\mathbf{i}]_t||}$ in Equation 5. In this way, the element-wise absolute difference between the intended output action and the action predicted by the memory module is clipped at a maximum of 1, resulting in lower values for ρ_t for bad actions and higher values for good actions, with $\rho_t \in [0, 1]$. The positive similarities $q \cdot K[n_p]$ from Equation (3) are then calculated as $\arg\max_i(q \cdot K[i]) \odot \rho_t$ and the negative similarities $q \cdot K[n_b]$ are calculated as $\arg\max_i(q \cdot K[i]) \odot (1 - \rho_t)$, where $\odot$ is the element-wise Hadamard product. In this way, the closest positive example is selected from its high similarity and high fidelity to the correct torque (embodied by ρ_t), and the closest negative example is selected for its similarity and low fidelity to the correct torque ($1 - \rho_t$ is high for bad outputs). Aside from these modifications, training proceeds in the manner as described in Section 2.3. The overall cost function that the model minimizes is the sum of the imitation learning loss given in Equation (2) and the memory loss given in Equation (3):

$$\begin{aligned}\mathcal{L}_{\mathcal{T}_i}(f_\phi, \mathcal{M}) = \sum_{\tau^{(j)} \sim \mathcal{T}_i} \sum_t \left\| f_\phi(\mathbf{o}_t^{(j)}) - \mathbf{a}_t^{(j)} \right\|_2^2 \\ + \max\{0, f_\phi(\mathbf{o}_t^{(j)}) \cdot K[n_b]_t - f_\phi(\mathbf{o}_t^{(j)}) \cdot K[n_p]_t + \alpha\}. \\ = \mathcal{L}_{\mathcal{T}_i}^{\text{MIL}}(f_\phi) + \mathcal{L}_{\mathcal{T}_i}^{\text{memory}}(f_\phi, \mathcal{M})\end{aligned} \tag{6}$$

As with our other implementations, we use the Adam optimizer and gradient descent for optmization. The learning process is summarized in Algorithm 3.

Algorithm 3 Meta-Imitation Learning with MAML and Explicit Memory

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β, η: step size hyperparameters
1: randomly initialize network parameters θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample demonstration $\tau = \{\mathbf{o}_1, \mathbf{a}_1, \ldots, \mathbf{o}_T, \mathbf{a}_T\}$ from $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using τ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation (6)
7: Compute adapted parameters: $\theta_i' \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}^{\text{MIL}}(f_\theta) - \eta \nabla_\theta \mathcal{L}_{\mathcal{T}_i}^{\text{memory}}(f_\theta, \mathcal{M})$
8: Sample demonstration $\tau_i' = \{\mathbf{o}_1', \mathbf{a}_1', \ldots, \mathbf{o}_T', \mathbf{a}_T'\}$ from $\mathcal{T}_i$ for the meta-update
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each τ_i' and $\mathcal{L}_{\mathcal{T}_i}$ in Equation (6)
11: **end while**
12: **return** parameters θ that can be quickly adapted to new tasks through imitations

4 Experiments

Our experiments on the Reptile meta-learning algorithm for MIL are designed to determine how the performance and efficiency of Reptile compares to MAML. As our performance metric we report the success rate of the model, and for our efficiency metric we report the time to train the model for 30,000 iterations on 15-way 1-shot mini-batches. We chose these values for the training iterations, way, and shot such to match the values used to train the 1-shot MIL model given in [1] to allow for a consistent comparison between the methods. In our experiments using the LRRE external memory, we explore using the memory in the pre-update head of the architecture as well as using the memory in both the pre and post-update heads of the architecture. For these LRRE experiments we report the these same metrics (success rate and training time).

4.1 Simulated Environment and Dataset

All of our experiments were conducted on the simulated pushing environment introduced in [1]. In this environment there 843 tasks where the goal of each task T_i is for the agent to push an object ob_i unique to task i onto a target location, which is marked by a red circle. For each task the object ob_i is placed on a table top in a random location next to a detractor object. Unless otherwise stated, we consider a trial a success if the agent is able to push the object onto the target location for 10 time-steps within a 100 time-step episode. Our training and test set splits are disjoint and contain 769 and 74 tasks respectively. For each task in the imitation learning dataset provided by [1] corresponding to the simulated pushing environment, there are a total of 24 demonstrations. Each of these demonstrations $\tau := \{o_1, a_1, ..., o_T, a_T\}$ was generated by an expert policy π^* that was trained on a single task T_i using trust-region policy optimization (TRPO). As previously mentioned, each observation $o_t \in \tau$ is a concatenation of a 125x125 dimensional RGB image (video-frame) of the expert agent at time t and a 20 dimensional vector containing the joint angles and end-effector pose of the expert agent at time t, which define it's state. In addition, each action $a_t \in \tau$ is a 7-dimensional vector containing the 7-DoF motor torques applied by the expert agent at time t. The figure below provides example images from the expert demonstrations for reference. In these images the target is the red circle, the task specific object is the multi-colored elephant, and the distractor object is the black cup.

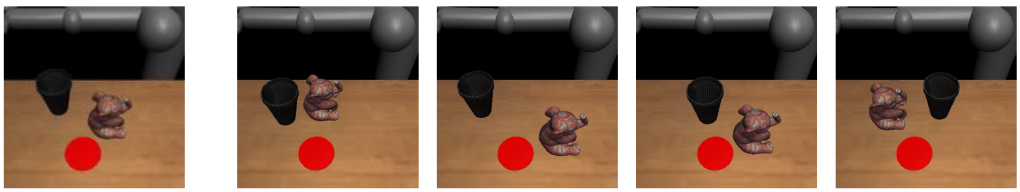

Figure 2: Examples of start of simulation for different initial placements of the same pair of target and distractor objects. In the simulation on the left, an expert demonstration generated by TRPO is performed. In the four simulations to the right, the MIL policy network performs the pushing task. This simulation environment was developed by Finn et al. using OpenAI Gym and MuJoCo [1].

4.2 Results

We follow the evaluation procedure used in [1] and report the 1-shot success rate of our models on the 74 test tasks for the pushing environment. Note that 1-shot indicates that the model is given one example from each task T_i to "learn" from at meta-test time before it must infer what action to take at each time-step in simulation on task T_i. In the table below we report the success rate, the training time (efficiency), and corresponding training speedup for each of our models. We note that all of these results were obtained on a single NVIDIA P100 GPU.

Method	Success Rate	Training Time	Speedup
LSTM	78.38*	NA	NA
MIL with MAML	**86.71****	24:52	1.00x
MIL with Reptile	18.24	**10:07**	**2.46x**
MAML + Memory (pre-update)	43.24	25:07	0.99x
MAML + Memory (pre and post-update**)	86.33	25:15	0.98x

Table 1: ** indicates the result is from our reproduction of [1]; the success rate of 85.81% given in [1] is slightly lower than the result we obtained. * indicates that the result was taken directly from [1]

We also tracked the usage rate of the memory module over training, as based on the memory confidence at each output step the model selects either the memory output or the original embedding. Desirably, the module's usage increases for both the pre-update and post-update heads over the course of training (viewable in Figure 3), demonstrating that the memory module adapts to meta-learning. For more analysis, see Section 4.3.

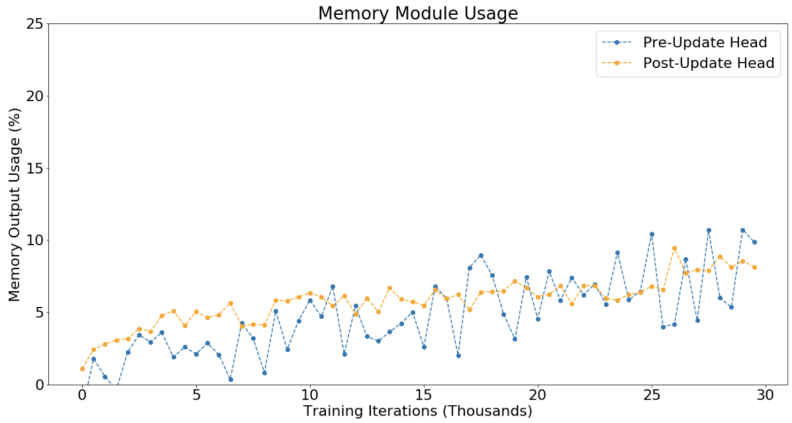

Figure 3: Memory usage increases during training for both the pre-update and post-update heads.

4.3 Discussion

Reptile

It is clear from the above table that the reptile method is significantly faster (roughly 2.5x) than the MIL with MAML method, but the performance of Reptile is much worse. The reason for this speedup is straight forward: Reptile saves time by not having to compute the second derivative, which is required to perform the MAML update. However, the reason for Reptile's under performance is not clear to us at this time. Our first working hypothesis on why Reptile does poorly is that Reptile does not perform as well as MAML with low-shot training batches. This hypothesis is supported by the fact that in all of the experiments presented with [6], the authors use a training shot of at least 10 examples per task. In our experiments, the training shot is set to 1 to match the training procedure of [1]. We believe this comparative lack of training data could be causing Reptile to perform poorly. The second plausible reason for Reptile's under performance is that we did not find the best settings for all of the hyper parameters used in the method. Of the nine total hyper parameters used in the method, 1. **meta-learning rate** 2. **Adam learning rate** 3. inner batch size 4. inner iterations 5. training shots 6. outer iterations 7. meta batch size 8. eval inner batch size 9. **eval inner iterations**, we only spent significant time tuning the three in bold. This was largely due to computational constraints as each training run was quite expensive; consequently, we focused on the hyper parameters we felt were most important.

MAML with Memory

We found that adding the explicit memory module to MAML provided both quantitatively and qualitatively different results from the original implementation, and the resulting performance was very sensitive to the location at which the memory module was attached. The best performance by far was found when the module was attached to the output layer. This is likely the case for two reasons. First, as referenced previously, attaching the module to a hidden layer in a feedforward network makes training more challenging because labeled output (in this case, desired hidden layer activations) are inherently difficult to produce due to the black-box nature of neural networks. This makes supervised learning of a hidden-layer memory module problematic. Second, even if the output of the memory module is linked directly to the output layer from the final convolutional layer in the policy architecture, all processing to that point has produced a representation only of the current state (the image and the robot position) and not of the task. All task-dependent processing occurs in the downstream fully-connected layers. In a meta-learning framework, where the agent must learn a variety of different tasks, task-dependent processing is highly relevant to the next action. We also strongly believe that a more thorough hyperparameter search would yield far more optimal results. The prolonged training time of the model and repeated hardware issues made such a search difficult.

However, the memory module also brought clear benefits. Quantitatively, the memory-enhanced model surpassed the performance of the original model from the paper [1], though not by a significant margin. Qualitatively, however, the memory-enhance module was observed to produce more precise pushes in comparison to the original model. To test this, we extended the number of time steps

(represented by video frames) that the target object was required to stay in the goal area to qualify a push as a success, with the default setting being 10 frames. Results, viewable in Table 2, show that as this requirement is made more strict, the performance of the memory-enhanced model begins to more noticeably surpass the original model. Qualitatively, this means that the memory-enhanced model was better able to definitively set the object in the correct place, without it later sliding out of bounds.

Number of Frames	MAML	MAML + Memory
10	86.7	86.3
20	85.8	85.6
30	82.7	83.3
40	77.7	79.1
50	72.1	73.7

Table 2: 1-Shot Success Rate for multiple trials of simulated pushing task. For each trial, each test object was required to remain on target for an increasing number of frames for simulated push to qualify as success.

Because the only difference between these models is the memory module, we can be sure that the different success rates are due to correct fetches from memory. The increased precision makes sense because the model is using exact movements that it has stored previously as being successful. Moreover, because the stored torques are averaged with each successful query during training, their precision is likely to be greater any single action produced by the deep network alone. For example, if two actions produce pushes that are successful but leave the object on different fringes of the target area, then their average (which is stored as the memory value) is likely closer to the center of the target than either of them individually.

Finally, there are several characteristics of the memory usage results plotted in Figure 3 that are worth noting. First, the memory module output is not used most on most outputs. This is not surprising, because the task of storing and selecting 7-degree real-valued torques is fundamentally more challenging than the classification setting in which the memory module was introduced. It is likely that the far greater size of the output space would be better served by a much larger memory size, though we lacked the computational resources to implement this. Second, though gradual, the module usage does increase throughout training, indicating that despite these obstacles, the memory module is learning effectively. Third, it is to be expected that the memory usage for the pre-update head is noisier than that of the post-update head, as because the pre-update head is processing previously-unseen tasks, its confidence in its output is more likely to vary.

5 Conclusions

In this work we have shown that Reptile is a much faster meta-optimization method for MIL; however, there is a large performance gap between Reptile and MAML that must be overcome before Reptile can become a viable alternative. We plan to continue work on our Reptile implementation to determine the exact cause of the failure mode we encountered and correct for it. We have also shown that external memory aids in more precise motor control. In future work, we plan to investigate this advantage further by developing a new environment with tasks that necessitate greater precision than pushing, such as placing different keys into locks. This would allow us to more rigorously quantify the impact of the addition of explicit memory to MAML. On a more general note, we would like to appeal to the community to focus on developing more sample efficient meta-learning algorithms. Although the methods we have used in this work (MAML and Reptile) are more data efficient than their supervised counterparts, they still require a large number training examples and many hours of training to achieve good performance.

References

[1] Chelsea Finn et al. “One-Shot Visual Imitation Learning via Meta-Learning”. In: *CoRR* abs/1709.04905 (2017). arXiv: 1709.04905. URL: http://arxiv.org/abs/1709.04905.

[2] Chelsea Finn, Pieter Abbeel, and Sergey Levine. “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks”. In: *CoRR* abs/1703.03400 (2017). arXiv: 1703.03400. URL: http://arxiv.org/abs/1703.03400.

[3] Lukasz Kaiser et al. “Learning to Remember Rare Events”. In: *CoRR* abs/1703.03129 (2017). arXiv: 1703.03129. URL: http://arxiv.org/abs/1703.03129.

[4] Marc G Bellemare et al. “The arcade learning environment: An evaluation platform for general agents”. In: *Journal of Artificial Intelligence Research* (2013).

[5] Pedro A. Tsividis et al. “Human Learning in Atari”. In: *Association for the Advancement of Artificial Intelligence* (2017).

[6] A. Nichol, J. Achiam, and J. Schulman. “On First-Order Meta-Learning Algorithms”. In: *ArXiv e-prints* (Mar. 2018). arXiv: 1803.02999 [cs.LG].

[7] Chelsea Finn. “Learning to Learn”. In: *Berkeley Artificial Intelligence Research Blog* (2017).

[8] Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural Turing Machines”. In: *CoRR* abs/1410.5401 (2014). arXiv: 1410.5401. URL: http://arxiv.org/abs/1410.5401.

[9] Sainbayar Sukhbaatar et al. “Weakly Supervised Memory Networks”. In: *CoRR* abs/1503.08895 (2015). arXiv: 1503.08895. URL: http://arxiv.org/abs/1503.08895.

[10] Diederik P. Kingma and Jimmy Lei Ba. “Adam: A Method For Stochastic Optimization”. In: *ICLR* (2015).